Karlsruher Institut für Technologie

Bachelor thesis

Engineering Asynchronous Label Propagation for Multilevel Graph Partitioning

Samuel Gil

Date: 1. Februar 2024

Supervisors: Prof. Dr. Peter Sanders
M.Sc. Daniel Seemaier
M.Sc. Tim Niklas Uhl

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

Abstract

Partitioning large graphs efficiently is becoming more important over the years, as the networks to be processed keep growing. Multilevel graph partitioning (MGP) is the most successful graph partitioning paradigm for processing large graphs. For extremely large graphs, the memory available in a single machine might not be sufficient for a MGP partitioner. This is where distributed MGP schemes, that make use of multiple compute nodes, come into play. In this work we explore the impact of asynchronous label exchanges during the coarsening phase. Our first algorithm, called `ThreadedLP`, uses threads to overlap communication and computation, while the second, called `MQLP`, uses a message queue for enabling interleaving of communication and computation. We performed experiments for both algorithms, using real-world and randomly generated graphs. `ThreadedLP` performs similarly to `dKaMinPar` and improves on the strong scalability of the twitter-2010 graph. `MQLP` performs significantly worse than `dKaMinPar`, and produces worse results due to more imbalanced cluster weights.

Effiziente Graph Partitionierer für große Graphen werden immer relevanter, da die betrachteten Netzwerke immer weiter wachsen. Multilevel Graph Partitionierung (MGP) hat sich als das erfolgreichste Paradigma zur Partitionierung von großen Graphen herausgestellt. Für extrem riesige Graphen kann es passieren, dass der verfügbare Hauptspeicher einer einzelnen Maschine nicht ausreicht, um eine Partitionierung durch einen MGP Partitionierer zu berechnen. In diesem Fall werden verteilte MGP Ansätze benötigt, die Partitioinierungen mit Hilfe mehrerer Rechner bestimmen. In dieser Arbeit untersuchen wir die Auswirkungen von asynchronem Label Austausch während der Coarsening Phase. Der erste Algorithmus, genannt `ThreadedLP`, verwendet Threads um Kommunikation und Berechnung zu überlappen, während der zweite Algorithmus, namens `MQLP` eine Message Queue benutzt um die Verschränkung von Kommunikation und Berechnung zu ermöglichen. Wir haben die beiden Algorithmen auf Echtwelt- und zufällig generierten Graphen getestet. `ThreadedLP` weist ähnliche Laufzeiten und Qualitäten wie `dKaMinPar` auf und verbessert die Skalierbarkeit des Algorithmus für den twitter-2010 Graphen. Für den `MQLP` Algorithmus sehen wir erheblich schlechtere Laufzeiten und schlechtere Cuts durch imbalanciertere Cluster Gewichte.

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.

Ort, den Datum

Contents

1 Introduction

Graphs are a powerful abstraction tool for modelling relations between objects. They are widely used in different fields, for representing social networks, road networks, web graphs and more. As networks, and therefore graphs, grow in size, graph partitioning becomes critical to reduce complexity, improve performance or enable parallel processing, when working with these graphs. However, the graph partitioning problem is NP-hard. Therefore we generally only expect to design polynomial time algorithms for approximations of the problem. The multilevel paradigm describes a group of algorithms attempting to compute such approximations, while providing a good trade-off between quality and compute time.

For extremely large graphs, the memory available in a single machine may not be sufficient for the computation of a graph partition using the multilevel paradigm. Because of this, distributed memory graph partitioners become essential in order to handle these cases.
In this thesis we present two algorithms that are based on the `dKaMinPar` [21] graph partitioner, which is a distributed deep-multilevel graph partitioner (DDMGP). In particular, we have developed two approaches to size-constrained label propagation clustering algorithms. We will refer to the first one as `ThreadedLP`, since it uses an extra thread to asynchronously perform the label communication while proceeding with the label computation. This is done to save time by using available resources not needed by the other thread. We will call the second algorithm Message Queue Label Propagation (`MQLP`). `MQLP` performs the label communication by using a message queue, that buffers messages until a threshold is surpassed. At which point the messages are sent to their targets. By manipulating the threshold, we are able to perform an arbitrary amount of computation before communicating. Thus we aim to improve the performance of the clustering algorithm, by converging to a small enough clustering more quickly or improve the quality of the clustering.

1.1 Structure of Thesis

We organize this thesis by first presenting some general definitions and concepts required throughout the thesis, in Chapter 2. In Chapter 3, we provide an overview of related works and introduce the multilevel graph partitioning paradigm, as well as a rough description of `ParHIP` and `dKaMinPar`, which are two algorithms closely related to the ones newly presented here. We will also describe the concept and motivation of a message queue in Section 3.7. Then we present and evaluate our first algorithm `ThreadedLP`, which is a version performing the MPI label communication asynchronously using an extra thread, in

Chapter 4 and 5. In Chapter 6 and 7 we present and evaluate our second algorithm `MQLP`, which makes use of a message queue [22]. Finally we conclude in Chapter 8.

2 Fundamentals

We base our algorithms on the `dKaMinPar` [21] algorithm. Which is the reason why we will derive our definitions from the terminology used by `dKaMinPar` [21]. We will begin by providing some definitions and explaining fundamental concepts required to understand this topic.

2.1 General Definitions

Let $G = (V, E, c, \omega)$ be an undirected, simple graph, as seen in Figure 2.1 with the vertex set $V = \{0, \ldots, n-1\}$, edge set $E \subseteq \{\{u, v\} \mid u, v \in V\}$, vertex weights $c : V \to \mathbb{N}_{\geq 0}$, edge weights $\omega : E \to \mathbb{N}_{\geq 0}$, $n := |V|$ and $m := |E|$. We also define the weights of sets, induced by the weight functions, as follows: $c(V') := \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{e \in E'} \omega(e)$. $N(v) := \{u \mid \{v, u\} \in E\}$ is the set of neighbors of v. An *isolated vertex* v is a vertex, that has no neighbors, i.e., $N(v) = \emptyset$. For an arbitrary subset $V' \subseteq V$, we define the naturally induced subgraph $G[V'] := G' = (V', E')$ with $E' = \{e = \{u, v\} \mid e \in E$ and $u, v \in V'\}$. The goal of the DDMGP is to compute a k-*block* partition $\Pi := \{V_1, \ldots, V_k\}$ of the input graph, i.e., $V_1 \cup \ldots \cup V_k = V$, $V_i \cap V_j = \emptyset$ for $i \neq j$ and $|\Pi| = k$. Figure 2.1 shows a 3-block partition of an example graph. The *balance constraint* is defined as $c(V_i) \leq L_{\max} := \max\{(1+\varepsilon)\frac{c(V)}{k}, \frac{c(V)}{k} + \max_v c(v)\}$ for each $i \in \{1, \ldots, k\}$ for an imbalance parameter ε. A vertex $u \in V_i$ that has a neighbor in V_j, $i \neq j$, is called a *boundary vertex*. A *clustering* $\mathcal{C} := \{C_1, \ldots, C_\ell\}$ is a partition of V, with an arbitrary number of blocks. The algorithms we use to compute such a clustering are *label propagation* algorithms. Label propagation algorithms use an *objective function*, such as *cut minimization*, in order to compute the labels of nodes. The goal of cut minimization is to minimze the total weight of all cut edges $e \in E_{ij}$, i.e, minimize $cut(\Pi) := \sum_{i<j} \omega(E_{ij})$, where $E_{ij} := \{\{u, v\} \mid u \in V_i$ and $v \in V_j\}$. We also extend this definition to the more specific case, that the partition is a clustering. We refer to the inter-cluster edge weight of a clustering as the cut weight of a clustering, i.e., $cut(\mathcal{C}) := \sum_{i<j} \omega(E_{ij})$, where $E_{ij} := \{\{u, v\} \mid u \in C_i$ and $v \in C_j\}$. A *label* l_i is a value assigned to a vertex v_i, that represents a cluster. We will use $l_i := label(v_i) = label(C_j) =: c_j$ to refer to the label of a vertex $v_i \in C_j$, with $i \in \{0, \ldots, n-1\}$ and $j \in \{1, \ldots, \ell\}$. The c_i-*cluster neighborhood* of a node v is defined as $N_i(v) := \{u \in N(v) \mid label(u) = c_i\}$. Since we present distributed algorithms, we also define *owned* and *unowned* clusters. A cluster is always represented by a vertex $repr(C_j) := v_x = c_j$, with $x \in \{0, \ldots, n-1\}$. An *owned cluster* is a cluster for which

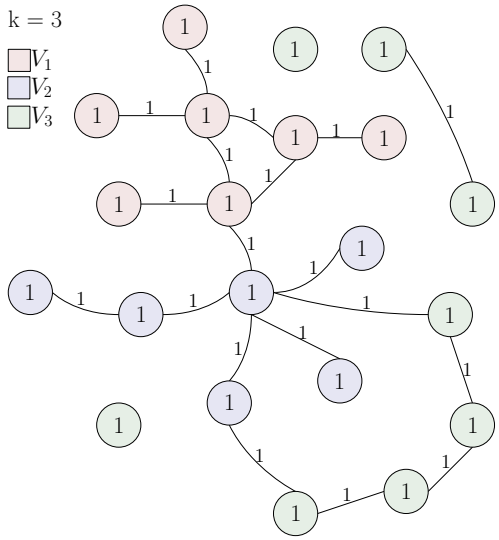

Figure 2.1: Graph partition with $k = 3$ of an undirected, simple graph. V_1, V_2, V_3 are the sets of vertices the graph is partitioned into. These sets of vertices share a common color if they belong to the same set. The weight of a vertex is depicted inside its vertex. The weight of an edge is depicted next to its edge. This particular partition has a cut weight of $cut(V_1, V_2, V_3) = 3$, as can be seen more easily in **Fig.** 2.2.

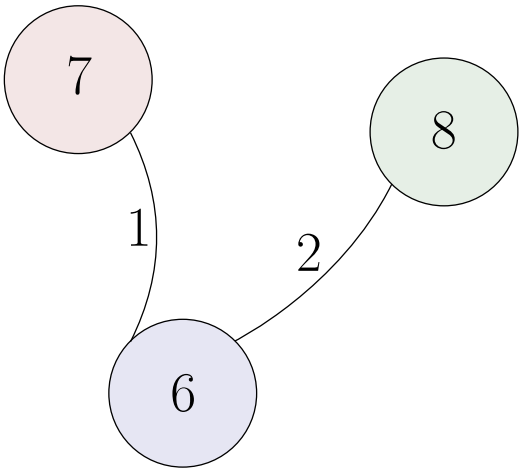

Figure 2.2: Graph contraction of **Fig.** 2.1. The contracted graph has a vertex for each vertex-set of the input partition. The weight of a vertex in the graph contraction equals the sum of the weights of vertices from that set. The weight of an edge equals the sum of the weights of edges between the vertices inside the corresponding vertex-sets.

the label $c_j = l_i$ corresponds to a vertex v_x that is owned. Similarly, an *unowned cluster* corresponds to an unowned vertex. A vertex is owned by a *processing element* (PE), if it is in the subset $V' \subset V$ that was assigned to the PE. A *cluster contraction* is a graph, that contracted each cluster into a vertex representing that cluster, or the operation that achieves this. *Contraction* in general, refers to the operation or result of an operation, that replaces multiple objects with a single object, e.g., during *vertex contraction*, two vertices are combined into one. The resulting vertex of this vertex contraction is referred to as the vertex contraction of the original vertices. Figure 2.2 shows the contraction of the graph partition seen in Figure 2.1.

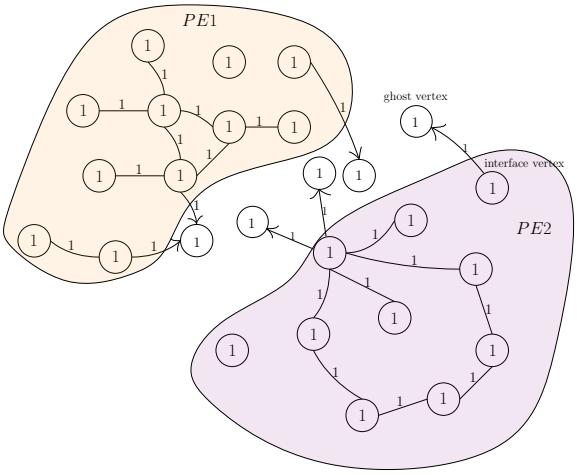

Figure 2.3: Graph representation of **Fig.** 2.1 split and distributed to two PEs. If a PE owns both directed edges corresponding to an undirected edge, the edge is drawn as an undirected edge to reduce cluttering. The ghost vertices are drawn outside the PE boundaries to differentiate between ghost vertices and local vertices.

2.2 Machine Model and Input Format

The experiments performed in this thesis make use of $\mathcal{N}$ *compute nodes* with P PEs each. The input graph is given with a (usually balanced) 1D vertex partition. Each PE is given a subgraph $G[V_i]$ of the input graph with consecutive vertices V_i. An undirected edge $\{u, v\}$ is represented by two directed edges (u, v), (v, u), which are stored on the PEs owning the respective tail vertices. Vertices adjacent to vertices owned by other PEs are called *interface vertices* and are replicated as *ghost vertices* (i.e., without outgoing edges) on those PEs. [21] Figure 2.3 shows such a PE partition. Because we construct a ghost vertex for each tail vertex of an edge, that is not owned, the maximum number of ghost vertices is limited by the number of edges of the graph representation. Note that, since we use a simple undirected graph as the input, the total number of edges of the input graph is limited by $\frac{n(n-1)}{2}$, which means that the total number of edges of the graph representation is limited by $n(n-1)$.
The vertices are initially assigned to their own clusters. Each PE holds local representations of the clusters of owned and ghost vertices. A local cluster representation is referred to as a *local cluster*. From the point of view of any PE, any other PE is called a *remote PE*. A cluster representation on a remote PE is called a *remote cluster*.

2.3 Collective MPI Operations

We will present some of the MPI communication operations used by us. The tables illustrating the send and receive buffers of each PE show the contents to be sent by a row to a

column or received on a row by a column.

Allreduce

Table 2.1: State on PEs before allreduce

Send Buffers			
PE	A	B	C
A	a		
B	b		
C	c		

Table 2.2: State on PEs after receiving the values for reduction

Receive Buffers			
PE	A	B	C
A	a	b	c
B			
C			

Table 2.3: Result of additive allreduce operation

Receive Buffers			
PE	A	B	C
A	$a+b+c$		
B	$a+b+c$		
C	$a+b+c$		

The allreduce function takes an operation and a value as an argument. Table 2.1 shows the initial state of the send buffers. The values are collected on the PE with rank 0, as seen in Table 2.2, which computes the result. The result is computed by reducing the set of values using the provided operation. It then returns the value to all PEs. The state of the receive buffers after the allreduce operation is shown in Table 2.3.

Alltoallv

Table 2.4: State on PEs before alltoallv

Send Buffers			
PE	A	B	C
A	$[a0, a1]$	$[a2, a3]$	$[a4, a5, a6]$
B	$[b0, b1, b2]$	$[b3]$	$[b4, b5]$
C	$[c0]$	$[]$	$[c1, c2]$

Table 2.5: State of the receive buffers of the PEs after alltoallv

Receive Buffers			
PE	A	B	C
A	$[a0, a1]$	$[b0, b1, b2]$	$[c0]$
B	$[a2, a3]$	$[b3]$	$[]$
C	$[a4, a5, a6]$	$[b4, b5]$	$[c1, c2]$

During an alltoallv operation, buffers of elements are exchanged by the processes. Each process specifies how many elements it sends and and receives from each process. The buffers are then sent to their target process and put into a receive buffer. Table 2.4 shows the send buffers before an alltoallv operation and Table 2.5 shows the corresponding state of the receive buffers afterwards.

Table 2.6: Grid layout for sparse alltoallv with 4 PEs

A	B
C	D

Table 2.7: Send buffer layout (as it would be for a generic alltoallv)

Send Buffers				
PE	A	B	C	D
A	$[a0, a1]$	$[a2, a3]$	$[a4, a5, a6]$	$[a7]$
B	$[b0, b1, b2]$	$[b3]$	$[b4, b5]$	$[b6]$
C	$[c0]$	$[]$	$[c1, c2]$	$[c3]$
D	$[d0]$	$[]$	$[d1, d2]$	$[d3]$

Table 2.8: State of the receive buffers of the PEs after first alltoallv

Receive Buffers				
PE	A	B	C	D
A	$[a0, a1, a2, a3]$	$[b0, b1, b2, b3]$	$[c0]$	$[d0]$
B	$[]$	$[]$	$[]$	$[]$
C	$[a4, a5, a6, a7]$	$[b4, b5, b6]$	$[c1, c2, c3]$	$[d1, d2, d3]$
D	$[]$	$[]$	$[]$	$[]$

Table 2.9: State of the receive buffers of the PEs after second alltoallv

Receive Buffers				
PE	A	B	C	D
A	$[a0, a1]$	$[b0, b1, b2]$	$[c0]$	$[d0]$
B	$[a2, a3]$	$[b3]$	$[]$	$[]$
C	$[a4, a5, a6]$	$[b4, b5]$	$[c1, c2]$	$[d1, d2]$
D	$[a7]$	$[b6]$	$[c3]$	$[d3]$

Sparse Alltoallv

Sparse alltoallv is a modification to the generic alltoallv operation provided by MPI. Sparse alltoallv performs two alltoallv operations to replace the generic alltoallv. For it, the PEs are arranged in a grid, as shown in Table 2.6. It reroutes the messages by first sending all buffers to the rows of their target using an alltoallv and then performing another alltoallv operation to send the buffers to their actual target. In our example all messages to PE A and B are routed to PE A first, and all messages to PE C and D are rerouted to PE C. This way, each PE only has $\mathcal{O}(\sqrt{P})$ communication partners instead of $\mathcal{O}(P)$, which reduces the total number of messages sent from $\mathcal{O}(P^2)$ to $\mathcal{O}(P)$. Table 2.7 shows the messages to be sent to each PE. Table 2.8 and Table 2.9 depict the receive buffers after the first and second alltoallv respectively. Figure 2.4 illustrates how the messages are pathed through the network.

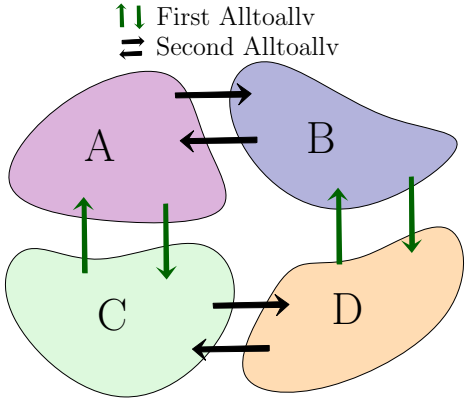

Figure 2.4: The sparse alltoallv communication paths. The PEs are labelled from A-D and positioned according to the grid layout. Direct diagonal communication paths are removed and each PE only has $\sqrt{P}$ communication partners per alltoallv operation.

3 Related Work

Graph partitioning is a highly researched problem. Over the years, multilevel graph partitioning (MGP) has proven to be the most successful approach with significant increases to performance and quality compared to single level partitioners. Distributed memory approaches allow for much larger graphs to be processed, but often lead to much lower quality. For further reading on the graph partitioning problem, the reader is referred to overview papers [2][3][4].

3.1 Multilevel Graph Partitioning

Multilevel graph partitioning is an approach, that consists of a coarsening, initial partitioning and uncoarsening phase. During coarsening multiple levels of coarser graphs are constructed. The coarsest graph is partitioned during the initital partitioning phase. This initial partition is then refined by subsequently projecting the partition onto the next finer level and improving it. We will provide an overview of the MGP paradigm while putting some partitioners into the context.

3.1.1 Coarsening

During coarsening, a hierarchy of successively coarser graphs is created by repeatedly computing a clustering $\mathcal{C} := \{C_1, \ldots, C_\ell\}$ on the input graph and then contracting the clustering by substituting each cluster by a vertex representing that cluster in the coarser graph G'. I.e., the coarser graph's set of vertices is defined as $V' := \{c = repr(C_i) \mid C_i \in \mathcal{C}\}$. The coarser graph's edges are created by contracting the inter-cluster edges, so that the edge set of the coarser graph is given by $E' := \{e' = \{c_i, c_j\} \mid c_i, c_j \in V'$ and $\exists e = \{u, v\} \in E : u \in C_i$ and $v \in C_j\}$. In order to accurately represent the finer graph, the weights of the coarser graph's edges and vertices are calculated. By doing this, it is possible to compute more balanced partitions. The weight of the new vertices is given by the total weight of a cluster's vertices, i.e., $c(c_i) = c(C_i)$, with $i \in \{1, \ldots, \ell\}$. And the weight of a new edge $\omega'(e')$ is given by the total weight of the edges contracted into the edge e'. This way, a hierarchy of coarse graphs is generated until the coarsest graph is small enough. A coarse graph is considered small enough, if its number of vertices is less or equal to Ck, for a *contraction limit* C and number of blocks k. There are also other

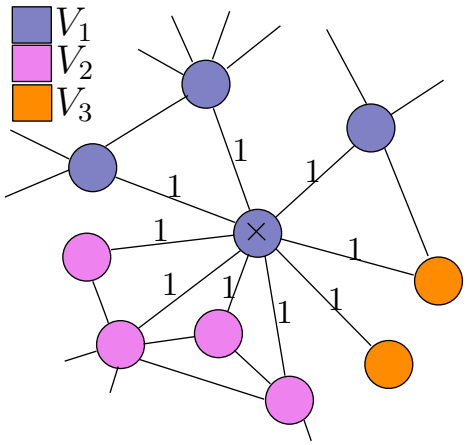

Figure 3.1: The marked vertex in the middle is the currently processed vertex. The gain for a move to V_1 is 0, the gain for a move to V_2 is 1 and the gain for a move to V_3 is -1. Since we maximize the gain, this means that the vertex will be put into V_2.

coarsening strategies [6][8][20] besides contraction based approaches, which we will not be covering here.

Matching

One widely used clustering method is matching based clustering, which selects certain edges as part of the matching and subsequently contracts the graph by replacing the matching edges with a vertex. This selection was usually done by iterating over the nodes in a random order and matching the heaviest edge in the neighborhood [28]. However, it has been found, that there are actually more advanced rating functions, that lead to better results [11][20]. `PT-Scotch` [5] is a distributed parallel matching and bisection based algorithm.
`ParMETIS` is a fast and high quality partitioner for irregular graphs, that provides multiple different matching-based coarsening strategies, as well as multiple refinement and initial partitioning algorithms for the MGP scheme [14].
Matching based schemes work well on meshes but struggle to compute balanced partitions on complex graphs and star-like structures. This is due to the fact that matchings usually leave the neighbors of high-degree vertices unmatched, which means that they do not get contracted for that level [4].

Label Propagation

Label propagation [18][19][21] is another method commonly used for graph partitioning, due to its simplicity and efficiency. It improves upon matching based approaches in terms of quality and performance when processing complex graphs while producing similar results

for meshes. It is initialized by assigning a label to each vertex. Label propagation then iterates over the vertices and computes a new label assignment for the current vertex, by looking at the labels in its neighborhood and choosing the label with the highest gain. Figure 3.1 shows a vertex that is currently assigned to the vertex-set V_1 and will be moved to V_2. In our case the new label is chosen by maximizing the cut weight between the current vertex v and vertices with the same label c_i, i.e., the new label is the c_i, for which $\omega(N_i(v)) = \max\{\omega(N_i(v))\}$ with $N_i(v) = \{u \mid u \in N(v) \text{ and } label(u) = c_i\}$. By maximizing the intra-cluster edge weight, we minimize the weight of the inter-cluster cut. If there are multiple labels that meet this condition, the label is chosen randomly. Label propagation terminates when the label assignments converge, i.e., there are no more label changes to be made. Since this procedure does not always converge, we limit the number of iterations to $maxIterations \in \{3, 5\}$. Examples for partitioners using label propagation are `ParHIP` [18], `dKaMinPar` [21] and `XtraPuLP` [23], which implements a multiple constraint and objective version. As opposed to the other two, it is a single level partitioner, which reports a higher performance than `ParMETIS` [14] and `ParHIP` [18], but also a much higher cut (up to 5x).

Active Set Strategy

Meyerhenke and Staudt [24] have used the active set strategy in their parallel label propagation algorithm for community detection in large graphs. They report extremely fast running times and reasonable quality. The active set strategy starts by declaring every vertex as active. The Label propagation using the active set strategy, then iterates over each active vertex. When a node has been processed, it is deactivated. If the node has changed its label, its neighbors are activated. This way, vertices in a neighborhood that did not change, do not get processed again. This is useful, since these vertices do not need to be rechecked, since the new label would most likely still be the same. The only way the label would change is if we were previously unable to move to a cluster, because it was too heavy and chose another cluster instead. If the cluster, that was our first choice lightened up sufficiently in the meantime, we would then add the vertex to this cluster, even though no label was changed in the neighborhood. Using the active set strategy therefore reduces unnecessary computation and optimizes the vertex traversal.

3.1.2 Initial Partitioning

After coarsening, an initial partition with k blocks is computed on the coarsest graph. The two main strategies for computing an initial partition are recursive bipartitioning and direct k-way partitioning.
For bipartitioning, the graph is first split into two parts. The resulting partition is recursively processed by dividing each subgraph into two parts again. This is repeated until the initial graph is split into k blocks of vertices. Some partitioners strengthen the requirements of

this procedure by not only computing two parts, but also ensuring that the two parts are of equal size. This is referred to as bisection.
For direct k-way partitioning, the k blocks are computed directly without bipartitioning the graph recursively [1].

3.1.3 Uncoarsening

The computed partition is then recursively projected onto the next finer graph and refined using local search algorithms [4], in order to get a more optimal partitioning in regards to the objective function. If necessary, the graph partition is balanced to meet the requirements of the balance constraint, before resuming the computation of the next level.

3.2 Distributed Multilevel Graph Partitioning

Compared to shared-memory partitioners, distributed-memory partitioners generally display slower running times and worse quality [9]. Nevertheless distributed graph partitioners are still useful for handling large instances, some of which do not fit into the memory of a single machine. At the start the input graph is split into subgraphs, which are distributed to the PEs. The PEs then each perform coarsening, while communicating changes to their neighboring PEs at designated times. Afterwards, the subgraphs are recombined in order to compute an initial partition. The resulting partition is then projected onto the finer levels while refining and balancing whenever necessary.

3.3 Distributed Deep-Multilevel Graph Partitioning

Deep MGP is a version of MGP, that continues coarsening until the coarsest graph has only $n' \leq 2C$ vertices left. In contrast to the generic MGP break condition, deep MGP's break condition is independent of k, and therefore an initial partition can be efficiently computed even for large k. Deep MGP does this by uncoarsening while recursively computing bipartitions. When the finest graph, for which $n' \leq kC$ is still given, is reached, the computed bipartition corresponds to the initial partitioning of the coarsest graph of MGP. [10] [21]

3.4 Streaming Graph Partitioning

Streaming algorithms partition the graph while reading the input graph. They read vertices and their neighborhood and assign them to their blocks solely based on that knowledge. This means that the partitioner only requires memory proportional to $\mathcal{O}(n)$, since the edges

do not need to be saved. Another consequence is, that they do not know the context of the whole graph while computing sections of the graph. This leads to very fast partitionings, which are even faster than MGP, but with much worse quality. [4]

3.5 ParHIP

As mentioned earlier, `ParHIP` [18] is a parallel label propagation MGP algorithm. In particular, it uses size-constrained label propagation, which was first proposed by Meyerhenke et al. [17]. Size-constrained label propagation differs from generic label propagation in the fact, that an additional constraint is added for the cluster weights. During the label propagation, an upper bound $W := \varepsilon \frac{c(V)}{k'}$ with $k' := \min\{k, \frac{|V|}{C}\}$ is defined as the maximum vertex weight of a cluster, i.e., $c(C_i) \leq W$ with $i \in \{1, \ldots, \ell\}$ [21]. By adhering to this condition, we get a more balanced clustering, which improves the quality of the coarsening [17]. In practice, the size-constraint is relaxed and only enforced locally during coarsening. The size-constraint of the blocks is only really enforced later during refinement.
`ParHIP` improves the performance of the size-constrained label propagation, by overlapping the communication of computed labels with the computation of the next labels. It does so by using non-blocking send and receive operations in combination with a specific version of OpenMPI, that provides an implementation of progress threads. This allows the Isend and Irecv operations to be progressed asynchronously. When comparing the results to `ParMETIS`, `ParHIP` reports improvements to both the cut and the performance, when computing complex networks [18].

3.6 dKaMinPar

The `dKaMinPar` [21] algorithm is a DDMGP. It uses size-constrained label propagation during coarsening and refinement. During coarsening the maximum cluster weight used by `dKaMinPar` is defined as $W := \varepsilon \frac{c(V)}{k'}$ with $k' := \min\{k, \frac{|V|}{C}\}$. Recall, that k is the number of blocks and C is the contraction limit. We will focus on the clustering phase, since both our algorithms are applied there. However, we will start by providing some general information to how `dKaMinPar` functions.
When `dKaMinPar` is run on P PEs, the input graph G is split into P subgraphs $G_i[V_i]$ with $i \in \{1, \ldots, P\}$ and each subgraph $G_i[V_i]$ is assigned to the corresponding PE i. In addition to the vertices owned by a PE, an additional ghost vertex is stored for each adjacent interface vertex. Iterating over vertices in increasing degree order, leads to a better result quality and performance [18]. To make use of this fact, the vertices are stored in exponentially spaced degree buckets and the graph is rearranged accordingly [21]. During Coarsening, `dKaMinPar` performs three iterations. Each iteration is divided into $\max\{8, \frac{128}{P}\}$ batches [21], that are processed sequentially. By default, `dKaMinPar` randomizes the order the vertices are handled in. However, in order to compare `dKaMinPar` to `MQLP`, we

use an implementation in which the vertices are iterated sequentially without randomization.
For each batch, the new label assignments for owned vertices are computed, by maximizing the gain of a vertex move. The gain of a move is defined as the corresponding change to the intra-cluster cut. After the label computation, the new weights of the clusters are computed by exchanging the total weight added to clusters during the batch. If the clusters are overweight, each PE reverts an amount of weight proportional to its part of the weight exceeding the limit. Then the new labels of interface vertices are communicated to neighboring PEs and applied there. Finally, the procedure ends by clustering isolated nodes of the current batch. This is done by repeatedly adding isolated nodes to one cluster, until the cluster is full and then repeating that step with a new cluster.

3.7 Message Queue

Asynchronously communicating introduces a new problem regarding the performance of MPI communication. Sending many small messages leads to performance deterioration due to cumulative communication startup overheads. A solution for this is message aggregation in order to reduce the number of messages sent. This also makes better use of the available network bandwith, while introducing additional latencies to send operations. [26]
Naively implementing asynchronous message aggregation is error prone and may even lead to worse performance. To combat this, a message queue is used, that handles the MPI communication and provides functions, that allows us to simplify the problem. [27]
Message queues encapsulate and abstract from the MPI communication procedures and provide easy to use functions communicating in a distributed memory setting. Posting messages to the message queue stores them in a local buffer, until a threshold is reached before sending them as one aggregated message. This reduces startup overheads, which can pose a problem to scalability when sending many small messages. The received messages can be accessed by polling the message queue, to check for and read any received messages. Additionally rerouting procedures defined by the message queue, or its user can further reduce communication startup overheads.
The message queue used by us is implemented by Uhl [22][27] and allows for messages of arbitrary size to be posted to the message queue. Self defined aggregation and deaggregation functions enable the user to store any message in an arbitrary way. By setting the thresholds at which the buffers are flushed, the user can fine tune the granularity of the communication. Usually only the global threshold is set. This way, all local buffers are flushed when the combined size surpasses the threshold. By doing this, it is ensured, that even buffers with very few entries are sent and not starved for a long time. The message queue implementation also enables the user to define a cleanup procedure, which is performed before sending the contents of a buffer. This can be useful when new messages are added to the buffer, that invalidate previous messages.
For further reading on the message queue concept, we refer the reader to the following pa-

per [22] and additional papers cited by the unpublished manuscript by Uhl [27]: [25][26].

3.8 Applications

Recently Merkel et al. [16] have tried to apply graph partitioning as a preprocessing stage to graph neural networks (GNNs). In their paper, they state distributed GNN training as a promising solution for the memory and processing power requirements for computing large graphs, which often times are not available on single machines. They report improvements to performance due to the fact, that the preprocessing cost can be amortized by faster GNN training times. Additionally computing a graph partition reduces memory overheads of GNN training.

4 Label Propagation Clustering Using Threading

This is the first algorithm we present, called `ThreadedLP`. We will first describe the general idea in Section 4.1 and then get into more detail in Section 4.2.

4.1 Overview

Due to the outstanding results, that `ParHIP` [18] could record, we try an approach, that overlaps the label communication with the computation of the next batch. As mentioned earlier, `ParHIP` has a similar approach that makes use of OpenMPI progress threads to speed up the non blocking MPI calls. Since OpenMPI discontinued the support of progress threads, we tried using Intel's progress thread implementation. Due to a common issue with the Intel progress thread feature, which we encountered, we were unable to reproduce the exact same strategy, that is used by `ParHIP`. Instead, we manually spawn a thread for concurrency.
This algorithm is based on the original synchronous LP clustering of `dKaMinPar` and makes use of asynchonicity by splitting the procedure into communication and computation parts, which are executed by two asynchronous threads.
However this algorithm does not execute fully asynchronously. Instead one computation and one communication block is performed simultaneously per step. In particular, the first computation and and the last communication blocks are performed separately, whereas for the rest, the communication block corresponding to the computation block performed in the previous step is executed simultaneously to the next computation step. We do this by spawning an extra thread, so that the computation and communication parts each have an own thread to make progress.
By interleaving the communication of labels and and the computation of new labels in this way, we aim to make use of potential idle resources and reduce the overhead created by the need to wait for certain resources to finish a request. For example we intend to make progress for the label computation, while waiting for MPI messages to be routed to their targets.

4.2 Implementation

A pseudocode for ThreadedLP is shown in Algorithm 1. ThreadedLP has a structure similar to the label propagation clustering algorithm defined in dKaMinPar. The main difference lies in the utilization of an additional thread in order to allow simultaneous execution of label computation and label communication. During label propagation, there are essentially two phases. One is the actual label propagation and the other is the weight handling for the size-constraint. Although we only asynchronize the first phase, we cannot directly use the weight handling procedure used by dKaMinPar. The reason for this is that we compute new labels without having completed the process for the first batch of nodes. By buffering some data from the previous computation, we are able to adapt the weight handling procedure to ThreadedLP. We will start with the description of the label propagation phase and will afterwards explain how we manage the size-constraint.

Label Propagation

We perform $maxIterations$ iterations and split the nodes into $\max\{8, \frac{128}{P}\}$ chunks, as seen in Algorithm 1, ll. 2-3. Just like dKaMinPar-Fast, we choose $maxIterations = 3$, since this has proven to be a sufficient number of iterations for a fast algorithm, with a fairly good quality. This is due to the fact that label propagation based clustering contraction aggressively contracts the graph, e.g., the first contraction already shrinks the graph by orders of magnitude [18]. In ll. 4-6 the very first step is performed, during which we compute the first chunk's labels and synchronize and enforce the modified cluster's weights. Unlike dKaMinPar, we do not fully process the chunks before continuing with the next chunk. Instead we make use of an extra thread and interleave the computation of the next chunk with the current chunk's computation. We will call a step, that does this an *inner* step. In ll. 14-18 such an inner step is processed.
This procedure therefore leaves us with a separate computation step in the beginning and a separate communication step at the end, i.e., for the first chunk we do not have any communication and when communicating the last chunk's labels, we do not have any computation to do. To decrease the overhead caused by these isolated parts, we also overlap the first chunk's computation with the communication of the previous iteration's last chunk, for later iterations. In ll. 8-12, we perform the first step of the second or third iteration. We treat this as an inner step, by keeping the last chunk as the previous chunk. By doing so, we reduce the maximum number of isolated parts to exactly two. However we cannot overlap the first computation with any communication and the last communication with any computation. As a result, we only need to perform one extra step when compared to dKaMinPar. We should mention, that the total number of computations and communications stays the same. The main difference lies in the fact, that we delay the communication for the previously computed chunk minimally and start working on the unsynchronized

Algorithm 1: ThreadedLP(G, P): Label Propagation on P PEs

```
Input: G = (V, E), W
Output: clustering C := {C_1, ..., C_ℓ} of G
1  numMoves[] := Array(numChunks)
2  for iteration < maxIterations do
3      B_1, ..., B_c := SplitIntoChunks(V, numChunks)
4      if iteration is first iteration then
5          numMoves[0] = ComputeLabels(B_1)
6          ControlClusterWeights(B_1)
7      else
8          T := Thread(numMoves[0] = ComputeLabels(B_1))
9          totalMoves += ExchangeLabels(B_c, numMoves[c − 1])
10         JoinThread(T)
11         ControlClusterWeights(B_1)
12         HandleLabels(B_c)
13     for i = 1 to (numChunks - 1) do
14         T := Thread(numMoves[i] = ComputeLabels(B_i))
15         totalMoves += ExchangeLabels(B_{i−1}, numMoves[i − 1])
16         JoinThread(T)
17         ControlClusterWeights(B_i)
18         HandleLabels(B_{i−1})
19     if iteration is last iteration then
20         totalMoves += ExchangeLabels(B_c, numMoves[c − 1])
21         HandleLabels(B_c)
22     if totalMoves == 0 then
23         break
24 return computed cluster
```

data. We just accept this potential source of worse quality. However, a much higher number of ghost nodes compared to local nodes can lead to significantly worse quality, since we repeatedly end up working on a much higher percentage of outdated data (only ghost nodes may have outdated labels).

Let us take a look at an inner step, i.e., a step in which both computation and communication is performed (neither the first nor the last step). In Algorithm 1, ll. 8-12 and ll. 14-18 such an inner step is processed. We first spawn a new thread and assign the label computation to it. We then concurrently perform the label communication for the previously computed labels. After waiting for both threads to finish their parts, we continue by synchronizing the weights of clusters we just modified during label computation. Lastly we apply the labels we just received during label communication. In Algorithm 1, ll.20-21, we perform the very last step of our clustering, i.e., we exchange and handle the labels of

the very last chunk.
In ll. 22-23, we define a break condition for our iteration. If no node moves have been performed during an iteration, we stop the clustering and return our result. As a sidenote, we do not need to handle the labels of the last iteration if we terminate the clustering function before reaching the last iteration. This is because we already know that no nodes have been moved, thus there are no label messages to process. To clarify what we execute asynchronously, we will take a closer look at the computation and communication parts.

Label Computation

We initialize our program with the parameter MPI_THREAD_FUNNELED, so only our base thread is allowed to utilize MPI calls. We do this, because we only need one thread to perform communication at a time and the use of a more powerful mode like for example MPI_THREAD_MULTIPLE is not necessary and would lead to higher communication overheads due to synchronization. This is why we assign the label computation to our newly spawned thread as seen in Algorithm 1, ll. 8,14. During this part, we iterate over the chunk's nodes in the order that they are provided in. Exactly like in `dKaMinPar`, our nodes are placed in exponentially spaced degree buckets and then stored in a roughly increasing degree order induced by the bucket assignments. For each node v, first the new cluster assignment is calculated. This is done by looking at the labels $l_i = label(u_i)$ of nodes u_i in the neighborhood $N(v)$, i.e., $u_i \in N(v)$. A rating map is then constructed by accumulating the edge weights $\omega(e_i)$ of edges $e_i = (v, u_i)$ with the same labels l_i, i.e., $rating(c_j) := \omega(N_j(v))$ is calculated for each cluster label c_j in the neighborhood. This rating map is then used to choose the cluster with the highest gain, by maximizing the difference $rating(c_j) - rating(label(v))$, i.e., maximizing the difference between the potential and the current cluster's ratings. By maximizing the rating, we maximize the intra-cluster weight, and thereby reduce the cut. If we choose to change the label of a node v to a cluster's label c_j, we need to check whether the node move is legal.A node move is legal if the resulting cluster is compliant to the size-constraint, i.e., if it is not overweight. For example, when adding a node v to a cluster $C \setminus \{v\}$, the combined weight should be less or equal to the maximum cluster weight: $c(C) \leq W$. The gain mentioned here refers to the change to the overall intra-cluster weight. Therefore we choose the cluster that has the highest cut $cut(\{\{v\}, N_i(v)\})$ between the processed node v and its neighbors $N_i(v)$ in the cluster C, with the label c_i, among the legal clusters. While performing these node moves we count the number of moves performed, which we return at the end. Before returning, we also cluster the isolated nodes by greedily adding them to a new cluster while maintaining the size-constraint. We do not add these label assignments to our number of node moves, since these nodes' labels never need to be communicated to neighboring PEs.

Figure 4.1: Label-Message

Label Message	
Fields	Size
Owner NodeID	32-bit
ClusterID	64-bit

Label Communication

The label communication part performed concurrently to the next label computation takes the number of moves that was counted during the corresponding computation part and uses an allreduce operation to check whether there have been any node moves globally. If there have been no changes to label assignments, we return with the result. Otherwise, we collect all new label assignments of our interface nodes and put them in buffers for each adjacent PE. We then exchange the labels with those PEs using a sparse-alltoall operation. The received labels are then stored in a buffer for later message handling.

Label Handling

HandleLabels is a function that handles the messages received during the ExchangeLabels function. For each received message $m := [ownerNodeID, clusterID]$ (as depicted in Figure 4.1), we apply the label to our ghost node that corresponds to the $ownerNodeID$ sent to us. We do this by first calculating the ID of our ghost node, by first determining its true NodeID, called *globalNodeID*, and then finding the corresponding local node. We then move the found ghost node into the cluster C_j with $c_j = clusterID$. If we have not received the weight change of the ghost node during the weight handling procedure described in the next section, we manually subtract the weight of the node from its old cluster and add its weight to the new cluster.

Cluster Weight Handling

In order to globally retain the size-constraint, we need to periodically synchronize and correct the weights of clusters. This algorithm reproduces the same logic used by `dKaMinPar`. To make it usable in `ThreadedLP`, we buffer a map from the previous weight handling step, that tracks which clusters's weights have been synchronized. This is necessary because we do not synchronize the weights of all clusters, but instead each PE only synchronizes the weights of clusters it has made a change to. Therefore the PE does not get updates to clusters of ghost nodes, that have not been modified by it and are not owned. While handling received labels, the receiving PE searches the map for the received clusterID. If it is not found, that means that the weight of the cluster with that clusterID has not been synchronized. In this case, the PE subtracts the weight of the ghost node from the old cluster

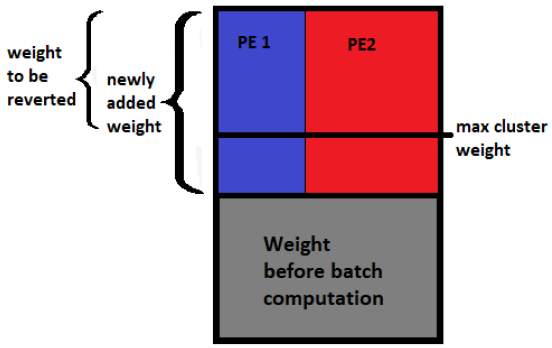

Figure 4.2: State of owned cluster after accumulating weight changes from remote PEs.

and adds it to the new cluster.
The weight handling procedure uses multiple sparse-alltoall operations to exchange weights. First, it accumulates the total locally caused weight change of each cluster on the PE and stores it in the map. Each PE then sends the weight deltas of unowned clusters to the owning PEs. These PEs aggregate the received weight deltas of each cluster and send the total weight of the cluster back to the PEs that sent a part of the total delta. Figure 4.2 shows the weight of a cluster after aggregating the weight deltas. Each PE then calculates its contribution to the part of the total delta, that does not fit in the cluster due to the size-constraint as the amount of weight that has to be reverted from that cluster. For example, let's say each node has a weight of 1. If one PE added a weight of 10 and another added a weight of 20, but only 6 fit into the cluster. The total weight over the maximum is then 24. So the first PE has to revert 8 nodes and the other PE has to revert 16 nodes, as the first PE only added a third of the weight. That amount added to the maximum cluster weight is then set as the new cluster weight of the cluster. Note, that the local cluster is now overweight by the part of the weight, that has to be reverted. Finally each PE iterates over its nodes once while checking for nodes that are inside an overweight cluster. When such a node is found, the node is reverted to its old cluster, i.e., the label is set to the node's old label and the weights of involved clusters are adjusted accordingly.

5 Experimental Evaluation of Threaded LP

In this chapter we will be exploring the performance and quality of `ThreadedLP` in comparison to the default label propagation algorithm of `dKaMinPar`, which we will refer to as `dKaMinPar-LP`. We implemented `ThreadedLP` in C++ and compiled it using g++-13.1.0. We use IntelMPI 2021.9.0 as our parallelization library and growt [15] for hash tables. We use generic C++ threads for our asynchronization. The plots presented here are generated using the open source library Plotly [13].

5.1 Experimental Setup

We run the experiments on two benchmark sets to evaluate our weak and strong scaling capabilities. The instances used are listed in Section 5.1.2.

5.1.1 Environment

We perform our experiments on the SuperMUC-NG high performance computing cluster, which is part of the Leibnitz Supercomputing Centre (LRZ). We make use of the thin compute nodes, i.e., compute nodes with 96GB of memory. Each compute node provides 48 cores, consisting of two Intel Xeon Platinum 8174 24C 3.1GHz CPUs. The processors are connected by an Intel Omni-Path network architecture to allow for low communication latency and high throughput. We only make use of 32 of the 48 processors, since the graph generator, KaGen, requires the number of processes to be a power of two. The cores support hyperthreading, which we enable for our experiments.

5.1.2 Instances

For all our experiments, we compute the same number of 8 blocks. We do not expect different values of k to make a difference in our comparison of the two algorithms, since we only look at coarsening, during which different k values only change the maximum cluster weight. We perform strong and weak scaling experiments, scaling the number of compute nodes up from 8 to 32 and finally 128 compute nodes with 32 processes each, i.e.,

Table 5.1: Strong scaling graphs, with their number of nodes and edges and their number of isolated nodes η, minimum degree greater than 0, average and maximum degree. The parameters used to generate the graphs are: (kronecker;N=18;M=22), (rgg2d;N=27;M=32), (rhg;N=26;M=30;gamma=3.0) and (rmat;N=20;M=24;a=0.1;b=0.2;c=0.3). Note, that the upper case N and M mean, that the graph will be generated with roughly 2^{N} nodes and 2^{M} edges.

Graphs Used in our Strong Scaling Experiments						
Type	Name	n	m	$\eta; \min(\delta > 0)$	avg(δ)	$\max(\delta)$
RealWorld	arabic-2005	22 744 080	553 903 073	199 ; 1	48.7075	575 628
	enwiki-2013	4 206 785	91 939 728	3462 ; 1	43.7102	432 260
	enwiki-2018	5 616 717	117 244 295	8012 ; 1	41.7483	248 444
	nlpkkt240	27 993 600	373 239 376	0 ; 4	26.6661	27
	sk-2005	50 636 154	1 810 063 330	95 ; 1	71.4929	8 563 820
	twitter-2010	41 652 230	1 202 513 046	0 ; 1	57.7406	2 997 487
	webbase-2001	118 142 155	854 809 761	2 587 710 ; 1	14.4709	816 127
Generated	kronecker	262 144	4 194 045	0 ; 10	31.998	62
	rgg2d	134 217 728	4 295 089 192	0 ; 13	64.0018	111
	rhg	67 108 864	1 056 616 051	0 ; 1	31.4896	148 178
	rmat	1 048 576	16 769 729	13 976 ; 1	31.9857	13 669

on 256, 1 028 and 4 096 processes.
We consider real world graphs and graphs generated by the KaGen [7][12] graph generator. The graph data provided in Table 5.1 and Table 5.2 describes the actual graph or graph family. Recall, that the graph representation used for the label propagation is a directed graph, where the edges have been duplicated in order to have one edge for each direction and the PEs also hold additional ghost nodes for unowned nodes. This means, that the total number of nodes that have to be loaded in memory is the number of nodes n plus the number of ghost nodes g, i.e., $total\_nodes := n + g$, and the number of edges is $total\_edges := 2 * m$.
We will call a system setup of a specific number of compute nodes an *execution mode*. We refer to the combination of a graph with an execution mode as an *instance*. We perform strong scaling experiments for 33 instances and 12 experiments for weak scaling.

5.2 Strong Scaling Experiments

We perform our strong scaling experiments on the system described above. We test how our algorithm scales on the fixed graphs, when scaling the number of compute nodes and therefore the number of processors.
Since our focus lies on improving the label propagation during the coarsening phase, will

Table 5.2: Weak scaling graph families, with their number of nodes and edges per compute node and maximum degree per execution mode. The parameters used to generate the graphs are: (kronecker;N=18;M=22), (rgg2d;N=26;M=30), (rhg;N=26;M=30;gamma=3.0) and (rmat;N=18;M=22;a=0.1;b=0.2;c=0.3). Note, that the upper case N and M mean, that the graph will be generated with roughly 2^N nodes and 2^M edges *per compute node*.

Graphs Used in our Weak Scaling Experiments						
Type	Name	n per node	m per node	$\max(\delta)$		
				8 Nodes	32 Nodes	128 Nodes
Generated	kronecker	262 144	4 194 304	64	67	66
	rgg2d	67 108 864	1 073 741 824	68	71	74
	rhg	67 108 864	1 073 741 824	127 786	1 638 340	1 203 110
	rmat	262 144	4 194 304	19 321	37 603	67 420

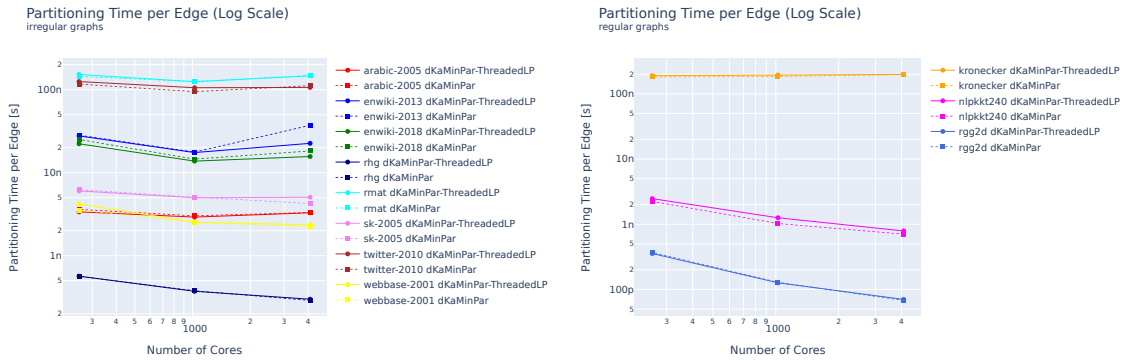

Figure 5.1: Strong scaling experiments' partitioning times per edge for the irregular graphs (left) and the regular graphs (right) over the number of cores; ThreadedLP (solid line), dKaMinPar (dotted line)

evaluate the total partitioning times and the coarsening times separately. We will first take a brief look at the total partitioning times. Looking at Figure 5.1, we can see that our approach does not significantly improve or worsen the performance of the algorithm. Moreover, when putting this into the context of the results of the other execution modes, we do not see any correlation between the type of the processed graph and the running times. Depending on the number of compute nodes used, we see 2-5 instances performing better, 2-4 instances performing the same, and 3-5 instances performing worse. Figure 5.2 shows the coarsening times per edge of all graphs over the number of cores. Similarly to the total partitioning times, we see 4 instances performing better, 1-2 instances performing the same, and 4-5 instances performing worse on the different execution modes. Note, that again, the time differences are very small. When looking at the coarsening times of the different numbers of compute nodes, we see a large diversity of plots both in the aspect of scalability

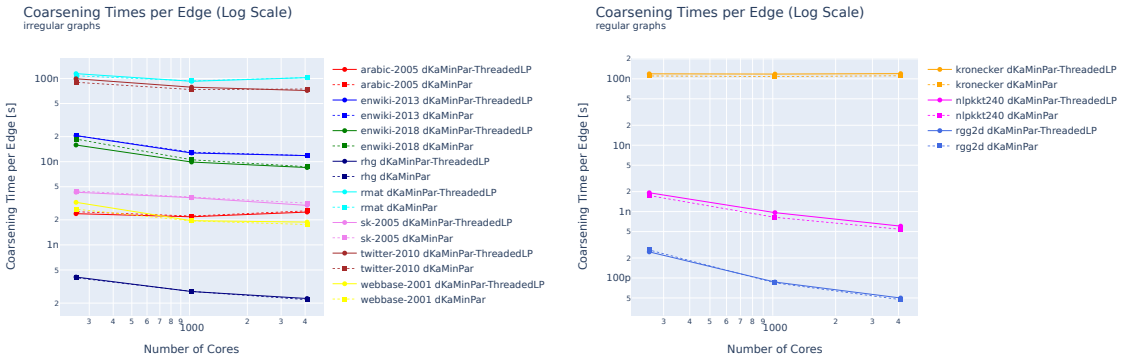

Figure 5.2: Strong scaling experiments' coarsening times per edge for the irregular graphs (left) and the regular graphs (right) over the number of cores; ThreadedLP (solid line), dKaMinPar-LP (dotted line)

and relative performance. As seen in Figure 5.2, the algorithm shows the same pattern of scalability as `dKaMinPar-LP` with the exception of twitter-2010, where `ThreadedLP` initially displays slower coarsening times, but gets better on 128 compute nodes, being 4% faster. `ThreadedLP` shows strong scalability for up to 32 compute nodes (1024 cores) for the rmat, kronecker and arabic-2005 graph. It even shows scalability to 128 compute nodes (4096 cores) for the rest (rgg2d, rhg, both enwiki graphs, nlpkkt240, sk-2005, twitter-2010 and webbase-2001). Figure 5.3 shows the difference between the coarsening times of the two algorithms. The coarsening times values are shown in relation to the coarsening times of `dKaMinPar-LP`. The greatest differences can be seen on 8 compute nodes, with the `ThreadedLP` algorithm being about 15.2% faster on the enwiki-2018 instance and about 20.8% slower on the webbase-2001 instance. We can also see the increase of performance on the twitter-2010 graph, that is a result of the better scalability of our algorithm.
As expected we see similar cuts being computed by the two algorithms. For most of the instances (rgg2d, rhg, enwiki-2013, webbase-2001, twitter on 8 compute nodes), the cuts after each level are almost identical. With "cut after each level" we refer to the cut weight of the intermediate clusterings computed in one coarsening level. We see slightly higher cuts on the rmat (by around .4%), kronecker (by around .3%) and arabic-2005 (up to 9.6%) instances and unexpectedly even slightly lower cuts on enwiki-2018 (by up to 1.8%). Our coarsening algorithm returns a solution of visibly lower quality on nlpkkt240 instances and the twitter instances run on 32 and 128 compute nodes. The tendencies seem to increase for some graphs when scaling the number of compute nodes up, i.e., enwiki-2018 starts out with almost the same cut values on 8 compute nodes, but `ThreadedLP` shows notably higher cuts on 128 compute nodes. Figure 5.5 shows the cuts for the webbase-2001 graph. The cuts of the initial partitions only differ by insignificant amounts. So do the final cuts computed by the partitioner (after uncoarsening and refinement). Running the algorithm on different numbers of compute nodes does not seem to significantly change the cut

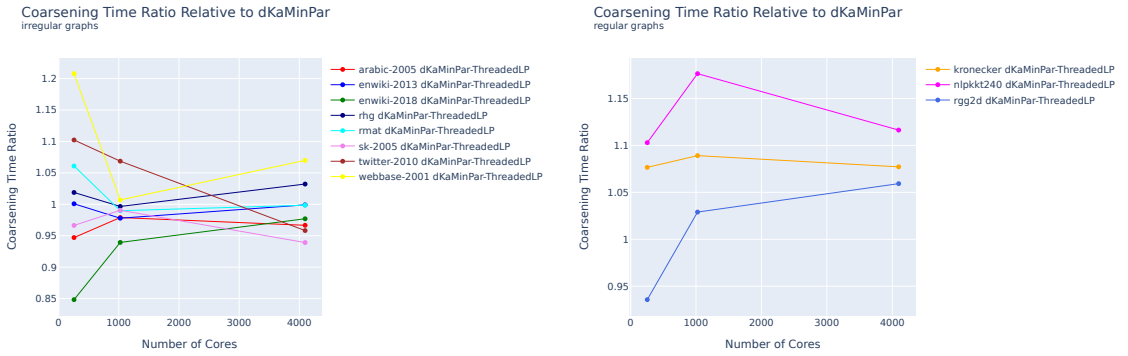

Figure 5.3: Strong scaling experiments' coarsening time ratios in relation to dKaMinPar-LP of the irregular graphs (left) and the regular graphs (right) over the number of cores; ThreadedLP (solid line), dKaMinPar-LP (dotted line)

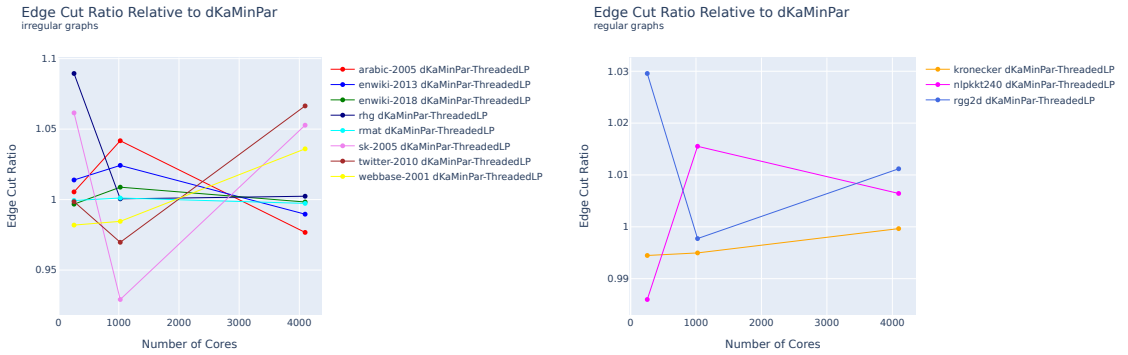

Figure 5.4: The ratio of the final cuts for the irregular graphs (left) and the regular graphs (right) in relation to dKaMinPar-LP; ThreadedLP (solid line), dKaMinPar-LP (dotted line)

computed.

In conclusion, `ThreadedLP` shows a better strong scalability on the twitter-2010 graph, while displaying similar performance and quality to `dKaMinPar-LP`.

5.3 Weak Scaling Experiments

For our weak scaling experiments, we use generated graphs, that grow proportionally to the number of compute nodes used. As we did before, we will be focusing on the evaluation of the coarsening phase.

We can see in Figure 5.6 and Figure 5.7, that the performance improves drastically in the number of compute nodes, when compared to `dKaMinPar-LP`. On lower numbers of

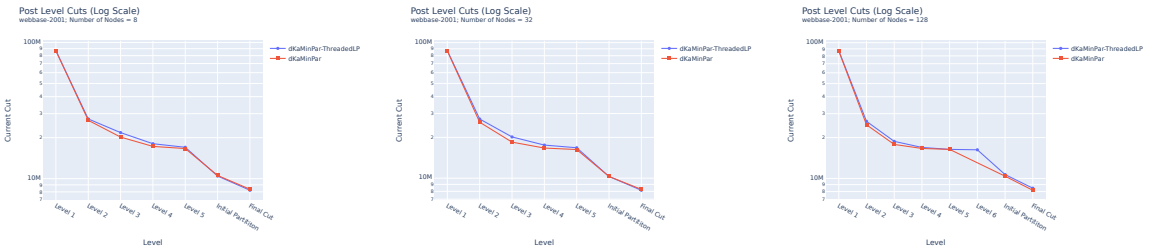

Figure 5.5: The cuts after each level and the initial partition as well as the final cut for the webbase-2001 graph on 8 (left), 32 (middle) and 128 (right) compute nodes; ThreadedLP (blue line), dKaMinPar-LP (red line)

compute nodes, `ThreadedLP` struggles to keep up with the performance of `dKaMinPar-LP`, only showing improvements on the communication-heavy rhg graph. With increasing numbers of compute nodes and therefore processors, the amount of communication performed during label propagation increases, due to the higher number of ghost nodes (higher inter-PE cut). We can see this effect in the coarsening times of the two algorithms. For 128 compute nodes the more communication efficient coarsening phase of `ThreadedLP` displays lower coarsening times on the kronecker, rhg and rmat graph when compared to `dKaMinPar-LP`, while taking almost the same amount of time for rgg2d, as seen in Figure 5.8. According to our experiments, the communication amount on the rgg2d graph instances increases to only 6 times the value on 8 compute nodes. Meanwhile, the other graph instances show increases to 10 to 12 times the communication amounts. It is also worth to mention, that the rmat graph does not scale for either algorithm. When looking at the throughput, we can see that the number of edges handled per second stays almost the same. In particular, it does not scale proportionally to the number of cores used.
Just like the strong-scaling experiments, the weak-scaling instances produce almost the same cuts for both coarsening strategies, while not deteriorating when increasing the number of compute nodes. Figure 5.9 shows the cuts produced by each level, the initial partition as well as the final cut. The increase in the computed cut as seen in the bottom -right plot is to be expected, since the total number of edges increases when increasing the size of the graph by scaling it up proportionally to the number of compute nodes.
In conclusion, `ThreadedLP` seems to show better results on the larger numbers of compute nodes with higher amounts of communication (up to 7.6% faster, but 1.9% slower on rgg2d). On 128 compute nodes it minimally improves upon the performance of `dKaMinPar-LP`, while producing similar cuts.

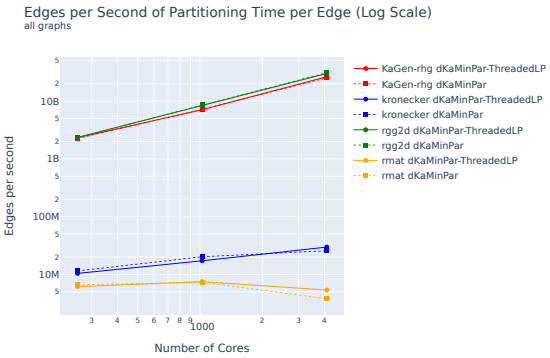

Figure 5.6: Weak scaling experiments' edges per second of partitioning time (throughput) for all graphs over the number of cores; ThreadedLP (solid line), dKaMinPar-LP (dotted line); the graphs are scaled so that, that the number of nodes and edges per core is kept relatively constant

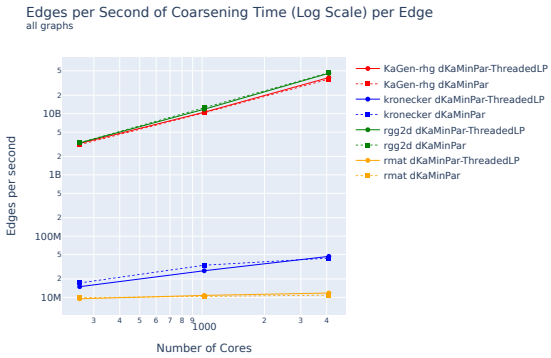

Figure 5.7: Weak scaling experiments' edges per second of coarsening time (throughput) for all graphs over the number of cores; ThreadedLP (solid line), dKaMinPar-LP (dotted line); the graphs are scaled so that, that the number of nodes and edges per core is kept relatively constant

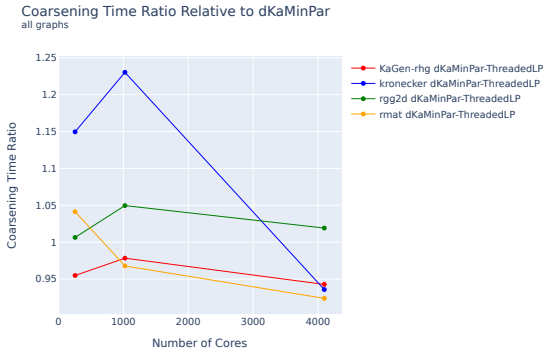

Figure 5.8: Weak scaling experiments' coarsening time ratios relative to dKaMinPar-LP for all graphs over the number of cores; ThreadedLP (solid line), dKaMinPar-LP (dotted line); the graphs are scaled so that, that the number of nodes and edges per core is kept relatively constant

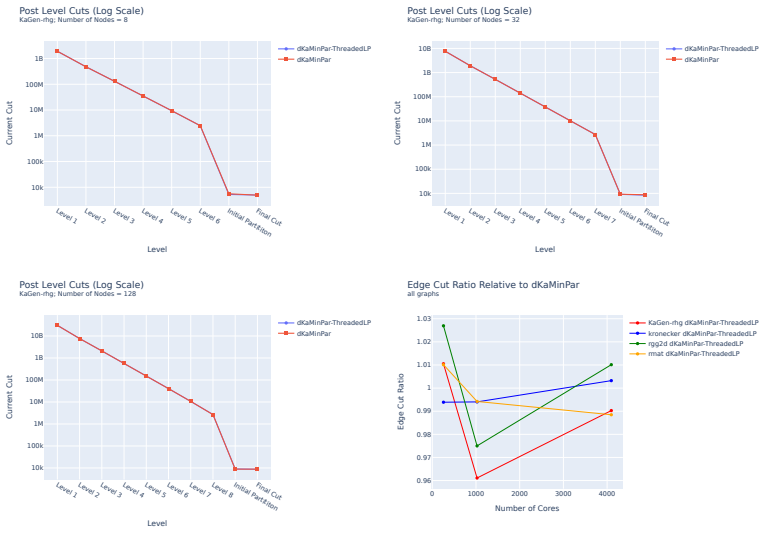

Figure 5.9: The cuts after each level and the initial partition as well as the final cut for the rhg graph on 8 (top-left), 32 (top-right) and 128 (bottom-left) compute nodes; ThreadedLP (blue line), dKaMinPar-LP (red line); the plot on the bottom-right shows the ratio of the final cuts of all graphs over the number of cores in relation to dKaMinPar-LP; ThreadedLP (solid line), dKaMinPar-LP (dotted line)

6 Message Queue Label Propagation Clustering

This is our second algorithm, which we call Message Queue Label Propagation (`MQLP`). As we did for our previously presented algorithm, we will first provide the general idea in Section 6.1 and then go into greater detail in Section 6.2.

6.1 Overview

This algorithm makes use of the message queue implemented by Uhl [22] to asynchronously communicate labels and weight changes. In contrast to our first algorithm, we do not use multithreading to separate the communication. Instead, we use two message queues, one of which is used to communicate labels and the other is used for synchronizing the cluster weights. The message queues use non-blocking MPI operations, which enables us to continue our computation while waiting for the messages to reach their targets. These operations are started when reaching a certain threshold in the number of messages posted in the message queue. Until then, the messages are put into a buffer for each target PE, that aggregates the messages. When the threshold is reached, the aggregated messages are sent using an Isend operation. The receiving PEs repeatedly poll for new messages and handle them when received. We will also be presenting a new strategy for approximating the weight constraint, while allowing asynchronous processing.
The goal is to increase the performance by allowing further progress, while the message queue sends messages. Additionally we expect to allow for lower cuts, due to more frequent communication. We also hope to allow for decent scalability, but due to higher memory usage compared to `dKaMinPar`, which stems from additional datastructures used, we might not be able to compute graphs as large as those tested by `dKaMinPar`. Since the additional datastructures are roughly proportional in size to the number of local nodes, we do not expect this to be a big issue.

6.2 Implementation

Like `ThreadedLP` and `dKaMinPar`, the algorithm takes a graph $G = (V, E)$, with the nodes in V being rearranged in increasing degree order, and computes a clustering

Algorithm 2: MQLP(G, P): Label Propagation on P PEs

```
Input: G = (V, E)
Output: clustering C := G' = (V', E') of G
1  MakeLabelMessageQueue
2  MakeWeightsMessageQueue
3  for iteration < maxIterations do
4  |  numMoves = 0
5  |  for each node v do
6  |  |  numMoves += PerformLPForNode(v)
7  |  |  if weight message threshold surpassed then
8  |  |  |  PostWeightMessages()
9  |  |  if should handle weights now then
10 |  |  |  HandleWeights()
11 |  |  if should handle labels now then
12 |  |  |  HandleMessages()
13 |  ClusterIsolatedNodes()
14 |  totalMoves := Allreduce(+, numMoves)
15 |  if totalMoves == 0 then
16 |  |  break
17 |  FixOverweightClusters()
18 |  PrepareNextIteration
19 return computed cluster
```

$C := G' = (V', E')$ of G. The algorithm then performs label propagation for up to $maxIterations$ iterations. During this process, nodes are traversed in the order in which they are stored. It shall be noted, that each node is traversed exactly once per iteration. Algorithm 2 outlines the procedure used for clustering the graph.

Besides the label-message queue used for the communication of new labels, we use a separate weight-message queue, to allow the weight-messages to reach their target independently from the current state of the label communication. Figure 6.1 describes the basic way we use the message queues. We will now describe both message queues separately.

Label-Message Queue

The label-message queue places messages into the send buffer and separates them with a sentinel element. Messages put into and read from the buffer, are wrapped in envelopes as seen in Figure 6.4, that hold the message and meta information needed for the communication. Label-messages (Figure 6.2) consist of the local ID of the node in question, and the

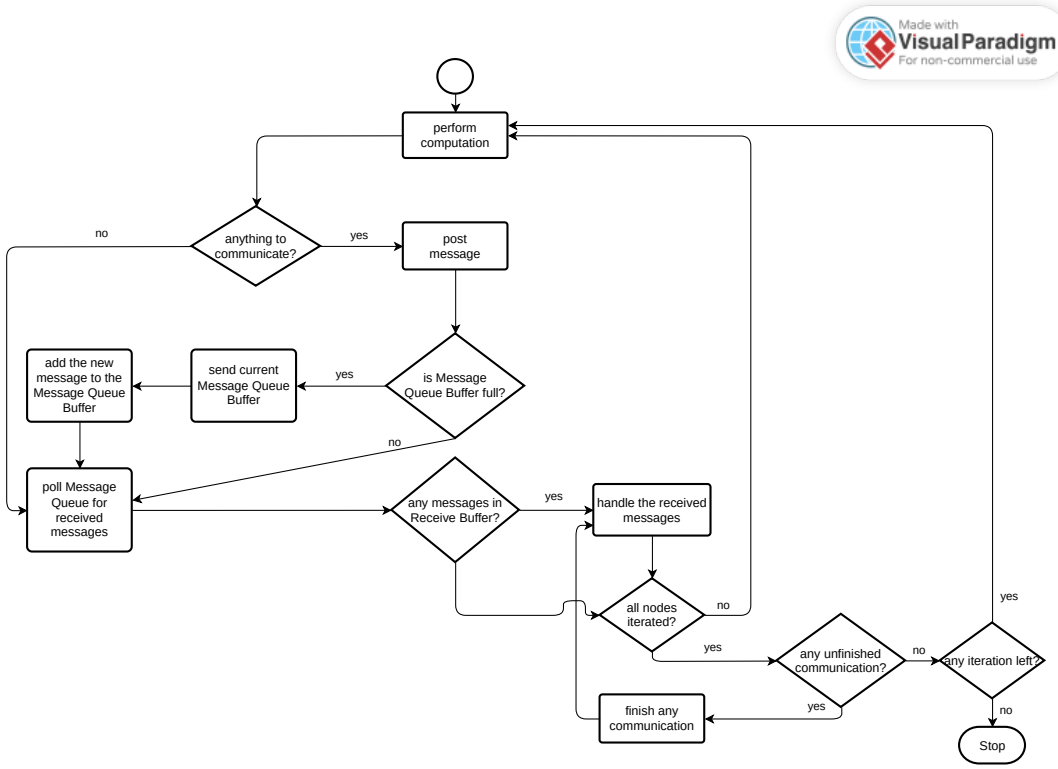

Figure 6.1: Flow chart modelling the usage of a message queue. Note, that the check whether the message queue buffer is full, is basically checking whether the global threshold would be exceeded by adding the newly posted messages.

label that the node is assigned. The use of sentinels allows the sender to encode the sender and receiver into the envelopes, which enables the use of indirection during communication. We use the generic grid-indirection scheme defined in the message queue library used by us [22]. This indirection scheme uses a grid to organize the PEs. It then divides the communication into two steps. First the message to a target PE is sent to the corresponding row, and then it is redirected from there. By using indirection, we reduce the total amount of startup overhead required for the communications, by reducing the number of communication partners for each PE from $\mathcal{O}(P)$ to $\mathcal{O}(\sqrt{P})$. Each PE holds send buffers for each PE, which in theory allows pre-send cleanup of deprecated labels. However we do not make use of this, since the amount of times, the same node's label is changed during an iteration is negligible (this can only happen, when a node reverts its label, since nodes are only traversed once per label propagation iteration). The buffers are flushed, when the global buffer threshold is exceeded. The global buffer threshold is checked against the combined

Figure 6.2: Label-Message

Label-Message	
Fields	Size
Owner NodeID	32-bit
ClusterID	64-bit

Figure 6.3: Weight-Message

Weight-Message	
Fields	Size
Flag	2-bit
ClusterID	62-bit
Weight Delta	64-bit

Figure 6.4: Message Envelope

Message Envelope	
Fields	Size
Message	2x 64-bit
Sender	64-bit
Receiver	64-bit

sizes of the send buffers to each target PE. This tuning parameter can be set manually, but is generally computed dynamically to be proportional to the size of the local subgraph. The dynamically calculated threshold is defined as $T := \min\{80000, \text{dynamicValue}\}$, where:

$$\text{dynamicValue} := \begin{cases} \lceil \text{numLocal}/(2 * \text{numChunks}) \rceil & \text{for numLocal} > \text{numGhosts} \\ \lceil \text{numGhosts}/(2 * \text{numChunks}) \rceil \end{cases}$$

The $numChunks$ used here is equal to the number of batches used by `dKaMinPar` and `ThreadedLP`. The reason why we set the upper limit of the threshold to 80000 is that some testing has shown this to be an efficient threshold, that makes good progress while not significantly increasing the overheads. We make use of that by shrinking the threshold down to this value whenever the dynamicValue would be larger than this. This basically increases the quality of the algorithm due to faster updates, without having any significant drawbacks.

Weight-Message Queue

The weight-message queue also places a sentinel element between its messages, for the same reasons stated previously. Differently to the label-message queue, we have two types of messages that are sent in the weight-message queue, namely a weight-delta message and a cluster-lock message. We differentiate between the two messages by setting a flag, that signals the type of the message. The weight-delta message consists of the cluster ID and the weight change of the corresponding cluster. The cluster-lock message simply provides the cluster ID, so that the receiving PE can lock the cluster to disallow further moves to it. Like for the label-message queue, the global buffer threshold is computed dynamically. By default, the threshold is roughly set to a quarter of the value that is used by the label-message queue, because some preliminary tests have shown the volume of weight-messages to be significantly lower than the label-messages. If the threshold would be larger than 80000, the threshold is set to 80000. So the weight global threshold is defined

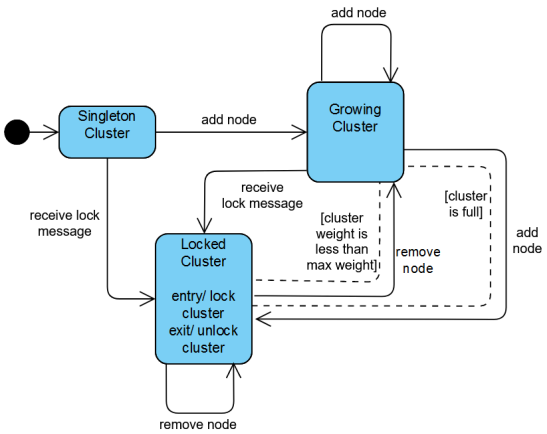

Figure 6.5: The states an unowned cluster can be in and the operations resulting in that state. The dotted lines mark conditions that have to be met to use the path. Note, that a cluster can be locked by completely filling it locally, or receiving a lock message due to it being locked on the remote owning PE. This lock message can arrive independently from the other operations. In particular, this means that the lock message can be received after multiple additional nodes have been added. (Made with Visual Paradigm)

as $T := \min\{80000, \text{dynamicValue}\}$, where:

$$\text{dynamicValue} := \begin{cases} \lceil \text{numLocal}/(8 * \text{numChunks}) \rceil & \text{for numLocal} > \text{numGhosts} \\ \lceil \text{numGhosts}/(8 * \text{numChunks}) \rceil \end{cases}$$

Size-Constraint Approximation

Due to the nature of the asynchronous processing and the use of two separate message queues for weight- and label-messages, we cannot use the weight-constraint handling performed by `dKaMinPar`. The reason is, that the weight handling used by `dKaMinPar` relies on the assumption, that a certain chunk of the nodes has been fully processed and the weight handling has finished before continuing with the next chunk. However this can not be assured by an asynchronous algorithm. Furthermore, naively using the weight handling strategy used by `dKaMinPar`, without adhering to the requirements, leads to worse performance and deterioration of quality.
As a result, we instead relax the size-constraint by only locally enforcing it on the PEs, similarly to `ParHIP`. Since this can lead to clusters with maximum sizes of $P * W$, i.e., the number of PEs times the maximum cluster weight, we introduce a "weight fixing" stage after each iteration. However the weight fixing procedure used can only revert moves performed during the iteration. It does not actually compute new cluster assignments. To improve on this, we also add a cluster locking procedure, that locks clusters, that are full, which prohibits further additions to the locked clusters. For unowned clusters, i.e., clusters, whose representing node is owned by a remote PE, this lock is received when trying

to move a node into a full cluster or when performing a move that fully satiates the cluster (i.e., the cluster is full after the move). Consequently, the impact of this method is heavily dependent on the granularity of the weight communication. This leads to the need to choose between better performance and better quality, since lower thresholds and therefore more frequent communication leads to better weight approximation, but comes at the cost of higher communication overheads. It is also worth to mention, that inserting the weight fixing stage after each iteration requires synchronizing the processes at that point, which further reduces the performance of the algorithm. Figure 6.5 shows the states an unowned cluster can be in.

Label Propagation

First, the new labels of traversed nodes are determined by exploring the neighborhood $N(v)$ of the currently processed node v. As described in Section 24, the legal node move to a cluster resulting in the highest gain is then chosen. In our case a node move is considered legal, if the cluster does not become overweight due to the weight change and the cluster is not locked already. There are two ways a cluster can become locked. Either by locally moving a node into the cluster, which leads to it having the maximum cluster weight W, or by receiving a lock message for an unowned cluster. In case that it is illegal, we do not choose the cluster and keep looking in the rating map. When we have checked all neighboring clusters, the change is applied, i.e., the label is changed and the weights of the involved clusters are adjusted. For interface nodes, the new label is wrapped in a label-message, that contains the ID of the node and the new label assignment, and is posted in the label-message queue to the adjacent ghost nodes' owners. Instead of posting the weight changes of unowned clusters immediately to the weight-message queue, they are buffered in a *weightDeltaMap*, as seen in Algorithm 3, that tracks the weight changes until the weights are communicated. This allows us to aggregate multiple changes to the same cluster, reducing the size of the messages sent. However, we check whether we know if the cluster is locked. We can know this by either having received a lock message from the owning PE, or by being the owning PE and tracking the true weight of the cluster. Therefore we only send the lock message, if the receiving PE does not own the cluster (the owning PE always is the first PE to lock its cluster). This way, if we are sending the new label to an adjacent PE, that does not own the cluster, that PE immediately knows it is not allowed to move nodes into that cluster.

Line 7 in Algorithm 2 checks, whether the global buffer threshold of the weight-message queue has been surpassed. In that case, the buffered weight changes stored in the *weight-DeltaMap* are posted in the weight-message queue, as seen in Algorithm 4, ll. 1-2. Since we aggregated the messages manually before posting them, we do not actually surpass the buffer threshold of the weight-message queue. This means, that the buffers are not automatically flushed by the message queue. But since we do want the messages to be sent, we

Algorithm 3: PerformLPForNode(v): Label Propagation for a Single Node

```
Input: NodeID v
Output: number of moves performed locally numMoves
1  numMoves := ComputeNewClusterForNode(v)
2  if old cluster is not owned then
3  |  add weight change to weightDeltaMap
4  if new cluster is not owned then
5  |  add weight change to weightDeltaMap
6  if v is interface node then
7  |  post message to adjacent PE
8  |  if new cluster is locked then
9  |  |  post lock message to adjacent PE
10 return numMoves
```

manually flush the buffers after posting all messages, as seen in Algorithm 4.

Algorithm 4: PostWeightMessages(): Posting Weight Messages from weightDeltaMap

```
1 for each entry in weightDeltaMap do
2 |  post weight-message to owning PE in MQ
3 FlushWeightsMessageQueueBuffers
```

We also define a weight handle threshold and a label handle threshold, which are by default set to 1, in order to immediately work on received messages. After performing said amount of steps, i.e., after processing one node, the weight handle and label handle procedures are executed. As shown in ll. 9-12 in Algorithm 2, the two procedures are handled separately to allow different thresholds for them. Calling these procedures essentially polls the corresponding message queue for received messages. We will now take a look at the two message handling procedures.

Algorithm 5 shows the procedure for handling a weight-message. The weight-message handling procedure has to process two types of messages. A weight-delta message $m := [clusterID, delta]$ and a cluster-lock message $m := [clusterID]$, as seen in Figure 6.3. The weight-message handler first checks the flag, to determine which type of message it is processing. In the case, that it is a weight-delta message, the receiving PE applies the weight change to the cluster, i.e., $c(C_m) += delta$ with $c_m = clusterID$. The delta may also be negative, e.g., if a few nodes have been removed from a cluster, the delta will be the amount of weight subtracted from that cluster. If the cluster is then full or overweight, the cluster is locked and the receiving PE replies to the sender with a cluster-lock message. If the cluster was already locked, the weight is applied anyway and the cluster-lock response is sent. As a sidenote, the PE receiving a weight-delta message is always the owner of the

cluster. In the case of the cluster-lock message, the receiver locks the cluster corresponding to the ID sent within the cluster-lock message. When creating a cluster-lock message, the weight delta field is left empty. Cluster-lock messages are always sent from owning PEs and received by non-owning PEs.

Algorithm 5: HandleWeights(): Weight-Message Handling

```
1 for each received message m do
2     switch m.messageType do
3         case cluster-lock message
4             lock cluster
5         case weight-delta message
6             apply weight change to cluster
7             if cluster is overweight then
8                 lock luster
9                 send lock-message to message's sender
```

The label-message handler shown in Algorithm 6 only processes label-messages $m := [ownerNodeID, label]$, the structure of which is also depicted in Figure 6.2. It first converts the received remote NodeID into the local NodeID, by applying the offset of the NodeID due to the distance between the PEs. Then the label of the ghost node is changed to the label received in the label-message. Additionally, the cluster weight changes created by the node move are applied, i.e., the old cluster c_i has a new weight of $c(C_{old}) = c(C_i) - c(v)$ and the new cluster c_j has a weight of $c(C_{new}) = c(C_j) + c(v)$.

Algorithm 6: HandleMessages(): Label-Message Handling

```
1 for each received message m do
2     (node, cluster) := m
3     move node to new cluster
```

After processing each node, we cluster the isolated nodes on each PE, i.e., nodes without any edges. We do this by simply locally adding isolated nodes to a new cluster, until it is full and then repeating this until all isolated nodes have been clustered. In Algorithm 2 l. 6 the number of local moves are accumulated for an iteration. This is later used in ll. 13-14 to check whether a change has occurred during that iteration. If there were no node moves during an iteration, the clustering is terminated regardless of the number of iterations performed. To get the actual number of moves performed globally over all PEs, we use an allreduce operation to sum up the local numbers of moves.
The FixOverweightClusters procedure seen in Algorithm 7 uses synchronous sparse-alltoall

communication operations in order to send the weights added to an unowned cluster, that has been locked, to the owning PE. The owning PE then accumulates the added weights and sends back the portion of the weight that each PE has to revert. It then applies the change of the weight of the owned cluster, that the other PEs are going to create by reverting moves. This removes redundant communication that would have been necessary otherwise. After receiving the weights it has to remove, each PE traverses its nodes once, while looking for nodes in overweight clusters, that have been added during the most recent iteration. If such a node is found, the node move is reverted, i.e., the node is assigned its old label and the weights of the clusters are adjusted. If this weight adjustment leads to the old cluster becoming overweight, it is locked. Also, clusters that begin to meet the size-constraint due to a weight change, are unlocked.

Algorithm 7: FixOverweightClusters(): Reverting Nodes from Overweight Clusters

```
1  for unowned locked clusters do
2  |  if added weight to cluster then
3  |  |  send locally added cluster weight to owning PE
4  for received requests do
5  |  accumulate weight for each cluster
6  |  calculate the parts that each PE has to revert
7  |  send back how much weight needs to be reverted
8  remove reverted weight in owned clusters
9  for all nodes do
10 |  if node has not been moved then
11 |  |  continue
12 |  if cluster of node is too heavy then
13 |  |  move node back to previous cluster
14 |  |  move node weight back to previous cluster
15 |  |  if previous cluster is full then
16 |  |  |  lock previous cluster
```

7 Experimental Evaluation of Message Queue LP

In this chapter we will be evaluating the performance and quality of `MQLP` in comparison to the default label propagation algorithm defined in `dKaMinPar`, which we refer to as `dKaMinPar-LP`. We implemented `MQLP` in C++ and compiled it using g++-13.1.0. We use IntelMPI 2021.9.0 as our parallelization library and a combination of growt [15] and Google Sparsehash dense hash map and dense hash set for hash tables. We use two types of hash tables, because we build our implementation on a hybrid parallel code base that uses a parallel hash table (growt), but do not use the thread parallelism for this work, hence we use a sequential hash table. We also use a message queue framework provided by Uhl [22][27]. The plots presented here are generated using the open source library Plotly [13].

7.1 Experimental Setup

We run the experiments on the two benchmark sets we already used to evaluate our weak and strong scaling capabilities of the `ThreadedLP` algorithm. Each instance is run with a time limit of 20 minutes to save expensive computation time. Most of our instances terminate during this time limit. For instances that exceed the time limit, the performance is too bad anyway, i.e., `dKaMinPar-LP` finishes in much less time. The instances used are listed in Section 5.1.2.

7.1.1 Environment

We perform our experiments on the SuperMUC-NG high performance computing cluster, which is part of the Leibnitz Supercomputing Centre (LRZ). We make use of the thin compute nodes, i.e., compute nodes with 96GB of memory. Each compute node provides 48 cores, consisting of two Intel Xeon Platinum 8174 24C 3.1GHz CPUs. The processors are connected by an Intel Omni-Path network architecture to allow for low communication latency and high throughput. We only make use of 32 of the 48 processors, since the graph generator, KaGen, requires the number of processes to be a power of two. For these experiments we do not make use of the hyperthreading capabilites available on the SuperMUC-NG compute nodes.

7.1.2 Tuning Parameters

We use the default values of our tuning parameters. As mentioned previously, the default for the polling interval is set to 1, to always poll the message queue for new entries. The global thresholds are set to:
$T \coloneqq \min\{80000, \text{dynamicValue}\}$, where:

$$\text{dynamicValue} \coloneqq \begin{cases} \lceil \text{numLocal}/(\gamma * \text{numChunks}) \rceil & \text{for numLocal} > \text{numGhosts} \\ \lceil \text{numGhosts}/(\gamma * \text{numChunks}) \rceil \end{cases}$$

with $\gamma \coloneqq 2$ for the global threshold of the label message queue and $\gamma \coloneqq 8$ for the global threshold of the weight message queue. The numLocal variable refers to the number of owned nodes, numGhosts is the number of locally tracked ghost nodes and numChunks is the same value as the default number of Chunks used during an iteration of `dKaMinPar-LP`.

7.1.3 Instances

For all our experiments, we compute the same number of 8 blocks. We do not expect different values of k to make a difference in our comparison of the two algorithms, since we only look at coarsening, during which different k values only change the maximum cluster weight. We perform strong and weak scaling experiments, scaling the number of compute nodes up from 8 to 32 and finally 128 compute nodes with 32 processes each, i.e., on 256, 1 028 and 4 096 processes.
We use the instances we previously used to evaluate `ThreadedLP`, which are described in Section 5.1.2.

7.2 Strong Scaling Experiments

We will first evaluate our strong scaling capabilities. As we can see in Figure 7.1 and Figure 7.2, the algorithm performs worse than `dKaMinPar-LP` by factors of 2 to roughly 106 (e.g., enwiki-2013 on 32 compute nodes) when comparing the coarsening times, as seen in Figure 7.3. The total partitioning times are up to 80 times the control value. The only exceptions for this are the rgg2d and rhg graphs on 128 compute nodes, which perform slightly better than `dKaMinPar-LP`. Note, that the twitter-2010 executions are not included, since they did not finish during the time limit. We have observed that some very skewed graphs take a lot longer by using our message queue approach. This may be caused by the fact, that our message queue does not regulate the amount of messages received. This means, that in some unfortunate cases, where multiple PEs send messages to one PE, the incoming messages can overwhelm the receiving PE. The only graphs for which `MQLP` seems to scale are the two aforementioned plus nlpkkt240. Against our expectations, we

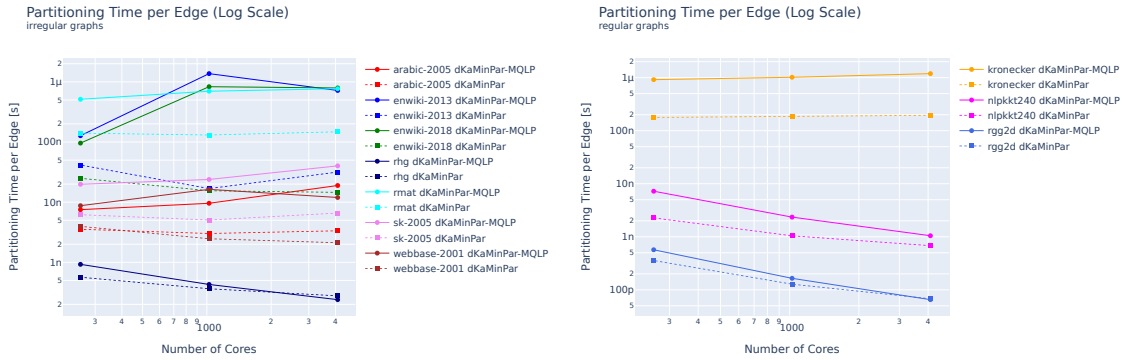

Figure 7.1: Strong scaling experiments' partitioning times per edge for the irregular graphs (left) and the regular graphs (right) over the number of cores; MQLP (solid line), dKaMinPar-LP (dotted line)

also see worse cuts for most of the instances, as seen in Figure 7.4. However, with the exception of rhg, and rgg2d, where we compute the same cuts, most instances generate better cuts during coarsening, which get worse after the initial partitioning step or after uncoarsening, due to our clusterings being less balanced than those computed by `dKaMinPar-LP`. While `dKaMinPar-LP` computes initial partitions with imbalances of only around 0.03, we generate imbalances of 0.3 (sk-2005), 0.38 (enwiki-2018), 0.39(enwiki-2013), 0.45 (webbase-2001), 0.75 (kronecker) and 0.78 (rmat). In fact, while only a few graphs keep the size-constraint (namely rgg2d, rhg and nlpkkt240, with imbalances of 0.27 to 0.29), for most instances we compute clusterings violating the size-constraint by reaching cluster weights of up to $26 * W$ (rmat on 128 compute nodes). Figure 7.5 shows a few instances illustrating this.

7.3 Weak Scaling Experiments

We will take a look at how our algorithm performs when scaling the size of the graph up proportionally to the number of compute nodes, i.e., each processor core has roughly the same number of nodes and edges for each execution mode.
Figure 7.6 show the graphs' throughputs relating to the coarsening times, plotted over the number of cores used. The data for the kronecker graph and the rmat graph instances on 128 compute nodes are missing because they did not finish before the time limit of 20 minutes. But the drop in throughput from 8 compute nodes to 32 compute nodes already shows that the algorithm does not scale at all for these graphs. The rgg2d and rhg graph instances scale similarly to the `dKaMinPar-LP` algorithm. We can also see that the overall performance of `MQLP` is worse than that of `dKaMinPar-LP` for all graphs, as seen in Figure 7.7. For the kronecker graph, `MQLP` reaches times that are 55 times the time needed

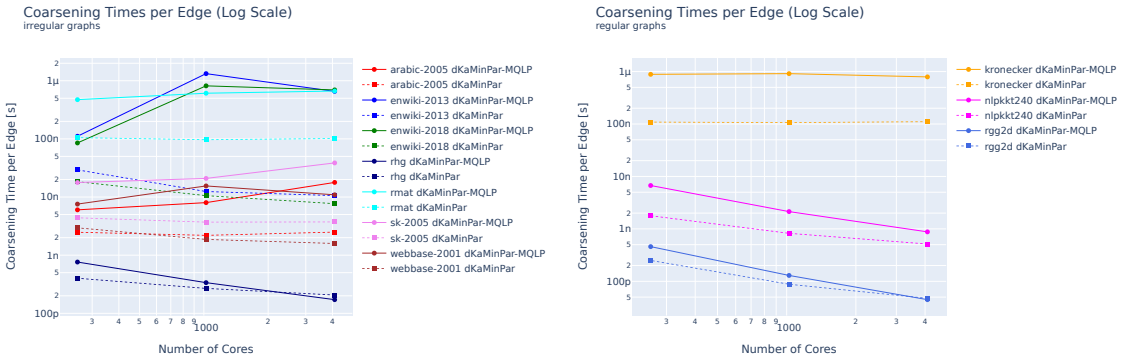

Figure 7.2: Strong scaling experiments' coarsening times per edge for the irregular graphs (left) and the regular graphs (right) over the number of cores; MQLP (solid line), dKaMinPar-LP (dotted line)

by `dKaMinPar-LP`. As we can see in Figure 7.8, the computed cuts are pretty much the same as the cuts computed by `dKaMinPar-LP`. Again, the reason why there are missing points for the kronecker and rmat graphs is that their experiments on 128 compute nodes did not terminate before the time limit was reached. Taking a closer look at the top two plots and the bottom-left plot in Figure 7.8, we can see, that the cuts computed during the coarsening phase are lower than the cuts computed by `dKaMinPar-LP` for the kronecker and rmat graphs. This time, we do not see the cuts getting significantly worse at the end, even though we violate the size-constraint by up to $42 * W$ (kronecker) in this case. The rgg2d and rhg instances produce the same cuts as `dKaMinPar-LP`.

7.4 Iterations

A possible improvement for the `MQLP` algorithm is the use of the active set strategy, by performing the first iteration of the label propagation for each node and then performing asynchronous computation, by considering active nodes sequentially. An active node looks at the labels of its neighbors and computes its new label. After changing the label, the node communicates the change if necessary (if it is adjacent to a ghost node) and becomes inactive. Neighbors of that node get activated, if the node changed its label. This will be repeated until a termination condition is fulfilled. Now to investigate, whether this approach could yield an algorithm, which is an improvement to `dKaMinPar-LP`, we will check how the `MQLP` algorithm performs in the separate iterations. We will take a look at the times taken for the `MQLP` algorithm to perform the first iteration in comparison to:

a) the other iterations

b) the time taken for `dKaMinPar-LP` to perform all three iterations

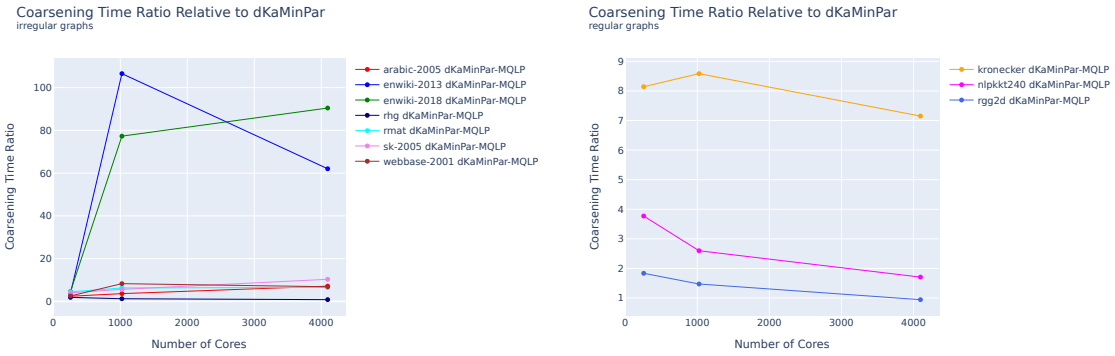

Figure 7.3: Strong scaling experiments' coarsening time ratios relative to dKaMinPar-LP for the irregular graphs (left) and the regular graphs (right) over the number of cores; MQLP (solid line), dKaMinPar-LP (dotted line)

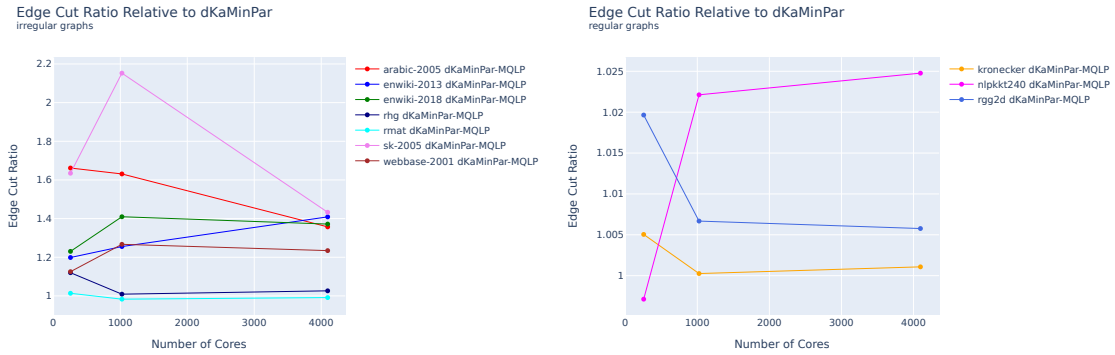

Figure 7.4: Ratio between the final cuts relative to dKaMinPar-LP for the irregular graphs (left) and the regular graphs (right); MQLP (solid line), dKaMinPar-LP (dotted line)

a) MQLP's First Iteration vs Following Iteratations

In Figure 7.9 we see the combined iteration times for each instance. The numbers plotted are the sums of the runtimes of the first iteration of each level l, i.e., $\Delta t_{first} := \sum_l \Delta t_{iteration0}$, and the sums of the runtimes of the second and third iteration of each level, i.e., $\Delta t_{rest} := \sum_l \Delta t_{iteration1} + \Delta t_{iteration2}$. We can see, that the times for the second and third iterations are roughly 0.5 to 2 times as high as the first iteration depending on the graph. This means, that an optimized algorithm, that would replace the second and third iteration of `MQLP`, could in theory reach a speedup of 3 compared to `MQLP`. Obviously, this speedup is not possible by only optimizing the second and third iteration.

Figure 7.5: The cuts after each level and the initial partition as well as the final cut for the enwiki-2013-128 (left), nlpkkt240-128 (middle) and sk-2005-128 (right) instances; MQLP (blue line), dKaMinPar-LP (red line)

b) MQLP's First Iteration vs dKaMinPar-LP's Full Clustering

For each instance, Figure 7.10 visualizes the time taken for the first iterations of each level of `MQLP` in comparison to the time taken by `dKaMinPar-LP` in order to compute the whole clustering. As we can see, that for most instances the first iteration actually already performs worse than all iterations of `dKaMinPar-LP` combined. Exceptions to this are the rgg2d and rhg graphs in the strong scaling experiments, which perform increasingly better by the number of cores used. They start out displaying roughy the same times as their counterparts and scale to taking roughly half of the time. This means, that an algorithm optimizing only the second and third iteration could yield a better performing algorithm for the rgg2d and rhg graphs. For the other instances, we would have to improve the first iteration as well.

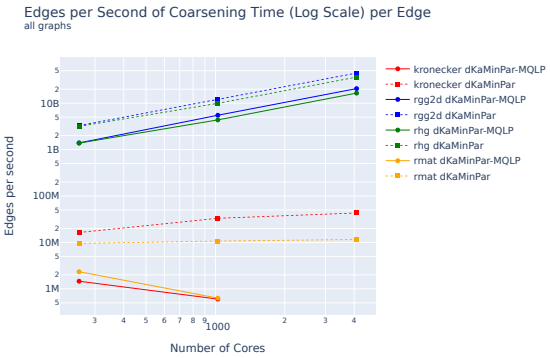

Figure 7.6: Weak scaling experiments' edges per second of coarsening time (throughput) for all graphs over the number of cores; MQLP (solid line), dKaMinPar-LP (dotted line); the graphs are scaled so that, that the number of nodes and edges per core is kept relatively constant

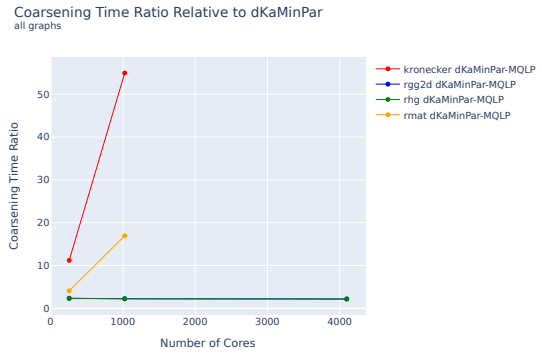

Figure 7.7: Weak scaling experiments' ratio of coarsening times relative to dKaMinPar-LP for all graphs over the number of cores; MQLP (solid line), dKaMinPar-LP (dotted line); the graphs are scaled so that, that the number of nodes and edges per core is kept relatively constant

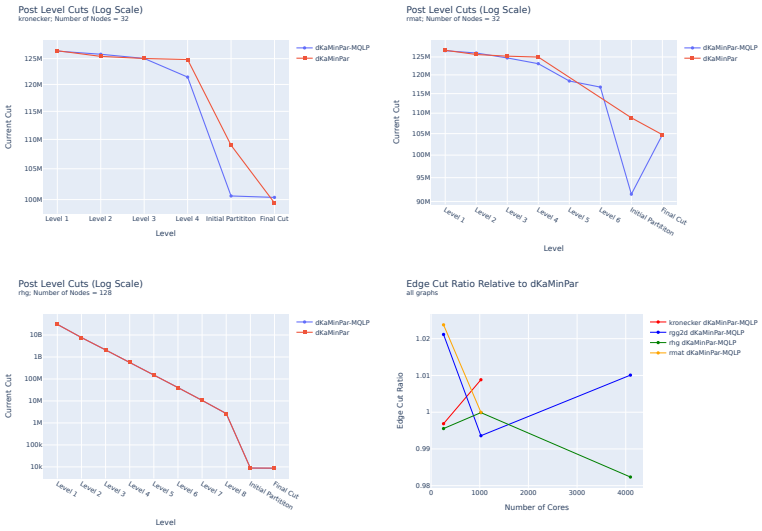

Figure 7.8: The cuts after each level and the initial partition as well as the final cut for the kronecker-32 (top-left), rmat-32 (top-right) and rhg-128 (bottom-left) instances; MQLP (blue line), dKaMinPar-LP (red line); the plot on the bottom-right shows the cuts of all graphs over the number of cores; MQLP (solid line), dKaMinPar-LP (dotted line); the points for 128 compute nodes of the kronecker and rmat graph are missing, since they did not finish before the time limit

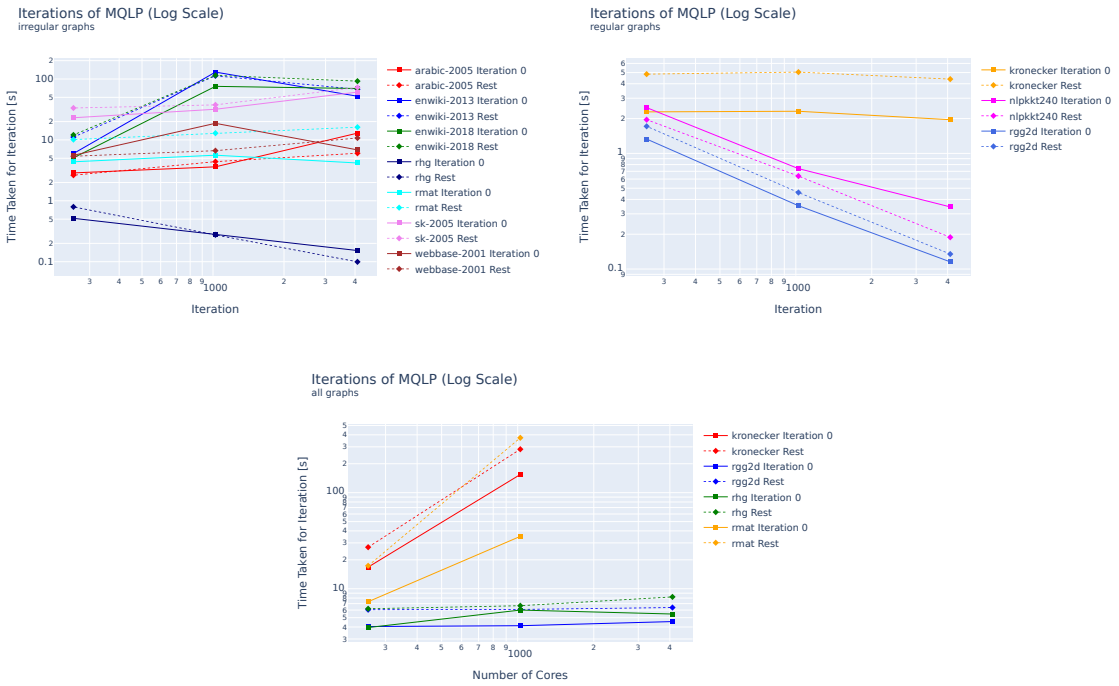

Figure 7.9: Iteration times for MQLP of strong scaling irregular instances (left), regular instances (right) and weak scaling instances (bottom) for all graphs over the number of cores; first iteration (solid line), 2nd + 3rd iteration (dotted line)

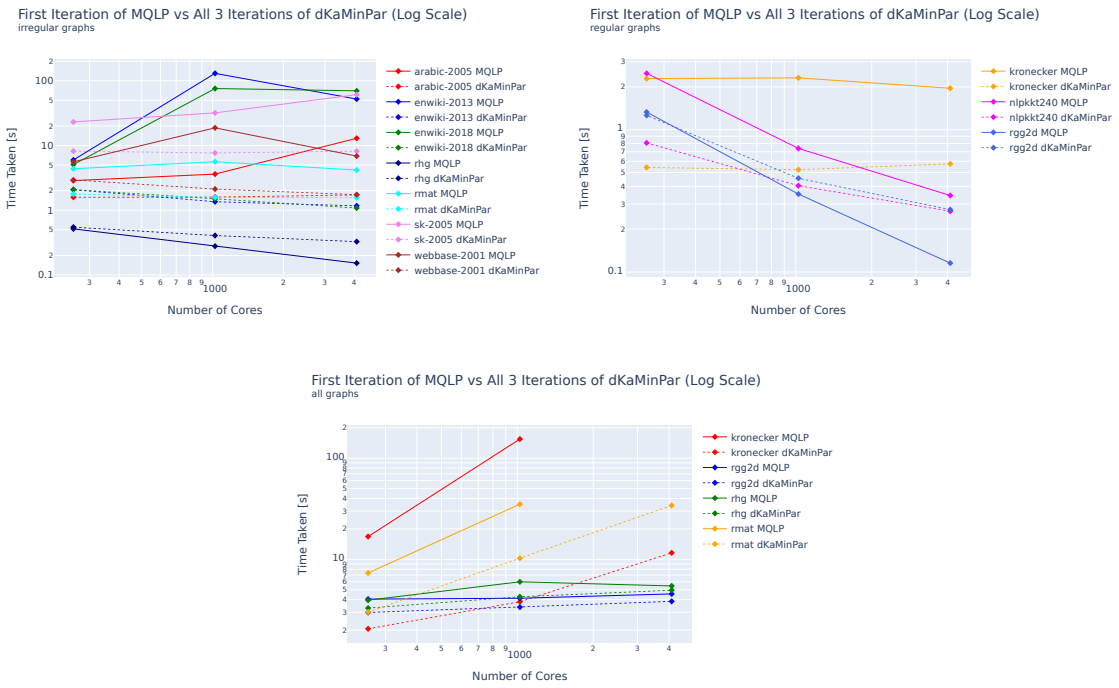

Figure 7.10: Iteration times of strong scaling irregular instances (left), regular instances (right) and weak scaling instances (bottom) for all graphs over the number of cores; MQLP's first iteration (solid line), all iteration of dKaMinPar-LP (dotted line)

8 Conclusion

Increasingly large graphs lead to the need of more scalable graph partitioners, that are able to process them. One approach to doing this is through the concept of multilevel graph partitioners. In the case of graphs that do not fit into the memory of a single machine, distributed multilevel graph partitioners become essential. We attempted to improve upon the performance of the commonly used label propagation algorithm. We designed and evaluated two approaches that perform the computation of labels asynchronously to the communication of new labels to the neighboring PEs.
Our first algorithm, called `ThreadedLP`, tries to do this by spawning an extra thread that progresses the computation, while the main thread communicates previously computed labels. It achieves similar quality and performance to `dKaMinPar-LP` and shows improved strong scalability on a very skewed graph (twitter-2010).
Our second algorithm, called `MQLP`, makes use of a message-queue framework, that encapsulates the MPI communication. The message queue is used to aggregate messages and overlap communication and computation through the message queue's internal use of non-blocking MPI operations. As we expected, we manage to compute more sparse clusterings during the coarsening phase. However we do not achieve this through more frequent updates and better label assignments, but rather due to the much worse approximation of the size-constraint, leading to highly imbalanced clusterings. The computed imbalance leads to higher cuts in the initial partition or in the final partition. In extreme cases, the algorithm also records coarsening times of up to roughly 100 times the amount taken by `dKaMinPar-LP`. For this algorithm, we only see weak and strong scalability for the rgg2d and rhg graphs and strong scalability for the nlpkkt240 graph.

8.1 Future Work

We could probably improve the performance of `ThreadedLP` even further, by also asynchronizing the communication in the cluster weight handling procedure. Also, the structure of the algorithm suggests, that concurrently processing the handling of labels and cluster weights should be possible, especially since the two procedures do not access the same chunks simultaneously.
Regarding `MQLP`, we may be able to improve its quality by using only one message queue and encoding the weight messages and label messages accordingly. By doing so, the weight messages should be "closer" to the event that caused their communication. A nice sideef-

fect is, that such a message queue only uses one global threshold instead of two. This means, that the weight and label messages get aggregated together and the messages can be sent more quickly without deteriorating the quality. For example, assuming two scenarios in which 20 weight messages and 20 label messages get posted in turns and an efficient threshold is 10. Now in the first scenario, we use two message queues with that efficient threshold, one for the label messages and one for the weight messages. The number of messages that need to be posted until a message queue communicates for the first time, is 19. The next message queue would start sending immediately after posting one more message, which also inefficiently makes use of the available bandwidth. Opposed to that, using only one message queue for both types of messages would always send in intervals of 10 messages and the first time messages get sent is also at an earlier time stamp. Combining this with the previously mentioned idea of using the active set strategy during the label propagation phase, might be able to produce a competitive algorithm. On a different note, using an indirection scheme that takes the distribution of the PEs to the different compute nodes into consideration, i.e., favouring communication between PEs on the same compute node and reducing the amount of communication between compute nodes, might increase the performance as well.

A Additional Plots for MQLP

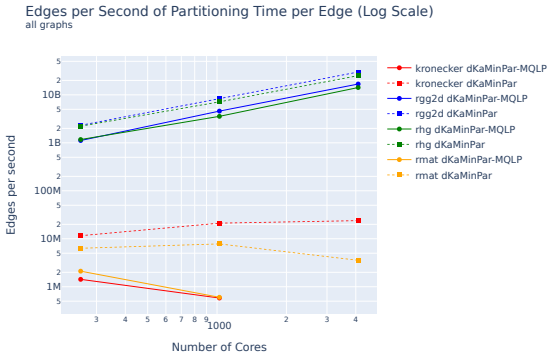

Figure A.1: Weak scaling experiments' edges per second of partitioning time (throughput) for all graphs over the number of cores; MQLP (solid line), dKaMinPar (dotted line); the graphs are scaled so that, that the number of nodes and edges per core is kept relatively constant

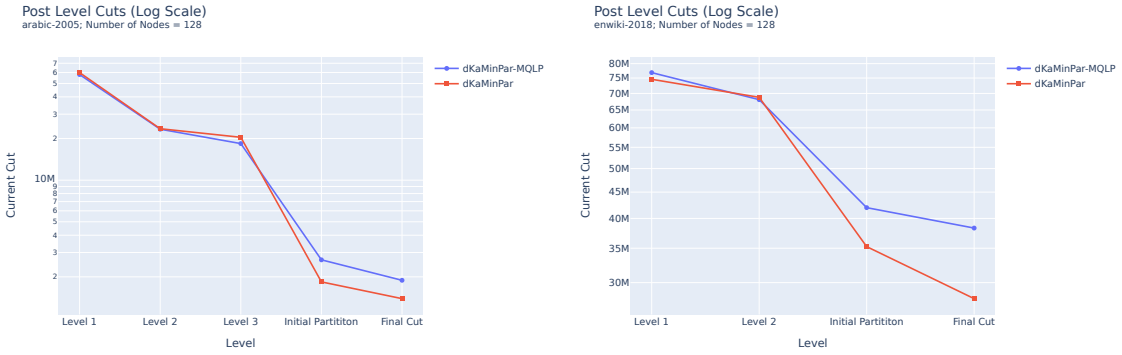

Figure A.2: The cuts after each level and the initial partition as well as the final cut for the arabic-2005-128 instance (left) and enwiki-2018-128 instance (right); MQLP (blue line), dKaMinPar (red line)

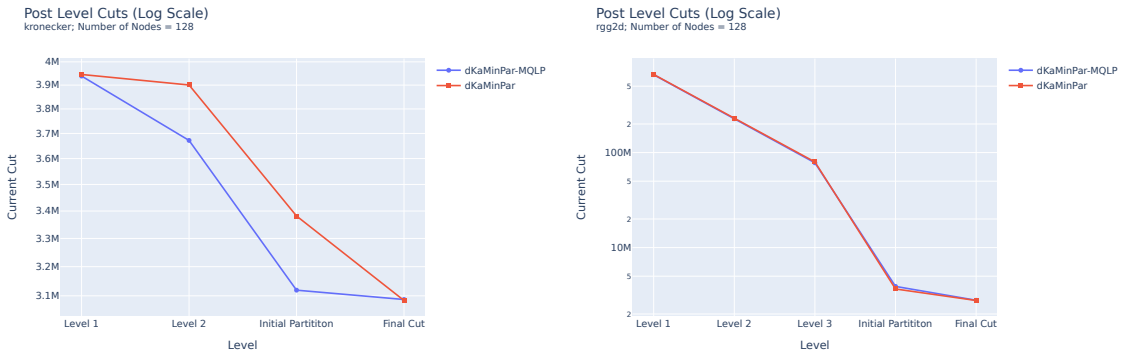

Figure A.3: The cuts after each level and the initial partition as well as the final cut for the kronecker-128 instance (left) and rgg2d-128 instance (right); MQLP (blue line), dKaMinPar (red line)

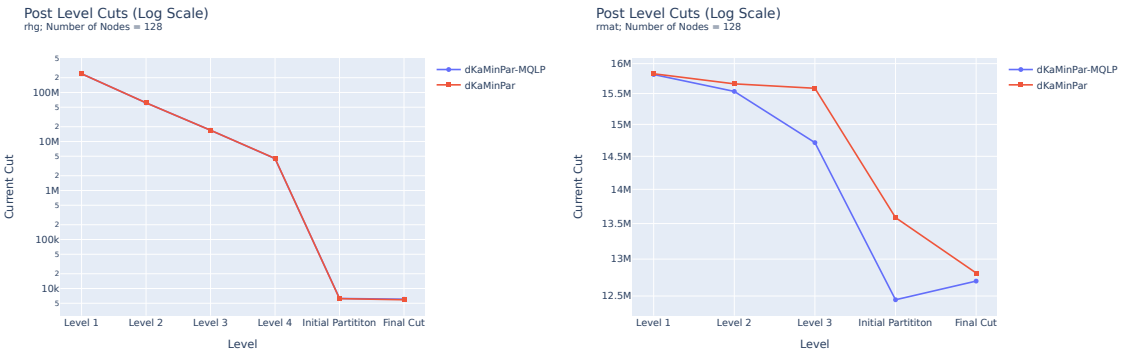

Figure A.4: The cuts after each level and the initial partition as well as the final cut for the rhg-128 instance (left) and rmat-128 instance (right); MQLP (blue line), dKaMinPar (red line)

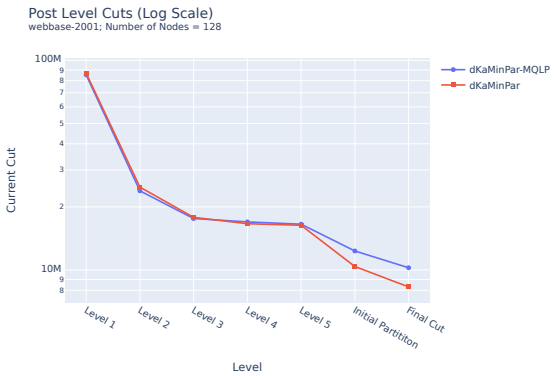

Figure A.5: The cuts after each level and the initial partition as well as the final cut for the webbase-2001-128 instance; MQLP (blue line), dKaMinPar (red line)

Bibliography

[1] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. *Engineering a direct k-way Hypergraph Partitioning Algorithm*, pages 28–42. Society for Industrial and Applied Mathematics, 2017.

[2] Charles-Edmond Bichot and Patrick Siarry. *Graph partitioning*. John Wiley & Sons, 2013.

[3] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. *Recent advances in graph partitioning*. Springer, 2016.

[4] Ümit Çatalyürek, Karen Devine, Marcelo Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, et al. More recent advances in (hyper) graph partitioning. *ACM Computing Surveys*, 55(12):1–38, 2023.

[5] Cédric Chevalier and François Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel computing*, 34(6-8):318–331, 2008.

[6] Timothy A Davis, William W Hager, Scott P Kolodziej, and S Nuri Yeralan. Algorithm 1003: Mongoose, a graph coarsening and partitioning library. *ACM Transactions on Mathematical Software (TOMS)*, 46(1):1–18, 2020.

[7] Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21 – May 25, 2018*, 2018.

[8] Michael S Gilbert, Seher Acer, Erik G Boman, Kamesh Madduri, and Sivasankaran Rajamanickam. Performance-portable graph coarsening for efficient multilevel graph analysis. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 213–222. IEEE, 2021.

[9] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Scalable shared-memory hypergraph partitioning. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 16–30. SIAM, 2021.

[10] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. Deep Multilevel Graph Partitioning. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms (ESA 2021)*, volume 204 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 48:1–48:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[11] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. Engineering a scalable high quality graph partitioner. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[12] Lorenz Hübschle-Schneider and Peter Sanders. Linear work generation of R-MAT graphs. *Network Science*, 8(4):543 – 550, 2020.

[13] Plotly Technologies Inc. Collaborative data science, 2015.

[14] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[15] Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent hash tables: Fast and general (?)! *ACM Transactions on Parallel Computing (TOPC)*, 5(4):1–32, 2019.

[16] Nikolai Merkel, Daniel Stoll, Ruben Mayer, and Hans-Arno Jacobsen. An experimental comparison of partitioning strategies for distributed graph neural network training. *arXiv preprint arXiv:2308.15602*, 2023.

[17] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Partitioning complex networks via size-constrained clustering. In *International Symposium on Experimental Algorithms*, pages 351–363. Springer, 2014.

[18] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel graph partitioning for complex networks. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2625–2638, 2017.

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

[20] Ilya Safro, Peter Sanders, and Christian Schulz. Advanced coarsening schemes for graph partitioning. *Journal of Experimental Algorithmics (JEA)*, 19:1–24, 2015.

[21] Peter Sanders and Daniel Seemaier. Distributed deep multilevel graph partitioning. *arXiv preprint arXiv:2303.01417*, 2023.

[22] Peter Sanders and Tim Niklas Uhl. Engineering a distributed-memory triangle counting algorithm. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023, St. Petersburg, FL, USA, May 15-19, 2023*, pages 702–712. IEEE, 2023.

[23] George M Slota, Cameron Root, Karen Devine, Kamesh Madduri, and Sivasankaran Rajamanickam. Scalable, multi-constraint, complex-objective graph partitioning. *IEEE Transactions on Parallel and Distributed Systems*, 31(12):2789–2801, 2020.

[24] Christian L Staudt and Henning Meyerhenke. Engineering parallel algorithms for community detection in massive networks. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):171–184, 2015.

[25] Trevor Steil, Tahsin Reza, Keita Iwabuchi, Benjamin W. Priest, Geoffrey Sanders, and Roger Pearce. Tripoll: Computing surveys of triangles in massive-scale temporal graphs with metadata, 2021.

[26] Trevor Steil, Tahsin Reza, Benjamin Priest, and Roger Pearce. Embracing irregular parallelism in hpc with ygm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Tim Niklas Uhl. Enabling scalability through asynchronous messaging and aggregation. unpublished manuscript by Tim Niklas Uhl, Karlsruhe Institute of Technology, uhl@kit.edu.

[28] Chris Walshaw and Mark Cross. Jostle: parallel multilevel graph-partitioning software–an overview. *Mesh partitioning techniques and domain decomposition techniques*, 10:27–58, 2007.