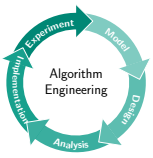

Bachelor Thesis

Optimizing a Parallel Graph Partitioner for Memory Efficiency

Daniel Salwasser

Date: March 28, 2024

Reviewers: Prof. Dr. Peter Sanders
T.T.-Prof. Dr. Thomas Bläsius
Advisors: Dr. Lars Gottesbüren
M.Sc. Daniel Seemaier

Institute of Theoretical Informatics, Algorithm Engineering
Department of Informatics
Karlsruhe Institute of Technology

Abstract

Graph partitioning is a classical NP-hard optimization problem with a wide range of practical applications. The problem is to divide the nodes of a graph into balanced blocks such that the sum of edge weights between different blocks is minimized. One challenge in graph partitioning is the partitioning of huge graphs due to their high memory footprint. Since graphs from real-world applications often outgrow the amount of RAM available on a single machine, recent work focuses on streaming, external memory or distributed partitioning. However, these approaches incur significant resource overheads or suffer from severe degradation in solution quality when compared to in-memory algorithms. In this thesis, we therefore deal with the memory optimization of the in-memory graph partitioner KaMinPar, to scale fast and high-quality in-memory graph partitioning to substantially larger graphs. We present algorithmic changes, more memory-efficient data structures, and a graph compression scheme and integrate them into KaMinPar. These memory optimizations enable us to partition some of the largest real-world graphs with less than 100 GB of RAM, down from over 800 GB without optimizations.

Graphpartitionierung ist ein klassisches NP-schweres Optimierungsproblem mit einer Vielzahl von praktischen Anwendungen. Das Problem besteht darin, die Knoten eines Graphen in balancierte Blöcke aufzuteilen, sodass die Summe der Kantengewichte zwischen verschiedenen Blöcken minimiert ist. Eine Herausforderung bei der Graphpartitionierung ist die Partitionierung von riesigen Graphen aufgrund ihres hohen Speicherbedarfs. Weil für Graphen von realen Anwendungen oft der verfügbare RAM einer einzelnen Maschine nicht ausreicht, beschäftigen sich neuere Arbeiten mit Streaming, External Memory und verteilten Graphpartitionierern. Diese Ansätze verursachen im Vergleich mit in-memory Algorithmen jedoch einen erheblichen Ressourcen-Overhead oder leiden unter einer starken Verschlechterung der Lösungsqualität. Deshalb beschäftigen wir uns in dieser Thesis mit der Speicheroptimierung des in-memory Graphpartitionierers KaMinPar, um schnelle und qualitativ hochwertige in-memory Graphpartitionierung auf wesentlich größere Graphen zu skalieren. Wir stellen algorithmische Änderungen, speichereffizientere Datenstrukturen und ein Graphkompressionsschema vor und integrieren diese in KaMinPar. Mit diesen Speicheroptimierungen können wir einige der größten realen Graphen mit weniger als 100 GB an RAM partitionieren, anstelle von über 800 GB ohne Optimierungen.

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

Karlsruhe, den 28. März 2024

Contents

1 Introduction

Graphs are often used abstractions to represent objects and the relationships between these objects. One fundamental problem in computer science is the graph partitioning problem. The problem is to determine a partition of the nodes into blocks such that the partition is fairly balanced and the weight of the edges between different blocks is minimized. Because the problem is NP-hard [22] and also NP-hard to approximate by a constant factor [7], heuristic methods are used in practice. A very successful heuristic method to compute such partitions is multilevel graph partitioning [20]. The multilevel scheme consists of three steps: coarsening, initial partitioning and uncoarsening. During coarsening, the graph to partition is first coarsened and thus approximated by smaller graphs. When the graph is small enough, an initial partition is computed on the coarsest graph. Because the graph is small, expensive methods can be used in this step. Subsequently, during uncoarsening, the partition is iteratively projected onto the finer graphs and improved by refinement algorithms. In this step, refinement algorithms can find improvements more easily because of the global overview of the input graph, which is provided by the coarse graphs [8].

In practice, there is a need to partition huge graphs [28], which are for example used to represent parts of the web, social networks or biological networks like the brain. However, due to memory constraints, they usually cannot be partitioned by an in-memory multilevel graph partitioner. Therefore, such graphs are partitioned using streaming [39, 13], external memory [1] or distributed [36] algorithms. Streaming partitioners only read in individual nodes or a group of nodes and assign them on-the-fly to a partition, allowing them to be run with less memory by only holding few nodes and their neighborhood in memory at a time. External memory partitioners use read and write operations on external memory to compute a partition and are therefore not dependent on the RAM of a machine. Distributed graph partitioners run on multiple machines and split the graph into parts, where each machine processes a part and communicates by messages with the other machines.

However, those approaches have disadvantages. Streaming partitioners produce partitions of lower quality because they do not utilize global properties of the graph. External memory partitioners use expensive IO operations, which is why there are slower than in-memory partitioners. Distributed partitioners require server clusters, which is why they are very expensive to run. In-memory partitioners on the other hand are fast, can be run inexpensively on a single machine and exploit the global structure of the graph and thus produce partitions of high quality. However, they are limited by the main memory of a single machine. It is therefore necessary to reduce the memory consumption of in-memory partitioners in order to partition huge graphs with them.

1.1 Problem Statement

The goal of this thesis is to optimize an in-memory multilevel graph partitioner for memory efficiency. With this we want to be able to partition huge graphs on a single shared-memory machine using an in-memory graph partitioner, which was not possible before due to high memory consumption or only possible on machines with a lot of RAM. The multilevel graph partitioner, which we are modifying to be more memory-efficient, is KaMinPar [19]. As we will see later in this thesis, the main memory consumption problem of KaMinPar is the coarsening step and the representation of the input graph in memory. In this thesis, we therefore want to reduce the memory consumption of the coarsening step and hold the input graph in memory with a more space-efficient representation. Furthermore, we still want to produce partitions of high quality and not lose much running time because of the memory optimizations.

1.2 Contribution

In this thesis, we present techniques to reduce the memory consumption of an in-memory multilevel graph partitioner. That include changes to the algorithms used during the clustering and contraction step of coarsening as well as the use of graph compression to store the input graph more space-efficiently. Furthermore, we integrate these memory optimization techniques into KaMinPar and thereby reduce its memory consumption. Figure 1.1 shows how each memory optimization, which we present in this thesis, affects the peak memory of KaMinPar on 64 threads for one of the largest real-world graphs. We reduce its memory consumption from 802.4 GB to 92.9 GB by a factor of 8.6. We therefore demonstrate that in-memory multilevel graph partitioners can be used to partition huge graphs on a single machine equipped with a reasonable amount of memory.

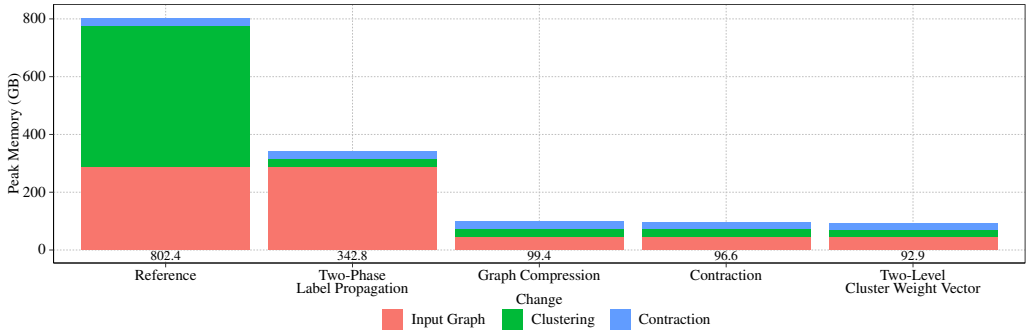

Figure 1.1: Burndown plot showing the reduction in peak memory from our optimizations for the clueweb12 graph on 64 threads, number of blocks $k = 128$, imbalance factor $\varepsilon = 0.03$ and not duplicating the coarse graphs during coarsening. Note that only the input graph, clustering step and contraction step are responsible for the peak memory for this graph.

1.3 Structure of Thesis

In Chapter 2, we first define notations, definitions and concepts for this thesis. Then, in Chapter 3, we present research on graph partitioning and graph compression, which is relevant to this thesis. Next, we present our memory optimizations. We begin Chapter 4 by analyzing the memory consumption of KaMinPar and identifying components that are responsible for a high memory consumption. Afterwards, in Chapter 4, we describe the changes to the coarsening phase. In Chapter 5, we describe how we use graph compression to store the input graph with a space-efficient representation in memory. Finally, we evaluate our memory optimizations in Chapter 6 and conclude this thesis in Chapter 7.

2 Preliminaries

In this chapter, we introduce the notations and concepts used in this thesis. We first introduce notations and definitions about graphs. Then, we define the problem of graph partitioning. We also introduce the adjacency array, which is a representation of graphs in memory. Next, we introduce graph compression and describe common techniques to represent graphs in a compressed form. Furthermore, we describe the size-constrained label propagation algorithm. Finally, we describe memory overcommitment, which is a technique to allocate memory before the required memory space is known.

2.1 Graphs

An undirected weighted graph, which we simply call graph, is a tuple $G = (V, E, c, \omega)$. It consists of finite sets V and $E \subseteq \binom{V}{2}$, whose elements are called nodes and edges, respectively. G also consists of a function $c : V \to \mathbb{N}_{>0}$, which assigns a weight to each node, and a function $\omega : E \to \mathbb{N}_{>0}$, which assigns a weight to each edge. We denote the size of the sets V and E as $n := |V|$ and $m := |E|$. We also identify the nodes with the natural numbers from 0 to $n - 1$ and the edges with the natural numbers from 0 to $m - 1$. Furthermore, we denote the number of nodes n as the order of G and the number of edges m as the size of G. An edge $e = \{ u, v \} \in E$ is also denoted by $e = uv$. We call two nodes $u, v \in V$ adjacent if there exists an edge $e = uv \in E$, and we say that an edge $e \in E$ is incident to a node $w \in V$ if w is one of the endpoints of e, i.e., $e = wx$ for another node $x \in V$. We call a node with no incident edges an isolated node. The neighborhood $N(u)$ of a node $u \in V$ is the set of nodes that are adjacent to u, i.e., $N(u) = \{ v \in V \mid uv \in E \}$. We call the nodes in $N(u)$ the neighbors of u. The degree $\deg(u)$ of a node $u \in V$ is the number of nodes that are adjacent to u, i.e., $\deg(u) = |N(u)|$. With $\Delta(G)$ and $d(G)$ we denote the maximum degree and average degree of G, respectively.

Let $C_1, \ldots, C_l \subseteq V$ be disjoint subsets of nodes, which collectively contain all nodes, a so-called clustering. Then, a new graph is created by contracting the clustering. A clustering is contracted by identifying each cluster C_i with a single node i. The weight of a new node i is thereby given by $\sum_{v \in C_i} c(v)$. Furthermore, there is an edge between two new nodes i and j with $i \neq j$ if there is an edge $uv \in E$ in the original graph with $u \in C_i$ and $v \in C_j$. Note that we do not allow self-loops and that we reduce parallel edges. The weight of such an edge is given by $\sum_{u \in C_i, v \in C_j, uv \in E} \omega(uv)$.

2.2 Graph Partitioning

A k-way partition $\Pi = \{ V_1, \ldots, V_k \}$ of a graph G is a partitioning of the nodes V into k disjoint sets, which we call blocks, i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i, j \in \{ 1, \ldots, k \}$ with $i \neq j$. We define the cut edges E_{ij} for $i, j \in \{ 1, \ldots, k \}$ with $i \neq j$ to be the set of edges with one endpoint in the vertex set V_i and one endpoint in the vertex set V_j, i.e., $E_{ij} = \{ uv \in E \mid u \in V_i, v \in V_j \}$. The cut of a k-way partition Π is the sum of the weights of all cut edges, i.e., $\text{cut}(\Pi) = \sum_{i \neq j} \omega(E_{ij}) \coloneqq \sum_{i \neq j} \sum_{e \in E_{ij}} \omega(e)$.

The balance constraint for a k-way partition Π requires that the total node weight of the graph is split evenly with respect to some imbalance parameter $\varepsilon > 0$ among the blocks $V_1, \ldots, V_k$ of the partition, i.e., $c(V_i) \leq (1 + \varepsilon) \frac{c(V)}{k}$ for all $i \in \{ 1, \ldots, k \}$.

The graph partitioning problem is defined to be the problem of finding a k-way partition with minimal cut, which satisfies the balance constraint for a given graph G, number of blocks k and imbalance parameter ε [18]. The problem is NP-hard [22] and NP-hard to approximate by a constant factor [7].

2.3 Adjacency Array

An adjacency array is a static representation of a graph $G = (V, E, c, \omega)$. In this context, static means that the graph structure cannot be modified, i.e., insertion and deletion operations of nodes or edges are not supported. The adjacency array $\mathcal{A} = (\mathcal{N}, \mathcal{E}, \mathcal{C}, \mathcal{W})$ consists of four arrays, which are each stored contiguously in memory. $\mathcal{N}$ is called the node array and stores for each node $u \in V$ the offset $\mathcal{N}[u]$ into the edge array $\mathcal{E}$ where the neighborhood of u is encoded. Furthermore, the node array stores the number of edges m in the entry n. This allows the degree of each node u to be determined by $\deg(u) = \mathcal{N}[u+1] - \mathcal{N}[u]$. The edge array $\mathcal{E}$ stores for each node $u \in V$ its adjacent nodes $N(u) = \{ v_1, v_2, \ldots, v_d \}$ contiguously, i.e., $\mathcal{E}[\mathcal{N}[u]] = v_1, \mathcal{E}[\mathcal{N}[u]+1] = v_2, \ldots, \mathcal{E}[\mathcal{N}[u]+d-1] = v_d$. Moreover, the adjacency array consists of a node weight array $\mathcal{C}$ and an edge weight array $\mathcal{W}$. The node weight array stores for each node $u \in V$ its weight, i.e., $\mathcal{C}[u] = c(u)$. Likewise, the edge weight array stores for each edge $e \in E$ its weight, i.e., $\mathcal{W}(e) = \omega(e)$. Figure 2.1 gives an example of an adjacency array.

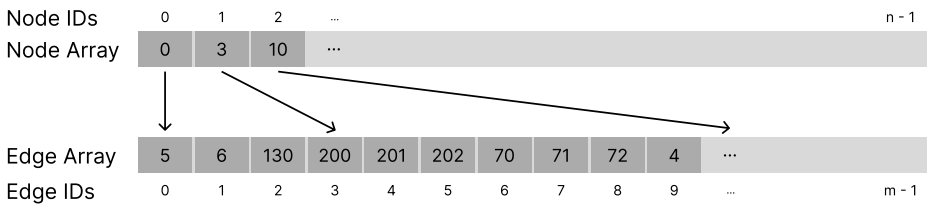

Figure 2.1: The structure of the node and edge array of an adjacency array.

2.4 Graph Compression Techniques

Graph compression is the process of representing a graph in memory with an encoding that uses less memory space than its original representation [2]. Lossless graph compression has the additional constraint that the compressed graph representation does not lose any information about the nodes, edges, node weights or edge weights of the graph. In the following, we present three techniques for lossless graph compression, which can be applied to the adjacency array to make it more space-efficient.

Gap Encoding. With gap encoding, node IDs are not stored directly but as differences between consecutive node IDs [2]. Thus, for a node u, the node IDs of its neighbors $N(u) = \{ v_1, v_2, \ldots, v_d \}$ are for example encoded as $v_1 - u, v_2 - v_1, \ldots, v_d - v_{d-1}$ or a similar construction. The idea of this encoding is that the differences are smaller values than the IDs themselves, and therefore the stored value takes up less space due to variable-length encodings.

Variable-Length Encoding. Variable-length encoding describe integer encodings with a variable number of bytes per integer, where the number of used bytes for each integer is proportional to its value [2]. This means that smaller values occupy fewer bytes and larger values occupy more bytes. This encoding can be applied to the node IDs that are stored in the edge array of an adjacency array. The advantage of this encoding is that possibly fewer bytes per integer are used to store them. This advantage is particularly utilized if the node IDs are not stored directly but stored in an encoded form, e.g., by gap encoding.

Variable-length integer (VarInt) is one type of variable-length encoding, which uses the most significant bit (MSB) of a byte to indicate, whether the following byte is part of the current integer [2]. Therefore, the MSB is also called the continuation bit. To encode a binary-encoded integer as a VarInt, first split it into chunks of 7 bits. Then, from the lowest to the most significant chunk add a continuation bit to each chunk such that it becomes a byte. Further, set the continuation bit if the following chunk is non-zero. Otherwise, stop at that chunk and concatenate the current and all previous created bytes. Figure 2.2 gives an example of a VarInt.

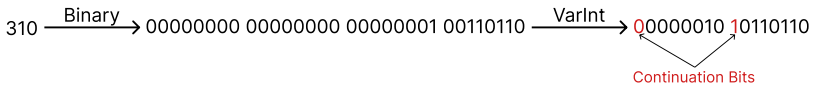

Figure 2.2: An example of a 32-bit integer encoded as a VarInt.

Because the node IDs can be stored with an alternative encoding such as gap encoding, which can lead to some node IDs becoming negative, the representation of signed integers as VarInts is required to save memory space when storing possibly negative integers. One way

to store signed integers as VarInts is to use zigzag encoding. Zigzag encoding transforms a signed integer to an unsigned integer with the following mapping [4]:

$$z(x) = \begin{cases} 2x, & x \geq 0 \\ 2|x| - 1, & x < 0 \end{cases}$$

Thus, to encode a signed integer x as a VarInt, first map it with zigzag encoding to an unsigned integer $z(x)$ and then encode $z(x)$ as a normal VarInt.

Interval Encoding. Interval encoding is a technique where instead of storing a consecutive part of a neighborhood $v, v+1, \ldots, v+l \in N(u)$ of some node u directly, only the start of the interval v and its length l are stored [2].

2.5 Size-Constrained Label Propagation

Label propagation is an algorithm for computing a node clustering in a graph [34]. At the start of the algorithm, each node in the graph is assigned to its own cluster. Then, it works in multiple rounds. At the beginning of each round, a random ordering of the nodes is selected. Afterwards, the nodes are iterated in the selected ordering. Each node is thereby moved to the cluster that most neighbors are assigned to. If there are several clusters that have the same amount of neighbors assigned to, then one of these clusters is selected uniformly at random. The rounds are repeated until the process converges.

Size-constrained label propagation is an extension of label propagation, which ensures that the weight of each cluster is constrained by a maximum weight, where the maximum weight is a parameter of the algorithm [32]. This algorithm works like label propagation, but instead of moving a node to a cluster with the most neighbors, a node is moved to a cluster with the most neighbors that additionally does not violate the maximum size constraint when the node is moved.

2.6 Memory Overcommitment

In some scenarios, it is necessary to store aggregated data, whereby the required storage space is only known after the data has been aggregated. Therefore, in such scenarios, the aggregated data is often stored in a temporary buffer or computed twice, determining the size the first time to allocate enough storage space, and then storing the data the second time. However, both approaches have disadvantages. With the former, additional memory is allocated and the latter costs running time. Another way to resolve the problem is to trick the virtual memory with a technique called memory overcommitment [31].

On operating systems that allow memory overcommitment, memory allocations that request more memory than is physically available are allowed. These operating systems allow this because they only assign virtual pages to physical pages when the corresponding page is touched. This means that only the memory pages that are touched are used and take up actual memory. This behavior can be used for the described scenario. For that, memory is overcommited by requesting enough memory to store the aggregated data. This can be done by either computing an upper bound or allocating all the physically available memory. Furthermore, only the memory that stores the aggregated data is written to and read from. This ensures that only these memory pages are mapped to physical pages and that therefore no memory is wasted.

3 Related Work

In this chapter, we provide an overview of literature and research that is relevant to this thesis. We first describe the multilevel paradigm, which is arguably the most successful approach for heuristically solving the graph partitioning problem. Then, we give an overview of the KaMinPar graph partitioner as the memory optimization techniques described in this thesis have been applied to and evaluated on KaMinPar. We also present the Ligra+ and the WebGraph Framework, which are frameworks for compressing graphs, that both have been very influential to our own graph compression scheme.

3.1 Multilevel Graph Partitioning

As the graph partitioning problem is NP-hard [22] and NP-hard to approximate by a constant factor [7], heuristic algorithms are used in practice to solve the problem. One technique that has been very successful in doing so is multilevel graph partitioning (MGP) [20]. The multilevel approach consists of three phases: coarsening, initial partitioning and uncoarsening.

In the first step, the so-called coarsening, the input graph is successively coarsened to smaller approximations of the input graph. Typically, this is done by computing a clustering of the nodes and contracting the clustering to create a coarse graph. Parallel edges, which might emerge from contraction, are thereby replaced by a single edge, whose weight is the sum of the weights of the parallel edges. Likewise, the weight of a coarse node is the sum of the node weights in the corresponding cluster. Graphs are coarsened until the order of the smallest graph reaches a threshold or the coarsening converges [20, 19]. After coarsening, an initial partitioning of the smallest coarse graph is computed. Since the coarsening step creates graphs, which maintain the weights of nodes and edges of the original graph, a balanced partition of a coarse graph corresponds to a balanced partition of the finer graph [20]. In the last step, starting from the initial partition, each partition of the coarse graph is successively applied to the finer graph. This is done by projecting the partition class of a coarse node onto the set of nodes from whose contraction the coarse node was created. This process is called uncoarsening. In addition, with each uncoarsening, the partition of the finer graph is improved with refinement algorithms [20].

There are two primary ways to implement MGP: recursive bipartitioning and direct k-way partitioning [19]. Recursive bipartitioning computes in the three steps described above a bipartition of the input graph. Then, it recursively computes further bipartitions of the subgraphs induced by the partition until the desired number of blocks is given. With direct

k-way partitioning only one coarsening and uncoarsening step is performed as the initial partitioning step directly computes a partition into k blocks.

One reason why the multilevel approach is so successful, is that expensive partitioning algorithms can be applied to the last coarse graph as it is typically much smaller than the input graph. This can include more powerful algorithms or a portfolio of different fast algorithms [8]. Further, during the refinement step, good improvements can be found on the coarse levels, which would be difficult to find on the finer levels, as the coarse graphs provide a more global overview of the fine graphs [8].

3.2 KaMinPar

KaMinPar is a parallel in-memory multilevel graph partitioner, which allows for balanced partitions of high quality for a large number of blocks [19]. For that, the authors introduce the deep multilevel graph partitioning scheme, which is a multilevel approach that combines recursive bipartitioning and direct k-way partitioning. With deep MGP, coarsening is applied until the coarsest graph has order of about $2C$, where C is an input parameter. Then, a bipartition of the coarsest graph is computed. During uncoarsening, the partition is extended through further bipartitioning such that the subgraphs induced by a partition have order of about $2C$ and the partition of the input graph has the desired number of blocks.

One benefit of the deep multilevel scheme compared to direct k-way partitioning is that only small graphs are bipartitioned. This is because the coarsening step of direct k-way partitioning typically terminates when the coarsest graph has order of about kC. Deep MGP coarsens, for large k, much deeper with only about $2C$ nodes left and tries to maintain this order for the subgraphs to bipartition during uncoarsening [19]. Therefore, deep MGP allows for partitions with larger number of blocks as initial partitioning only operates on small graphs, and thus does not become a bottleneck for running time or parallelism [19].

Moreover, deep MGP only performs one coarsening and uncoarsening step, unlike recursive bipartitioning, and therefore has a better asymptotic running time. It also allows for refinement algorithms to operate on more than two blocks, and therefore has the potential for better partition improvements during the refinement step compared to recursive bipartitioning, which can only refine two blocks [19].

Furthermore, the deep multilevel scheme fully exploits parallelism, by ensuring that parallel work of p processing elements (PEs) is performed on graphs of order at least pC [19, 10]. This is ensured by duplicating the coarse graph and splitting the PEs into two groups of size $\lceil P/2 \rceil$ and $\lfloor P/2 \rfloor$ when the coarse graph becomes smaller than pC. Each group of processors then works on a separate copy of the duplicated coarse graph. This results in different partitions of the duplicated graph. During uncoarsening, the partition that is balanced and has the smallest edge cut is selected. This allows the diversified search for a good partitioning with only minimal additional running time costs.

3.3 Ligra+

Ligra+ is a framework for in-memory graph processing, which builds upon the Ligra framework by integrating graph compression techniques [38]. The Ligra framework itself consists of a directed graph and node subset data structure and provides, besides basic graph operations such as order, size or degree queries, two operations: NodeMap and EdgeMap [37]. The operations are used for mapping over the nodes and edges of a subset.

Ligra+ extends Ligra by storing the graph in a compressed form and adapting the EdgeMap operation to support the compressed graph form. The compressed graph representation is an adjacency array with a compressed edge array, which integrates both edges and edge weights. The compressed edge array uses two compression techniques, namely variable-length encoding and gap encoding.

With this compression scheme, the authors achieve compression ratios between 1.79 and 2.04 with running times ranging from 1.1 times slower to 2.2 times faster than Ligra on their graph benchmark set and six applications: Breadth-first search, betweenness centrality computation from a source node, graph radii estimation, connected components, PageRank and Bellman-Ford shortest path [38]. Note that the authors use different graph reorderings and pick for each graph the reordering with the best compression ratio.

3.4 WebGraph Framework

The WebGraph Framework is a framework equipped with graph compression, which is targeting web graphs [4]. It uses variable-length encoding, reference encoding, gap encoding and interval encoding as its compression techniques. The framework provides several variable-length encodings such as ζ, which is designed for integers with a power-law distribution with small exponent, Elias γ or Golomb [3]. Reference encoding is a technique that identifies two nodes with a similar neighborhood and stores one of the neighborhoods as a reference to the other neighborhood using a copy list. The copy list marks all the adjacent nodes in the referenced neighborhood that are also adjacent nodes in the own neighborhood. The remaining adjacent nodes of a neighborhood, which are not covered through the referenced neighborhood, are stored using interval and gap encoding. Interval encoding identifies intervals in the neighborhood of a node of a minimum length, which is a parameter chosen by the user. An interval is then stored by encoding its left extreme and its length. The remaining adjacent nodes, which are not covered through reference and interval encoding, are stored with gap encoding.

With this compression scheme, the authors achieve a compression ratio of 6.40 resp. 8.09 for webbase2001 resp. uk2002 with the best configuration that they tested. With the worst configuration that they tested, they achieve a compression ratio of 4.12 resp. 4.23 for webbase2001 resp. uk2002. Note that the compression scheme with the best configuration is slower than with the worst configuration in decoding due to additional decoding work.

4 Reducing Memory Consumption during Coarsening

In this chapter, we describe how we reduce the memory consumption of KaMinPar during coarsening. To this end, we first analyze the memory consumption of KaMinPar and identify the components of the algorithm with the highest memory consumption. Then, we describe how we change these components to consume less memory space.

4.1 Memory Consumption Analysis of KaMinPar

We illustrate the memory consumption of KaMinPar using an example. To do this, we use the graph with the highest memory consumption in our benchmark set. Figure 4.1 shows the memory consumption of the KaMinPar algorithm before and after our optimizations for this graph. In this figure, the memory consumption is logically split into two parts, namely coarsening and uncoarsening as initial partitioning is fused into uncoarsening in KaMinPar. We see that the memory peak is reached in the first coarsening level and that clustering requires 57.4 GB of memory, followed by the input graph with 7.7 GB of memory and then followed by contraction with 3.8 GB of memory. We also see that the other components as well as uncoarsening are not responsible for the memory peak.

Therefore, besides the input graph that we address in Chapter 5, the two components of the coarsening phase must be changed to reduce the memory consumption. Recall that during coarsening, a clustering of the graph to coarse is first computed. The graph is then contracted according to that clustering to create a coarse approximation of the input graph. For computing a clustering, KaMinPar uses the label propagation algorithm. It accounts for most of the memory consumption of clustering. As we see, the memory required for contracting is also important for the total memory consumption. Moreover, clustering and contraction only have a high memory consumption in the first coarsening level because the graph to process is still large at that level, as opposed to the following levels, where the graphs are much smaller. We also see in Figure 4.1 that the memory used for initial partitioning and uncoarsening is not relevant for the total memory consumption. This is because initial partitioning in the deep multilevel scheme only works on very small graphs. Similarly, the memory consumption during uncoarsening is small because it mainly needs memory for the refinement step. The refinement algorithm used is also label propagation, where the initial number of clusters is the number of blocks of the partition. Thus, it has

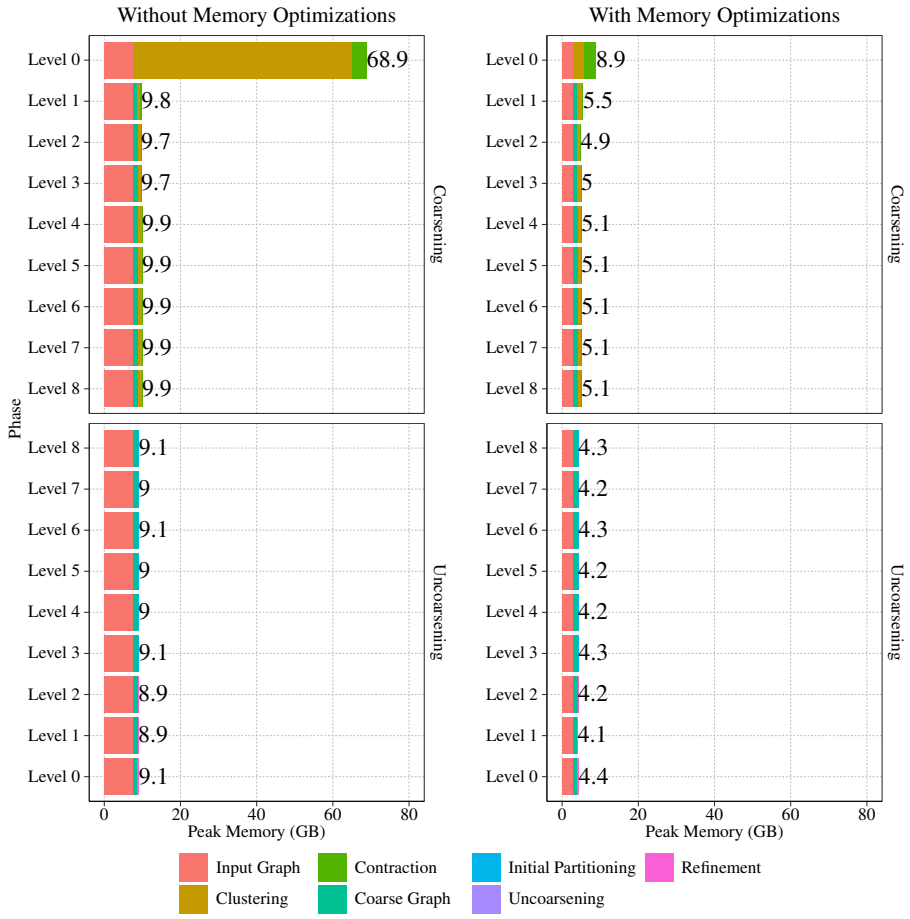

Figure 4.1: Peak memory during the different phases of the algorithm. The memory measurements were carried out for the webbase2001 graph with number of blocks $k = 128$ and imbalance parameter $\varepsilon = 0.03$ on 64 PEs and not duplicating graphs during coarsening. The left plot shows the memory consumption before memory optimizations and the right plot with our memory optimizations enabled.

a memory consumption of $\mathcal{O}(k)$ per PE, which is typically much smaller than the graph to partition. In addition, a balancer is used during refinement because the partition may become unbalanced on a fine graph due to the balance constraint equation that KaMinPar uses. This balancer consumes memory in $O(\widetilde{n})$, where $\widetilde{n}$ is the order of the graph to balance. This is therefore also usually smaller than the input graph.

As we see, changes in the clustering implementation are crucial for reducing the memory consumption. In Section 4.2, we therefore describe a modification of the label propagation algorithm to reduce the memory consumption during clustering. We also reduce the amount of memory required to compute a clustering by using a more memory-efficient data structure in the process. We describe this change in Section 4.3. Finally, in Section 4.4, we describe changes to the contraction algorithm to reduce its memory consumption.

4.2 Two-Phase Label Propagation

As label propagation during coarsening is the largest contributor to the peak memory usage, we focus on optimizing this stage first. Recall that label propagation visits nodes in a random order in parallel and moves them to the adjacent cluster with the majority of neighbors.

Determining the best target cluster is implemented by counting the weight of edges to adjacent clusters and selecting the cluster with the highest weight. As the maximum number of adjacent clusters is $n - 1$, this computation uses an array of size n per PE, which is responsible for the largest amount of memory consumed by webbase2001 with 57.4 GB. Additionally, for low-degree nodes, a small fixed-capacity hash table is used to attain better cache efficiency, as random access writes to the large array are avoided.

Ideally, one would like to use the hash table for all nodes. However, the number of adjacent clusters for each node is unknown prior to counting. Therefore, growing hash tables have to be used, which are slower than the full-size array. In addition, nodes with many adjacent clusters can be present, which is why the hash tables might even grow to size n per PE.

Algorithm 1 depicts one iteration of label propagation as used in KaMinPar. Note that this implementation is simplified, for the reason that we are interested in memory efficiency. Furthermore, the capacity of the small hash table is 10 000, which is why this value is used as the threshold in Line 5.

Algorithm 1: One iteration of label propagation

```
1  Function PerformIteration(G = (V, E, c, ω), C, W)
       Input: Graph for which a clustering is computed, mapping from nodes to
              clusters, mapping from clusters to weights
2      Ṽ ← SelectNodeOrder(G)
3      for u ∈ Ṽ do parallel
4          R ← ⊥
5          if deg(u) ≤ 10000 then
6              R ← Thread-local small hash table
7          else
8              R ← Thread-local array
9          for v ∈ N(u) do
10             c_v ← C[v]
11             R[c_v] ← R[c_v] + ω(uv)
12         c_new, δ_weight ← SelectBestCluster(C, W, R)
13         MoveNode(C, W, u, c_new, δ_weight)
14         Reset(R)
```

We see in Line 8 of Algorithm 1 that in the worst case it is possible for each PE to use an array of size n. Since the hash tables have constant size, the memory consumption is thus $\mathcal{O}(np)$, where p is the number of PEs. Our goal is now to modify the implementation of the label propagation algorithm used by KaMinPar to avoid memory consumption that scales with the number of PEs. We want the total memory consumption of the label propagation implementation to be in $\mathcal{O}(n)$. For our example graph webbase2001, this would mean that instead of 57.4 GB only 3.2 GB of memory would be required for label propagation. To achieve this, we split label propagation into two phases. In the first phase, we process all nodes whose adjacent clusters can be aggregated in a hash table. Note that we still iterate over the nodes in parallel during the first phase. In the second phase, all nodes for which an array is required are processed. However, we no longer iterate over the nodes in parallel in the second phase. Instead, we iterate sequentially so that only one array has to be allocated because the adjacent clusters of only one node are aggregated at a time. To utilize parallelism nevertheless, we iterate in parallel over the neighborhood of a node, which is fine because the degree is large enough to achieve efficient per-node parallelization. With this two-phase approach, we reduce the memory consumption because in the first phase a hash table of constant size is used per PE and in the second phase only a single array of size n is used. Therefore, the memory consumption is in $\mathcal{O}(n)$ instead of $\mathcal{O}(np)$.

Algorithm 2 shows the second phase of two-phase label propagation. To ensure that the second phase is executed correctly in parallel, a few aspects must be taken into consideration. First, atomic fetch-and-add operations are used to update the edge weights to the adjacent clusters. This is implemented in Line 6. Moreover, used entries in the array are stored for faster iterations and resets by inserting the position of an entry to a dynamically growing vector the first time it is incremented. Therefore, to avoid race conditions when storing an entry as used, only the PE that first increments an entry inserts the corresponding position to a dynamically growing thread-local vector. Here, a PE determines whether it is the first one that increments an entry by comparing if the return value of the atomic fetch-and-add operation is equal to zero. This works because initially all clusters have weight zero and through the atomic fetch-and-add operation only one PE is guaranteed to read the zero value when updating. This is implemented in Lines 7–8. Finally, to select the cluster to move the node to, each PE determines in Lines 10–12 the cluster with the most neighbors, from the stored clusters in its own thread-local vector. In Lines 13–14, the overall best cluster is then selected from all the clusters selected as the best by the individual PEs.

So far, we have described how we split the label propagation algorithm, such that in the first phase all nodes that can be aggregated in a hash table are processed and in the second phase all remaining nodes are processed. In the following, we describe how we actually select nodes for the first and second phase. The requirements for a selection strategy are that it does not perform too much redundant work in the first and second phase as well as not flood the second phase with nodes with few adjacent clusters, which results in bad parallel efficiency. We have therefore explored two selection strategies, both of which fulfill the requirements with certain trade-offs.

Algorithm 2: Second phase of two-phase label propagation

```
1  Function PerformSecondPhase(G = (V, E, c, ω), C, W, U)
       Input: Graph for which a clustering is computed, mapping from nodes to
              clusters, mapping from clusters to weights, nodes of the second phase
2      R ← Array used for the second phase
3      for u ∈ U do
4          for v ∈ N(u) do parallel
5              c_v ← C[v]
6              ω_prev ← R[c_v] +=_atomic ω(uv)
7              if ω_prev = 0 then
8                  Add c_v to thread-local vector
9          T ← Array of size p
10         for all threads i do parallel
11             Select best cluster c_local, δ_weight from thread-local vector
12             Store c_local, δ_weight at T[i]
13         c_new, δ_weight ← SelectBestCluster(C, W, T)
14         MoveNode(C, W, u, c_new, δ_weight)
15         Reset(R)
```

High-Degree Selection. One strategy to select nodes for the second phase is to make it dependent on whether they have a high degree, and thus potentially many clusters in their neighborhood. In other words, the nodes with a degree greater than a threshold are selected for the second phase. We choose 10 000 as the threshold because it is the capacity of the hash table. This means, that all nodes whose adjacent clusters potentially cannot be aggregated in a hash table are processed in the second phase. However, nodes that can theoretically be processed in the first phase are also processed in the second phase. This is exactly the case when the degree exceeds the threshold, but the number of clusters in the neighborhood is smaller than or equal to the threshold.

Full-Rating-Map Selection. If nodes are selected for the second phase, whose degree is greater than the threshold but whose actual number of adjacent clusters is lower, the algorithm can be slowed down because processing in the second phase requires additional synchronization due to parallel aggregation. Furthermore, the synchronization to determine the best adjacent cluster is also a potential issue since many short bursts of parallelism can be harmful. We therefore present another strategy, which selects nodes for the second phase depending on whether the neighborhood must be aggregated in the second phase because it contains too many adjacent clusters.

With this strategy all nodes are processed in the first phase using a hash table and only bumped to the second phase when the hash table is full. The downside of this selection strategy is that it wastes work by throwing away partially aggregated results of a node when it is selected for the second phase. However, a reason why this selection strategy can work well is that the nodes are visited in increasing order of degree with KaMinPar. When nodes with low degree are processed first, they are processed in the first phase because they cannot have many adjacent clusters due to their small neighborhood. Thus, at the beginning, the number of possible clusters is reduced while only using the first phase. When nodes with a higher degree, which are more likely to be selected for the second phase due to their bigger neighborhood, are processed, fewer clusters are present since the low degree nodes have already been moved. Therefore, the likelihood of being selected for the second phase intuitively decreases as more nodes are processed. Algorithm 3 shows one iteration of two-phase label propagation, which uses one of the two presented selection strategies.

Algorithm 3: One iteration of two-phase label propagation

```
1  Function PerformTwoPhaseIteration(G = (V, E, c, ω), C, W)
       Input: Graph for which a clustering is computed, mapping from nodes to
              clusters, mapping from clusters to weights
2      U ← ∅
3      Ṽ ← SelectNodeOrder(G)
4      for u ∈ Ṽ do parallel
5          if deg(u) > 10000 and use high-degree strategy then
6              U ← U + u
7          else
8              R ← Thread-local small hash table
9              move ← true
10             for v ∈ N(u) do
11                 c_v ← C[v]
12                 R[c_v] ← R[c_v] + ω(uv)
13                 if |R| = 10000 and use full-rating-map strategy then
14                     U ← U + u
15                     move ← false
16                     break
17             if move then
18                 c_new, δ_weight ← SelectBestCluster(C, W, R)
19                 MoveNode(C, W, u, c_new, δ_weight)
20             Reset(R)
21     PerformSecondPhase(G, C, W, U)
```

Buffered Aggregation. The algorithm can suffer from high contention during the second phase caused by the atomic fetch-and-add operations when many neighbors belong to the same cluster. To avoid this scenario, the PEs can temporarily buffer the aggregated values in their thread-local hash table used in the first phase instead of writing them directly to the array. When a thread-local hash table is full, it is flushed by transferring the aggregated values to the array with atomic fetch-and-add operations. Additionally, after aggregation, each thread-local hash table is flushed to ensure that all aggregated values are transferred to the array. By buffering the values, fewer atomic fetch-and-add operations can be performed because no atomic operations are required for writing to the hash tables as these are thread-local. Therefore, this could potentially improve the running time by using less atomic fetch-and-add operations, which can cause high contention. However, a downside of this approach is the overhead of transferring the aggregated values from the thread-local hash tables into the array.

4.3 Two-Level Cluster Weight Vector

Through two-phase label propagation, we were able to reduce the memory consumption of label propagation for webbase2001 from 57.4 GB to 3.2 GB. In this section, we now describe how we can further reduce its memory consumption to 2.5 GB (with $L = 16$, see below) by using a more memory-efficient data structure for storing the cluster weights.

KaMinPar stores the weight of each cluster in order to fulfill the balance constraint and thus ensures that a node is not moved into a cluster that then has too high of a weight. To do this, it stores the weight of each cluster in an array with n entries with 8 bytes per entry. However, the cluster weights can be stored more space-efficiently when an unweighted graph is partitioned if the distribution of the cluster weights is taken into consideration. If we consider an unweighted graph, then in the first coarsening level the weight of a cluster, which is the sum of the node weights in the cluster, corresponds to the number of nodes in the cluster because in an unweighted graph each node has weight one. In the following coarsening levels, the weight of a cluster corresponds to the number of fine nodes in the cluster if the graph were uncontracted. Therefore, because in most cases, the number of large clusters is small, many of the cluster weights are small and can thus be stored with less than 8 bytes.

Taking advantage of this observation, we use a two-level data structure for storing the cluster weights. The data structure consists of an array with n entries and $L \in \{ 8, 16, 32 \}$ bits per entry, where L is a user-chosen parameter, and a dynamically growing concurrent hash table. Small cluster weights are stored in the array and large cluster weights are stored in the hash table. More precisely, the array stores cluster weights that are smaller than $L_{\max} \coloneqq 2^L - 1$. The value $L_{\max}$ is used to indicate, that the weight of a cluster in the corresponding entry is large, and is therefore stored in the hash table. Figure 4.2 illustrates the two-level cluster weight vector.

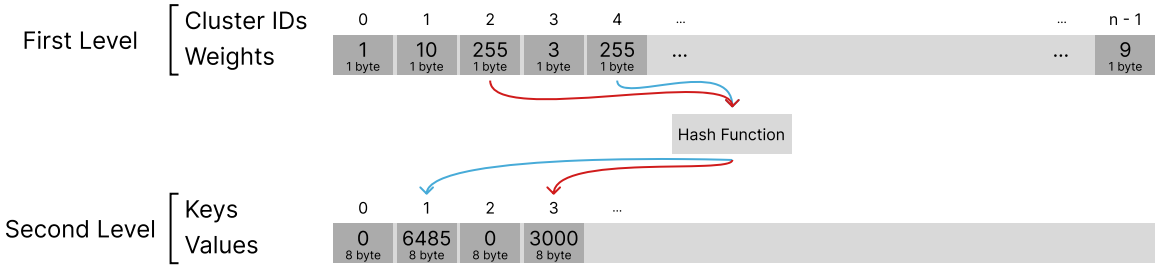

Figure 4.2: The structure of the two-level cluster weight vector with $L = 8$. The arrays are filled with sample data. The lower value in each entry of the arrays specifies how many bytes the corresponding encoded integer takes up.

Because label propagation is used for clustering, which iterates over the nodes in parallel, it is necessary to change the cluster weights atomically. For this we use atomic compare-and-swap operations. Note that we only allow values to be added to the hash table if the corresponding cluster becomes too large. Therefore, we do not support removing values from the hash table when the cluster becomes smaller than $L_{\max}$ again.

To increase the weight of a cluster with ID i because we are moving a node into it, we first read the value of the array at entry i and check whether the value is equal to $L_{\max}$. If this is the case, the value of key i is updated in the hash table. The concurrent hash table is thereby responsible for the synchronization. Moreover, because we do not support removal from the hash table, a race condition cannot occur since the value in the array stays the same once it is set to $L_{\max}$. If the value does not correspond to $L_{\max}$, the value read is already the weight of the cluster. We add the given value to the weight and check whether the new weight exceeds $L_{\max} - 1$. If not, we use an atomic compare-and-swap operation to change the value in the array at entry i and use the value read at the beginning as the expected value. Moreover, if the operation fails, we start from the beginning and try again. If the new weight is too large for the array, we use an atomic compare-and-swap operation to write the value $L_{\max}$ to the array at entry i and also use the value read at the beginning as the expected value. We also start from the beginning and try again if the operation fails. Further, if the operation does not fail, we add the entry to the hash table or update the value if another thread has already read the $L_{\max}$ value and added the entry.

The reduction of a cluster weight is easier because we do not have to consider the case that a value stored in the hash table becomes smaller than $L_{\max}$ again and should be removed from the hash table. To reduce a cluster weight, the value is first read at the corresponding position in the array. If this value is equal to $L_{\max}$, the value in the hash table is updated. Otherwise, if the value is less than $L_{\max}$, the new value is written to the array using an atomic compare-and-swap operation. The value read at the beginning is used as the expected value. If the operation fails, we start from the beginning and try again.

4.4 Contraction

As we have seen in the analysis of the memory consumption, contraction is also a contributor to the peak memory usage. In this section, we therefore present how we reduce the memory consumption of KaMinPar's contraction implementation. We begin with a description of the contraction implementation to understand where memory space can be saved.

Recall that the contraction step during coarsening creates the coarse graph using the clustering computed by label propagation. To create the coarse graph, the coarse nodes are visited in parallel and for each coarse node its neighborhood is aggregated. A neighborhood is aggregated by iterating over the adjacent nodes of the set of nodes that are contracted to the coarse node, remapping the adjacent nodes to their corresponding coarse nodes, and summing up the edge weights for each adjacent coarse node since we reduce parallel edges to a single edge. The next step would be to store the aggregated neighborhood in the edge and edge weight array of the coarse graph's adjacency array. However, this step is challenging because of the structure of the adjacency array and because the neighborhoods are aggregated in parallel. This is because the edge offset is necessary in order to write the edges and edge weights of a coarse node into the edge arrays. However, the edge offset depends on the degrees of other coarse nodes, namely those with smaller IDs, and therefore additional effort is required when aggregating the neighborhoods in parallel.

One solution, which is used by KaMinPar, is the use of an edge buffer to temporarily store the aggregated edges and edge weights. This approach is outlined in Algorithm 4. First, as described above, the coarse nodes are visited in parallel and the neighborhood is aggregated for each one. This is implemented in Lines 4–12. After a neighborhood has been aggregated, it is temporarily stored in the edge buffer. Furthermore, the coarse node degree, i.e., the number of aggregated edges, is stored in the node array. This is implemented in Lines 11–12. Then, when all coarse nodes have been visited, a prefix sum over the node array is computed in Line 13. This results in the correct offsets into the edge array. Finally, the coarse nodes are visited again in parallel and their neighborhoods are transferred from the edge buffer to the adjacency array using the offsets in the node array. This is implemented in Lines 15–16. For the sake of simplicity, we omit the computation of the node weight array as this can be done during the aggregation of the coarse neighborhoods.

Our goal in the following is to reduce the memory consumption of the edge buffer, in which the aggregated edges are temporarily stored, since it consumes $\Theta(\widetilde{m})$ memory space, where $\widetilde{m}$ is the number of edges of the coarse graph.

4.4.1 Partially Filling the Edge Buffer

The first variant trades off additional synchronization overhead for reduced memory consumption by only partially filling the edge buffer. Therefore, when aggregating the neighborhoods of the coarse nodes, not all coarse nodes are visited at once, but only a part of them. To do this, the coarse nodes are subdivided and processed from small to large IDs.

Algorithm 4: Contraction

```
1  Function Contract(G = (V, E, c, ω), C)
       Input: Graph to contract, mapping from nodes to clusters
2      Allocate node array 𝒩
3      Initialize edge buffer ℬ
4      for each coarse node c_u do parallel
5          R ← Thread-local small hash table or array
6          for each fine node v that is contracted to c_u do
7              for w ∈ N(v) do
8                  c_w ← C[w]
9                  if c_w ≠ c_u then                          // no self-loops
10                     R[c_w] ← R[c_w] + ω(vw)
11         𝒩[c_u] ← |R|                                  // store degree of c_u
12         Transfer aggregated neighborhood from R to ℬ
13     Compute prefix sum over 𝒩 in parallel
14     Allocate edge array ℰ and edge weight array 𝒲
15     for each coarse node c_u do parallel
16         Transfer N(c_u) from ℬ into edge arrays ℰ, 𝒲 at offset 𝒩[c_u]
17     return (𝒩, ℰ, 𝒲)
```

After the edges and edge weights of some coarse nodes have been aggregated in the edge buffer and the respective degrees have been written to the node array, a prefix sum is computed for the entries in the node array that have been processed. These entries then contain the correct offsets into the edge and edge weight array. The edges and their weights can then be transferred from the buffer to the adjacency array. This process is repeated until all coarse nodes have been processed.

In order to implement this contraction variant, the edge and edge weight array must be allocated in advance. The problem thereby is that the size of the edge arrays is only known once the neighborhood of all coarse nodes has been aggregated. Therefore, we overcommit memory for the arrays and only touch the memory that is actually used.

Furthermore, to subdivide the coarse nodes, which are processed in groups one after the other, an upper bound of the coarse node degrees is considered. The coarse nodes are merged into groups with contiguous coarse node IDs, so that each group has no more than $M \cdot m$ edges, where $M \in (0, 1)$ is a user-chosen parameter. With smaller M values, the coarse nodes are divided into more groups and thus the edge buffer is filled to the maximum with fewer edges. However, smaller M values also mean that more iterations are performed. Note that with $M = 1$ this contraction variant is identical to the original implementation.

4.4.2 Aggregating the Edges Twice

The second variant of the contraction algorithm completely eliminates the edge buffer by aggregating twice. More precisely, it aggregates the neighborhood of all coarse nodes in a first pass, writes the coarse node degrees into the node array and discards the aggregated edges. Then, it computes a prefix sum over the node array, which afterwards stores the correct offsets into the edge and edge weight array. Finally, in a second pass, the neighborhood of all coarse nodes is aggregated again and the computed edges and their weights are placed in the edge and edge weight array using the computed offsets.

4.4.3 Remapping the Coarse Nodes

The third variant also avoids the edge buffer, but it avoids it by writing a neighborhood directly into the edge and edge weight array after it has been aggregated for a coarse node. Two problems must be overcome for this. First, the number of edges is not known in advance, which is why it is not possible to allocate the edge and edge weight array properly. The problem is overcome by overcommitting memory for the arrays and only touching the memory that is actually used.

The second problem is that the offsets into the edge arrays are at the time of aggregation not known, which is why the neighborhood cannot be written to the right place. To avoid this problem, the coarse node IDs are remapped and the PEs use atomic operations to determine the new coarse node IDs and their offsets into the edge arrays. This works exactly as follows. Two counters are initialized with zeros. One counter is used to remap the coarse node IDs. The other counter is used to store the offset into the edge arrays for the next coarse node to be remapped. When a PE has aggregated a neighborhood and needs the offset into the edge arrays, it atomically reads both counters and also increments the coarse node ID counter by one and the edge offset counter by the degree of the coarse node, i.e., the number of edges it writes. Note that in order to be able to read and write the two counters atomically, wide words are required. The value read from the coarse node ID counter is then used to store the offset into the edge array, which is read from the edge offset counter, in the corresponding entry in the node array. Furthermore, the read edge offset is used to store the neighborhood in the edge and edge weight array.

Because the aggregated neighborhoods still consist of the old coarse node IDs, the IDs of the adjacent nodes stored in the edge array must subsequently be remapped. Moreover, the contraction algorithm computes a mapping, which stores the assignment of fine nodes to the coarse nodes, in the process. It must also be changed because it is used for projecting the partition in the uncoarsening phase. To do this, the remapping is explicitly stored in an array of size $\widetilde{n}$, where $\widetilde{n}$ is the number of coarse nodes, and the entries in the edge array and mapping are adjusted afterwards.

Synchronization. For synchronization, the double-width compare-and-swap instruction is used as an atomic operation. More precisely, the two counters are stored as a 128-bit integer, with the lower 64-bit being used for one counter and the upper 64-bit for the other. Then, when a coarse node is to be remapped, the 128-bit integer is read, the counters extracted and updated with the double-width compare-and-swap instruction. For this, the value read at the beginning is used as the expected value and the incremented counter is used as the new value. If the operation fails, the process starts from the beginning until it succeeds.

Buffering. In order to perform fewer compare-and-swap operations and thus improve performance, we increase the counters for several coarse nodes at once. To do this, we use a thread-local buffer of constant size to cache the neighborhood of multiple coarse nodes. That means that each PE stores the data about aggregated neighborhoods in the thread-local buffer until it is full. Then, a single compare-and-swap operation is used to read the coarse node ID and the edge offset when the buffer is full in order to transfer the data to the adjacency array. The coarse node counter is thereby not increased by one but by the number of cached nodes and the edge offset counter is increased by the total number of cached edges. Thus, fewer compare-and-swap operations are performed by fetching and updating coarse node IDs and edge offsets in batches.

5 Graph Compression

We observed in Section 4.1, in the analysis of the memory consumption, that the input graph is another critical component for reducing the total memory consumption of KaMinPar. In this chapter, we therefore present a graph compression scheme, which we use for storing the input graph space-efficiently. In Section 5.1, we describe what the compression scheme looks like. The goal is to provide a static and compressed representation of a graph. As operations on the graph data structure, we provide simple operations such as order, size, weight and degree queries as well as iterating over the neighborhood of a node. Another motivation behind graph compression is the potential speed up of the program because fewer memory accesses take place, and thus the load on the memory subsystem is reduced [2]. Therefore, in Section 5.2, we describe how iterating over the neighborhood of a node can be accelerated to provide fast operations on the compressed graph data structure. Finally, in Section 5.3, we describe how we read the input graph from disk in a single read operation and compress it in the same process.

5.1 Compression Scheme

The basis of our compression scheme is the adjacency array, which we call the uncompressed adjacency array. Recall that the adjacency array $\mathcal{A} = (\mathcal{N}, \mathcal{E}, \mathcal{C}, \mathcal{W})$ consists of a node array $\mathcal{N}$, which stores for each node $u \in V$ the offset $\mathcal{N}[u]$ into the edge array $\mathcal{E}$, where the IDs of the adjacent nodes of u are stored contiguously. We transform the uncompressed adjacency array by applying several compression techniques to the edge array and storing the node array more compact. The node weight array $\mathcal{C}$ and the edge weight array $\mathcal{W}$ remain unchanged. Note that since the graph data structure is static, we do not provide graph operations that modify the graph structure, i.e., insertion and deletion of nodes or edges is still not provided.

5.1.1 Compressed Edge Array

In the following, we describe the changes to the edge array. Since we have certain goals, the edge array must be adapted accordingly. One requirement is that the degree queries should be in constant time because many parts of the KaMinPar algorithm require the node degree. Another requirement is that the edge weights must be available while iterating over the neighborhood of a node. For this purpose, the IDs of the edges must be provided during

iteration in order to be able to access the edge weight array. The last requirement is that we want to iterate in parallel over the neighborhood of nodes with high degree to exploit parallelism.

Variable-Length Encoding

The first compression technique we apply to the edge array is variable-length encoding. With this technique, we encode the IDs of the adjacent nodes of each neighborhood in the edge array as VarInts with the format described in Section 2.4. Furthermore, the node array must be adapted. In the uncompressed adjacency array, the node array stores the ID of the first edge in the neighborhood for each node, because that is the index into the edge array where the edge is stored. The other edges of the neighborhood are stored at the subsequent indices. However, changing the edge array to use variable-length per edge means that an entry of the node array can no longer store the ID of the first edge of the respective neighborhood. Instead, the relative position of the first byte at which the neighborhood is encoded in the edge array must be stored. This is because each encoded edge no longer has a fixed length and therefore cannot be indexed consistently. In Figure 5.1, we give an overview of the structure of the compressed adjacency array, which uses variable-length encoding.

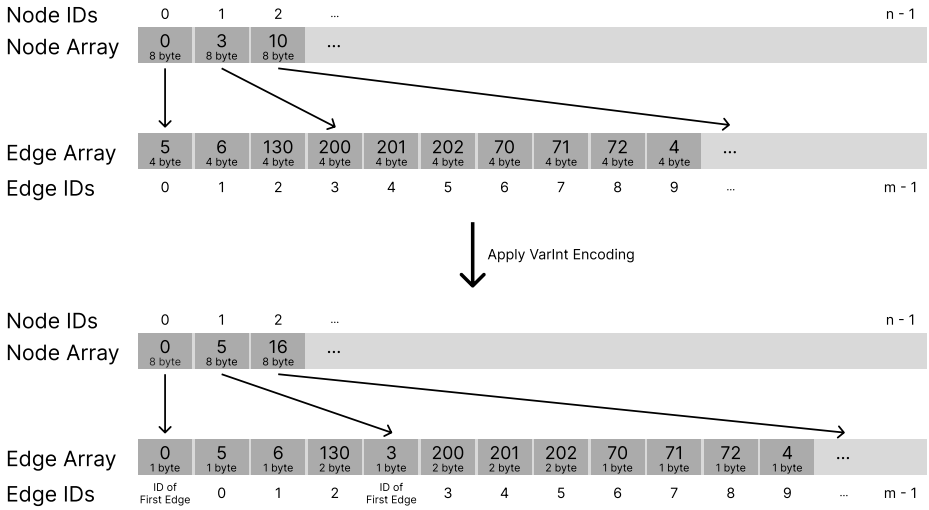

Figure 5.1: The structure of the compressed adjacency array, whose edge array uses variable-length encoding and explicitly stores the first edge ID of each neighborhood. The upper part shows the uncompressed adjacency array and the lower part the compressed adjacency array. Moreover, the lower value in each entry of the arrays specifies how many bytes the encoded integer takes up.

One problem with variable-length encoding is that the degree of a node is no longer implicitly stored by the offsets in the node array. With the uncompressed adjacency array, the

degree of a node u can be obtained by computing the difference between the offset into the edge array of the following node and its own offset, i.e., $\deg(u) = \mathcal{N}[u+1] - \mathcal{N}[u]$. This is true because the edge array stores the neighborhood contiguously and the neighborhood of the next node is stored directly afterwards. Therefore, the offset into the edge array of the following node is exactly $\deg(u)$ larger. This no longer applies to the edge array, which stores the node IDs with variable length. Another problem is that the ID of an edge is no longer stored implicitly. This ID is given with the uncompressed adjacency array, as the node array stores the ID of the first edge in the neighborhood for each node. The IDs of the following edges in a neighborhood can be obtained by incrementing the ID of the first edge. Due to the variable-length encoding, the value stored in the node array is no longer the ID of the first edge of a neighborhood, but the relative starting position in bytes at which the adjacent node IDs of the neighborhood are encoded as VarInts.

Storing the First Edge ID. For these two reasons, additional information must be explicitly stored for each neighborhood because we want to provide degree queries in constant time and provide IDs for the edges of a neighborhood. We choose to store the ID of the first edge of a neighborhood explicitly at the start of each neighborhood in the edge array. With this information, the IDs of the edges of a neighborhood can be obtained by incrementing the ID of the first edge, as with the uncompressed adjacency array. Further, we can compute the degree based on that information. For that, we can take advantage of the property that the IDs of the edges increase incrementally in the order in which they are stored in the edge array. Formally, this means that for the edges $uv_1, uv_2, \ldots, uv_{\deg(u)}$ of a neighborhood its IDs are $e, e+1, \ldots, e+\deg(u)-1$ and the ID of the first edge of the next neighborhood is $e + \deg(u) - 1 + 1$. Therefore, the difference between the IDs of the first edge in the neighborhood of the next node and its own is exactly the degree, i.e., $e + \deg(u) + 1 - 1 - e = \deg(u)$.

Further, the n-th entry in the node array must also be changed. In the uncompressed adjacency array, the number of edges is stored in it, i.e., $\mathcal{N}[n] = e$. The degree of the last node $n-1$ can therefore be obtained by $\mathcal{N}[n] - \mathcal{N}[n-1] = \deg(n-1)$ as for any other node. As the degree is obtained differently with the compressed adjacency array, the n-th entry of the node array now points to the position in the edge array where the number of edges is encoded. More precisely, the n-th entry in the node array now stores the relative position in bytes to the end of the last encoded neighborhood in the edge array. Moreover, at that position in the edge array the number of edges m is encoded as a VarInt. With this change, the degree of the last node can be obtained as for any other node in the compressed adjacency array.

Isolated Nodes. For an isolated node, we do not store any information about it in the edge array. Therefore, an isolated node can be identified by checking that the offset does not increase. Thus, a node u is isolated if and only if $\mathcal{N}[u+1] = \mathcal{N}[u]$ holds.

Gap Encoding

The next compression technique we apply to the edge array is gap encoding [4, 38]. For this purpose, the neighborhood of each node u is sorted in ascending order, i.e., $v_1 < v_2 < \cdots < v_{d-1} < v_d$ with $N(u) = \{ v_1, v_2, \ldots, v_{d-1}, v_d \}$. The IDs of the adjacent nodes are now stored as differences of consecutive adjacent nodes in the respective order. The first adjacent node is treated differently. It is stored as the difference with respect to the source node u. Moreover, all differences except the first difference are subtracted by one. The resulting gaps are $v_1 - u, v_2 - v_1 - 1, \ldots, v_d - v_{d-1} - 1$, which are stored in that order. Note that the gaps are stored as VarInts. Since the neighborhood has been sorted in ascending order, all gaps, except possibly the first one, are non-negative. Therefore, the first gap is stored as a signed VarInt using the zigzag encoding described in Section 2.4. It should also be noted that the IDs of the edges may change due to the sorting of each neighborhood. However, this is not an issue as the structure of the graph does not change and as the IDs are relabeled consistently.

First Edge ID Gap. An optimization regarding gap encoding can be applied to the edge array if it is known that the graph has no isolated nodes. It can also be applied when there are isolated nodes $x_1, \ldots, x_i$, but only if they are the nodes with the highest IDs, i.e., $x_1 = n - i - 1, \ldots, x_i = n - 1$. In both cases, the ID of the first edge e of the node u that is stored at the beginning of each encoded neighborhood can be stored as the gap $e - u$. As there are no isolated nodes or the isolated nodes are the ones with the highest IDs, the ID of the first edge is greater than or equal to the node ID because each previous neighborhood contains at least one edge. Therefore, the gap is non-negative and can still be stored as an unsigned VarInt. It is also smaller than or equal to the first edge ID and can therefore due to variable-length encoding only reduce the used memory space. With this change, the degree of a node u is still implicitly stored, but has to be obtained differently. Let $\widetilde{e}_u = e_u - u$ and $\widetilde{e}_{u+1} = e_{u+1} - (u+1)$ be the gaps of the first edge IDs of a node u and $u+1$, where e_u and e_{u+1} are the first edge IDs respectively. The degree of u is then obtained by $\widetilde{e}_{u+1} - \widetilde{e}_u + 1$ since $\widetilde{e}_{u+1} - \widetilde{e}_u + 1 = e_{u+1} - (u+1) - (e_u - u) + 1 = e_{u+1} - e_u = \deg(u)$ holds. Furthermore, in order for the degree computation to be correct for the last node, the value at the position where $\mathcal{N}[n]$ points to must be changed. Instead of storing m, the gap $m - (n - i)$ must be stored, where i is the number of isolated nodes labeled last. The changes to the degree computation are also the reason why the first edge ID cannot be stored as a gap when the graph contains isolated nodes not labeled last. When the node u is followed directly by isolated nodes $u + 1, u + 2, \ldots, u + j$, the first edge ID stored of the next neighborhood is not that of the node $u + 1$ but of node $u + j + 1$ because we do not store any information in the edge array for isolated nodes. Therefore, $\widetilde{e}_{u+1} - \widetilde{e}_u + 1$ is off by exactly j. This could be corrected by determining the amount of isolated nodes j that are directly stored after the node. However, the degree query would then no longer run in constant time. In Figure 5.2, we illustrate how gap encoding is applied to the compressed adjacency array.

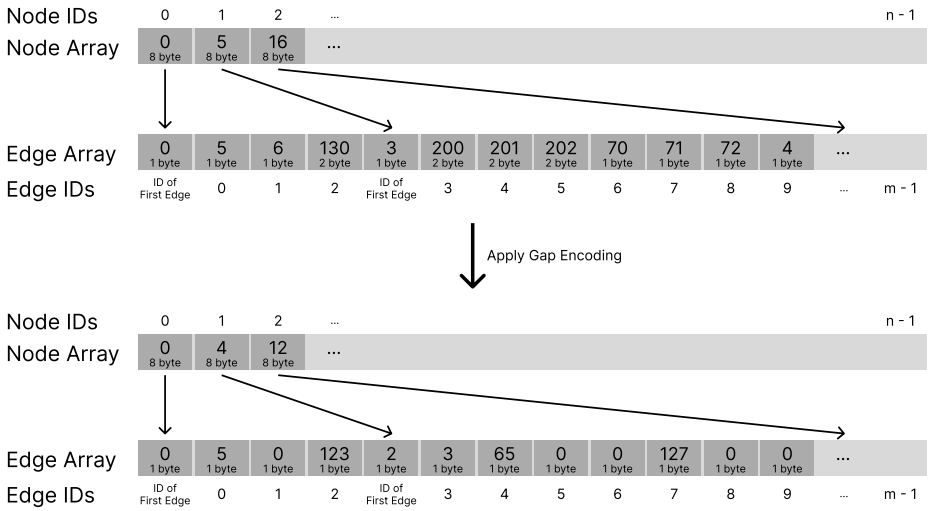

Figure 5.2: The structure of the compressed adjacency array, whose edge array uses variable-length encoding and gap encoding, where the first edge ID of each neighborhood is also stored as a gap. The upper compressed adjacency array only uses variable-length encoding and the lower compressed adjacency array additionally uses gap encoding.

Gap Decoder. We now present an algorithm to decode the neighborhood of a node u, which is encoded with gap encoding. First, the relative position into the compressed edge array to the start and the end of the encoded neighborhood of u is determined. The start is given by $\mathcal{N}[u]$ and the end is given by $\mathcal{N}[u+1]$. Then, a VarInt at the start of the encoded neighborhood in the compressed edge array is decoded. This is the ID of the first edge. In Algorithm 5, we illustrate how the gaps are then decoded. In Lines 4–5 the first gap is decoded and the first adjacent node is derived from it. The first gap is treated specially as it is encoded as a signed VarInt whereas the remaining gaps are normal VarInts. After that, the first adjacent node and the ID of the respective edge is passed to a caller, who requested to decode the neighborhood. Then, the edge ID is incremented and a variable that holds the previous decoded adjacent node, which is needed to decode the next gap, is initialized. This is implemented in Lines 9–10. The Lines 11–19 show how the remaining adjacent nodes are iteratively decoded. Each gap stored as a VarInt is decoded and the adjacent node is derived from it. Then, the caller is invoked and the ID of the next edge and the previous adjacent node is updated.

Interval Encoding

The last compression technique applied to the edge array is interval encoding [4]. Interval encoding identifies maximal contiguous intervals of adjacent nodes with consecutive IDs for the neighborhood of each node. Additionally, only intervals whose length is greater than or equal to a threshold T_{Interval} are considered. Let $I_1 = [v_1, v_1 + l_1], \ldots, I_i = [v_i, v_i + l_i]$ be all such intervals in the neighborhood of a node. Then, instead of storing each member of

```
1  template <typename Lambda>
2  void decode_gaps(uint8_t *data, uint8_t *end, NodeID u, EdgeID e,
3                   Lambda lambda) {
4    NodeID first_gap = decode_signed_varint(&data);
5    NodeID first_adjacent_node = first_gap + u;
6
7    lambda(first_adjacent_node, e);
8
9    e += 1;
10   NodeID prev_adjacent_node = first_adjacent_node;
11   while (data != end) {
12     NodeID gap = decode_varint(&data);
13     NodeID adjacent_node = gap + prev_adjacent_node + 1;
14
15     lambda(adjacent_node, e);
16
17     e += 1;
18     prev_adjacent_node = adjacent_node;
19   }
20 }
```

Algorithm 5: An algorithm to decode adjacent nodes encoded as gaps.

an interval as a gap, only the left extremes $v_1, \ldots, v_i$ and lengths $l_1, \ldots, l_i$ of the intervals are stored. The encoded neighborhood in the compressed edge array is split up for this purpose. As before, the ID of the first edge in the neighborhood is encoded at the beginning. Then, the number of intervals i is encoded. After that, one after another, for each interval $I_j = [v_j, v_j + l_j]$ its left extreme v_j and length l_j are encoded. The first left extreme is stored as it is, whereas the following left extremes are stored as the difference to the previous right extreme. Further, it is decremented by two as this is the minimum distance between the end of one interval and the start of the next interval. That is to say, if $I_{j-1} = [v_{j-1}, v_{j-1} + l_{j-1}]$ is the previous interval for the interval $I_j = [v_j, v_j + l_j]$, then the left extreme of I_j is stored as $v_j - (v_{j-1} + l_{j-1}) - 2$. The length is decremented by the minimum length of an interval, i.e., $l_j - T_{\text{Interval}}$ is stored. Figure 5.3 depicts the compressed adjacency array with interval encoding.

Avoid Storing Unused Interval Counts. Even if some nodes do not have any intervals of minimum length T_{Interval}, the interval count is still stored at the beginning of its neighborhood after the first edge ID. This results in an unused byte for each neighborhood without proper intervals. To avoid storing unused bytes, we make use of the compact format of the VarInt to only store the interval count if the neighborhood has at least one interval. To implement that, we use a bit of the first encoded byte of the first edge ID as a boolean value. The boolean thereby encodes whether the neighborhood has at least one interval. Note that the data bits in the VarInt must therefore be shifted by one.

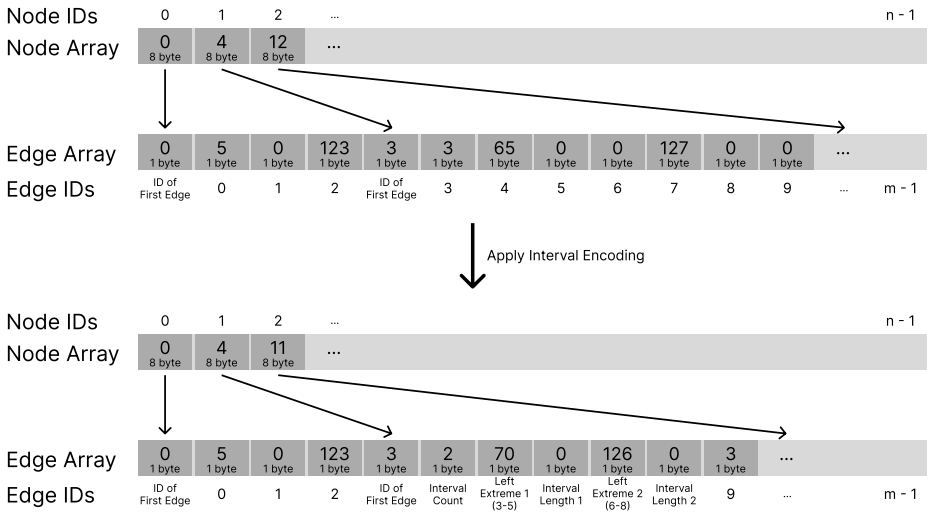

Figure 5.3: The structure of the compressed adjacency array, whose edge array uses variable-length encoding, gap encoding and interval encoding with $T_{\text{Interval}} = 3$. The upper part shows the compressed adjacency array that only uses variable-length and gap encoding and the lower part shows the compressed adjacency array with additional interval encoding.

Interval Decoder. Algorithm 6 illustrates how adjacent nodes encoded as intervals are decoded. First, the interval count is decoded. This is implemented in Line 4. The value of the previous right extreme is then initialized in Line 5. The initial value is 2, since the first interval is stored as it is. This cancels the subtraction by 2 in Line 11, which is only required for the following intervals. In Lines 7–19 the intervals are decoded. For each interval, its left extreme and length are derived from the decoded gaps. Then, the caller is invoked with all elements in the interval. Finally, the previous right extreme is updated, such that the left extreme of the next interval can be decoded correctly.

Decoding the Neighborhood in Parallel

Next, we present an encoding of the neighborhood of a node that allows parallel iteration [38]. With the uncompressed adjacency array, a neighborhood can be iterated in parallel by splitting the neighborhood into parts and then processing the parts in parallel. This is possible because all node IDs have a fixed width and the boundaries of the parts can therefore be determined only using the start and end of the neighborhood. With our compression scheme of the edge array, two problems arise with this approach. The first problem is that, due to the variable-length encoding, the boundaries of the parts cannot be determined using the start and end position of the neighborhood. If the neighborhood were split in some non-trivial way, it would have to be aligned, such that the start of each part begins with a new VarInt. This is possible, but the ID of the first edge of this part is not available. To compute the first edge ID of a part, the number of edges in the previous parts must be known and added to the first edge ID of the neighborhood. However, to determine the number of edges in the

```
1  template <typename Lambda>
2  uint8_t* decode_intervals(uint8_t *data, EdgeID e,
3                            Lambda lambda) {
4    NodeID interval_count = decode_varint(&data);
5    NodeID prev_right_extreme = 2;
6
7    for (NodeID i = 0; i < interval_count; ++i) {
8      NodeID left_extreme_gap = decode_varint(&data);
9      NodeID interval_len_gap = decode_varint(&data);
10
11     NodeID left_extreme = left_extreme_gap + prev_right_extreme - 2;
12     NodeID interval_len = interval_len_gap + kIntervalTreshold;
13
14     for (NodeID j = 0; j < interval_len; ++j) {
15       lambda(left_extreme + j, e + j);
16     }
17
18     prev_right_extreme = left_extreme + interval_len - 1;
19   }
20
21   return data;
22 }
```

Algorithm 6: An algorithm to decode adjacent nodes encoded as intervals.

previous parts, these must be decoded or their length must be stored explicitly. The second problem is that a data dependency arises due to the gap encoding. This is because in order to decode a gap, the previous decoded gap must be known.

High-Degree Encoding. To solve both problem, we modify the compressed edge array in two ways. First, the neighborhood of each node is split into parts of fixed-length L_{Part}, which are encoded separately. This means that the encoding previously described, including gap encoding and interval encoding, is applied to each part independent of the other parts. It is therefore treated like a separate neighborhood in the context of the previously described compression scheme. Second, the relative starting position at which each part is encoded is stored explicitly at the beginning of the neighborhood after the first edge ID. Note that the relative position is stored as a fixed-width integer. If it were stored as a VarInt, then not all decoding routines could be started simultaneously because each part position must be obtained in the stored order due to the variable-length encoding. Moreover, the amount of parts is not stored because it can be derived from the size of the neighborhood and L_{Part}. These two modification allow for the parallel iteration of a neighborhood since each part location is now known in advance and can be decoded independently. It also integrates into the compression scheme described earlier, the only problem being that all parts use the marked VarInt of the first edge ID to store whether its part has at least one interval. To

avoid this, we include a bit marker in the integer of the relative starting position of each part and use it to indicate whether the respective part has at least one interval.

Furthermore, parallel decoding of a neighborhood is only advantageous if the neighborhood is large, as otherwise the overhead of starting the parallel tasks dominates. Therefore, we only split large neighborhoods into parts. More precisely, we only apply this technique to nodes with degree greater than or equal to a threshold T_{Part}. We call this technique high-degree encoding.

Parallel Decoder. Algorithm 7 outlines how to decode a neighborhood encoded with high-degree encoding in parallel. The algorithm starts by computing the number of parts in the neighborhood in Line 5. As each part contains L_{Part} edges, the total number of parts is the ceiling of the size of the neighborhood divided by L_{Part}. Then, each part is decoded in parallel. For that, the relative position of each part and whether it has at least one interval is decoded. This is implemented in Lines 7–8. Next, the start, the ID of the first edge and the end of each part are derived in Lines 10–20. The start of each part is the relative part position added to the start of the encoded parts. The ID of the first edge of each part is the amount of edges in the previous parts added to the ID of the first edge of the neighborhood. As each part has L_{Part} edges, there are $i \cdot L_{\text{Part}}$ edges encoded in front of part $i + 1$. To obtain the end of each part, two cases are considered. If it is not the last part, the end is the start of the next part. If it is the last part, then the end is the end of the neighborhood. Lastly, the adjacent nodes are decoded in Lines 22–28. For that, the intervals are decoded if there are any, and if not all adjacent nodes have been encoded by intervals, the adjacent nodes encoded as gaps are decoded.

Algorithm for Decoding the Compressed Edge Array

All presented algorithms can be combined to decode the whole neighborhood of a node u. This results in Algorithm 8. It first determines the start and end of the neighborhood via the node array. If the start is the same as the end, the node is isolated and the algorithms stops. This is implemented in Lines 3–7. If the node is not isolated, the gap of the first edge ID of u and $u + 1$ is determined. From the gaps the actual first edge ID of u and the degree of the neighborhood are derived. This is implemented in Lines 9–13. The next step is to check whether the node has a high degree, i.e., whether $\deg(u) \geq T_{\text{Part}}$ holds. If this is the case, the neighborhood is encoded using high-degree encoding and the corresponding decoding routine is called in Line 16. Otherwise, the intervals are decoded first, if there are any, and then the gaps are decoded, if not all nodes are encoded by the intervals. This is implemented in Lines 18–24.

If the algorithm is used in a program where parallel decoding is not desired, the algorithm must be adapted. One possible change would be to provide a second algorithm, which replaces the parallel for loop in Algorithm 7 with a sequential for loop. This would be favorable if sequential decoding is required but parallel decoding is also desired because

```
template <typename Lambda>
void decode_parts(uint8_t *data, uint8_t *end,
                  NodeID u, EdgeID e, NodeID degree,
                  Lambda lambda) {
  NodeID part_count = std::ceil(degree / kPartLength);
  tbb::parallel_for(0, part_count, [&](NodeID i) {
    uint8_t *pos_data = data + i * sizeof(NodeID);
    auto [part_pos, has_intervals] = decode_marked_int(pos_data);

    uint8_t *part_data = data + part_pos;
    EdgeID part_edge = e + i * kPartLength;

    uint8_t *part_end;
    if (i + 1 < part_count) {
      uint8_t *next_pos_data = data + (i + 1) * sizeof(NodeID);
      auto [next_part_pos, _] = decode_marked_int(next_pos_data);
      part_end = data + next_part_pos;
    } else {
      part_end = end;
    }

    if (has_intervals) {
      data = decode_intervals(data, part_edge, lambda);
    }

    if (data != end) {
      decode_gaps(part_data, part_end, u, part_edge, lambda);
    }
  });
}
```

Algorithm 7: An algorithm to decode adjacent nodes encoded with high-degree encoding in parallel.

then a sequential and parallel algorithm are available. One possible other change would be to not use high-degree encoding. This would be favorable when only sequential decoding is used because it avoids the overhead of determining a part position and first edge ID to decode each part. Furthermore, it makes the compression scheme more space-efficient by not storing the part count and the relative part positions as well as potentially encoding larger intervals, which overlap different parts.

5.1.2 Compact Node Array

The uncompressed adjacency array uses fixed-width integers for storing edge IDs in the node array, where the edge IDs are typically 64-bit integers. This is independent of the graph to store and is space-inefficient if for example the number of edges of a graph is much smaller than the maximum number that can be represented with an edge ID. Therefore, we

```
template <typename Lambda>
void decode_neighborhood(NodeID u, Lambda lambda) {
  uint8_t *data = node_array[u];
  uint8_t *end = node_array[u + 1];
  if (data == end) {
    return;
  }

  auto [first_edge_gap, has_intervals] = decode_marked_varint(&data);
  auto [next_first_edge_gap, _] = decode_marked_varint(&data);

  EdgeID e = first_edge_gap + u;
  NodeID degree = next_first_edge_gap - first_edge_gap + 1;

  if (degree >= kHighDegreeThreshold) {
    decode_parts(data, end, u, e, degree, lambda);
  } else {
    if (has_intervals) {
      data = decode_intervals(data, e, lambda);
    }

    if (data != end) {
      decode_gaps(data, end, u, e, lambda);
    }
  }
}
```

Algorithm 8: An algorithm to decode a compressed neighborhood of a node.

store the edge IDs as fixed-width integers, but select the width based on the graph. Ideally, one would use the size of the compressed edge array in bytes as the width because this is the largest value stored in the node array. However, since we compress the input graph during reading, we do not know in advance how large the compressed edge array is, which is why we select the width using an upper bound of the size that depends on the number of edges.

5.2 Accelerating the Decoding Routine

In addition to the goal of reducing memory consumption, graph compression has the potential to reduce the running time of the algorithm, especially if it is run in parallel on many PEs. This is because the compressed graph can reduce the number of memory accesses and thus reduce the load on the memory subsystem [2]. Parallel algorithms that are run on many PEs are particularly bottle necked by the memory subsystem, which is why the potential is greatest in this case. However, this requires fast decoding routines of the compressed graph to provide graph operations that can keep up with their uncompressed counterparts. Therefore, we now present various techniques for speeding up the decoding of the VarInts.

A straightforward implementation of an algorithm for decoding VarInts is a loop that iterates over the encoded bytes until the continuation bit is not set. In each iteration, the least significant seven bits are extracted and accumulated into a variable. Algorithm 9 illustrates such an algorithm. For the sake of simplicity, Algorithm 9 and the following algorithms do not increment the pointer to the next encoded VarInt, in contrast to the decoding routines used in Algorithm 8. Moreover, the algorithms only decode 32-bit integers. In order to decode 64-bit integers with Algorithm 9, the type of the accumulation variable and the return type have to be adapted.

```
1  int32_t decode_varint(const uint8_t *ptr) {
2    int32_t result = 0;
3    int32_t shift = 0;
4
5    while (true) {
6      const int32_t byte = *ptr;
7
8      if ((byte & 0b10000000) == 0) {
9        result |= byte << shift;
10       break;
11     } else {
12       result |= (byte & 0b01111111) << shift;
13     }
14
15     ptr += 1;
16     shift += 7;
17   }
18
19   return result;
20 }
```

Algorithm 9: A straightforward implementation of a VarInt decoder.

The algorithm can be improved by replacing the operations `and`, `or` and `shift` with a specialized bit-level operation. This results in a faster algorithm because less arithmetic has to be performed due to the more efficient bit manipulation operation. We use the parallel bit extract (PEXT) instruction to replace the basic bit-level instructions in the algorithm. The PEXT instruction receives two arguments: data bits and a mask. It extracts the data bits according to the bits set in the mask and returns the bits so that they are right-aligned and contiguous [21]. If the PEXT instruction is called with the mask, which has zeros at the continuation bits and ones at the remaining places, the result is exactly the decoded integer. Therefore, the previous algorithm is improved by unrolling the loop and using the PEXT instruction with the mask that corresponds to the length of the VarInt. Algorithm 10 depicts the improved implementation. Note that in the case where the VarInt has length one, the `and` operation can be used instead. To decode 64-bit integers, the loop has to be unrolled to handle ten cases because the maximum byte length of a VarInt that encodes a 64-bit integer

is ten. Moreover, since a VarInt with byte length at least nine cannot fit into a 64-bit register, the PEXT instruction can only be used to decode the lower 8 bytes of the VarInt. The upper bytes are decoded with `and`, `or` and `shift` instructions and are appended.

Since the algorithm reads either 4-byte or 8-byte chunks, which contain the encoded integer, but might also contain other data, it must be ensured that the chunks can be read from the passed pointer. This could pose a problem when decoding the compressed edge array. For example, if the last VarInt in the compressed edge array has length five, an 8-byte chunk is read, which contains three bytes that do not belong to the compressed edge array. The compressed edge array must therefore be padded with three bytes at the end in order to avoid segmentation faults.

```
1  int32_t decode_varint(const uint8_t *ptr) {
2    if ((ptr[0] & 0b10000000) == 0) {
3      return *ptr & 0b01111111;
4    }
5
6    if ((ptr[1] & 0b10000000) == 0) {
7      return _pext_u32(
8        *reinterpret_cast<const uint32_t *>(ptr),
9        0x7F7F
10     );
11   }
12
13   if ((ptr[2] & 0b10000000) == 0) {
14     return _pext_u32(
15       *reinterpret_cast<const uint32_t *>(ptr),
16       0x7F7F7F
17     );
18   }
19
20   if ((ptr[3] & 0b10000000) == 0) {
21     return _pext_u32(
22       *reinterpret_cast<const uint32_t *>(ptr),
23       0x7F7F7F7F
24     );
25   }
26
27   return static_cast<uint32_t>(
28     _pext_u64(
29       *reinterpret_cast<const uint64_t *>(ptr),
30       0x7F7F7F7F7F7F
31     )
32   );
33 }
```

Algorithm 10: A decoder for VarInts using the PEXT instruction, where the number of cases considered corresponds to the number of bytes decoded.

In the following, we present two further variable-length encodings, which can be used instead of VarInts. With these encodings, the decoding can be accelerated. However, the disadvantage of these encodings is that they have worse compression rates, i.e., they require more bits per encoded integer on average. These encodings are therefore optional if one wants better running times and can tolerate worse compression ratios.

5.2.1 Run-Length Encoding

Run-length encoding (RLE) is a variable-length encoding, which identifies consecutive integers of the same length and stores each integer with the minimal number of bytes behind a byte header [38]. The byte header encodes how many integers with the same length follow and what length they have.

A reason to use this encoding is that it is faster than VarInts to decode because branch mispredictions can be avoided. This is the case because the continuation bit does not have to be checked as it is known how many integers are encoded in a run. Therefore, the decoding loop is avoided for the integers in a run. Note that we only use it to encode the gaps in the compressed edge array. This means that the first edge ID and interval data are still encoded with VarInts when using RLE. The reason for this is that the algorithms for decoding the different parts remain independent, which simplifies the implementation. Furthermore, we do not apply RLE to the interval data because the data from one interval could be contained in two runs, which would complicate the decoding routine.

The exact format of RLE used for the compression scheme is as follows. To encode 32-bit node IDs, the byte header is split into two least significant bits and six upper significant bits. The header uses the least significant two bits to encode the number of bytes used to store the integers in the respective run. It uses the upper significant six bits to encode the length of the run. We decrement the length of the run by one before storing it because each run has a minimum length of one. Therefore, a run can only contain a maximum of 65 integers. To encode 64-bit node IDs, the byte header is split into three least significant bits and five upper significant bits. The run is encoded in the header analogously to the 32-bit encoding. Therefore, a run can only contain a maximum of 33 integers. Figure 5.4 depicts the layout of RLE with 32-bit node IDs.

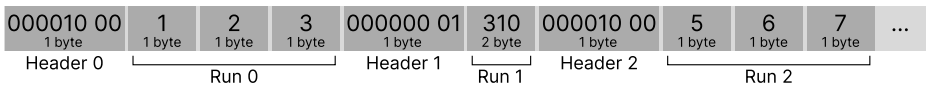

Figure 5.4: The layout of byte headers and encoded integers when stored with RLE. The array is filled with sample data. The lower value in each entry of the arrays specifies how many bytes the encoded integer takes up. Moreover, the value of the byte headers are shown in binary.

5.2.2 Stream VByte

Besides run-length encoding, vectorization of the decoding routine is another way to speed up the decoding of the variable-length encoded integers in the compressed edge array. This is because with Single-Instruction-Multiple-Data (SIMD) instructions multiple variable-length encoded integers can be simultaneously decoded. Additionally, branch mispredictions can be avoided by decoding multiple integers in blocks. Therefore, we provide an alternative variable-length encoding, which uses SIMD instructions to speed up the decoding routine.

We use the Stream VByte encoding for this, which decodes four variable-length integers simultaneously using SIMD instructions [26]. Stream VByte stores each integer with only the minimum number of bytes needed. To decode the integers, they are grouped into blocks of four and a control byte is added for each such block. The control byte is split into parts of 2 bits, each part storing the number of bytes an integer in the block is stored with. For now, we assume that the number of stored integers is a multiple of four. The Stream VByte encoding stores the control bytes and encoded integers separately in memory. This means that all control bytes are stored contiguously at the beginning and the integers are stored contiguously directly behind them. With this layout, the control bytes are stored in such a way that the control byte of the next block does not depend on the length of the previous block and can therefore be read predictably, unlike when control bytes and integers are stored interleaved. This allows the processor to read the control bytes without data dependencies [26]. Figure 5.5 shows the layout of Stream VByte.

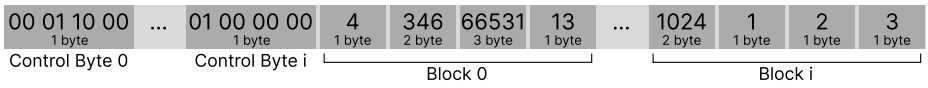

Figure 5.5: The layout of control blocks and encoded integers when stored with Stream VByte. The array is filled with sample data. The lower value in each entry of the arrays specifies how many bytes the encoded integer takes up. Moreover, the value of the control bytes are shown in binary.

We will now explain how to decode a block of integers. Two lookup tables are used for this purpose. The first lookup table stores for each possible control byte the number of bytes that the corresponding block takes up in memory. It is used to determine the position of the next block in memory. The second lookup table stores for each possible control byte a shuffle mask. This shuffle mask is used to rearrange the encoded integers, which are loaded into a SIMD register, such that the decoded integers can be extracted from the register. In Algorithm 11, we show how the integers are decoded using the lookup tables. First, the number of control bytes is determined from the number of encoded integers and then a pointer to the start of the encoded integers is created. This is implemented in Lines 2–4. Then, the blocks are visited and decoded. In Lines7–8, the current control byte is read and the length of the block is derived from the lookup table. Then, the next 16 bytes, which include the data of the current block, are loaded into a SIMD register. Further, the bytes

of the current block are skipped such that the next block can be loaded afterwards. This is implemented in Lines 10–11. Next, in Lines 13–14, the shuffle mask is loaded from the lookup table and is applied to the data loaded into the SIMD register. Finally, in Lines16–20, the decoded values are extracted and can be processed. Note that for decoding, the number of encoded integers is required. For our case, this is given because we are storing the degree of a neighborhood explicitly.

```
1  void decode_stream(const uint8_t *ptr, const size_t size) {
2    const size_t num_control_bytes = size / 4;
3    const uint8_t *control_bytes = ptr;
4    const uint8_t *data = ptr + num_control_bytes;
5
6    for (size_t i = 0; i < num_control_bytes; ++i) {
7      const uint8_t control_byte = control_bytes[i];
8      const uint8_t length = kLengthTable[control_byte];
9
10     __m128i value = _mm_loadu_si128((const __m128i *)data);
11     data += length;
12
13     const __m128i shuffle_mask = kShuffleTable[control_byte];
14     value = _mm_shuffle_epi8(value, shuffle_mask);
15
16     uint32_t value1 = _mm_extract_epi32(value, 0);
17     uint32_t value2 = _mm_extract_epi32(value, 1);
18     uint32_t value3 = _mm_extract_epi32(value, 2);
19     uint32_t value4 = _mm_extract_epi32(value, 3);
20     // Decoded values can be passed to a caller, for example
21   }
22 }
```

Algorithm 11: A decoder for integers encoded with Stream VByte.

If the number of encoded integers is not a multiple of four, the last block has to be treated specially. The control byte is read first and is used to obtain the shuffle mask. Then the data is loaded into a SIMD register and the shuffle mask is applied to it. Finally, only the number of integers stored in the last block are extracted from the SIMD register.

As with RLE, we only apply Stream VByte encoding to the gaps because this simplifies the decoding algorithms. Moreover, a padding of three bytes is added to the compressed edge array. This is necessary because the decoder reads 4-byte chunks into a SIMD register, which could lead to a segmentation fault if the number of encoded integers is not a multiple of four. In addition, Stream VByte can only encode 32-bit integers. So when the encoding is used, node IDs are 32-bit, which allow for graphs with up to 2^{32} nodes.

5.3 Compressing a Graph in a Single Disk-Read Operation

A bottleneck for the practical use of a graph partitioning application is the time required to read a graph from disk. This is particularly problematic for our use case because graph compression is usually applied to large graphs. These graphs take up a lot of space on the hard disk, and therefore have slow read times. To prevent reading the graph multiple times due to the building of the compressed graph, we now describe a way to read the graph from disk in a single read operation while compressing it, whereby only additional memory in the size of the largest neighborhood of the graph is required.

One problem that makes it difficult to compress graphs in a single disk-read operation without temporarily storing the graph in an uncompressed format is that the length of the compressed edge array is not known prior to compression and therefore the amount of memory that needs to be allocated is unknown. This is because the length of the compressed edge array is the sum of the lengths of the individual integers, most of which are encoded with variable length. In order to determine the lengths of all variable-length encoded integers, they must be encoded or their encoded lengths have to be determined. This means that the length is not available before the actual compression or before being aware of all integers to encode. Therefore, it is not known how much memory needs to be allocated before the graph to be compressed is read.

To solve this problem, we overcommit enough memory to store the compressed graph. This can be done by either computing an upper bound of the maximal size of the compressed graph if the order and size of the graph is known beforehand, or allocating all the physically available memory. Further, only the memory that stores the compressed graph is written to and read from. This ensures that only these memory pages are mapped to physical pages and therefore no memory is wasted.

Moreover, we want to reduce the additional memory consumption during compression. We therefore compress nodes individually and sequentially. In addition, when reading the graph from disk and compressing it, we only use a single dynamically growing temporary buffer, which stores exactly one neighborhood of a node at each time. The temporary buffer is reused when compressing the nodes. Therefore, the additional memory used during compression is in the size of the largest neighborhood of the graph.

6 Experimental Evaluation

In this chapter, we evaluate the memory optimizations that we introduced in Chapter 4 and Chapter 5. By means of these experiments, we want to find out how the changes affect the memory consumption, the running time and the quality of the produced partition. To this end, we explain the setup of the experiments and methodology in Section 6.1. In Section 6.2, we then describe how we determine the memory consumption for the experiments using a specially integrated heap profiler in KaMinPar. Next, in Section 6.3, we describe the experiments and evaluate the results obtained with our benchmark sets. In Section 6.4, we evaluate our memory optimizations on three huge graphs. Finally, in Section 6.5, we compare memory-efficient KaMinPar against other shared-memory multilevel graph partitioners.

6.1 Experimental Setup and Methodology

The presented memory optimizations have been applied to KaMinPar, which is implemented in C++. For the following experiments, we compiled it using g++ version 12.1.0 and used Intel oneTBB [33] as a parallelization library. The flags `-O3 -mtune=native -march=native -msse4.1 -mcx16` were used for compilation. The additional compiler flag `-mbmi2` was used for the experiment about the fast VarInt decoder using the PEXT instruction.

Systems. The experiments were conducted on two machines. All experiments except the one about the fast VarInt decoder using the PEXT instruction were conducted on a machine equipped with an AMD EPYC 7702P processor, which has 64 cores running at 2.0-3.35 GHz, and with 1024 GB of main memory. Since the machine uses an AMD processor of the second Zen generation, which needs 18 machine cycles for the PEXT instruction [16], the experiment about the fast VarInt decoder was conducted on a machine with an Intel processor. We used a machine with an Intel Xeon Gold 6314U processor, which has 32 cores running at 2.3 GHz, and with 512 GB of main memory. Such a processor only uses 3 machine cycles for the PEXT instruction [16]. Furthermore, both machines run Ubuntu 20.04. For all experiments, we used the taskset program[1] to bind the execution of KaMinPar to one thread of each core in order not to use hyper threading.

[1] `https://github.com/util-linux/util-linux`

Instances. For the evaluation of our changes to the KaMinPar algorithm, the benchmark set by Maas, Gottesbüren and Seemaier [29] is used. It consists of 71 graphs from the SuiteSparse Matrix Collection [11], Network Repository [35], graphs created by compressing texts from the Pizza&Chill corpus [23, 14] and artificially created graphs [17]. The graphs are mostly unweighted except for the six graphs from the compression class. More details about the benchmark set can be found in Appendix A.

We have reordered the graphs for all experiments so that the nodes are sorted into exponentially spaced degree buckets, i.e., a node u with $2^i \leq \deg(u) < 2^{i+1}$ is placed in bucket i. Usually this is done in-memory in KaMinPar but to avoid the memory overheads we do it offline. We reorder the graphs to improve the quality of the clustering produced by label propagation [19]. In addition, all isolated nodes are labeled as the last nodes with this reordering. This allows us to store the first edge IDs at the beginning of each neighborhood as gaps and thus improve the compression ratio for our compression scheme.

Methodology. To evaluate the change in quality, running time and memory consumption from our modifications, we use two different benchmarks. The first benchmark is a complete run of the KaMinPar algorithm. We use this benchmark to measure the impact on partition quality, total running time and total memory consumption. The second benchmark is a label propagation microbenchmark. It consists of one round of label propagation as performed in the KaMinPar algorithm to compute a clustering. We use this benchmark to compare different configurations of the compressed graph scheme and the choice of parameter L for the two-level cluster weight vector. This allows us to obtain information about the effect on the total running time and at the same time shorten the duration of the experiments. It also gives us cleaner measurements because other factors, which are present in a full KaMinPar run, are excluded.

For the experiments using the first benchmark, we run the graphs on five different numbers of blocks $k \in \{ 8, 37, 91, 128, 1000 \}$. We also use five different seeds for each pair of graph and k. For the second benchmark, we just use $k = 128$ because this only has an influence on the maximum cluster weight. Likewise, we run the experiments for each graph on five different seeds. Furthermore, we use $\varepsilon = 0.03$ as the imbalance parameter. To aggregate the results of the experiments, we calculate the edge cut, running time and memory consumption for each graph and k by calculating the arithmetic mean of the results over all seeds. To aggregate the results for one graph, we use the geometric mean over all k-value results, and to aggregate the results for one algorithm, we use the geometric mean over all graph results.

Performance Profiles. We use performance profiles [12] to evaluate how the quality, i.e., the cut, of the partition produced by the algorithm changes with the memory optimizations. Let $\mathcal{A}$ be the algorithms we want to compare and $\mathcal{I}$ the graph instances on which the quality measurement was performed. In addition, let $q_A(I)$ be the quality of algorithm

$A \in \mathcal{A}$ on instance $I \in \mathcal{I}$ and $r_A(I) = \frac{q_A(I)}{\min_{A' \in \mathcal{A}} q_{A'}(I)}$ the so-called performance ratio. Then, the performance profile plots for each algorithm the proportion of instances whose quality is within a factor τ of the best quality. The curve of the following function is therefore drawn for each algorithm $A \in \mathcal{A}$:

$$\rho_A(\tau) := \frac{|\{ I \in \mathcal{I} \mid r_A(I) \leq \tau \}|}{|\mathcal{I}|}$$

An algorithm that achieves higher fractions for smaller τ values is considered better.

6.2 Heap Profiler

To evaluate the memory consumption of the algorithms, we have built our own heap profiler into KaMinPar. The heap profiler works by overriding the standard functions provided by the C standard library for allocating, reallocating and freeing memory, such as `malloc` and `free`. Note that we do not overwrite the memory operations from the C++ standard library, such as `new` and `delete`, because these call the memory operations of the C standard library internally in the implementation of GCC [9], thus avoiding double counting. Since in the Linux-based C standard library of GCC the memory allocation routines are defined as weak symbols [27], which are special symbols during linking that can be overridden by (strong) symbols of the same name, the override is done by simply redefining the corresponding routines. The redefined routines then call the memory routines against which they should actually be linked and additionally track the memory operation. This means that by redefining the memory routines, the memory operations are forwarded as if they had been linked correctly, with the change that the operation is also tracked by our heap profiler. Because memory routines can be called from different PEs in parallel, the tracking of the memory operations is synchronized by locks. Therefore, the measured running time of the algorithms can increase while using the heap profiler.

One advantage of the heap profiler in contrast to an external tool that measures memory consumption is the possibility of a detailed and customizable overview of the memory consumption because it is integrated directly into the program. In addition to the measurement of the overall memory consumption, the heap profiler also allows us to mark sections in the algorithm in which memory measurements are recorded separately. This makes it possible to examine how much memory is consumed by the various components of the algorithm, such as clustering or contraction, and thus obtain a detailed overview of the algorithm's memory consumption.

6.3 Memory Optimizations

In this section, we evaluate our presented memory optimizations. We start the experiments with all memory optimizations disabled and enable them one by one. In addition to the

memory optimizations that we have presented in this thesis, we have made further minor optimizations to KaMinPar, which we also enabled in all the following experiments. We begin by first briefly presenting these minor optimizations.

6.3.1 Minor Optimizations

The first minor optimization is a change in the allocation of a memory buffer used to store subgraphs induced by a partition. This buffer is used for bipartitioning during initial partitioning [19]. It was previously allocated depending on the size of the input graph. Now we allocate it with the size of the coarse graph on which the last bipartitioning takes place because the partition then consists of the desired number of blocks and is therefore not bipartitioned any further. This means that the buffer is not required for larger coarse graphs and memory space can be saved if the buffer is allocated with a minimum size. With this change, the memory consumption of KaMinPar on 64 PEs decreases from 5.87 GB to 4.65 GB and the running time from 3.63 s to 3.48 s.

We have also made changes to the initialization of the greedy balancer, which is responsible for balancing the partition [19]. As the balancer is not always needed, we initialize it lazily to save memory space in such cases. With this additional change, the memory consumption of KaMinPar on 64 PEs decreases from 4.65 GB to 4.46 GB and the running time from 3.48 s to 3.43 s.

As a final minor optimization, we deallocate the data structures of label propagation and contraction directly after execution during coarsening. This means that they are not held in memory at the same time and the memory peak can therefore be reduced. We only apply the technique to coarsening, because coarsening is the decisive factor for the peak memory. With this final change, the memory consumption of KaMinPar on 64 PEs decreases from 4.46 GB to 3.93 GB while the running time increases from 3.43 s to 3.69 s. This means that this change has running time costs of about 7.6%.

6.3.2 Two-Phase Label Propagation

We now present the experiments for evaluating the change in memory consumption, running time and quality when using two-phase label propagation. Note that no further memory optimizations from Chapter 4 or Chapter 5 are enabled for these experiments.

Memory Consumption. In Figure 6.1, we compare the memory consumption of single-phase label propagation and two-phase label propagation. In the plot, we have excluded the data for two-phase label propagation with 4 PEs because its memory consumption roughly corresponds to the memory consumption of two-phase label propagation with 64 PEs. This is because at most one array is allocated and the memory consumption otherwise only differs due to the thread-local hash tables, which are each at most 1 MB in size

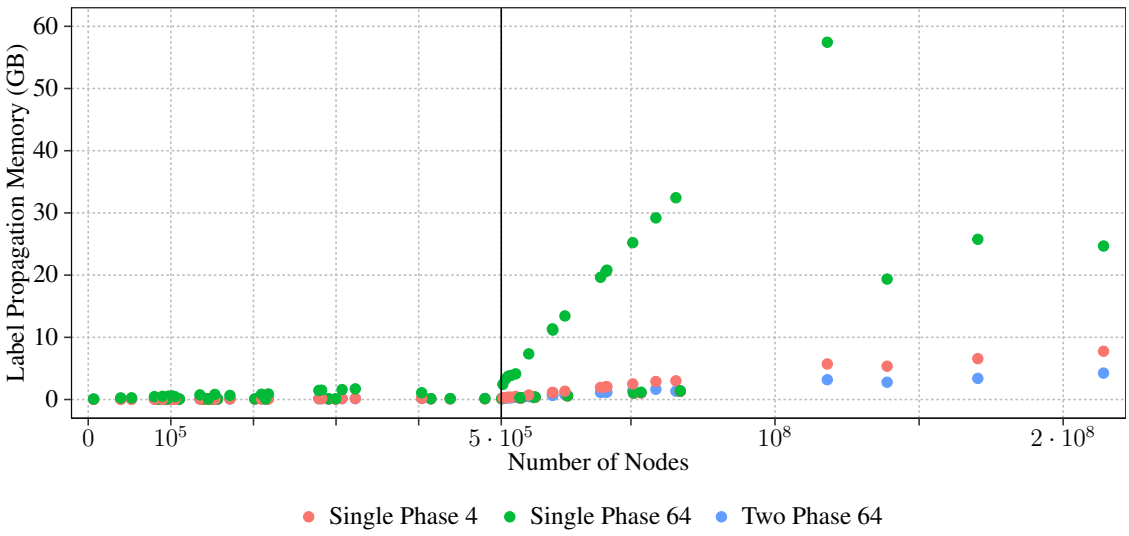

Figure 6.1: The maximum memory consumption of label propagation during the coarsening phase of KaMinPar. Each point (x, y) represents a graph instance, where x is the number of nodes of the graph and y is the corresponding memory consumption of label propagation when run on the graph.

We can observe that single-phase label propagation on 64 PEs has the highest memory consumption, followed by single-phase label propagation on 4 PEs and that two-phase label propagation has the lowest memory consumption. This is expected, because at most one array with memory space in $\Theta(n)$ is allocated for two-phase label propagation in contrast to single-phase label propagation, which in the worst case allocates 4 resp. 64 such arrays when run on 4 resp. 64 PEs. Note that the memory consumption is not strictly linear with respect to the number of nodes because not every PE has to allocate an array or a hash table.

The total memory consumption of KaMinPar on 64 PEs is 3.93 GB with single-phase label propagation and 2.89 GB with two-phase label propagation. With this change, we therefore reduce the total memory consumption on our benchmark set by a factor of 1.36 when run on 64 PEs. When we look at the total memory consumption of the largest graphs ($\Delta(G) \geq 250\,000$) from the benchmark set on 64 PEs, we see that it drops from 12.87 GB to 4.92 GB by a factor of 2.62. In Figure 6.2, we compare the total memory consumption relative to single-phase label propagation on 4 PEs for these graphs. The graph with the best reduction of its memory consumption is webbase2001 with a factor of 5.23, where we reduce its total memory consumption from 65.82 GB to 12.59 GB. However, the maximum memory consumption of KaMinPar with two-phase label propagation is still 25.18 GB for the friendster graph because other factors such as the graph itself still consume memory.

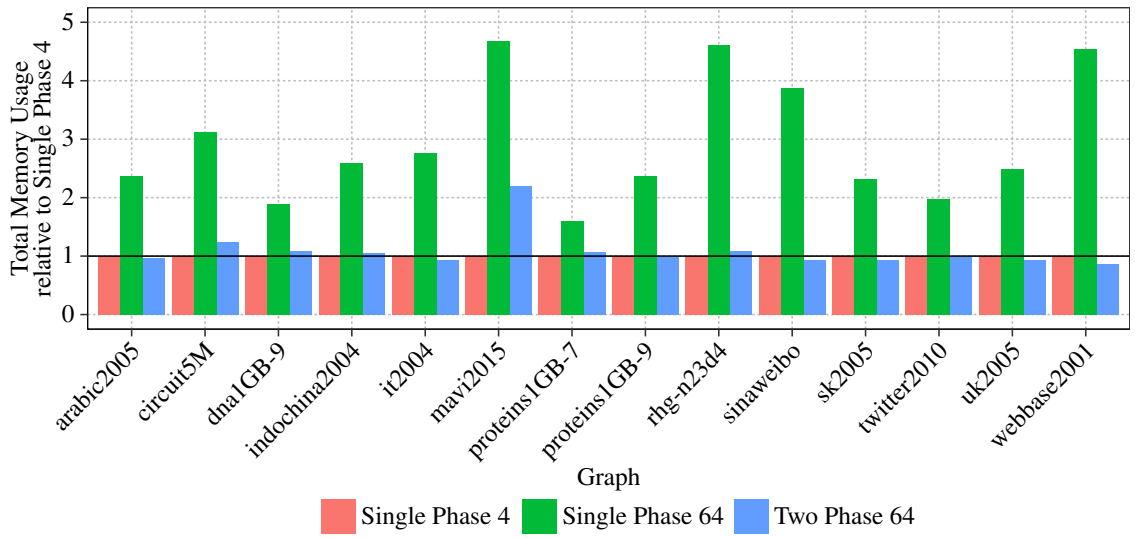

Figure 6.2: The total memory consumption of KaMinPar relative to single-phase label propagation on 4 PEs of the largest graphs ($\Delta(G) \geq 250\,000$) from the benchmark set.

Running Time. Next, we evaluate the running times of KaMinPar when the different variants of two-phase label propagation are used. First, we want to find out which strategy works best to select nodes for the second phase. Then, we also want to find out whether buffering in the second phase provides running time advantages because it may prevent high contention caused by the atomic fetch-and-add instructions.

In Table 6.1, we present the running times of single-phase label propagation and the variants of two-phase label propagation. For two-phase label propagation, we compare High-Degree (HD) and Full-Rating-Map (FRM) as the selection strategies. In addition, we evaluate the advantages of buffering for both selection strategies.

Table 6.1: Geometric mean running times T of KaMinPar for the different variants of two-phase label propagation when run on p PEs.

Algorithm	$T[p = 4]$	$T[p = 64]$
Single-Phase	8.10 s	3.84 s
Two-Phase HD	8.15 s	3.79 s
Two-Phase HD Buffered	8.17 s	3.63 s
Two-Phase FRM	8.27 s	3.48 s
Two-Phase FRM Buffered	8.28 s	3.46 s

We observe that single-phase label propagation is the fastest algorithm on 4 PEs. The algorithm is followed by two-phase label propagation with HD as the selection strategy and then with FRM as the selection strategy, both of which are slightly slower. Furthermore,

the buffered variant brings no running time advantages for both strategies on 4 PEs. This is possibly due to the fact that with 4 threads there is little contention and therefore the overhead when flushing the buffer is predominant.

However, we observe that on 64 PEs all two-phase variants are faster than single-phase label propagation. Furthermore, we can observe that FRM is the faster strategy on 64 PEs. We also see that buffering provides running time advantages for both selection strategies. However, the running time advantage is especially evident for HD as a selection strategy. For the HD selection strategy, using buffers in the second phase reduces the running time by about 4.4% and for the FRM selection mode by about 0.6%. When we look at the running times of individual graphs, we see that for FRM the running time of none of the graphs of our benchmark set decreases noticeable. For HD, however, a noticeable improvement can be observed for individual graphs. For example, the running time for the sk2005 graph and HD as the selection strategy decreases from 65.7 s to 14.6 s due to buffering, which is a speed-up factor of 4.5.

Furthermore, if we only look at the running time required for label propagation and divide it into the time for the first and second phase (only considering graphs that require a second phase), we see that the second phase without buffering uses 1.6% resp. 6.3% of the time for $p = 4$ resp. $p = 64$ with the FRM strategy and 8.7% resp. 54.8% of the time for $p = 4$ resp. $p = 64$ with the HD strategy. This shows that with the FRM strategy, fewer nodes enter the second phase and therefore the percentage of time used for the second phase is less compared to HD.

Quality. Finally, we want to evaluate how the quality is affected by two-phase label propagation. In Figure 6.3, we compare the quality of single-phase label propagation and the variants of two-phase label propagation. We find that two-phase label propagation has slightly better quality than single-phase label propagation and that the FRM strategy results in better quality than the HD strategy.

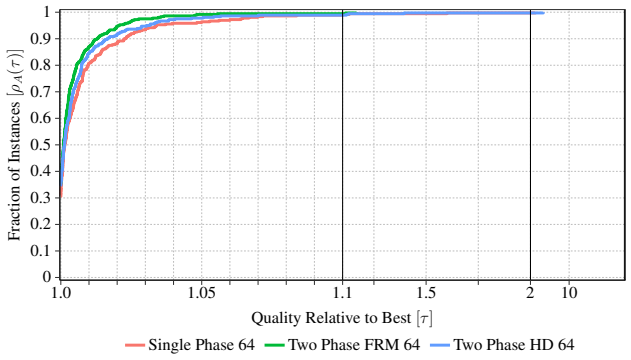

Figure 6.3: Performance profile comparing the quality of KaMinPar when using single-phase label propagation and the different variants of two-phase label propagation on 64 PEs.

6.3.3 Graph Compression

In this section, we evaluate the memory consumption, running time and impact on quality when our graph compression scheme is used to store the input graph in KaMinPar. For all these experiments, we use two-phase label propagation with FRM as the selection strategy and we do not use buffering in the second phase.

Compression Scheme. First, we evaluate the different variants of our compression scheme. As compression techniques for the adjacency array we use the compact node array and the edge array with variable-length encoding and gap encoding, which we call gap encoding as a whole. Furthermore, we investigate the effects of applying interval encoding and high-degree (HD) encoding to the edge array. We use $T_{\text{Interval}} = 3$ as the threshold for the minimum interval length, $T_{\text{Part}} = 10000$ as the degree threshold for the use of HD encoding for a node and $L_{\text{Part}} = 1000$ as the length of the parts when splitting the neighborhood of a high-degree node. In Table 6.2, we compare the compression ratios and the running times of these compression schemes.

Table 6.2: Geometric mean compression ratios and running times T of a label propagation micro benchmark with p PEs when graph compression is used with the corresponding scheme.

Compression Scheme	Ratio	$T[p = 4]$	$T[p = 64]$
Uncompressed	-	2.23 s	0.47 s
Gap Encoding	2.28	2.63 s	0.50 s
Gap HD Encoding	2.27	2.75 s	0.48 s
Gap Interval Encoding	2.80	2.80 s	0.52 s
Gap Interval HD Encoding	2.80	3.08 s	0.51 s

We observe that with gap encoding alone we already achieve an average compression ratio of 2.28, i.e., we can reduce the memory space of the input graph by more than half on average. Furthermore, we can achieve even higher compression ratios if we additionally use interval encoding for the edge array. With such a scheme we achieve compression ratios of 2.80 on average. However, the running time when we use graph compression increases on average. This applies to the running time on both 4 and 64 PEs, whereby the relative slowdown is higher with 4 than with 64 PEs. We also see that HD encoding improves the running time on 64 PEs and has very little impact on the compression ratio.

In Figure 6.4, we present the detailed compression ratios of all graphs. The best compression ratio we can achieve with gap encoding is 3.86 for the mycielskian19 graph. With interval encoding, the best compression ratio is 8.79 for the HV15R graph. Compression ratios of less than one are also achieved. This concerns the kmer-A2a and kmer-V1r graphs, whereby the kmer graphs are all poorly compressible with our scheme. This is due to the fact that the node IDs are not particularly reduced by gap encoding, and thus, despite variable-length encoding, these IDs are stored on average with about 4 bytes. Then, there is the fact that

we store the first edge ID for each neighborhood, which causes more bytes to be required overall than in the uncompressed adjacency array.

Interval Length Threshold. Because the running time of interval encoding increase compared to gap encoding, which is especially the case on 4 PEs, we now investigate the influence of the minimum interval length threshold T_{Interval} on the running time and compression ratio. With this, we hope to find a threshold that has better running times with only a small loss in the compression ratio. Table 6.3 shows the compression ratios and running times of various thresholds. We find that even increasing the threshold by one or two does considerably reduce the compression ratio and has only minimal benefits for the running time. For the following experiments, we therefore continue to use $T_{\text{Interval}} = 3$.

Table 6.3: Geometric mean compression ratio and running times T of a label propagation microbenchmark with p PEs when using the corresponding interval length threshold.

T_{Interval}	Ratio	$T[p = 4]$	$T[p = 64]$
3	2.80	3.08 s	0.51 s
4	2.74	3.06 s	0.50 s
5	2.71	3.04 s	0.50 s

Variable-Length Encoding. In Table 6.4, we compare the compression ratios and running times when using VarInts, RLE and Stream VByte encoding (SVE). As expected, the compression ratios of both alternative variable-length encodings decrease because RLE stores one byte header for each run of integers with the same byte width and SVE stores two additional bits for each integer. On 64 PEs, both encodings offer running time advantages compared to VarInt. On 4 PEs, however, only RLE brings running time advantages. In the following, we will continue to use VarInt as the variable-length encoding for all experiments because of the better compression ratio and because the running time improvements of the alternative variable-length encodings are only minimal.

Table 6.4: Geometric mean compression ratio and running times of a label propagation microbenchmark with p PEs when using the corresponding variable-length encoding.

Encoding	Ratio	$T[p = 4]$	$T[p = 64]$
Uncompressed	-	2.23 s	0.47 s
VarInt	2.80	3.08 s	0.51 s
RLE	2.66	3.06 s	0.50 s
SVE	2.69	3.09 s	0.50 s

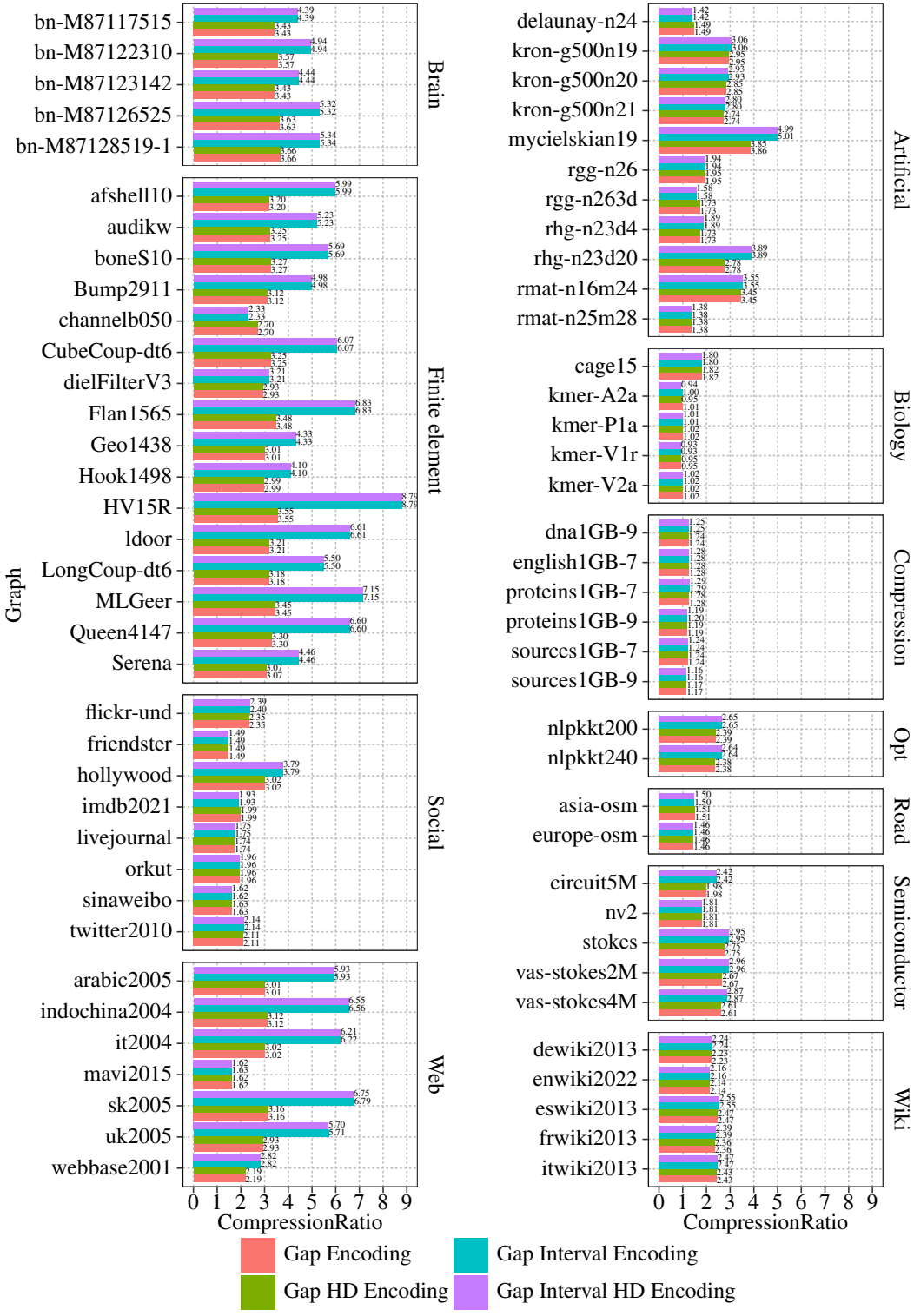

Figure 6.4: Compression ratios for all graphs in the benchmark set.

Fast VarInt Decoder. In Table 6.5, we compare the running time when using the VarInt decoder, which is implemented using a while loop, and the fast decoder, which is implemented using the PEXT instruction. We observe that the fast VarInt decoder has better running times. On 4 PEs it improves the running time by about 7.5% and on 64 PEs by about 8.2%. In the following, however, we do not use the fast VarInt decoder for the reason that it requires a natively supported PEXT instruction and the running time improvement therefore depends on the system. For example, the system on which we conducted all other experiments implements the PEXT in microcode [16], which is why the running time advantages are not available on this system.

Table 6.5: Geometric mean running times of a label propagation microbenchmark with p PEs when using the corresponding VarInt decoder. Note that this experiment was carried out on a different machine than the other experiments.

VarInt Decoder	$T[p = 4]$	$T[p = 32]$
Uncompressed	2.00 s	0.40 s
VarInt	2.73 s	0.53 s
Fast VarInt	2.54 s	0.49 s

Total Memory Consumption. The total memory consumption of KaMinPar on 64 PEs is 2.89 GB without graph compression and is 2.33 GB with graph compression. We therefore reduce the memory consumption of KaMinPar on our benchmark set with graph compression by a factor of 1.26.

Total Running Time. The total running time of KaMinPar with graph compression is 9.55 s for $p = 4$ and 3.62 s for $p = 64$. KaMinPar without graph compression has running times of 8.27 s for $p = 4$ and 3.48 for $p = 64$. Therefore, we have a slowdown of 1.16 for $p = 4$ and of 1.04 for $p = 64$ on average. However, by using graph compression, on some inputs we also achieve faster running times. Figure 6.5 shows the relative total running times of KaMinPar for the graphs that are faster with graph compression. This is given for 13 graphs.

Quality. The left plot in Figure 6.6 compares the quality of KaMinPar when it uses graph compression and when it does not use graph compression. There we can see that KaMinPar produces better partitions with the uncompressed graph. The loss in quality is due to the edge reordering of the compressed graph. This can be seen in the right plot in Figure 6.6, which compares the quality of KaMinPar using the compressed graph and the uncompressed graph, whose edges are reordered according to the order of the compressed graph. There we can see that both have the same (worse) quality. We explain the worse quality due to edge reordering by the fact that in label propagation, when selecting one of several clusters

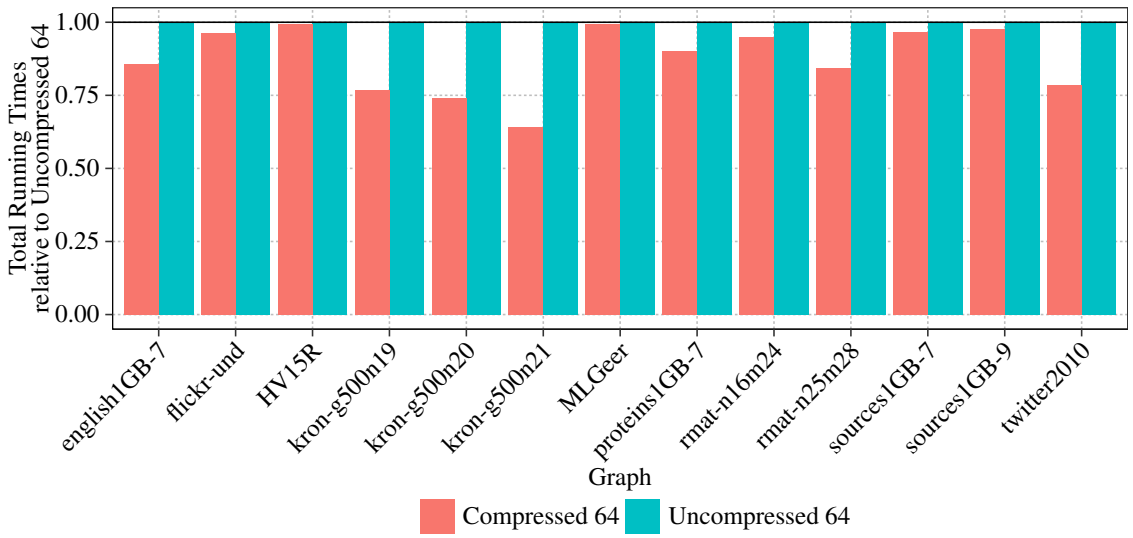

Figure 6.5: The total running time of KaMinPar relative to using the uncompressed graph representation on 64 PEs for the graphs that are faster with graph compression.

with the same number of members in a neighborhood, the clusters of the adjacent nodes at the end of a neighborhood are strongly preferred. Therefore, we believe that we can restore the quality loss with a uniform random tie-breaking strategy. However, this requires further experiments, which we do not address in this thesis.

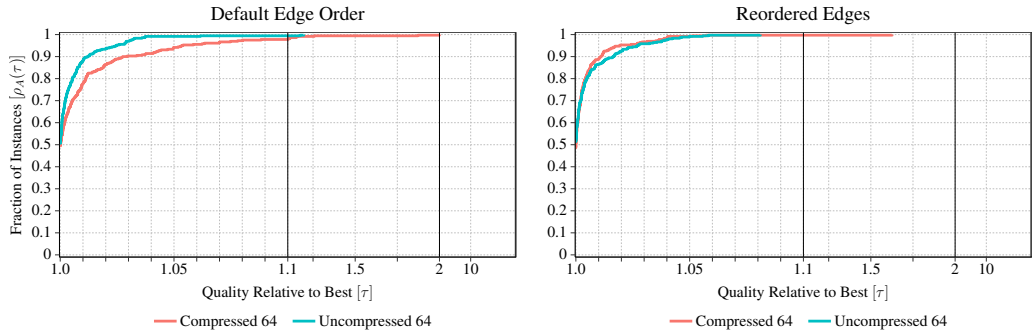

Figure 6.6: Performance profile comparing the quality of KaMinPar when using and not using graph compression. The plot on the left compares the quality when the uncompressed graph uses its default edge order and the plot on the right compares the quality when the uncompressed graph uses the edge order of the compressed graph.

6.3.4 Contraction

In this section, we evaluate the three proposed changes to the contraction algorithm to reduce the memory consumption of the edge buffer. In the following experiments, we still use two-phase label propagation with FRM as the selection strategy and without buffering during the second phase. We also use graph compression with gap encoding, interval encoding, high-degree encoding and VarInts as the compression scheme. Furthermore, we use $M = 1/10$ as the parameter for the contraction variant that fills the edge buffer partially.

Memory Consumption. In Figure 6.7, we compare the memory consumption of all four contraction implementations. We observe that all three proposed changes reduce the memory consumption of contraction. Moreover, as expected, the two variants that completely eliminate the edge buffer perform better than the variant that only partially fills the edge buffer. If we look at the maximum memory consumption of contraction on 64 PEs aggregated over all graphs, the original variant requires 0.36 GB, the variant that fills the edge buffer partially 0.29 GB and the variants that eliminate the edge buffer 0.25 GB. However, if we look at the total memory consumption, it is 2.33 GB for the original implementation, 2.31 GB for the variant that partially fills the edge buffer and 2.25 GB for the variants without edge buffer. The reason why the total memory consumption decreases less than the memory consumption for contraction is due to the change that the data structures of label propagation are deallocated afterwards. As a result, contraction is not always responsible for the peak memory and therefore the change is not always reflected in the total peak memory.

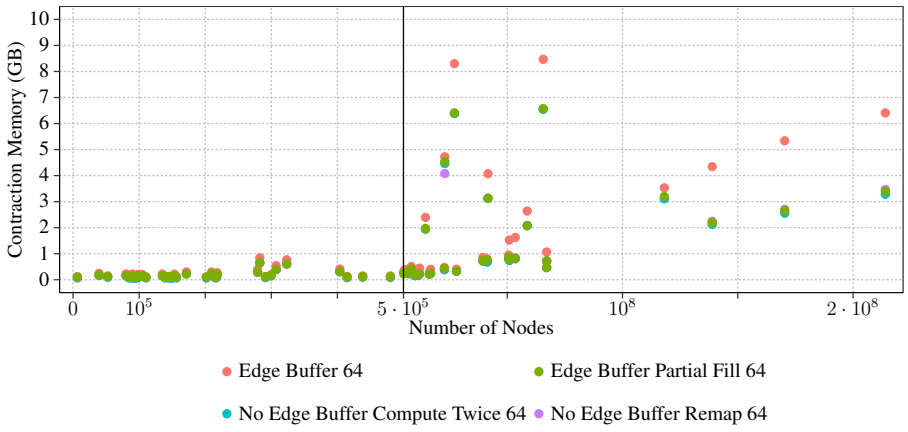

Figure 6.7: The maximum memory consumption of contraction during the coarsening phase of KaMinPar. Each point (x, y) represents a graph instance, where x is the number of nodes of the graph and y is the corresponding memory consumption of contraction when run on the graph.

Running Time. In Table 6.6, the running times of the contraction algorithms are compared. We find that on 4 PEs, the variant that aggregates the neighborhoods of the coarse nodes twice is the slowest. On 64 PEs, however, the variant that only partially fills the edge buffer is the slowest. Moreover, in both cases, the variant that eliminates the edge buffer by remapping is the fastest. This variant is even faster than the original implementation.

Table 6.6: Geometric mean running times T of KaMinPar with p PEs for the different contraction algorithms.

Algorithm	$T[p = 4]$	$T[p = 64]$
Edge Buffer	9.55 s	3.62 s
Edge Buffer Partial Fill	9.64 s	3.74 s
No Edge Buffer Compute Twice	10.02 s	3.65 s
No Edge Buffer Remap	9.49 s	3.60 s

Quality. Finally, in Figure 6.8, we compare the change in quality when using the different contraction algorithms. As we can see, the curves overlap, which means that all algorithms compute the best partition on about the same fraction of graphs. We therefore observe no loss of quality when a different contraction algorithm is used.

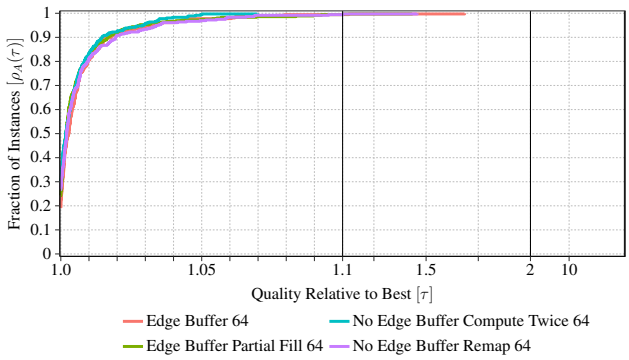

Figure 6.8: Performance profile comparing the quality of KaMinPar when used with the different contraction algorithms on 64 PEs.

6.3.5 Two-Level Cluster Weight Vector

In this section, we evaluate the two-level vector for storing the cluster weights. As before, we use two-phase label propagation with graph compression for all experiments. This time we also use the contraction algorithm that eliminates the edge buffer by remapping.

Impact of the Vector Entry Size L. First, we evaluate the influence of the parameter $L \in \{8, 16, 32\}$ (word width in bits of the first level) on the running time and memory consumption. As concurrent hash tables we have used the hash tables of oneTBB [33] and growt [30]. Table 6.7 shows the running time for the different L values and the two hash tables. For both hash tables and $L = 8$ is the original data structure for storing the cluster weights faster. However, for $L = 16$ and $L = 32$ the two-level cluster weight vector improves the running time except for growt with $L = 32$ and $p = 4$, where it performs slightly worse. Furthermore, oneTBB's hash table is faster than growt's hash table in all configurations.

Table 6.7: Geometric mean running times T of a label propagation microbenchmark when run on p PEs and using the two-level cluster weight vector with the corresponding parameter L.

	oneTBB		growt	
L	$T[p=4]$	$T[p=64]$	$T[p=4]$	$T[p=64]$
Reference	3.08 s	0.51 s	3.08 s	0.51 s
8	3.24 s	0.66 s	3.64 s	1.57 s
16	2.85 s	0.49 s	3.04 s	0.50 s
32	2.86 s	0.49 s	3.11 s	0.51 s

In Table 6.8, we compare the memory consumption for the different L values and the two hash tables. The best memory consumption can be seen with the two-level cluster weight vector and $L = 8$, then with $L = 16$ and then with $L = 32$. As expected, the original implementation has the worst memory consumption. We also see that the hash table of oneTBB has the lower memory consumption.

Table 6.8: Geometric mean memory consumption M of a label propagation microbenchmark when run on p PEs and using the two-level cluster weight vector with the corresponding parameter L.

	oneTBB		growt	
L	$M[p=4]$	$M[p=64]$	$M[p=4]$	$M[p=64]$
Reference	116.81 MB	228.15 MB	116.81 MB	228.15 MB
8	84.90 MB	189.91 MB	87.32 MB	191.44 MB
16	89.49 MB	195.53 MB	91.92 MB	197.03 MB
32	98.63 MB	206.60 MB	101.09 MB	208.13 MB

Total Running Time and Memory Consumption. The running time of KaMinPar without the two-level cluster weight vector is 9.49 s for $p = 4$ and 3.60 for $p = 64$. The running time with the two-level cluster weight vector and $L = 16$ is 9.18 s for $p = 4$ and

3.62 s for $p = 64$. Therefore, we find that the running time decreases slightly for $p = 4$ and increases slightly for $p = 64$. Furthermore, when we look at the total memory consumption of KaMinPar we find that it decreases from 2.25 GB to 2.20 GB.

6.4 Memory Savings for Huge Graphs

The experiments in Section 6.3 show that our memory optimizations reduce the memory consumption of KaMinPar on 64 PEs for our benchmark set from 3.93 GB to 2.20 GB and that the running time even decreases from 3.84 s to 3.60 s. However, the benchmark set consists of no huge graphs, which means that they can be partitioned on a machine with a reasonable amount of main memory even before our memory optimizations. In the following, we therefore want to evaluate how much the memory consumption of huge graphs is reduced by our memory optimization and what effect it has on the running time. For this purpose, we use three of the largest graphs available: gsh2015, clueweb12 and uk2014 [4, 5, 6]. Details about the graphs are listed in Table 6.9.

Table 6.9: Number of nodes n, number of edges m, average degree $d(G)$, max degree $\Delta(G)$ and class of the three evaluated huge graphs.

Graph G	n	m	$d(G)$	$\Delta(G)$	Class
gsh2015	988 490 691	51 381 410 236	76.4	75 611 696	Web
clueweb12	978 408 098	74 744 358 622	52.0	58 860 305	Web
uk2014	787 801 471	84 928 431 100	107.8	8 605 492	Web

We performed the experiments for the huge graphs with 64 PEs. Furthermore, we only used $k = 128$ because we are primarily interested in the reduction of the memory consumption, thus shortening the duration of the experiments. In addition, we disabled the optimization that the data structures of label propagation and contraction are deallocated after each coarsening level. We did this because the huge graphs require a considerable amount of time for deallocation (in contrast to the graphs in our benchmark set) because the data structures used for them are so large, which can therefore bias the running times.

Memory Consumption. In Figure 6.9, we show the reduction of the memory consumption for each graph, where we apply the memory optimizations one by one. Through two-phase label propagation, the memory consumption is on average reduced by a factor of 2.40. When graph compression is additionally used, the memory consumption is further reduced by a factor of 3.86 on average. Here, we achieve compression ratios of 5.53 for gsh2015, 6.68 for clueweb12 and 9.76 for uk2014. Moreover, through the use of the contraction algorithm that avoids the edge buffer by remapping, we further reduce the memory consumption by a factor of 1.02 on average. Note that this change has no effect

on the memory consumption of uk2014 because KaMinPar's contraction implementation performs precomputations before the actual contraction, with the necessary memory being released afterwards. Thus, the peak memory of contraction is dependent on either the precomputations or the actual contraction. In this case, the precomputations determine the peak memory, which is why our memory change is not visible in the peak memory. Lastly, through the two-level cluster weight vector, we can reduce the memory consumption by a factor of 1.04 on average. Therefore, we have an average reduction in peak memory of 9.85 due to the memory optimizations.

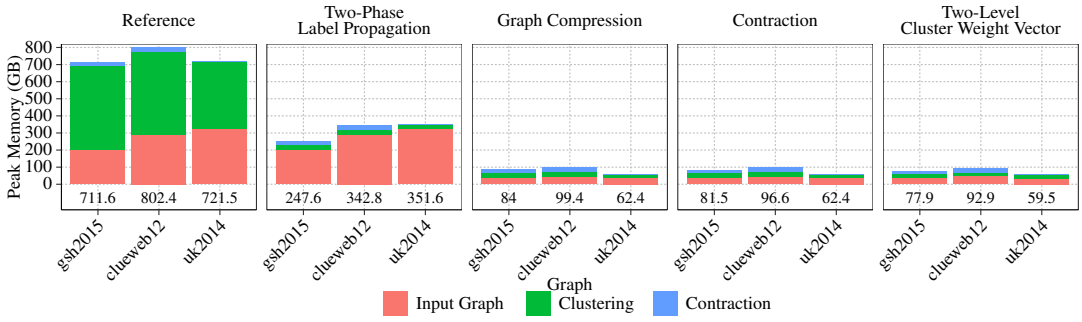

Figure 6.9: Peak memory of KaMinPar for the huge graphs when the memory optimizations are applied one by one.

Running Time. In Table 6.10, we compare the running times of KaMinPar for the huge graphs, where the memory optimizations are enabled one by one. We find that two-phase label propagation reduces the running time on average by a factor of 1.19. Moreover, through the use of graph compression, the running time increase on average by a factor of 1.31. The contraction algorithm, which avoids the edge buffer by remapping, also increases the running time on average, however only by about 0.8%. Lastly, through the use of the two-level cluster weight vector, the running time is on average reduced by a factor of 1.02. Therefore, when all memory optimizations are used, the average runtime increases by a factor of 1.08 for the huge graphs, with the slowdown coming primarily from graph compression.

Table 6.10: Running times and geometric mean running time of KaMinPar for the huge graphs when the memory optimizations are applied one by one.

Memory Optimization	gsh2015	clueweb12	uk2014	Average
Reference	140.5 s	192.6 s	129.5 s	151.9 s
Two-Phase Label Propagation	124.9 s	164.6 s	100.6 s	127.4 s
Graph Compression	165.8 s	219.2 s	126.8 s	166.4 s
Contraction	166.8 s	223.8 s	126.3 s	167.7 s
Two-Level Cluster Weight Vector	165.2 s	215.8 s	123.8 s	164.0 s

6.5 Comparison to Other Shared-Memory Graph Partitioners

In this section, we compare the memory consumption of KaMinPar to the memory consumption of other multilevel graph partitioners. For the comparison, we use Metis (sequential) [24] and MtMetis (multi-threaded) [25] as in-memory partitioners and HeiStream [13] as a streaming partitioner.

We run the multi-threaded partitioners KaMinPar and MtMetis on 64 PEs. Metis and HeiStream run as sequential algorithms by default on one PE. Furthermore, we only run the experiments with $k = 16$ because we are again primarily interested in the reduction of the memory consumption and can therefore shorten the duration of the experiments. To control the size of the input graphs, we use randomly generated graphs for this experiment [17]. rgg$N$ refers to a family of 2D random geometric graphs generated with 2^N nodes and average degree $d(\text{rgg}N) = 8$. These graphs do not have any high-degree nodes and loosely represent the class of regular graphs. To represent social networks with a skewed power-law degree distribution, we employ a family of randomly generated hyperbolic graphs denoted rhg$N$. These graphs are generated with 2^N nodes, power-law exponent $\gamma = 3.0$ and average degree $d(\text{rhg}N) = 8$. Note that we did not run Metis and MtMetis on the huge web graphs from Section 6.4 because they run out of memory for these graphs on the testing machine.

Memory Consumption. In Figure 6.10, we compare the memory consumption of HeiStream, KaMinPar with memory optimizations, KaMinPar without memory optimizations, Metis and MtMetis, where all failed runs are marked with ✗. We see that HeiStream has the lowest memory consumption. This is because its memory consumption as a streaming algorithm is mainly determined by the partition of the graph with memory space in $\mathcal{O}(n)$. Therefore, its memory consumption is on average 16.4 times less than the memory consumption of memory-efficient KaMinPar. We also see that memory-efficient KaMinPar uses on average 1.7 resp. 5.5 times less memory for the rgg resp. rhg graphs than KaMinPar without memory optimizations. Furthermore, it uses on average 5.6 times less memory than Metis and 5.3 times less memory than MtMetis. Thereby, on average, we achieve a compression ratio of 2.33 for the rgg graphs and a compression ratio of 1.82 for the rhg graphs.

We observe that the memory consumption of all competing algorithms increases linearly with the order of the graph. By extrapolating the memory consumption for Metis and MtMetis, we find that Metis resp. MtMetis would roughly require 1 071.4 GB resp. 987.4 GB for rgg31 and 1 164.8 resp. 1 108.3 GB for rhg31 by our calculation. Therefore, as expected, they run out of memory on the largest graphs in the families. We also find that KaMinPar without memory optimizations would require 1 213.2 GB for rgg31. Furthermore, assuming that the memory consumption of memory-efficient KaMinPar continues to scale linearly, it would require 506.7 GB resp. 621.6 GB for rgg33 resp. rhg33 and 1 013.4 GB reps. 1 243.1

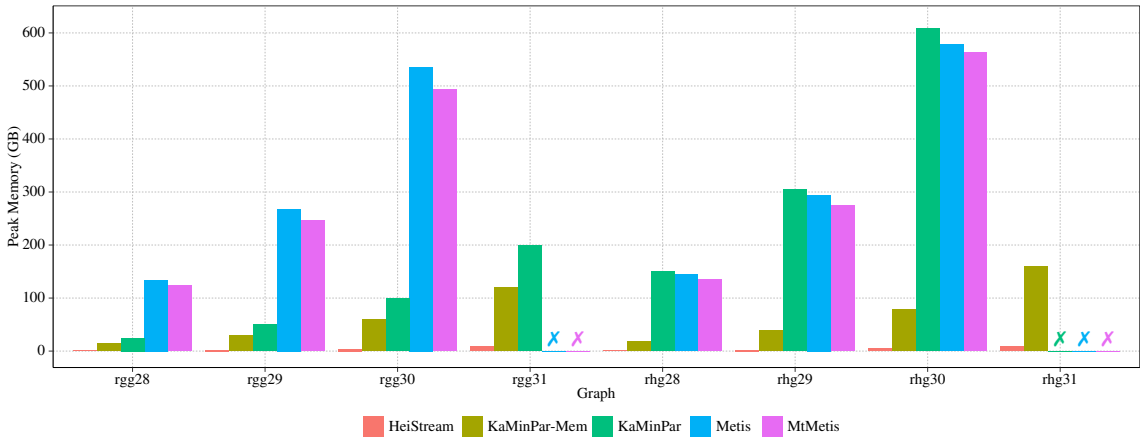

Figure 6.10: Peak memory of the graph partitioners we compare, where all failed runs are marked with ✗.

GB for rgg34 resp. rhg34. Therefore, memory-efficient KaMinPar would presumably no longer be able to partition rgg34 and rhg34 on the machine we conduct the experiments. However, these are only calculations, which we cannot verify experimentally because of limitations of the used graph generator.

Running Time. In Figure 6.11, we compare the running times of the graph partitioners. KaMinPar without memory optimizations is the fastest algorithm with 43.7 s on average, followed by memory-efficient KaMinPar with 44.8 s on average and then followed by MtMetis with 100.6 s on average. Therefore, we find that despite the overhead of graph compression, we are faster than MtMetis on the same number of PEs by a factor of 2.25 and 2.24 on the rgg and rhg graphs, respectively. Moreover, Metis requires 577.9 s and HeiStream 813.3 s on average because, unlike KaMinPar and MtMetis, these are sequential algorithms. Note that we have excluded the graphs rgg31 and rhg31 for the average results above as they fail under Metis and MtMetis.

Quality. In Table 6.11 and Table 6.12, we compare the quality of the partitions produced for the rgg and rhg graphs by the algorithms we compare. KaMinPar produces the best cut, followed by Metis and MtMetis. HeiStream has the worst cut as a streaming algorithm. In addition, the largest graphs rgg31 and rhg31 do not run on Metis and MtMetis, which is why there is only the streaming algorithm for these graphs when compared to KaMinPar. Thus, compared to HeiStream, memory-efficient KaMinPar produces partitions with a cut that is 8.64 times better for rgg31 and 59 452.23 times better for rhg31. We see that HeiStream's low memory consumption is possible due to compromises in quality, with partitions having such a high cut that they are unusable for many applications.

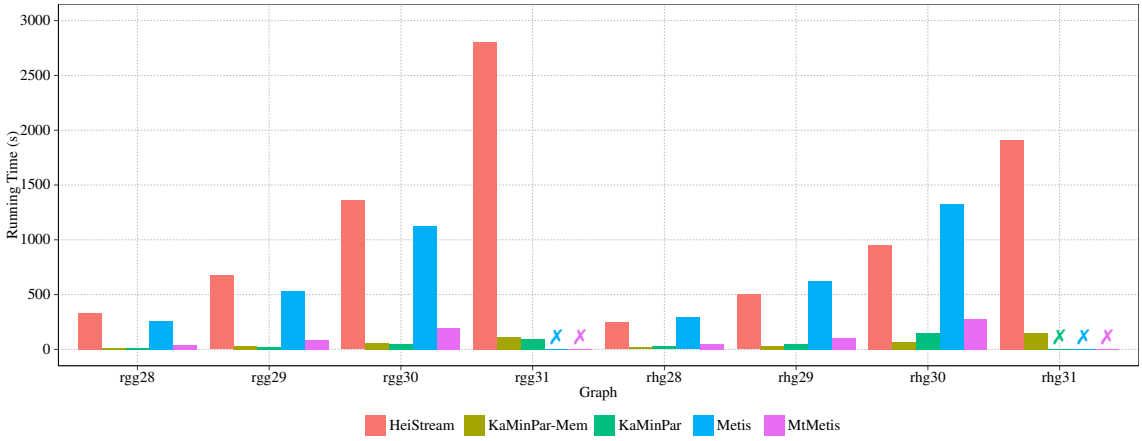

Figure 6.11: Running times of the graph partitioners we compare, where all failed runs are marked with ✗.

Table 6.11: Cuts of the partitions produced by the algorithms for the rgg graphs and the geometric mean cut relative to memory-efficient KaMinPar (excluding rgg31).

Graph	HeiStream	KaMinPar-Mem	KaMinPar	Metis	MtMetis
rgg28	3 787 408	406 480	407 490	456 658	465 934
rgg29	5 148 035	584 279	589 249	650 954	669 445
rgg30	7 549 167	835 903	836 530	935 634	946 685
rgg31	10 252 472	1 186 065	1 205 191	✗	✗
Relative Cut	9.10	1.00	1.01	1.12	1.14

Table 6.12: Cuts of the partitions produced by the algorithms for the rhg graphs and the geometric mean cut relative to memory-efficient KaMinPar (excluding rhg31).

Graph	HeiStream	KaMinPar-Mem	KaMinPar	Metis	MtMetis
rhg28	62 175 712	5 105	4 224	5 117	8 773
rhg29	118 444 885	3 022	2 857	5 272	8 946
rhg30	219 850 739	6 145	5 037	7 286	11 818
rhg31	465 570 405	7 831	✗	✗	✗
Relative Cut	25 752.46	1.00	0.86	1.28	2.14

7 Conclusion

In this thesis, we presented algorithmic modifications for the coarsening phase of a multilevel graph partitioner and a graph compression scheme, both aimed at reducing the memory consumption of the in-memory parallel multilevel graph partitioner KaMinPar. The changes to the clustering step of the coarsening phase include a two-phase implementation of label propagation, such that its memory consumption does not scale with the number of threads and is instead in $\mathcal{O}(n)$, and a more memory-efficient data structure for storing the cluster weights. For the contraction step of the coarsening phase, we presented various changes to the contraction implementation, such that the memory consumption of a temporary edge buffer is reduced. Moreover, we presented a graph compression scheme, which enables us to store the input graph with less memory space. We also looked at various methods to accelerate the decoding of compressed neighborhoods. Finally, we presented a way to compress the graph in a single-read disk operation.

Furthermore, we integrated these memory optimizations into KaMinPar and evaluated them. With two-phase label propagation, we can reduce the memory consumption of label propagation for the largest graph that we tested, namely clueweb12, by a factor of about 2.34. By additionally using graph compression, we can store the clueweb12 graph more space-efficiently, consuming about 3.45 times less memory, where we achieve a compression ratio of 6.68. Finally, through changes to the cluster weight vector and the contraction implementation, we reduced the memory consumption of clueweb12 by a factor of about 1.07. Therefore, we reduced the memory consumption for clueweb12 from 802.4 GB to 92.9 GB. As a consequence, we can now partition some of the largest graphs available with an in-memory graph partitioner on a machine with a reasonable amount of main memory.

7.1 Future Work

Although we have improved the memory consumption of KaMinPar in several places in this thesis, there is still potential to further reduce its memory consumption. One component of KaMinPar that could be considered in future work is the greedy balancer. It is responsible for balancing a partition during the uncoarsening phase if it becomes unbalanced by being projected onto a finer graph [19]. Furthermore, the implementation of the Fiduccia-Mattheyses (FM) algorithm [15] could also be considered in order to reduce memory consumption. FM can be used instead of label propagation in the refinement step during uncoarsening to improve the quality of the partition as it is a more advanced tech-

nique [19]. However, the memory consumption of KaMinPar's current implementation is in $\mathcal{O}(nk)$, rather than $\mathcal{O}(k)$ as with label propagation. Further work is therefore required on the implementation of the algorithm in order to use FM during refinement while keeping the memory consumption low. In addition, future work must investigate whether a uniform tie-breaking strategy in the label propagation algorithm can restore the loss of quality due to the edge reordering of the compressed graph, as we suspect it can.

Finally, considering the fact that even larger graphs will be partitioned in the future and that in-memory partitioners are ultimately limited by the main memory of a single machine, it should be explored to what extent our presented memory optimizations can be applied to other partitioner types. For example, it could be explored whether our optimizations can be applied to the distributed KaMinPar algorithm [36], which looks promising since the distributed and in-memory KaMinPar share the same basic partitioning scheme. Transferring our techniques would then make it possible to partition very huge graphs, on which memory-efficient KaMinPar fails due to memory constraints, with a distributed algorithm that use smaller server clusters or clusters with machines using less memory, and thus being more cost-effective.

Bibliography

[1] Yaroslav Akhremtsev, Peter Sanders and Christian Schulz. "(Semi-)External Algorithms for Graph Partitioning and Clustering". In: *Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments, ALENEX 2015, San Diego, CA, USA, January 5, 2015*. Ed. by Ulrik Brandes and David Eppstein. SIAM, 2015, pp. 33–43. DOI: 10.1137/1.9781611973754.4.

[2] Maciej Besta and Torsten Hoefler. "Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations". In: *CoRR* abs/1806.01799 (2018). arXiv: 1806.01799.

[3] Paolo Boldi and Sebastiano Vigna. "Codes for the World Wide Web". In: *Internet Math.* 2.4 (2005), pp. 407–429. DOI: 10.1080/15427951.2005.10129113.

[4] 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. DOI: 10.1145/988672.988752.

[5] Paolo Boldi et al. "BUbiNG: Massive Crawling for the Masses". In: *ACM Trans. Web* 12.2 (2018), 12:1–12:26. DOI: 10.1145/3160017.

[6] 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. DOI: 10.1145/1963405.1963488.

[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. DOI: 10.1016/0020-0190(92)90140-Q.

[8] Aydin Buluç et al. "Recent Advances in Graph Partitioning". In: *Algorithm Engineering - Selected Results and Surveys*. Ed. by Lasse Kliemann and Peter Sanders. Vol. 9220. Lecture Notes in Computer Science. 2016, pp. 117–158. DOI: 10.1007/978-3-319-49487-6_4.

[9] Paolo Carlini et al. *The GNU C++ Library Manual*. Free Software Foundation. 2023. URL: https://gcc.gnu.org/onlinedocs/gcc-12.1.0/libstdc++/manual.

[10] Cédric Chevalier and François Pellegrini. “PT-Scotch: A tool for efficient parallel graph ordering”. In: *Parallel Comput.* 34.6-8 (2008), pp. 318–331. DOI: 10.1016/J.PARCO.2007.12.001.

[11] 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. DOI: 10.1145/2049662.2049663.

[12] Elizabeth D. Dolan and Jorge J. Moré. “Benchmarking optimization software with performance profiles”. In: *Math. Program.* 91.2 (2002), pp. 201–213. DOI: 10.1007/S101070100263.

[13] Marcelo Fonseca Faraj and Christian Schulz. “Buffered Streaming Graph Partitioning”. In: *ACM J. Exp. Algorithmics* 27 (2022), 1.10:1–1.10:26. DOI: 10.1145/3546911.

[14] Paolo Ferragina and Gonzalo Navarro. *Pizza&Chili Corpus (Compressed Indexes and their Testbeds)*. Sept. 2005. URL: http://pizzachili.dcc.uchile.cl/index.html.

[15] 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. DOI: 10.1145/800263.809204.

[16] Agner Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Technical University of Denmark. Nov. 2022. URL: https://www.agner.org/optimize/#manuals.

[17] Daniel Funke et al. “Communication-free massively distributed graph generation”. In: *J. Parallel Distributed Comput.* 131 (2019), pp. 200–217. DOI: 10.1016/J.JPDC.2019.03.011.

[18] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN: 0-7167-1044-7.

[19] Lars Gottesbüren et al. “Deep Multilevel Graph Partitioning”. In: *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*. Ed. by Petra Mutzel, Rasmus Pagh and Grzegorz Herman. Vol. 204. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 48:1–48:17. DOI: 10.4230/LIPICS.ESA.2021.48.

[20] Bruce Hendrickson and Robert W. Leland. “A Multi-Level Algorithm For Partitioning Graphs”. In: *Proceedings Supercomputing ’95, San Diego, CA, USA, December 4-8, 1995*. Ed. by Sidney Karin. ACM, 1995, p. 28. DOI: 10.1145/224170.224228.

[21] Yedidya Hilewitz and Ruby B. Lee. “Fast Bit Compression and Expansion with Parallel Extract and Parallel Deposit Instructions”. In: *2006 IEEE International Conference on Application-Specific Systems, Architecture and Processors (ASAP 2006), 11-13 September 2006, Steamboat Springs, Colorado, USA*. IEEE Computer Society, 2006, pp. 65–72. DOI: 10.1109/ASAP.2006.33.

[22] Laurent Hyafil and Ronald L. Rivest. *Graph Partitioning and Constructing Optimal Decision Trees are Polynomial Complete Problems*. Tech. rep. Rapport de Recherche no. 33. IRIA – Laboratoire de Recherche en Informatique et Automatique, Oct. 1973.

[23] Artur Jez. “Faster Fully Compressed Pattern Matching by Recompression”. In: *ACM Trans. Algorithms* 11.3 (2015), 20:1–20:43. DOI: 10.1145/2631920.

[24] 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. DOI: 10.1137/S1064827595287997.

[25] 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. DOI: 10.1109/IPDPS.2013.50.

[26] Daniel Lemire, Nathan Kurz and Christoph Rupp. “Stream VByte: Faster byte-oriented integer compression”. In: *Inf. Process. Lett.* 130 (2018), pp. 1–6. DOI: 10.1016/J.IPL.2017.09.011.

[27] Sandra Loosemore et al. *The GNU C Library Reference Manual*. Free Software Foundation. 2024. URL: https://sourceware.org/glibc/manual/2.31/.

[28] Andrew Lumsdaine et al. “Challenges in Parallel Graph Processing”. In: *Parallel Process. Lett.* 17.1 (2007), pp. 5–20. DOI: 10.1142/S0129626407002843.

[29] Nikolai Maas, Lars Gottesbüren and Daniel Seemaier. “Parallel Unconstrained Local Search for Partitioning Irregular Graphs”. In: *2024 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 32–45. DOI: 10.1137/1.9781611977929.3.

[30] Tobias Maier, Peter Sanders and Roman Dementiev. “Concurrent Hash Tables: Fast and General(?)!” In: *ACM Trans. Parallel Comput.* 5.4 (2019), 16:1–16:32. DOI: 10.1145/3309206.

[31] Tobias Maier, Peter Sanders and Stefan Walzer. “Dynamic Space Efficient Hashing”. In: *Algorithmica* 81.8 (2019), pp. 3162–3185. DOI: 10.1007/S00453-019-00572-X.

[32] Henning Meyerhenke, Peter Sanders and Christian Schulz. “Partitioning (hierarchically clustered) complex networks via size-constrained graph clustering”. In: *J. Heuristics* 22.5 (2016), pp. 759–782. DOI: 10.1007/S10732-016-9315-8.

[33] Chuck Pheatt. “Intel® threading building blocks”. In: *J. Comput. Sci. Coll.* 23.4 (Apr. 2008), p. 298. ISSN: 1937-4771.

[34] Usha Nandini Raghavan, Réka Albert and Soundar Kumara. “Near linear time algorithm to detect community structures in large-scale networks”. In: *Phys. Rev. E* 76 (3 Sept. 2007), p. 036106. DOI: 10.1103/PhysRevE.76.036106.

[35] Ryan A. Rossi and Nesreen K. Ahmed. “The Network Data Repository with Interactive Graph Analytics and Visualization”. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. Ed. by Blai Bonet and Sven Koenig. AAAI Press, 2015, pp. 4292–4293. DOI: 10.1609/AAAI.V29I1.9277.

[36] Peter Sanders and Daniel Seemaier. “Distributed Deep Multilevel Graph Partitioning”. In: *Euro-Par 2023: Parallel Processing - 29th International Conference on Parallel and Distributed Computing, Limassol, Cyprus, August 28 - September 1, 2023, Proceedings*. Ed. by José Cano et al. Vol. 14100. Lecture Notes in Computer Science. Springer, 2023, pp. 443–457. DOI: 10.1007/978-3-031-39698-4_30.

[37] Julian Shun and Guy E. Blelloch. “Ligra: a lightweight graph processing framework for shared memory”. In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’13, Shenzhen, China, February 23-27, 2013*. Ed. by Alex Nicolau et al. ACM, 2013, pp. 135–146. DOI: 10.1145/2442516.2442530.

[38] Julian Shun, Laxman Dhulipala and Guy E. Blelloch. “Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+”. In: *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015*. Ed. by Ali Bilgin et al. IEEE, 2015, pp. 403–412. DOI: 10.1109/DCC.2015.8.

[39] Isabelle Stanton and Gabriel Kliot. “Streaming graph partitioning for large distributed graphs”. In: *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’12, Beijing, China, August 12-16, 2012*. Ed. by Qiang Yang, Deepak Agarwal and Jian Pei. ACM, 2012, pp. 1222–1230. DOI: 10.1145/2339530.2339722.

A Properties of the Graphs in the Benchmark Set

Table A.1: Number of nodes n, number of edges m, average degree $d(G)$, max degree $\Delta(G)$ and class of the graphs in the benchmark set.

Graph G	n	m	$d(G)$	$\Delta(G)$	Class
delaunay-n24	16 777 216	100 663 202	6.0	26	Artificial
kron-g500n19	524 288	43 561 574	83.1	80 674	Artificial
kron-g500n20	1 048 576	89 238 804	85.1	131 503	Artificial
kron-g500n21	2 097 152	182 081 864	86.8	213 904	Artificial
mycielskian19	393 215	903 194 710	2 296.9	196 607	Artificial
rgg-n26	67 108 864	1 149 107 290	17.1	45	Artificial
rgg-n263d	67 108 864	755 904 090	11.3	34	Artificial
rhg-n23d20	10 000 000	399 184 966	39.9	96 981	Artificial
rhg-n23d4	8 388 608	64 163 456	7.6	415 850	Artificial
rmat-n16m24	65 527	33 554 422	512.1	26 068	Artificial
rmat-n25m28	27 089 643	536 831 408	19.8	24 179	Artificial
cage15	5 154 859	94 044 692	18.2	46	Biology
kmer-A2a	170 372 459	359 883 478	2.1	40	Biology
kmer-P1a	138 896 082	296 930 692	2.1	40	Biology
kmer-V1r	214 004 392	465 409 664	2.2	8	Biology
kmer-V2a	53 500 237	114 152 252	2.1	39	Biology
bn-M87117515	891 589	97 339 212	109.2	4 546	Brain
bn-M87122310	924 284	188 741 772	204.2	8 135	Brain
bn-M87123142	846 535	107 650 654	127.2	8 356	Brain
bn-M87126525	975 930	292 218 600	299.4	8 009	Brain
bn-M87128519-1	861 636	338 735 702	393.1	7 397	Brain
dna1GB-9	3 233 125	50 570 172	15.6	342 348	Compression
english1GB-7	801 514	26 125 176	32.6	100 816	Compression
proteins1GB-9	14 537 567	148 617 134	10.2	660 097	Compression
proteins1GB-7	2 825 742	87 714 764	31.0	373 750	Compression
sources1GB-7	898 704	14 544 726	16.2	32 091	Compression
sources1GB-9	2 792 175	24 376 602	8.7	37 881	Compression

Table A.2: Number of nodes n, number of edges m, average degree $d(G)$, max degree $\Delta(G)$ and class of the graphs in the benchmark set.

Graph G	n	m	$d(G)$	$\Delta(G)$	Class
afshell10	1 508 065	51 164 260	33.9	34	Finite element
audikw	943 695	76 708 152	81.3	344	Finite element
boneS10	914 898	54 553 524	59.6	80	Finite element
Bump2911	2 911 419	124 818 480	42.9	194	Finite element
channelb050	4 802 000	85 362 744	17.8	18	Finite element
CubeCoup-dt6	2 164 760	125 041 384	57.8	67	Finite element
dielFilterV3	1 102 824	88 203 196	80.0	269	Finite element
Flan1565	1 564 794	115 841 250	74.0	80	Finite element
Geo1438	1 437 960	61 718 730	42.9	56	Finite element
Hook1498	1 498 023	59 419 422	39.7	92	Finite element
HV15R	2 017 169	324 715 138	161.0	492	Finite element
ldoor	952 203	45 570 272	47.9	76	Finite element
LongCoup-dt6	1 470 152	85 618 840	58.2	755	Finite element
MLGeer	1 504 002	109 375 970	72.7	73	Finite element
Queen4147	4 147 110	325 352 174	78.5	80	Finite element
Serena	1 391 349	63 140 352	45.4	248	Finite element
nlpkkt200	16 240 000	431 985 632	26.6	27	Optimization
nlpkkt240	27 993 600	746 478 752	26.7	27	Optimization
asia-osm	11 950 757	25 423 206	2.1	9	Road
europe-osm	50 912 018	108 109 320	2.1	13	Road
circuit5M	5 558 326	53 967 852	9.7	1 290 500	Semiconductor
nv2	1 453 908	51 274 454	35.3	83	Semiconductor
stokes	11 449 533	515 962 626	45.1	1 728	Semiconductor
vas-stokes2M	2 146 677	96 703 558	45.0	1 307	Semiconductor
vas-stokes4M	4 382 246	195 417 042	44.6	1 139	Semiconductor
flickr-und	1 715 255	31 110 082	18.1	27 236	Social
friendster	65 608 366	3 612 134 270	55.1	5 214	Social
hollywood	2 180 759	228 985 632	105.0	13 107	Social
imdb2021	2 996 317	10 738 944	3.6	833	Social
livejournal	4 036 538	69 362 378	17.2	14 815	Social
orkut	3 072 627	234 370 166	76.3	33 313	Social
sinaweibo	58 655 849	522 642 066	8.9	278 489	Social
twitter2010	41 652 230	2 405 026 092	57.7	2 997 487	Social

Table A.3: Number of nodes n, number of edges m, average degree $d(G)$, max degree $\Delta(G)$ and class of the graphs in the benchmark set.

Graph G	n	m	$d(G)$	$\Delta(G)$	Class
arabic2005	22 744 080	1 107 806 146	48.7	575 628	Web
indochina2004	7 414 866	301 969 638	40.7	256 425	Web
it2004	41 291 594	2 054 949 894	49.8	1 326 744	Web
mavi2015	22 914 771	49 124 430	2.1	10 327 637	Web
sk2005	50 636 154	3 620 126 660	71.5	8 563 816	Web
uk2005	39 459 925	1 566 054 250	39.7	1 776 858	Web
webbase2001	118 142 155	1 709 619 522	14.5	816 127	Web
dewiki2013	1 532 354	66 186 058	43.2	118 246	Wiki
enwiki2022	6 492 490	289 177 312	44.5	231 674	Wiki
eswiki2013	972 933	42 369 862	43.5	145 310	Wiki
frwiki2013	1 352 053	62 074 604	45.9	148 758	Wiki
itwiki2013	1 016 867	46 859 288	46.1	91 517	Wiki