Karlsruher Institut für Technologie

Bachelor Thesis

How to Partition a Graph When You Think Like A Vertex

Jan Ebbing

Date of Submission: 07.12.2015

Supervisor: Prof. Dr. rer. nat. Peter Sanders
Dr. rer. nat. Christian Schulz
Dr. Darren Strash

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

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

Ort, Datum

Abstract

High-quality graph partitionings are useful for a wide range of applications, from distributing data of a social network to route planning to simulations using the finite element method. In the graph partitioning problem, the goal is to distribute all vertices of a given graph onto k blocks in a way that ensures every block has a weight below the maximum allowed weight of $(1+\varepsilon)$ times the average block weight and optimizes a certain metric - often minimizes the global cut, meaning the sum of the weights of all edges that run between distinct blocks.

In this thesis, we engineer a multi-level graph partitioning algorithm in Giraph using the perspective of a vertex so that no global view of the graph is necessary. Shortly summarized, our algorithm uses a label propagation algorithm to iteratively compute a clustering of the process graph and contract this clustering, partitions the coarsest level of that contraction using a centralized partitioning algorithm and then performs a local search iteratively on each level using the same label propagation algorithm as before to improve the partitioning. After introducing and explaining the algorithm, we present and evaluate results from experiments on a Hadoop machine. Compared to the main competitor Spinner in this framework, our algorithm computes partitions with a global cut that is lower by a factor of up to 10 while being slower by a factor of 3 to 20. In our experiments, our algorithm also met the balance constraint more often than Spinner.

Zusammenfassung

Graphpartitionierungen von hoher Qualität sind für eine große Zahl an Anwendungen nützlich, von der Verteilung der Daten eines sozialen Netzwerks über Routenplanung bis hin zu Simulationen mit der Finiten-Elemente-Methode. Beim Graphpartitionierungsproblem sollen alle Knoten eines gegebenen Graphen auf k Partitionen dergestalt aufgeteilt werden, dass das Gewicht jeder Partition unter dem erlaubten Maximalgewicht von $(1+\varepsilon)$ mal dem Durchschnittsgewicht liegt und eine gegebene Metrik optimiert - meist soll der globale Schnitt, also die Summe der Kantengewichte aller Kanten, die zwischen unterschiedlichen Partitionen verlaufen, minimiert werden.

In dieser Arbeit wurde ein Multi-Level-Graphpartitionsalgorithmus in Giraph aus der Sicht eines einzelnen Knotens implementiert, sodass keine globale Sicht auf den Graphen nötig ist. Kurz zusammengefasst berechnet der Algorithmus iterativ mit einem Label Propagation-Algorithmus eine Partitionierung des bearbeiteten Graphen und kontrahiert diese, partitioniert dann die kleinste Ebene mit einem zentralisierten Graphpartitionierungsalgorithmus und führt dann iterativ eine lokale Suche mittels des gleichen Label Propagation-Algorithmus auf jeder Ebene durch, um die erhaltene Partitionierung zu verbessern. Nach Einführung und Erklärung der Funktionsweise werden die Ergebnisse von Experimenten auf einer Hadoop-Maschine präsentiert und evaluiert. Verglichen mit dem Hauptkonkurrenten Spinner in diesem Framework berechnet unser Algorithmus Partitionierungen, deren globaler Schnitt um einen Faktor von bis zu 10 niedriger ist, mit einer Laufzeit die um etwa das fünf- bis zwanzigfache höher ist. In unseren Experimenten konnte die Balance-Nebenbedingung von unserem Algorithmus häufiger erfüllt werden als von Spinner.

Contents

1 Introduction **5**
1.1 Contribution of the Thesis . . . 6
1.2 Structure of the Thesis . . . 6

2 Preliminaries **7**
2.1 General Definitions . . . 7
2.2 The Think Like a Vertex Model . . . 7

3 Related Work **9**
3.1 Multi-Level Graph Partitioning . . . 9
3.2 Initial Partitioning with KaHIP . . . 11

4 Graph Partitioning in the Think Like a Vertex Model **13**
4.1 Clustering . . . 13
4.2 Contraction . . . 15
4.3 Initial Partitioning . . . 16
4.4 Local Search . . . 17

5 Implementation in Giraph **19**
5.1 Giraph Programming Concepts . . . 19
5.2 Spinner Variant . . . 21
5.3 Contraction . . . 23
5.4 Local Search . . . 25

6 Experimental Results **27**
6.1 Experimental Setup . . . 27
6.2 Results . . . 28

7 Conclusion & Outlook **31**
7.1 Future Work . . . 31

1 Introduction

To efficiently process large graphs such as modern social or biological networks, one needs to use parallel computing and thus distribute the graph on a cluster. In this context the running time of the vast majority of applications will be majorly influenced by the necessary communication effort of the computation. *Graph partitioning* can be used to optimize the distribution of the graph across a cluster so that applications will run with minimal communication effort [22].

To be able to store a graph on a cluster of equal machines, each node of the cluster has to store a certain part of the graph. Since each node only has limited memory, these parts of the graph need to be smaller than the memory of each node - this is taken into account by adding a restriction to only allow distributions that assign parts of the graph smaller than a certain limit to every node of the cluster. Additionally, to minimize the communication effort of the cluster, the goal of graph partitioning is to minimize the edges that run between distinct blocks. More specifically, the task is to assign every vertex of a graph to one of k blocks with the following restriction and goal:

- For every block, the weight of the block (which is the sum of the weights of each vertex in the block) must be $\leq (1+\varepsilon)\frac{|V|}{k}$ for a given ε
- The sum of the weights of all edges that run between distinct blocks is to be minimized

Thus, if you interpret vertices as data to be distributed as well as computation and edges as the necessary communication effort between data a high quality graph partitioning will improve performance of computations on this graph. For example Zeng et al. [33] evaluated their computed graph partitioning and compared it to the default hashing method for a PageRank iteration and reported speedups of roughly 3.

Since the graph partitioning problem is an abstraction of this use case there are also lots of other applications of the problem: Simulations using the finite element method [29], processes during VLSI design [3, 4], route planning [17, 11, 20] and scientific computing [10, 13] all lead to or benefit from solving a graph partitioning problem. One of these cases is example the iterative solving of the linear system $Ax = b$ on a parallel computer. During an iteration, the multiplication of a sparse matrix and a dense vector is computed. With a high-quality graph partitioning of the graph of the matrix A, communication effort can be significantly reduced as Kumar et al. [18] showed.

Our approach to solve these problems is to engineer a multi-level graph partitioning algorithm in a distributed environment. Multi-level algorithms are the most successful heuristic in centralized graph partitioning [16, 9] and consist of three phases. In the initial *coarsening* phase, details of the graph are omitted to create smaller graphs that still contain the "characteristics" of the input graph. In our algorithm, this is achieved by iteratively computing *contractions* of the graph that merge vertices and edges, reducing the size of the graph. Then, the smallest graph is *initially partitioned*, solving the problem on a small instance. Lastly, the coarsening is reversed and on each graph a local search to improve the current solution is performed.

We choose Giraph [2] as the framework of our algorithm since it is a highly scalable graph processing system following the Pregel model [21]. This has the advantage that we can use the partition we computed in the same framework, without needing to transfer the solution and redistribute the graph to a different graph tool. Also, Hadoop and Giraph are used in many cloud systems, making our algorithm available to many potential users. Giraph is an implementation of the "Think Like a Vertex" model which means that in each step of the algorithm we assume the perspective of a single vertex and decide what to do based on locally available data. To achieve this, we modify the label propagation algorithm Spinner introduced by Martella et al. [22] and add the required steps for a multi-level scheme. The original label

propagation algorithm proposed by Raghavan et al. [25] is used to find communities in complex networks and has a near-linear running time.

As a side note, there exist some variations to the metric that better judge the quality of a partition to reflect the communication effort of a given partitioning. In our primary example of distributing graphs among the nodes of a cluster the metric that has the highest impact on the communication overhead is the *maximum communication volume*. For a block V_i, we define the communication volume $\text{comm}(V_i) := \sum_{v \in V_i} c(v)D(v)$, where $D(v)$ is the number of distinct blocks v is adjacent to. Then the maximum communication volume is defined as $\max_i \text{comm}(V_i)$. This metric accurately reflects the overhead of a computation that distributes the data of each vertex across all blocks that have an edge to that vertex. Another variant of this approach is to look at the *total communication volume* which is defined as $\sum_i \text{comm}(V_i)$. The notation is from Buluç et al. [9]. They also give an overview of other objective functions and further literature on the topic.

There are other metrics that are more appropriate for specific applications but the global cut is the most commonly used and usually the standard metric to judge partition quality. A big advantage of the global cut is the easy definition and computation that makes optimizing the global cut of a graph partitioning easier than e.g. the total communication volume especially in distributed environments where the computation of globally interdependent values is sophisticated. Also the global cut often correlates with the other metrics and is thus preferred due to its simplicity [9]. Lastly, multi-level partitioning approaches have proven to produce very good results in practice but different levels of the graph can have different score function values which makes using them problematic. For these reasons, we will only consider graph partitioning that minimizes the global cut.

1.1 Contribution of the Thesis

We present the first multi-level graph partitioning algorithm in the Think Like a Vertex model. Using a label propagation algorithm based on Spinner, we coarsen the graph and later perform local search to improve the solution on each level. For initial partitioning we use KaHIP on one machine of the cluster. This allows us to compute partitions of very high quality which are vastly superior to current decentralized partitioners: Using the geometric mean, the global cuts our algorithm computes are on average 2.88 times smaller than the ones Spinner computes. We evaluate our algorithm in experiments on various graphs in comparison to Spinner. Due to the more complex nature of the approach as well as the coarsening phase having a communication requirement, the algorithm is quite a bit slower than Spinner, however.

1.2 Structure of the Thesis

The remainder of this work is organized as follows. In Section 2, we give an overview of the notations and exact definitions surrounding the graph partitioning problem used in this thesis. Moreover we explain the “Think Like a Vertex" programming model used by Giraph. In Section 4, we describe the algorithms used in this thesis. We will also discuss the time complexity and performance guarantees. Following on this Section 5 will discuss implementation details when engineering this algorithm in Giraph. In Section 6 we report on the experiments we performed with the partitioning algorithm. We discuss the overall improvement in solution quality as well as running time of our algorithm and compare it to Spinner. In the end, we summarize the main points and results of the thesis and give an outlook on the future work in the area of graph partitioning.

2 Preliminaries

2.1 General Definitions

An unweighted undirected graph with no self-loops $G = (V, E)$ consists of two sets $V = \{1, \ldots, n\}$ and $E \subset V^2$, $\forall v \in V : (v, v) \notin E$. Despite the graphs being unweighted we still define weight functions for vertices and edges: $c\colon V \to \mathbb{R}_{\geq 0}$, $\omega\colon E \to \mathbb{R}_{\geq 0}$ with $\forall v \in V : c(v) = 1, \forall e \in E : \omega(e) = 1$. In the later stages of our algorithm we will use these functions.

The *degree* of a vertex $v \in V$ in $G = (V, E)$ is $\deg(v) := |\{(v, x) \in E\}|$. We also use the *weighted* degree $\deg_\omega(v) := \sum_{(v,x)\in E} \omega(v, x)$. The *neighbourhood* of a vertex v is denoted by $N(v) := \{(v, u) \in E\}$. A graph G is *connected* when there is a path between every pair of vertices. For the rest of the thesis we assume the graph is connected.

To ease notation, we write $n = |V|$, $m := |E|$, and for any set $V' \subseteq V : c(V') := \sum_{v\in V'} c(v)$, likewise for all $E' \subseteq E : \omega(E') := \sum_{e\in E'} \omega(e)$. A *clustering* of a graph $G = (V, E)$ are pairwise disjoint sets of vertices $V_1, \ldots, V_j$ with $\bigcup_i V_i = V$. A *partitioning* of a graph $G = (V, E)$ into k partitions (for a given k and ε) is a clustering $V_1, \ldots, V_k$ of G with the added restriction

$$\forall 1 \leq i \leq k : c(V_i) \leq (1 + \varepsilon)\frac{c(V)}{k}. \tag{2.1}$$

Note that there are instances in which (2.1) can never be fulfilled, such as graphs with few vertices that have one vertex with an extremely high weight. To guarantee solvability you can change (2.1) to:

$$\forall 1 \leq i \leq k : c(V_i) \leq (1 + \varepsilon)\frac{c(V)}{k} + \max_{v\in V} c(v). \tag{2.2}$$

(2.1) is known as the *balancing constraint* and restrains arbitrary partitionings of graphs to "useful" ones since it guarantees a certain degree of balance between blocks. Also note that our restriction to unit weights for the input graph alleviates the solvability issue, for example for all input graphs with $n \bmod k = 0$ the graph partitioning problem is solvable for all $\varepsilon \geq 0$. For the rest of the thesis, we will ignore it and assume the problem to be solvable. The current partitioning will be represented as a function $p\colon V \to \{1, \ldots, k\}$.

The parameter ε allows for some variety and a limited imbalance that can improve the quality of a partition. Since all valid partitionings for an ε are also valid for any $\varepsilon' > \varepsilon$, increasing the ε increases the size of the solution space and thus can improve the metric we are trying to optimize. As stated in the introduction, we want to minimize the global cut $\sum_{i<j} \omega(E_{ij})$ with $E_{ij} = \{(u, v) \in E : u \in V_i, v \in V_j\}$.

In our algorithm, the input graph will have unit node and edge weights, but will be transformed into one having non-trivial edge and vertex weight functions in the multi-level scheme. Theoretically, the input graph could have vertices or edges with differing weights. To keep the notation simple, $G = (V, E)$ will refer to the current state of the graph during coarsening and uncoarsening. The only exceptions to this are when we explain the transition between levels, where G and G' will denote the two subsequent levels of the graph.

2.2 The Think Like a Vertex Model

In this thesis, we implement our algorithm using the "Think Like a Vertex" (TLAV) programming model. One of the first TLAV frameworks was the Google Pregel model [21] that allowed for iteratively executing graph algorithms on large graphs in a scalable way. It is in

turn based on the Bulk Synchronous Parallel model for parallel computation [31]. Pregel was developed due to the increasing demand for graph and network analysis which required a distributed and parallel computation due to the size of the graphs. Existing distributed models like MapReduce were not suited to process large graphs, e.g. in MapReduce the graph was written to and read from disk in between each superstep, resulting in poor performance due to the overhead as well as other problems. The alternative of programming in MPI was prone to errors and still contained many low-level details.

In the TLAV model the programmer writes code from the perspective of a single vertex of the graph that is then executed on all vertices in parallel. An algorithm is divided into iterations called *supersteps* which roughly consist of the execution of one method on all vertices of the graph. Information is exchanged between vertices using message passing for which the sender needs to know the ID of the receiver.

We will now quickly cover the core principles necessary to implement an algorithm in the TLAV model. For more detailed information how the implementation works refer to the original Pregel paper [21].

The core idea of the TLAV model is that graph algorithms are implemented from the perspective of a single vertex and executed in parallel on all vertices at the same time. Each vertex has information it can store locally, depending on the algorithm, as well as edges to all its neighbours which also offer the space to store information. Synchronisation is achieved through several mechanisms. A graph algorithm is organized in *supersteps*, meaning that after one execution of the code the framework will wait until all vertices have finished and then start the new superstep. This allows the sending of *messages* between vertices. A vertex can send a message containing information to any other vertex as long as it knows the ID of the receiver. In practice, this usually means that messages are exchanged in some subset of the close neighbourhood of a vertex, although it is possible for two arbitrary vertices to start communicating (one could even imagine random communication). The synchronisation via supersteps is implemented so that messages sent in superstep i will be delivered to the vertices at the beginning of superstep $i + 1$ so they can use the contained information in their computation of the current step. Each vertex is also able to retrieve the ID of the current superstep to switch phases of the algorithm or something similar.

In this thesis, we use Giraph as an open source implementation of the Pregel model. Giraph is a graph processing framework designed for scalability and is used for example by Facebook to analyse graphs with up to a trillion edges. It runs on top of Hadoop, an implementation of the popular MapReduce framework. Hadoop serves both as a distributed file system to store large amounts of data redundantly on a cluster and a processing framework for that data. Both Hadoop and Giraph are written in Java and developed as top level projects by the Apache Software Foundation. More information on Giraph can be found on its website [2].

In Giraph, the Pregel model was extended by adding sharded aggregators that are able to store global variables in a scalable way. Hereby, the aggregators are distributed across workers in the cluster and changes to the variable are batched at the end of a superstep. This allows for more applications to be implemented but should be utilised sparingly since it requires synchronisation and will be sent over the network most of the time. Giraph also added a central instance to regulate the control flow of the algorithm that we explain in detail in Section 5.2.

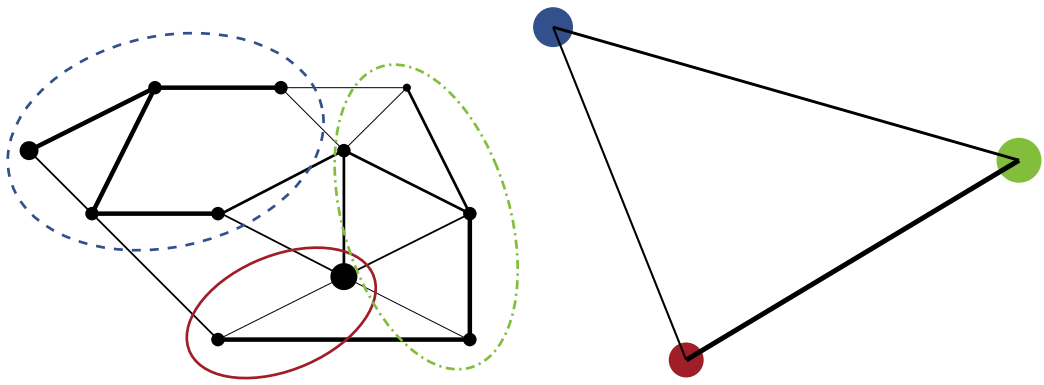
3 Related Work

Graph partitioning, also called k-way partitioning, denotes the problem of putting each vertex of a graph into one of k blocks in a way that maximizes the quality of the partition. The partitioning has to fulfil the *balancing constraint* introduced in Section 2.1 for a given ε and should minimize the total cut. That way, data represented as a graph can be distributed among several machines while minimizing the dependencies/communication between nodes. The corresponding decision problem to graph partitioning is NP-complete [14] and since the graphs worth distributing are big, heuristics are used in practice. It has also been shown that even finding approximations of guaranteed quality is NP-hard in some cases, for greater detail refer to [9, 14, 5]. There is a huge amount of work done on graph partitioning which led to an enormous diversity of heuristics used, from branch-and-bound over flow algorithms that are based on computing maximum flows of a graph to spectral partitioning which performs computations on the adjacency matrix of the input graph. For a more detailed overview, we refer to [9, 29, 6].

3.1 Multi-Level Graph Partitioning

In multi-level approaches, the coarsening phase usually consists of iterative *contractions* of a graph $G = (V, E)$. A contraction of a graph merges (non-empty) subsets of V as well as their incident edges by replacing them with a new vertex/edge with a weight equal to the sum of the weights of the original vertices/edges. Each contraction operates on a higher level of the graph, beginning with the input graph, and computes the next lower level at which point the process is repeated (after clustering the new level). Figure 1 depicts a simple contraction of a small graph.

Contraction is one of the most commonly used methods for the coarsening phase. In the sequential case, many algorithms compute matchings on the graph and contract the matched edges which halves the number of vertices per contraction. To keep the explanation more general, assume that for every vertex $v \in V$ there is a new vertex $\mathcal{C}(v)$ which denotes the vertex v is contracted to. $\mathcal{C}$ would then be the contraction function, assigning (non-empty) subsets of V to the same new vertex. Then the weight of the new vertices is computed as $c(\mathcal{C}(v)) = \sum_{w \in V:\ \mathcal{C}(w)=\mathcal{C}(v)} c(w)$. This simply means that the contracted vertex has the weight of all its vertices combined. The edges are a similar case, for every edge (v, w) we add an edge $(\mathcal{C}(v), \mathcal{C}(w))$ with the same weight. This will usually create parallel edges, in which case we



(a) Clustered graph (b) And the newly created level

Figure 1: A small example of a contraction.

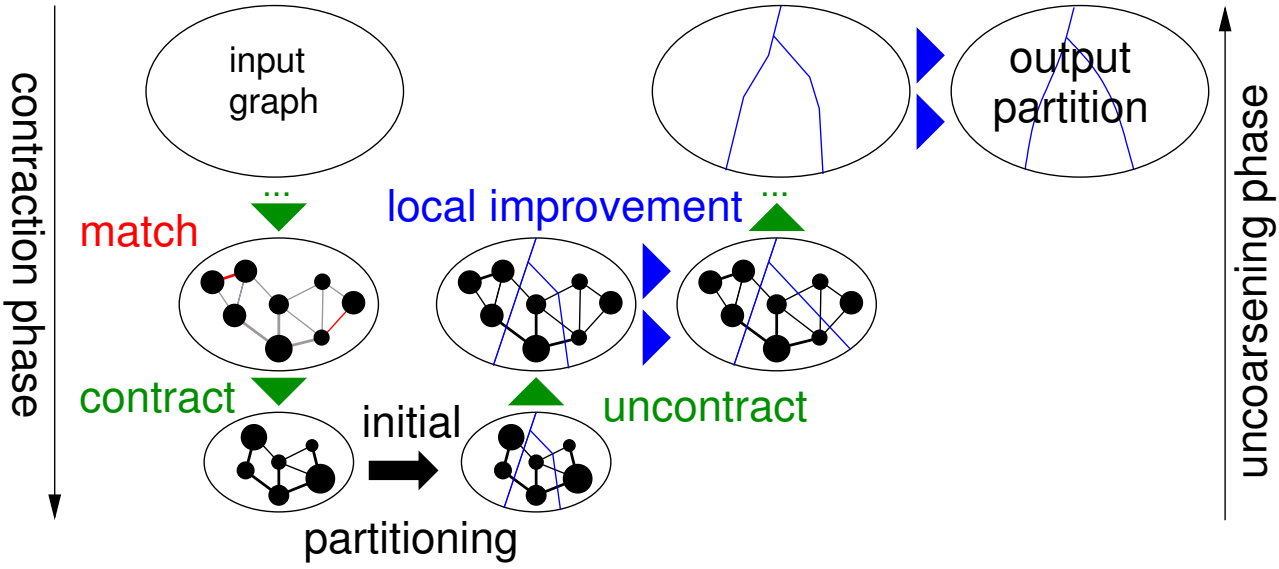


Figure 2: An overview of the general multi-level graph partitioning scheme, taken from [30].

merge the parallel edges by adding up their weights. More specifically, for each edge (v, w) there is a contracted edge $(\mathcal{C}(v), \mathcal{C}(w))$ and one has $c(\mathcal{C}(v), \mathcal{C}(w)) = \sum_{(x,y)\in E:\mathcal{C}(x)=\mathcal{C}(v),\mathcal{C}(y)=\mathcal{C}(w)} c(x, y)$. For a graphic explanation, see Figure 1. Note that we do not use matchings to coarsen the graph. With this definition of contraction, it is easy to see that cuts (referring to the edges inducing the cut) in a lower level of the graph have the same weights as the induced cuts in the original graph and clusters in a lower level have the same weight as the induced clusters in the original graph. This property is important to warrant the multi-level approach.

This method ensures that partitions of the coarsest level have the same global cut and imbalance as the corresponding clustering on the input graph. By contracting the graph iteratively, we get several levels of the graph. This also shrinks the graph, so we can use more expensive partitioning algorithms (which are expected to yield high-quality partitionings) on the most contracted level, use that partitioning one level above and then run the cheaper label propagation algorithm to improve the current solution.

For a given graph G and the allowed imbalance ε, we define the *capacity* C of a block as the maximum allowed block size: $C = (1 + \varepsilon)\frac{c(V)}{k}$. An important property of the state of the current partition p is the size of a block with label l: $b(l) = \sum_{v\in V} \delta(l, p(v))c(v)$ where δ is the Kronecker delta.

Our label propagation algorithm that computes a clustering is based on Spinner [22]. Spinner is a single-level graph partitioning algorithm in Giraph that extends the label propagation algorithm to compute size-constrained clusters. However, Spinner does not use the exact same problem definition we do. Instead of balancing clusters based on the weight of the contained vertices, it balances clusters based on the weight of the contained edges. For this, they change the balancing constraint to

$$\forall 1 \leq i \leq k : \sum_{v\in V_i} \deg(v) \leq (1 + \varepsilon)\frac{|E|}{k}.$$

In the first step every vertex chooses a partition randomly from $\{1, \ldots, k\}$ and sends this information to its neighbours. Then the label propagation starts: Each vertex v computes the score of all labels l according to

$$score''(v,l) = \sum_{u \in N(v)} \frac{w(u,v)\delta(p(u),l)}{\sum_{u \in N(v)} w(u,v)} - \frac{b(l)}{C}$$

where δ is the Kronecker delta. This score function consists of the objective function that maximizes the number of edges to the current partition of the vertex and the penalty term that punishes partitions that are almost full or even overloaded. If a different partition than the current one has the highest score, the vertex will be marked as a candidate.

In the next step, all vertices that were marked as a candidate compute if they will change their label. For this purpose, all of those vertices request the size of the block they want to switch to $b(l)$ as well as the amount of candidate vertices that want to switch to that block in total $m(l)$. Then they compute their switching probability as

$$p = \frac{C - b(l)}{m(l)}.$$

Note that with this probability, the expected value of block sizes of blocks that are overloaded is exactly C. This is further amplified by the law of large numbers and the scale of the graphs we partition in Giraph. When a vertex switches partitions, the global counters are updated and it sends a message to all of its neighbours containing its new partition. The algorithm halts once the global score (calculated as the sum of the scores of each vertex) does not improve by a certain value Δ in a maximum of w steps. Due to the nature of the label propagation algorithm used in Spinner, a partitioning can only be optimized locally. This means partitioning quality is not that high. Also, due to the linear penalty function, there is no guarantee the balance constraint will be fulfilled and in practice it will not always be fulfilled as we will see in the experiments.

3.2 Initial Partitioning with KaHIP

We use KaHIP to initially partition the coarsened graph. Hereby the following advanced techniques are employed to compute an initial partition of high quality. This overview is an adaptation of [27, 26].

FM algorithm. The general local improvement algorithm KaFFPa is a variant of the FM algorithm [12]. It is a series of rounds with each round traversing the graph in a random order, skipping vertices that are incident to only one block. The visited vertices are put into a priority queue based on the maximum gain in edge cut one would achieve by moving them to a different block. The first vertex in the queue is then moved to the corresponding block and its neighbours are added to the queue. Each vertex can only be moved once per round. After a stopping criterion, the best found partitioning used for the rest of the rounds. Then a new round begins until no improvement occurs during a round.

Max-Flow Min-Cut Local Improvement. To further improve the FM variant, KaHIP also employs more advanced techniques such as Max-Flow Min-Cut improvement. This technique creates a flow problem around vertices at block boundaries such that every $s-t$ cut corresponds to a valid bipartition in the original graph. Then a minimal cut in the area is computed, improving the original partition. This method is extended by multiple runs, increasing the search area or applying heuristics that also factor in balance. These techniques are covered in [27].

Multi-try FM. This method moves vertices between blocks to achieve a lower cut. To further decrease the cut and be able to escape local maxima, the multi-try FM is initialized with a single node, generating a more localized search than previous approaches initializing with all boundary nodes. A more detailed explanation is given in [30, 27].

Iterated Multi-Level Algorithms. This is a metaheuristic introduced for graph partitioning by Walshaw et al. [32]. Using different seeds for a random number generator, the method iterates through coarsening and uncoarsening steps trying to find an optimal solution. If the current partition is not good enough, the uncoarsening can be reversed and a different uncoarsening option can be chosen on a lower level. If a coarsening matches an edge between two blocks, it is not contracted so that the partition can still be used on the coarsest level if the contraction is being reversed. This ensures that partition quality will never decrease since the local improvement also guarantees that the global cut will never increase. For more detail, e.g. on the used F-cycles, we refer to [27].

4 Graph Partitioning in the Think Like a Vertex Model

In this section, we present the general concept of our multi-level graph partitioning algorithm.

To partition a given graph $G = (V, E)$ our multi-level scheme first computes a clustering of the graph using a modified version of the Spinner algorithm [22]. In turn the Spinner algorithm is based on the *label propagation algorithm* which is changed to produce balanced partitions with a low edge cut. Spinner can be parallelized very well in the Pregel model and requires very little global communication or synchronization. Our version of the Spinner algorithm starts with a high amount of blocks (usually n, depending on the graph and the cluster) a vertex can belong to and then merges these blocks step by step. When the global score does not improve by a factor of at least Δ after w steps – both values are tuning parameters – the current clustering is contracted and a new level is created. We then run our modified Spinner version on that level and repeat these two steps until the resulting level is smaller than a certain halt size n_h. This approach of contracting clusterings computed by a label propagation algorithm is described by Meyerhenke et al. [23] in a centralized setting. We follow this idea and adapt it to the Think Like a Vertex model. After coarsening, we import this coarsest level into KaHIP and compute an initial partitioning with it. KaHIP is a centralized graph partitioning tool that requires a global view of the graph but produces results of very high quality. It is described in detail by Sanders et al. [28].

Lastly, we import this initial partitioning of the coarsest level into our extended input graph to try and improve it on every level by performing a local search. This local search uses the same label propagation algorithm we used in the coarsening phase to compute a contraction. Once the label propagation fulfils the stop criterion on the level of the original input graph the whole algorithm halts and prints out the computed partitioning.

4.1 Clustering

For the contraction phase we receive an unweighted graph G as input with $V = \{1, \ldots, n\}$. We also receive the parameters ε, k of the graph partitioning problem as well as the tuning parameters w, Δ and n_h. Our label propagation algorithm consists of two phases that will be repeated alternately. Firstly we will compute a clustering of the current level using our label propagation algorithm. After this algorithm converged, we contract all the blocks to single vertices and thereby create the next level of the graph. This process is repeated until the current level is small enough to be partitioned by KaHIP (specified by the halting size n_h). The clustering phase starts with the initialization: Every vertex v of the current level sets its initial label as its ID. Each vertex also stores the labels of its neighbours in the corresponding edge.

Algorithm 1: Coarsen Graph

`Data`: Input graph G, k, ε, n_h, Δ
`Result`: Input graph G together with several levels of iterative contractions of G

```
1 while G.size > n_h do
2     InitializeClustering(G)
3     while !stopcriterion do
4         ComputeLabels(G, ε)
5         SwitchLabels(G)
6     G = ContractLevel(G);
```

Algorithm 2: InitializeClustering

```
Data: Graph consisting of multiple levels with the current level G = (V, E)
Result: Labels for every vertex v ∈ V
for v ∈ V do
    p(v) := v
    for e = (v, u), u ∈ N(v) do
        storeLabel u FOR u
```

After this the label propagation starts. In the first of the two steps every vertex iterates over its received messages and updates the labels that are stored in the edge to the sender (A message contains the ID of the sender as well as the new label of the sender). Following this, every vertex only continues executing the current step with a certain probability starting at 50% and increasing with every step. This is to avoid an endless loop that can occur with our changed way to initialize the Spinner variant by adding "stability" to the graph, i.e. not literally every vertex changes partition in the first step. We will discuss the endless loop at the end of Section 5.2. If the vertex continues execution it computes the scores of all neighbouring partitions. We define the score function as:

$$score(v, b) = \sum_{(v,w)\in E'} \frac{\omega(v, w)\delta(b, p(w))}{\deg_\omega(v)} - \frac{l(b)}{C}.$$

This score function optimizes the local cut for every vertex and a simple linear penalty term proportional to the block size to cause balance. This is very similar to the Spinner score function, however, our blocks are vertex size-constrained while Spinner uses edge weights to measure balance. Lastly, we define the global score of a level according to intuition $score(G) := \sum_{v\in V} score(v, p(v))$.

If the vertex received a message in this step and the current label of the vertex is not among the labels with maximum score the vertex chooses one of the labels with maximum score at random and becomes a *candidate* to switch to that label. This means the vertex is *marked* as a candidate as well as increasing a global variable counting the amount of vertices that want to switch to a certain label. If the current label has the highest score it is always preferred. This concludes the first of the two label propagation steps. In the pseudocode, $M(v)$ denotes the set of messages received by a vertex v in the current superstep.

In the second step the vertices migrate between labels. We need to split this up into two steps to be able to fulfil the balancing constraint. At the start of the second step every vertex checks if it is marked. If that is not the case the step ends. If it is the case, however, the vertex computes its probability to change labels. Since we stored the amount of vertices that want to switch to a label l (we denote this by $candidates(l)$) in the last step and always have the information how many vertices can switch to label l without violating the balance constraint (denoted by $freeCapacity(l)$) this allows for a probabilistic approach. We set the changing probability of a vertex to label l to $p(l) = \frac{freeCapacity(l)}{candidates(l)}$. If a vertex ends up switching its label it sends a message to all its neighbours containing its ID and its new block and we go back to step one. Label propagation stops once the current global score was not better than the best global score so far by a factor of at least Δ in a maximum of w steps. After label propagation halts the contraction of the graph begins.

4.2 Contraction

In the first step, vertices are created. Each vertex checks if its corresponding block is empty and if it is not, a corresponding vertex is created in the new level. The vertex is initialized with weight 0.

After the contracted vertices have been created, each vertex sends its weight to its corresponding contracted vertex. This allows them to sum up the vertex weights easily.

In the following superstep the contracted vertices sum up all their received messages and set their new vertex weight to that value while the higher level sends a message for every outgoing edge to its corresponding vertex. This message contains the target ID and the weight of the edge.

Finalizing the creation of the new level each new vertex collects all these edge messages,

Algorithm 3: ComputeLabels

```
Data: Current level G = (V, E) with every vertex belonging to a block
Result: G with appropriate vertices being marked as a candidate
for v ∈ V do
    for Message m : M(v) do
        e = (v, m.targetID)
        storeLabel m.newLabel FOR m.targetID
    if [0, 1] ∋_rand p ≤ participationProbability() then
        newLabel = v.computeScores(ε)
        if p(v) ≠ newLabel then
            markAsCandidate(v WITH newLabel)
```

Algorithm 4: SwitchLabels

```
Data: Current level G with some vertices being marked as candidates
Result: G with the respective vertices having switched labels and sent the appropriate
        message to their neighbours
for Vertex v ∈ G do
    if isMarked(v) ∧[0, 1] ∋_rand p ≤ calcSwitchProbability() then
        p(v) := getCandidateLabel(v)
        updateAggregators()
        sendMessageToAllEdges WITH (v, p(v))
        unmarkVertex(v)
```

Algorithm 5: ContractLevel

```
Data: Level G with a computed partition stored in the vertices
Result: New level G that is a contraction of the old G
Graph G'
ContCreateVertices(G, G')
ContSendVWeights(G, G')
ContSendEWeights(G, G')
ContFinalize(G')
return G'
```

Algorithm 6: ContCreateVertices

```
Data: Current level G = (V, E), new level G'
Result: Vertices of the contraction of G in G'
for v ∈ V do
    if getBlockLoad(getCorrespondingBlockID(v)) then
        G'.addVertex(createVertex(getCorrespondingVertexID(v))
```

Algorithm 7: ContSendVWeights

```
Data: Current level G = (V, E), new level G'
Result: Messages from G to G' containing the vertex weights
for v ∈ G do
    sendMessage TO C(v) WITH v.getWeight())
```

Algorithm 8: ContSendEWeights

```
Data: Current level G = (V, E), new level G' = (V', E'
Result: Level G' with accurate vertex weights, messages from G to G' containing edge
        weights.
for v ∈ V' do
    for Message m ∈ M(v) do
        v.weight += m.getWeight()
for v ∈ V do
    for e = (v, u), u ∈ N(v)) do
        sendMessage TO C(v) WITH (C(u), e.getWeight())
    setInactive(v)
```

Algorithm 9: ContFinalize

```
Data: New level G' = (V', E')
Result: G' is a contraction of G and therefore used as the current level
for v ∈ V' do
    for Message m ∈ M(v) do
        e = (v, m.getTargetId())
        e.weight + = m.getWeight()
```

sums up the edge weights of edges to the same partition and sets the weight of the edge to the corresponding vertex to the sum of the received weights. After this step, the clustering starts anew.

4.3 Initial Partitioning

After contracting the graph to a level $G = (V, E)$ with $|V| < n_h$ we use KaHIP to compute an initial partitioning. KaHIP [28] is a graph partitioning framework developed by Peter Sanders and Christian Schulz at KIT. It contains several graph partitioning algorithms. we use KaFFPa (Karlsruhe Fast Flow Partitioner) to compute an initial partitioning locally on one node of the

Hadoop cluster. This parameter can be changed according to the memory and speed of a node to facilitate the best initial partitioning possible. The coarsest level of the graph is copied to node, partitioned and then copied back into the distributed file system.

KaFFPa itself uses a multi-level graph partitioning algorithm as well. Note that this algorithm possesses a global view and is thus able to employ advanced techniques during the coarsening and local improvement phases. The coarsening is performed by iteratively computing a matching of the graph and then contracting the vertices adjacent to the matched edges. The initial partitioning is performed by Scotch [24] or Chaco [15]. The focus of KaFFPa is however on the local improvement methods presented in Section 3.2. After transferring the initial partition we calculated with KaHIP to the multi-level graph our algorithm then continues with the local search phase. Note that the initial partition from KaHIP is guaranteed to have exactly k blocks since we started with n blocks in the coarsening phase and the number of blocks at the end of the coarsening phase is non-deterministic.

4.4 Local Search

The local search phase is divided into two phases. In the first part we transfer the partition of a coarser level to the level above. The second part is the actual local optimization and that improves the partition on the current level. For this we use the same label propagation algorithm we also used to compute a clustering in the coarsening phase.

Transferring a partition from a high level to one level above. In this phase we consider two levels G and G' with G being the coarser level and already partitioned, either from the initial partitioning phase or from this phase. At the start each vertex in G' sends a message to its corresponding vertex in G. In the next step the corresponding vertices reply to the messages and send their current label to the higher level. In the last step each vertex in the higher level updates its partition with the received value.

Local optimization. In this phase we only consider one level $G = (V, E)$. We employ the label propagation algorithm from Section 4.1 with minimal changes. We do not need initialization since we receive the partition either from KaHIP or from the level above. Again each vertex computes the score of neighbouring blocks and migrates to the best block with a probability depending on the amount of vertices that want to switch to that block. We do not need to disable some vertices in the `LabelPropagationComputeLabel` step to avoid the endless loop in this step. This is due to the fact that we operate on a clustering in the coarsening phase whereas KaHIP gives us a partition. This leads to only few migrations per superstep, mitigating the back and forth between the same two states. Also note that in this phase our algorithm is very similar to Spinner.

We use the same stop criterion as well, halting when there was no improvement by at least Δ in the last w steps. When the stop criterion is met, we transfer the partitioning one level below again. Once the stop criterion is met on the lowest level – the input graph – the complete algorithm halts and the partition is put out.

Due to KaHIP's partitioning and label propagation not creating any new blocks, there are only k blocks in the output as opposed to the n blocks during coarsening. There is no strict balance guarantee, however, balance is achieved through the probabilistic measures introduced by Spinner such as the probabilistic migrations and the penalty term in the objective function.

5 Implementation in Giraph

In this section we explain the implementation of our algorithm in detail. We start by giving a brief introduction to programming in Giraph and then explain the changes we made to the Spinner label propagation algorithm to suit our needs as well as the implementation of the multiple levels.

5.1 Giraph Programming Concepts

Giraph is the most widely-used graph processing framework for large-scale computing and was our first choice to implement our algorithm. It allows for easy implementation of graph algorithms using the perspective of a local vertex and thus high scalability due to following the Pregel model. It is for example used by Facebook since 2012 to analyse their graphs with up to a trillion edges. In the following we present the most important parts of the framework that the algorithm is run in.

A graph algorithm in Giraph consists of a series of *supersteps*. At the beginning of a superstep each vertex receives *messages* sent by other vertices in the last superstep. Then, every vertex executes the code of the algorithm and may send messages to other vertices during this period. At the end of the superstep, it is ensured that all vertices have finished the current superstep – since every vertex executes the code in parallel – and messages are being delivered for the next superstep. The main graph algorithm is implemented as a subclass of the `AbstractComputation` class. It provides a `compute()` method that will be executed by each vertex in the graph in each superstep. The compute method has two parameters, the vertex that executes the code and the messages that this vertex received in this superstep. To implement more complex graph algorithms that for example have different phases, there are several possibilities. One can use the `getSuperstep()` method to retrieve the current superstep and make a case differentiation this way. Every vertex also has a user-defined value field that can store information. In the `compute()` method it is also possible to add vertices and edges to the graph. The created objects are passed as a parameter in `addVertexRequest()`/`addEdgeRequest()` and are added to the graph before the next superstep.

To be able to structure graph algorithms better – among other reasons – the `MasterCompute` class was created. It is a centralized master instance and contains a `compute()` method that will be executed once each superstep, before the vertices start with their current superstep. In this class it is possible to choose an `AbstractComputation` to be used from this superstep on. This allows for better object-oriented programming by writing simple `AbstractComputation` classes and then using a `MasterCompute` class to handle the control flow of the algorithm and switch between phases. It also allows storing values over time as static fields to allow more complex computations.

One of the most important aspects of the `MasterCompute` class, however, is that it contains the method necessary to register *Aggregators*. Aggregators are global variables combined with a binary operation like $\min, \max, +$ and are necessary for most non-trivial graph algorithms. For example, in graph partitioning the balancing constraint can not be locally checked which creates the need for aggregators. Before it is possible to use an aggregator it has to be registered in the `initialize()` method of the `MasterCompute` class. This requires the name and type of aggregator to be set. The name is basically an ID of the aggregator and is used to store and retrieve values from it. The type of the aggregator is a subclass of `Aggregator` and defines the combination of data type, operation and initialization that the aggregator uses. Many standard aggregators are already implemented in Giraph, for example the `LongSumAggregator`, which is initialized with the value 0 and each time a vertex calls `aggregate()`, adds the passed

long to the current value of the aggregator. The current value can be requested with the `getAggregatedValue()` method. To implement an own aggregator, one needs to define instructions how to create an initial value, how to process a value being stored via `aggregate()` and how to retrieve the current value via `getAggregatedValue()`. Note that the operation that combines the current value and the one to be processed should be commutative and associative since the order of the store commands is not deterministic.

A Giraph algorithm will halt if one of two conditions is met.

- The method `haltComputation()` in `MasterCompute` is called. This instantly halts the algorithm.
- Every vertex of the graph called its `voteToHalt()` method.

The `voteToHalt()` method is an interesting concept. Once a vertex calls this method, it will no longer execute code during a superstep until it receives an arbitrary message by another vertex. Once all vertices of the graph are in this "sleeping" state, the algorithm halts as well. We will use this to deactivate parts of the graph that we don't need any more and never wake them up.

In the Giraph implementation there is a class `Vertex<I,V,E>` as well as a class `Edge<I,E>`. Hereby, the generic type variable `I` parametrizes the class whose objects act as the ID of a vertex, `V` is the *vertex value* class and `E` is the *edge value* class. The vertex value class is a class field of of `Vertex` that is used to store information used by the algorithm. The same applies to edge values.

For our algorithm the vertex value object stores

- The current block (a 64-bit integer)
- The "candidate block" , the block the vertex might switch to in the second label propagation phase (a 64-bit integer)
- A boolean indicating whether the vertex is in the coarsest level of the graph or not (used for processing the result of the contraction and to distinguish between active and inactive levels in local search; 1-bit boolean)
- A boolean indicating whether the vertex is marked as a candidate to switch blocks in the second phase of label propagation (1-bit boolean)
- Its weight. This is necessary due to contraction (64-bit floating point number) when subsets of V' should have the same weight as their corresponding subsets in V.

The edge value object stores

- The weight of the edge (64-bit floating point number)
- The partition of the target vertex of this edge. This is used to calculate the score function since a vertex only has its own local information. Every time a vertex switches partitions, it sends an update message to its neighbours so they update this value. (64-bit integer)

With this in mind we can now discuss in detail how to implement the rough directives of Section 4. One of the first problems with implementing the multi-level graph partitioning scheme is that in several steps different levels of the graph have to perform different steps. This means, that in several steps each vertex has to "know" in which level it is. For this purpose we will often use the `voteToHalt()` method of a vertex as well as the `isInCoarsest()` field. In theory vertices that have voted to halt can be reactivated by sending them a message, however, we will not use this feature. Once a vertex has voted to halt it will not execute code any more for that phase of the algorithm.

The implementation of our algorithm follows this plan: Firstly we coarsen the input graph by iteratively computing clusterings and coarsen the computed clusterings. Once the halting

condition is met, vertices from all levels including their edges, weight and labels are written to the HDFS. We then iterate over the output file and write the coarsest level to a separate file. We then copy this file to local disk and transform the graph file so it can be partitioned by KaHIP (this requires a renaming of the vertices to $1, \ldots, n'$ for an appropriate n' as well as changing the graph file format). KaHIP then produces an output file with a mapping of vertices to their labels. We undo the transformation on the vertices from the output file and copy it into the HDFS. Subsequently, we iterate over our first output file containing all levels of the processed graph and overwrite the labels of the coarsest graph with the better partition computed by KaHIP. Lastly, we iteratively perform a local search on each of the computed levels and write the computed partition of the input graph to disk.

5.2 Spinner Variant

Initialize partitions. In the initialization step every vertex of the input graph chooses its initial label. In Spinner, each vertex chooses a random label from $\{1, \ldots, k\}$. To be able to iteratively contract the computed clusterings, we changed this and made each vertex choose its ID as its label. In the higher levels we will subtract an offset to ensure the resulting label l fulfils $1 \leq l \leq n$. This creates the need for $2n$ aggregators to store block sizes as well as the amount of vertices that want to migrate to a block in a certain superstep. It should be noted that only a fraction of these aggregators will actually be used after a few supersteps, since many blocks that started with just one vertex would be merged with others and disappear. The aggregators that store the load will sum up all aggregators that are aggregated on them and store them throughout the algorithm. This means when a vertex migrates from one block to another, it needs to aggregate its negative weight to the load aggregator of the old label as well as its positive weight to the load aggregator of the new label. The candidate aggregators, however, will be reset after every superstep. They also sum up the values aggregated to them.

In case the cluster does not have enough memory to create $2n$ aggregators we also implemented a second mode that consumes less memory. By setting a parameter $k' < n$, only $2k'$ aggregators will be registered. In that case, every vertex chooses its label l as $l = ((\text{id} - \text{offset}) \bmod k') + 1$, ensuring that all labels l fulfil $1 \leq l \leq k'$. To ensure a fast initialization the following trick has been employed in our Spinner version: After every vertex initialized its own label, instead of sending a message to all neighbours which would result in $\mathcal{O}(m)$ messages in the first step each vertex calculates the initial label of all of its neighbours. This is possible just using information the vertex needs anyway to calculate its own label. Since only message-receiving vertices are allowed to change label every vertex has to send an invalid message to themselves for the next superstep. This means we could reduce the amount of messages sent in the first superstep from $\mathcal{O}(m)$ to $\mathcal{O}(n)$.

Compute new labels. In the first label propagation step, after iterating over the received messages and updating the corresponding fields every vertex computes the label with the highest score function value. However, the vertex only computes its new best label with a certain probability. An aggregator stores the superstep s in which the last contraction phase ended – initialized with 0 – and every vertex computes its probability not to participate in this step as $p_{part} = 0.5t^s$, with $0.5 \leq t \leq 1$ being a tuning parameter. This ensures that the probability in each first label propagation step is 50% which maximizes the chances to escape an endless loop explained in the following paragraph.

Computation of the score function values is achieved by iterating over the outgoing edges and adding these `<block, weight>` entries into a hash table. If a block was already in that

hash table the weight was added onto his entry before else a new entry with their weight is being created. At the same time we add up all weights of all outgoing edges in a separate field. After all edges have been processed the score function is evaluated using almost exclusively local information. After this step we iterate over the entries of the hash table and compute the score function for each label. For this we need the weight of all outgoing edges of that label, the weight of all outgoing edges of the vertex, both of which we just computed in linear time as well as the load of each block l which is stored in an aggregator that is updated after each migration and the capacity C of a block which can be computed from the input parameters n, ε and k as $C = (1 + \varepsilon)\frac{n}{k}$ since the input graph is unweighted. Note that the only non-local value used is the load of a partition (and some sort of global communication is necessary since the balance constraint is a global constraint). The marking as a candidate also includes increasing a global candidate counter for the respective label by one. To be able to calculate the stop criterion we also add the score of the current partition to a global counter. Spinner uses the `preSuperstep()` method to reduce the amount of aggregator calls, greatly improving performance. This method is called once before the `compute()` method is executed for all vertices. Thus, you can receive the block sizes for all blocks with a single call and store them in an array of size k, avoiding a call to an aggregator for each vertex. Since we start with k' blocks, however, we cannot use this optimization in the coarsening phase. It is used in the implementation of the local search phase.

If the score function of a different label than the current one is higher the vertex will be marked as a candidate and prepared for the switch. If there is a tie and the current label is among the highest score function values the current one is preferred else a random label out of those will be chosen. This is to improve stability of the graph since every vertex makes decisions based on the current state of the graph which might change to the worse for its decision as it makes it. For example if a vertex has an edge with a higher weight than all other edges combined it will most likely switch to the label of the target vertex but that vertex itself might switch to a different label at the same time.

Endless loop during label propagation after initialization. A small example (since the smallest example with two vertices is not very helpful) of a graph which will keep our label propagation running forever is depicted in Figure 3. The basic problem occurs when the union of each edge with the highest weight per vertex forms a matching of the current level. Since each vertex then initializes in its own block, the penalty scores of all blocks are the same. This means the partition belonging to the edge with the highest weight is optimal for every vertex. However, consider such an edge (u, v) from the perspective of u as well as v at the same time. u will want to switch to the label of v since it is the label with the optimal score function value. v wants to switch to u's label. Since no partition is overloaded yet, the migration probability for all partitions is 1.0. This means u and v will swap their labels at the same time in every superstep. When we introduce a probability to not participate in this step, we create the possibility for one of the many u's to switch and one of the many v's not to participate. When this happens, the vertices locally escaped the endless loop. When this happens on a larger scale, the partition of the graph becomes more stable and the label propagation algorithm can run as it is supposed to. Of course, 50% optimizes the probability that one vertex does and another vertex does not participate in the label propagation in that step. Our solution also does not increase run time significantly since the participation probability approaches 100% very quickly. Every vertex that did not participate in label propagation in an iteration will get the chance to do so in the next iteration of label propagation.

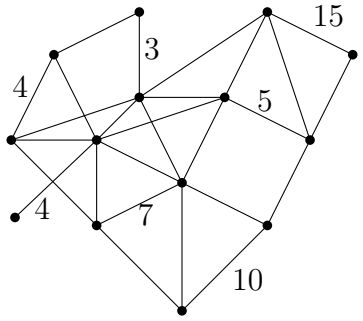


Figure 3: A small "realistic" example graph in which the endless loop would occur. All vertices are in their own block, edge weights are annotated. Where missing, they are set to 1.

Compute label migrations. In the second step of label propagation all vertices that are marked as a candidate calculate their probability to switch to the vertex. Like in step one the capacity of a block can be computed from input parameters and the load of the block is retrieved from an aggregator. Additionally we also need the amount of vertices that want to switch to that label which is also stored in a global variable. Then, with a certain probability some vertices will change label and send a message containing their new label to all of their neighbours and the step is repeated. The load variables of the old and the new label are adjusted and the candidate variables reset automatically after each step due to the setting of the corresponding aggregator. If the vertex did not participate in the label computation in the last step it now sends a message to itself for the next superstep since only vertices that received messages are considered to change blocks.

Control flow. Lastly, we want to explain how to implement switching between the several phases of the algorithm. As introduced in Section 5.1, we use the `MasterCompute` class to switch between `AbstractComputation` implementations of different steps. For this purpose, the `getSuperstep()` method is very important. This method allows us to implement case differentiations. We store an offset for the current superstep as a class field as well as a boolean to indicate whether we are currently clustering or contracting the graph. In each iteration we then retrieve the current superstep, subtract our offset from it and do a simple `switch`: First we initialize the level, then we alternately compute best labels and perform label switches. During this, the central `MasterCompute` class also retrieves the current global score and checks if it is at least by a factor of Δ better than the last score. If it is not, a counter is increased. Once the counter hits w, clustering on this level halts. If it is, the counter is reset and the last score is updated. Via the `getTotalNumVertices()` method and saving the amount of vertices after the last contraction, we check the size of each newly created level and halt once the condition is met.

5.3 Contraction

After our Spinner variant meets its halting criterion on the current level we compute a contraction that will become the next level. This level is then used in the multi-level scheme to run on until the resulting contraction has fewer than n_h vertices. All levels except the current level are put to sleep, meaning the steps below are only executed on the current level and newly created vertices. Note that for the input graph $G = (V, E)$ we assume $V = \{1, \ldots, n\}$. When we create new vertices during contraction, we need to give them IDs as well. We want the following property: For the ith level the vertex IDs should be $\in \{(i-1)n+1, \ldots, in\}$ with the

input graph being the first level. When creating a new level, we need *corresponding* vertices to a vertex v. The vertex w corresponding to the ID of v has the ID $v.id \bmod n + (\lfloor \frac{v.id}{n} \rfloor + 1)n$ and the vertex corresponding to the block of v has the ID $p(v) + (\lfloor \frac{v.id}{n} \rfloor + 1)n$. We will contract v to the vertex corresponding to its block, the vertex corresponding to the ID is necessary for the creation of the new level in Giraph.

Of course, the new level also has to be a contraction of the current level. This is achieved through the following steps:

Create Vertices. Each vertex v checks if the partition p corresponding to its own ID $p = v.id - (i-1)n$ is empty or not. If it is not, the vertex creates a request to Giraph to create a vertex with ID $vertexID + n$ and weight 0.

Send vertex weights. Each vertex sends a message to the vertex corresponding to its block containing the weight of the vertex.

Receive vertex weights & send edge weights. We differentiate between the lower and higher level by the amount of messages received this step. 0 means the vertex is in the higher level, 1 or more means the vertex is in the lower level. Each vertex in the lower level sums up the values it received as messages and set that as its new weight. Each vertex in the higher level iterates over all outgoing edges and sends a message to its corresponding vertex for each edge to a vertex with a different partition containing the weight and the partition of the neighbour vertex. Then, all vertices in the higher level are set to sleep.

Receive edge weights. Each vertex creates a hash table in which they save the weight of all edges to certain partitions. Then they iterate over all their messages and add the weight contained in the message to the stored weight of the partition contained in the message (if there is no entry in the hash table yet, it is treated as zero). After all messages are processed, each vertex requests Giraph to create an edge from itself to the vertex in the new level corresponding to the partition from the hash table with the weight from the hash table.

After this, the label propagation starts anew on the newly created level until the level is small enough. When the Giraph job ends, the whole graph including all levels is written to disk, we extract the highest level via a simple Hadoop job and copy it to the local disk (this file is very small since it is heavily contracted).

After preprocessing, we partition this level using KaHIP, which gives us a high-quality partitioning for that level. The preprocessing is necessary since the highest level doesn't consist of vertices from 1 to n' – which KaHIP assumes its input to – and to convert between graph file formats. We use the strong social network setting for KaFFPa. After reversing the preprocessing on the output file that KaFFPa produces, we use a Hadoop job to overwrite the partitioning of the coarsest level in the output file of the contraction run by copying it and replacing the lines in question as well as deleting the old file.

When uncoarsening the graph we need iterate through the levels starting at the highest and send the partitioning of current level to the level below that one. Since by default every level executes the code however, we need to differentiate between the current level i and the level $i-1$ below that and all other levels. For this purpose we have a global variable id_f that stores the first vertex of the level $i-1$. This way we can determine the level of a vertex based on its id id and that id. If $id_f \leq id$ and $id < id_f + n$ then the vertex is in the level $i-1$, if $id_f + n \leq id \wedge id < id_f + 2n$ then the vertex is in level i. In all other cases the vertex is inactive

for this iteration. After sending the partition information to the lower level, we can put all vertices from level i to sleep again and start the label propagation on level i. We set a flag in the vertices from other levels so we do not have to retrieve the value from the global variable each superstep. Once the level of these vertices becomes level $i-1$, the flag is set appropriately when level $i-1$ sends messages to level i.

5.4 Local Search

After transferring the partitioning from a higher level to a lower one we run the label propagation algorithm on that level. The only difference to the one used for contraction is that in the contraction we start in the lowest level and create the higher levels so we can put the lower levels to sleep after we are done with them. Since in local search we start at the highest level and move down we need to ensure that the rest of the graph is not executing any code yet. For this purpose, we set a `boolean` of the vertex to *false* in the lower levels and check at the start of each `compute()` method where it is necessary whether the flag is *true* or not and only continue if it is true. If we put the lower vertices to sleep we could avoid this but we would face the problem of waking these vertices up. Either vertices of the higher levels need to store which vertices in the level one below them belong to their partition which would require a big amount of memory (although the memory would be in $\mathcal{O}(n)$ or we would need to send a message to all IDs in the range of the lower level which would be $\mathcal{O}(n)$ messages for each level which would be $\mathcal{O}(n \log n)$ in the case of matching-based contraction.

6 Experimental Results

In this chapter we present and discuss the results of our experiments after introducing the methodology and setup used to acquire them.

We implemented the contraction and local search phases as Java code in Giraph which we run on Java 1.8. KaHIP is written in C++ and the steps to copy the interim results to the local disk and back into HDFS are realized as Hadoop Jobs and HDFS commands. We use Hadoop 1.2.1 in pseudo-distributed mode and Giraph 1.1.0. To transfer the coarsest level and later the initial partitioning by KaHIP we use small linear time converters. They are executed on local disk and were written in C++ and compiled using g++ 4.8.4, using optimization level 3 for KaHIP. We use the strong social configuration of KaHIP, for more information about the configurations we refer the reader to [23]. We execute all steps of our algorithm via a simple Python script that calls the necessary commands in order.

Since Spinner uses an edge balance constraint we changed our configuration to use weighted vertices and set $\forall v : c(v) = \deg(v)$. This makes our vertex size constrained problem equal to the one Spinner uses and allows us to compare our results. All of our calculations and algorithms also work with the weighted version of the problem so we can easily implement this change. The reported maximum block sizes denote the amount of edges in a block.

6.1 Experimental Setup

The experiments were conducted on a machine provided by the Institute of Theoretical Informatics with the following specs:

We partitioned various graphs from the 10th DIMACS Implementation Challenge [1] and the Stanford Large Network Dataset Collection [19] with $k = 32$ and $\varepsilon = 0.03$. Some graphs from the DIMACS challenge are from the Laboratory for Web Algorithmics [8, 7] from where we also used the Amazon graph. Since uk-2002 contains isolated vertices which are not allowed for Spinner (and make no sense since their weight and cut would be 0) we removed these vertices. Our own tuning parameters were set to: Δ during coarsening was 1.3, during local optimization was 1.02. w was 3, n_h =75 000 and k' was set to $\min(n, 900000)$. We use 48 Giraph threads. For Spinner, we set $c = \varepsilon$ as suggested in the Spinner paper [22]. It would also be impractical to adjust c based on past runs on the same graph (c controls the degree of balance of the Spinner solution and can be used to achieve a valid partition when the first solution was invalid or a better global cut as well as faster convergence speed if the balance constraint is easily met by the first solution). Each experiment is repeated 5 times (except uk-2002 which was repeated 3 times) and average running time, average and best cut as well as biggest block are reported.

Property	Value
Processor	Intel Xeon CPU E5-2670 v3
Architecture	x86_64
Cores	12
Clock frequency	2.30 GHz
Cache	30 MB
RAM	128 GB
OS	Ubuntu 14.04
Kernel	3.13.0-45-generic

Table 1: Basic properties of the graphs used in our experiments. They are sorted alphabetically.

graph	n	m	$\min_v \lvert N(v)\rvert$	$\max_v \lvert N(v)\rvert$	Reference
amazon-2008	735 323	3 523 472	1	1 077	[8, 7]
as-skitter	554 930	5 797 663	1	29 874	[19]
citationCiteseer	268 495	1 156 647	1	1 318	[1]
cnr-2000	325 557	2 738 969	1	18 236	[1]
coAuthorsCiteseer	227 320	814 134	1	1 372	[1]
eu-2005	862 664	16 138 468	1	68 963	[1]
uk-2002	18 520 486	261 787 258	1	194 955	[1]
wiki-Talk	753 323	7 046 944	1	100 029	[19]

Table 2: Average global cut and runtime of our multi-level algorithm (left) compared to Spinner (right)

Graph	Avg. Cut	Avg. t [s]	Avg. Cut	Avg. t [s]
amazon-2008	703 800.4	1 302.9	2 575 545.6	204.0
as-skitter	3 215 195.0	2 293.3	5 040 332.8	206.2
citationCiteseer	438 870.0	788.9	897 556.0	177.5
cnr-2000	1 684 627.6	955.8	2 894 320.4	181.9
coAuthorsCiteseer	133 256.4	583.2	677 534.8	162.4
eu-2005	2 587 074.8	3 827.5	10 412 803.2	209.4
uk-2002	8 014 413.3	24 107.0	81 607 518.0	1 339.3
wiki-Talk	2 203 763.2	5 235.5	2 419 945.2	151.8

The block size is to confirm that the partitioning is valid since there is no guarantee that the balancing constraint is fulfilled.

To compare Spinner's and our output quality we measure the *global cut* G_C as well as the *maximum block size* $b_{\max}$, defined as

$$G_C = \sum_{v \in V} \Big(\sum_{(v,u) \in E} c(v,u) * \delta(p(v), p(u)) \Big), \; b_{\max} = \max_{1 \le i \le k} \sum_{v \in V, p(v)=i} \deg(v).$$

To compare the balance we compute the ratio r_{bal} of $b_{\max}$ to the average block size as

$$r_{\text{bal}} = \frac{b_{\max} k}{c(V)}.$$

We only compare our results with Spinner since according to their experiments [22] they outperform the other stream-based algorithms by a large margin.

6.2 Results

In Table 2 we compare Spinner's cut quality with ours, for example on the big uk-2002 graph we only needed a tenth of Spinner's cut. Every partition we compute has a better global cut than the one Spinner computes. With the exception of wiki-Talk, which appears to be an

Table 3: Average running time of the three phases of our algorithm in seconds for various graphs

Graph	t coarsening	t initial partitioning	t local improvement
amazon-2008	1 006.9	118.0	178.1
as-skitter	896.2	1 102.8	244.9
citationCiteseer	276.8	357.6	174.5
cnr-2000	690.8	28.8	236.3
coAuthorsCiteseer	259.9	152.4	170.8
eu-2005	2 723.9	599.3	504.3
uk-2002	19 647.0	1 523.3	2 936.7
wiki-Talk	1 072.4	3 606.7	556.5

Table 4: Average size of the coarsest level for several graphs

Graph	$\|V\|$ of coarsest level
as-skitter	51 416.8
citationCiteseer	60 780.0
cnr-2000	33 410.6
coAuthorsCiteseer	54 124.8
eu-2005	44 943.8
wiki-Talk	74 798.0

outlier since neither Spinner nor our algorithm can meet the balance constraint on that graph, our worst cut is almost 60% better than the one Spinner computes. On the geometric average, the cuts our algorithm computes are around 2.88 times better than Spinner's. As presented in Table 5 we are also better balance-wise, Spinner will sometimes calculate an invalid partition by violating the balance constraint, which happens very rarely with our algorithm due to the multi-level approach. The maximum partition in the Spinner solution is often smaller than ours, since Spinner focuses more on balance. However, there's no graph for which Spinner's solution meets the balance constraint and ours does not.

A downside to the current multi-level algorithm is its runtime however. Ignoring the outlier wiki-Talk, in our experiments our algorithm took 3-20 times as long as Spinner. For the large graphs eu-2005 and uk-2002, our algorithm takes 20 and 18.8 times as long as Spinner, indicating a scaling around this factor. This is mainly due to the contraction step since the trick described in Section 5.2 can not be employed there. The distribution of runtime across phases, excluding IO, can be found in Table 3. On the biggest graph, uk-2002, the coarsening phase was responsible for 82.5% of the total runtime. Note that the runtime of our initial partitioning and local improvement phase is often below twice the runtime of Spinner, allowing for a comparable runtime with a faster coarsening phase. To give further perspective on these runtimes, Table 4 contains the average size of the coarsest level of selected graphs.

Table 5: Average size of the biggest block of the solution of our multi-level algorithm (left) compared to Spinner's solution (right)

Graph	$b_{\max}$	r_{bal}	$b_{\max}$	r_{bal}
amazon-2008	225 036.4	1.022	223 464.4	1.015
as-skitter	372 138.8	1.027	385 423.0	1.063
citationCiteseer	73 992.8	1.024	74 328.8	1.028
cnr-2000	193 911.2	1.133	183 948.8	1.075
coAuthorsCiteseer	52 320.6	1.028	51 084.2	1.000
eu-2005	1 038 766.2	1.030	1 046 010.8	1.037
uk-2002	16 796 936.7	1.026	16 602 397.3	1.015
wiki-Talk	112 400.8	1.233	118 313.8	1.298

7 Conclusion & Outlook

We have introduced the first multi-level graph partitioning in a Think Like a Vertex Model. It is integrated into the Giraph framework and implemented in a scalable way. We presented the changes we made to use Spinner as a multi-level clustering and local optimizing algorithm and how we use the initial partition computed by KaHIP. In the implementation chapter we explained the way we differentiate between levels based on the ID of a vertex and which optimizations we use. When comparing our algorithm to our main contender Spinner, our experiments have shown that we compute partitions with vastly superior global cut that usually fulfil the balance constraint and are never worse than Spinner. The cost for this, due to the more complex approach, is that our runtime is 3 to 20 times slower. However, this is mostly due to the contraction phase and can be solved in future implementations. Other works have shown that using a computed graph partition to distribute vertices can vastly improve application performance, so our algorithm is interesting for any problem that requires a high-quality graph partition.

7.1 Future Work

The most obvious extension of our work would be employing techniques to make contraction faster. To achieve this, the number of partitions must be minimized very quickly to reduce the number of aggregators, e.g. through a phase of constant length at the beginning of the first contraction that merges blocks faster than the current clustering scheme. If a significant reduction in coarsening runtime is achieved, our algorithm will have a runtime comparable to Spinner while computing vastly superior partitions. Another possible extension would be to add the ability to repartition a graph due to a change of the state (e.g. vertices removed/added, partitions removed/added, ...) like in Spinner. This is very useful to reduce computation overhead when using the partition to distribute the graph in Giraph but will not work as well as it does in Spinner since the multi-level approach still needs to execute initial partitioning and uncoarsening from scratch.

References

[1] *10th DIMACS implementation challenge website*. http://www.cc.gatech.edu/dimacs10/archive/clustering.shtml. Retrieved on 01/11/2015.

[2] *Apache Giraph Project*. http://giraph.apache.org/, 2012. Retrieved on 01/11/2015.

[3] Alpert, Charles J and Andrew B Kahng: *Recent directions in netlist partitioning: a survey*. Integration, the VLSI journal, 19(1):1–81, 1995.

[4] Alpert, Charles J, Andrew B Kahng and So-Zen Yao: *Spectral partitioning with multiple eigenvectors*. Discrete Applied Mathematics, 90(1):3–26, 1999.

[5] Andreev, Konstantin and Harald Räcke: *Balanced Graph Partitioning*. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 120–124, New York, NY, USA, 2004. ACM.

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

[7] Boldi, Paolo, Marco Rosa, Massimo Santini and Sebastiano Vigna: *Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks*. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[8] Boldi, Paolo and Sebastiano Vigna: *The WebGraph Framework I: Compression Techniques*. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[9] Buluç, Aydin, Henning Meyerhenke, Ilya Safro, Peter Sanders and Christian Schulz: *Recent Advances in Graph Partitioning*. CoRR, abs/1311.3144, 2013.

[10] Çatalyürek, Ümit V and Cevdet Aykanat: *Decomposing irregularly sparse matrices for parallel matrix-vector multiplication*. In *Parallel algorithms for irregularly structured problems*, pages 75–86. Springer, 1996.

[11] Delling, Daniel, Andrew V. Goldberg, Thomas Pajor and Renato F. Werneck: *Customizable Route Planning*. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, Lecture Notes in Computer Science. Springer Verlag, May 2011.

[12] Fiduccia, C. M. and R. M. Mattheyses: *A Linear-time Heuristic for Improving Network Partitions*. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press.

[13] Fietz, Jonas, Mathias J Krause, Christian Schulz, Peter Sanders and Vincent Heuveline: *Optimized hybrid parallel lattice Boltzmann fluid flow simulations on complex geometries*. In *Euro-Par 2012 Parallel Processing*, pages 818–829. Springer, 2012.

[14] Garey, M. R., D. S. Johnson and L. Stockmeyer: *Some Simplified NP-complete Problems*. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.

[15] Hendrickson, B.: *Chaco: Software for Partitioning Graphs*. http://www.cs.sandia.gov/CRF/chac_p2.html. Retrieved on 08/11/2015.

[16] Karypis, George and Vipin Kumar: *A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering*. J. Parallel Distrib. Comput., 48(1):71–95, January 1998.

[17] Kieritz, Tim, Dennis Luxen, Peter Sanders and Christian Vetter: *Distributed time-dependent contraction hierarchies*. In *Experimental Algorithms*, pages 83–93. Springer, 2010.

[18] Kumar, Vipin, Ananth Grama, Anshul Gupta and George Karypis: *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[19] Leskovec, Jure and Andrej Krevl: *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data, June 2014. Retrieved on 01/12/2015.

[20] Luxen, Dennis and Dennis Schieferdecker: *Candidate Sets for Alternative Routes in Road Networks*. In Klasing, Ralf (editor): *Experimental Algorithms*, volume 7276 of *Lecture Notes in Computer Science*, pages 260–270. Springer Berlin Heidelberg, 2012.

[21] Malewicz, Grzegorz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser and Grzegorz Czajkowski: *Pregel: A System for Large-scale Graph Processing*. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[22] Martella, Claudio, Dionysios Logothetis and Georgos Siganos: *Spinner: Scalable Graph Partitioning for the Cloud*. CoRR, abs/1404.3861, 2014.

[23] Meyerhenke, Henning, Peter Sanders and Christian Schulz: *Partitioning Complex Networks via Size-Constrained Clustering*. In *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, volume 8504, pages 351–363. Springer, 2014.

[24] Pellegrini, F.: *Scotch Home Page*. http://www.labri.fr/pelegrin/scotch. Retrieved on 08/11/2015.

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

[26] Sanders, Peter and Christian Schulz: *KaHIP – Karlsruhe High Qualtity Partitioning User Guide*. http://algo2.iti.kit.edu/schulz/software_releases/kahipv0.73.pdf/. Retrieved on 08/11/2015.

[27] Sanders, Peter and Christian Schulz: *Engineering Multilevel Graph Partitioning Algorithms*. In *ESA*, pages 469–480. Springer, 2011.

[28] Sanders, Peter and Christian Schulz: *Think locally, act globally: Highly balanced graph partitioning*. In *Experimental Algorithms*, pages 164–175. Springer, 2013.

[29] Schloegel, Kirk, George Karypis, Vipin Kumar, J. Dongarra, I. Foster, G. Fox, K. Kennedy, A. White and Morgan Kaufmann: *Graph Partitioning for High Performance Scientific Simulations*, 2000.

[30] Schulz, Christian: *High Quality Graph Partitioning*. PhD thesis, Karlsruhe.

[31] Valiant, Leslie G.: *A Bridging Model for Parallel Computation*. Commun. ACM, 33(8):103–111, August 1990.

[32] Walshaw, Chris: *Multilevel refinement for combinatorial optimisation problems*. Annals of Operations Research, 131(1-4):325–372, 2004.

[33] Zeng, ZengFeng, Bin Wu and Haoyu Wang: *A Parallel Graph Partitioning Algorithm to Speed Up the Large-scale Distributed Graph Mining*. In *Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, BigMine '12, pages 61–68, New York, NY, USA, 2012. ACM.