Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 344

Distributed Graph Partitioning for large-scale Graph Analytics

Lukas Rieger

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Dipl.-Inf. Christian Mayer, Dr. Adnan Tariq
Commenced:	1. June 2016
Completed:	1. December 2016
CR-Classification:	G.2.2.

Abstract

Through constant technical progress the amount of available data about almost anything is growing steadily. Since hardware has become very cheap in the last decades, it is also possible to store and process huge amounts of data. Moreover companies like Google have sprung up that generate a large part of their revenues by extracting valuable information from collected data. A common approach to compute large inputs efficiently is to use a distributed system and concurrent algorithms. Therefore it is necessary to distribute the data to be processed intelligently among the cores or machines. Thereby the distribution of the input has strong effects on the efficiency of algorithms processing it. Furthermore data which can be represented as graph is ubiquitous, Facebook and the world wide web are well known examples of it. In this case, graph partitioning algorithms are used to distribute the input data in the best possible way. Since graph partitioning is a NP-hard problem, heuristic methods have to be used to keep execution time within reasonable bounds. This thesis presents and analyses a divide-and-conquer based framework for graph partitioning which aims to approximate linear runtime with better partitioning results than linear time algorithms.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

Today it is possible to gain more insight than ever before from collected data. However, the percentage of data that can be processed goes down, thus loosing valuable information [EDD+12]. This demonstrates the importance of processing big amounts of data efficiently. Furthermore IBM states that every day around 2.5 quintillion (1 quintillion $= 1 \times 10^{18}$) bytes of new data are generated [IBM16]. Many times data comes in the representation of a graph, e.g., biological networks, economical networks or social networks[FFF99]. Moreover graphs make it easier for humans to understand and work with big amounts of data. Today graphs can have millions of vertices and even more edges representing relations between the vertices. As an example, the graph representation of Facebook has over 1.39 billion vertices and more than 400 billion edges [CEK+15; Cia15]. Online shops like Amazon or platforms like Netflix are just one example of applications for large-scale graph analysis. They use it for product recommendation or collaborative fitting. Using intelligent algorithms to recommend products to customers they can generate higher revenues. This example shows the vivid interest such companies have in effective algorithms to process graph data.

The standard approach to achieve effectiveness and scalability on big data is distributed computing. This makes it necessary to partition the input data into several smaller data sets that can be processed on different machines. In the case of graphs this is called graph partitioning. In order to use resources optimally, the goal of graph partitioning is to distribute the input data proportionally to the amount of resources of the respective machines. Data set size can either be defined as the number of edges or as the number of vertices that are contained in one data set. In this thesis all machines are considered to have the same amount of resources, so that input data should be distributed equally among the machines. Another limiting factor is the communication between the machines during the computation process. The more interdependencies exist between the data sets, the more synchronization is necessary. Therefore graph partitioning algorithms should not only compute equilibrated data set sizes, but also reduce the communication cost between machines. This is achieved by making the data sets as independent as possible from one another.

Graphs can be partitioned either by cutting through edges or through vertices. Traditionally graph partitioning problems focus on edge-cut, e.g., the min cut problem. But it has

been shown [AHB00] that especially on graphs with a power-law degree distribution very good partitioning results can be achieved using vertex-cut algorithms. However the balanced graph partitioning problem is known to be NP-hard [TGRV12], independently of the cut variant that is used [PQD+15].
The two different cut variants are illustrated in figure 1.1.

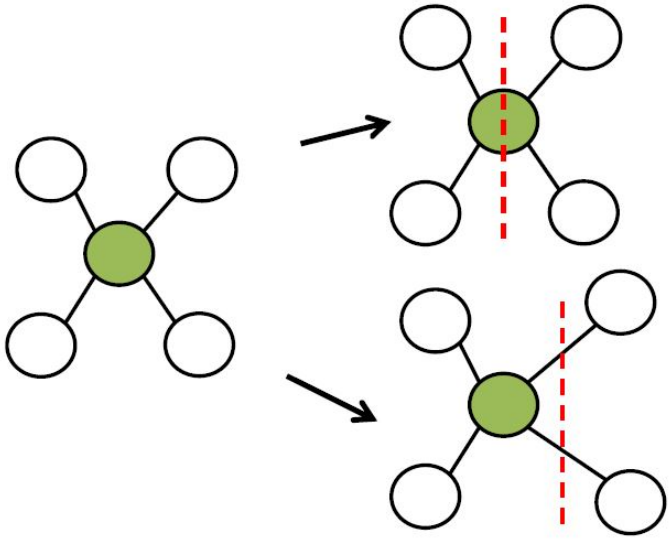

Figure 1.1: Edge-cut and vertex-cut

Furthermore graph partitioning algorithms can be divided into online and offline algorithms. Offline algorithms first read the whole graph into memory before computation starts. This allows them to compute a partitioning with full knowledge of the whole graph, e.g., for every vertex the exact in/out degree is know. Offline partitioning can be done in many different ways, for instance there are genetic algorithms [KHKM11] or iterative algorithms [RPG+13]. Especially for big graphs, graph processing frameworks have been proposed. Well known examples of such frameworks are Pregel, PowerGraph, GraphX and GrapH [MAB+10; MMTR16; MTLR16] which are used for offline partitioning.
Online algorithms on the other hand read the input graph as a stream and assign the elements contained in the stream immediately to partitions using a heuristic function, thus usually seeing each element only once [PQD+15]. In the case of vertex-cut, this stream consists of the edges of the graph. Each edge is represented as a tuple of two vertices if the graph is directed and as a set of two vertices if the graph is undirected. In the following all graphs are undirected. The order in which the edges arrive in the stream can have significant effects on an algorithm's performance. Most streaming algorithms need a randomized order of edges to perform optimally [SZCH15], others try

to avoid bad performance on breadth-first-order or depth-first-order input by using more complex heuristic functions [XLZ15]. Offline algorithms take more time for computation but achieve better results than online algorithms. This reflects the tradeoff between computation time and result quality. A good result can be defined as a partitioning for which the mean standard deviation from the average number of edges over the edges on all partitions, called imbalance, is less than 5% and the communication cost is as low as possible. Since every graph has a different structure, a different average vertex degree and a different vertex degree distribution, it is not possible to give any general fixed value for the computation cost, it depends on the graph. The vertex-cut variant generates additional communication cost whenever a vertex is cut and thereby replicated on another machine. Therefore the communication cost for a given partitioning can be approximated as the average number of replications that exist for each vertex. As can be seen in figure 1.1, there is always a trade off between low imbalance and low replication degree. If the graph in the vertex-cut case was not cut at all, the replication degree would be 1.0 but the partitioning to two partitions would be completely imbalance. If the vertex is cut as shown, the partitioning is completely balanced but a new replica had to be introduced. In the case of edge-cut the communication cost amounts to the number of edges cut by the partitioning. If an edge-cut approach is used, the objective of the partitioning is to assign the same amount of vertices to each partition and reduce the number of edges that have to be cut.

Since the problem is NP-hard, any algorithm that solves the problem optimally would have a very high execution time for large problem instances. That is why computation time is another factor influencing the result of a partitioning. Every graph partitioning algorithm has to make a tradeoff between low imbalance, low replication degree and low computation time. This can be visualized as shown in figure 1.2.

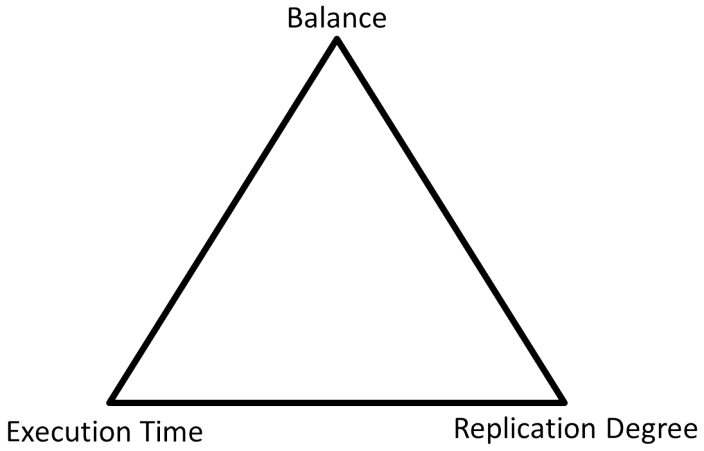

Figure 1.2: Partitioning trade-off

In this work a new approach for graph partitioning is presented. The idea is to reduce the problem size through a divide-and-conquer based approach and subsequently identify dense clusters on the subgraphs concurrently. Since the subgraphs are smaller than the whole input graph, it requires less computation cost to find better partitionings than on a big problem instance. After this step was performed, the clusters are recombined with an iterative algorithm so that in the end the process results in a partitioning of the graph. The overall goal of this procedure is to get close to linear execution time and simultaneously achieve better results than linear time algorithms. Figure 1.3 visualizes the idea but assumes sequential computation of the subgraphs.

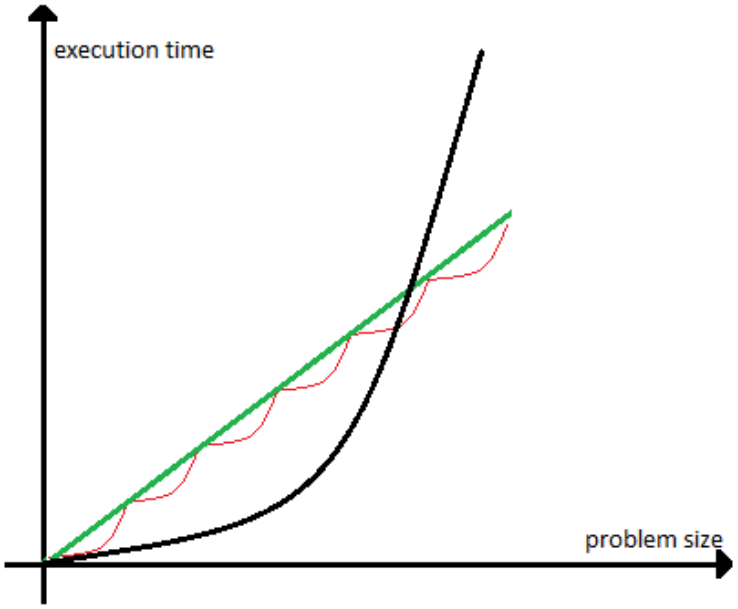

Figure 1.3: Linear run time approximated by exponential run time on smaller sub problems

In the evaluation chapter the approach will be compared to state-of-the-art online and offline algorithms. Runtime and partitioning results have to be compared to see if the idea can be implemented practically. In addition the effects of different algorithms for each step are evaluated. In the end the best settings depending on the input will be presented.

1.1 Contributions

The contributions of the present work are:

1. Every step of the divide-and-conquer based approach is described in detail. After a general introduction and a description of the step's task, possible methods to solve the task are presented. Then the implementation of the methods that were effectively used are shown and analyzed theoratically. A conclusion section completes each of these chapters. In the conclusions advantages and disadvantages of the chosen methods are discussed.
2. The divide-and-conquer based framework was implemented in JAVA. This implementation will be called multistep graph partitioning algorithm.
3. A new variant of the HDRF algorithm, called PSHDRF, is presented, analyzed and evaluated. PSHDRF aims to make better use of breadth-first-sorted edge streams so that it fits better for its task in the context of the divide-and-conquer based framework.
4. Karger's minimal cut algorithm is adapted from edge-cut to vertex-cut.
5. Iterated local search (ILS) is a common approach to solve problems iteratively. An implementation of ILS for partitioning graphs with vertex-weights and edge-weights is presented, implemented and evaluated. Another approach which derives from ILS but is much faster is also presented and evaluated. Strictly speaking this fastILS approach is not an ILS but is still an iterative algorithm.
6. The multistep graph partitioning algorithm is evaluated with different settings. Therefore every new step is introduced gradually so that the reader can see how it affects the computation process or the final results of the partitioning. From the evaluation's results the best settings are derived for different inputs.

1.2 Outline

After a general introduction to graph partitioning in chapter 1, the problem is defined and presented in several variants in chapter 2. Chapter 3 presents the divide-and-conquer framework and gives a general overview of its steps. From chapter 4 to chapter 6 every single step is described in detail and several possible methods that can be used to solve the step's task are presented and analyzed. The implementation of the divide-and-conquer framework with the presented methods is evaluated in chapter 7.

Finally related work is presented in chapter 8 and the discussion follows in chapter 9. The work is concluded by a summary of the whole in chapter 10.

2 Problem Formulation

This chapter gives a mathematical definition of the graph partitioning problem in general and for variants of the graph partitioning problem where edges and vertices can have weights. Mathematically a graph $G = (V, E)$ is defined as a set $V = \{v_1, v_2, ..., v_n\}$ of vertices and a set $E = \{e_1, e_2, ..., e_m\}$ of edges. Each edge consists of two vertices $v_i, v_j \in V$ and can be either directed $(v_i, v_j) \in V \times V$, thus represented as a tuple, or undirected $\{v_i, v_j\}$, thus represented as a set of two vertices. If two vertices are connected to each other by a common edge, they are called neighbors. On a higher level this can be seen as two data points that are related to each other in some way. The number of edges or vertices contained in a graph, is given as cardinality of the respective set, e.g., $|V| = n$ is the number of vertices.

Graph partitioning in general aims to split a graph G into several subgraphs thereby minimizing the cut size. A subgraph G' is defined as $G' = (V', E')$ where $V' \subset V$ and $E' \subset E$. The graph can be partitioned in two different ways, first by cutting through edges and second by cutting through vertices. Normally the objective of a partitioning is to achieve equal sized subgraphs and minimize the edges or vertices that have to be cut.

2.1 Definitions

variable	**definition**
e	edge
v	vertex
$\|V\|$	total number of graph vertices
$\|E\|$	total number of graph edges
p	total number of partitions
$\|p_i\|$	number of edges (size) on partition i
A(x)	set of partitions containing x
λ	constant for weighting the load balance
ξ	constant for allowed imbalance

2.2 Edge-Cut

In [XLZ15] the definition of the classic graph partitioning problem by edge-cut is given as follows

$$\min_{A} |\{e \,|e = (v_i, v_j) \in E, v_i \in V_x, v_j \in V_y, x \neq y\}| \tag{2.1}$$

s.t.

$$\frac{\max_i |V_i|}{\frac{1}{p}\sum_{i=1}^{p} |V_i|} \leq 1 + \xi \tag{2.2}$$

ξ is a small constant with $0 \leq \xi \leq 1$ that defines the allowed imbalance. p is the number of partitions into which the graph was partitioned and A is a partitioning of the graph. Every partitioning A is defined as $A = V_1, V_2, ..., V_p$ and $V = \bigcup_{i=1}^{p} V_i$ where $V_x \cap V_y = \emptyset$ for $x \neq y$. In other words, the graph is partitioned by cutting through edges with the constraint that every partition should contain more or less the same number of vertices. For edge-cut partitioning there exist online algorithms and offline algorithms. LDG [SK12] is an example of an online edge-cut algorithm and Metis [KK95] for an offline edge-cut algorithm.

2.3 Power-law Graphs

Most graph problems, such as the min-cut problem, have been formulated based on edge-cut. Edge-cut is first of all suitable for sparse graphs [XLZ15] but most of the real world graphs are not sparse. Natural graphs are typically power-law graphs, which means their vertex degree distribution can be modeled as:

$$P(d) \propto d^{-\alpha} \tag{2.3}$$

$P(d)$ indicates the probability that a vertex has degree d. α is a positive constant that controls the skewness of the degree distribution. For natural graphs it is $\alpha \approx 2.0$ [FFF99]. In [KLPM10] it was shown that the constant α equals 2.276 for the graph of the social network Twitter of 2010 and in [FFF99] it is shown that α is around 2.2 for the internet interdomain graph. For social graphs this distribution comes from the fact that there are little very famous people that have many connections to other people and many little known people who have few connections to others. For those reasons, vertex-cut algorithms are more suitable for natural graphs.

2.4 Vertex-Cut

A vertex-cut can be computed by assigning edges $e \in E$ of the input graph to partitions/-machines $A(e) \in \{1, ..., p\}$. Therefore each vertex $v \in V$ is contained, or replicated, on a set of machines $A(v) \subseteq \{1, ..., p\}$. The definition of the graph partitioning problem for balanced p-way vertex-cuts is given [GLG+12] as:

$$R = \min_{A} \frac{1}{|V|} \sum_{v \in V} |A(v)| \tag{2.4}$$

s.t.

$$I = \max_{m} |\{e \in E \, | A(e) = m\}| \leq \lambda \frac{|E|}{p} \tag{2.5}$$

λ is the imbalance factor and should be chosen as a small constant bigger than 1.0. R is called the replication degree and I is called Imbalance of the partitioning.

2.5 Cuts on Graphs with Edge Weights and Vertex Weights

So far all the problem formulations are only suitable for unweighted graphs. But there are also cases where edges have weights and/or vertices also have weights. Until now the majority of graph partitioning algorithms focuses on unweighted graphs. If edge weights and vertex weights are to be taken into account, a weight function $\gamma(x)$ has to be introduced. $\gamma(x)$ returns the weight of x, where x is either an edge or a vertex.

$$\gamma(x) = \begin{cases} \text{if x} \in V \text{ then return weight of edge x} \\ \text{if x} \in E \text{ then return weight of vertex x} \end{cases} \tag{2.6}$$

If only vertices or only edges are considered to have weights, $\gamma(x)$ returns 1 for all vertices or edges without weight.

2.5.1 Weighted Edge-Cut

The problem formulation of the edge-cut graph partitioning problem can be adopted for weighted edge-cut as shown in equation 2.7.

$$\min_{A} \sum_{e=(v_i,v_j)\in E \wedge v_i \in V_x \wedge v_j \in V_y \wedge x \neq y} \gamma(e) \tag{2.7}$$

s.t.

$$\frac{\max_i \left| \{g \mid g = \sum_{v \in V_i} \gamma(v)\} \right|}{\frac{1}{p} \sum_{i=1}^{p} \sum_{e \in V_i} \gamma(e)} \leq 1 + \xi \tag{2.8}$$

This means that it is no longer the mere number of edges, but the weight of the edges which are cut, that has to be reduced. At the same time the sum of vertex weights on each partition should be balanced, though the balance criteria can be relaxed increasing the constant ξ.

2.5.2 Weighted Vertex-Cut

The problem formulation of the vertex-cut graph partitioning problem can be adopted for weighted vertex-cut as follows in equation 2.10.

$$\min_{A} \frac{1}{|V|} \sum_{v \in V} |A(v)| \, \gamma(v) \tag{2.9}$$

s.t.

$$\max_{m} \sum_{e \in E \wedge A(e) = m} \gamma(e) < \lambda \frac{\sum_{e \in E} \gamma(e)}{p} \tag{2.10}$$

Once again it is not the number but the weight of cut elements that has to be reduced. For weighted vertex-cut the balance is defined by the sum of edge weights on each partition. Increasing the constant λ the balance criteria can be relaxed.

3 Approach Overview

Many different methods to tackle the graph partitioning problem have been investigated and described before. Yet there has not been given much attention to the divide-and-conquer paradigm in this context. Many efficient algorithms to solve different types of problems are based on the divide-and-conquer paradigm which shows how powerful a tool it is. In the following a divide-and-conquer based framework will be presented and evaluated. It is designed to be used on distributed systems or on a multi-core system. Theoretically any divide-and-conquer algorithm can be divided into three steps: initialization, recursion, recombination. These steps are also reflected in the framework.

3.1 Roles

The three steps of the divide-and-conquer paradigm are reflected in the framework as three roles. The methods that each role uses can be implemented independently of the other roles. Possible implementations are described in the respective chapters.

3.1.1 Splitter

In the initialization step the input problem has to be split into several subproblems. In the context of graph partitioning that means the input graph has to be distributed so that a partitioning can be computed on smaller subgraphs, which can then be processed independently in concurrent executions. In the framework this role is called "splitter" since it splits the input graph into several subgraphs. The splitter does not have to compute a fully balanced partitioning, however it should assign a reasonable amount of work to every available machine or thread.

3.1.2 Worker

The classical understanding of the recursion step would be that it divides subproblems in even smaller subproblems. It recurses until the subproblems are small enough to solve them directly. In terms of graph partitioning this means that subgraphs are partitioned until they reach a reasonably small size. In the framework this step is performed by the workers. After the splitter assigned a subgraph to every worker, this subgraph is partitioned into even smaller subgraphs. The workers run concurrently and use either a clustering algorithm or a minimal cut algorithm. Most likely the workers' algorithm has exponential runtime, due to the NP-hardness of the problem and the goal to find dense clusters or good minimal cuts. This reflects the actual idea behind the framework to approximate linear runtime through exponential time executions on smaller subproblems.

3.1.3 Merger

After the input problem was split into small subproblems the solutions of the subproblems have to be recombined into a solution for the initial problem. In the context of graph partitioning this step deals with subgraphs which have to be joined together. This role is called "merger" since it merges small subgraphs together so that they result in bigger subgraphs. It has to merge the subgraphs so that a previously given number of balanced partitions results of it. For this purpose the small subgraphs can be seen as meta-vertices. Every meta-vertex contains a number of edges which defines the weight of such a meta-vertex. If the splitter uses a vertex-cut algorithm for the prepartitioning, every meta-vertex also contains some vertices of the input graph. A vertex that is replicated in two meta-vertices can be seen as a meta-edge between two meta-vertices, if there is more than one vertex that is contained in both meta-vertices, a weight can be attributed to the meta-edge. This weight reflects the number of common vertices of both meta-vertices. For better understanding the idea is visualized in figure 6.1

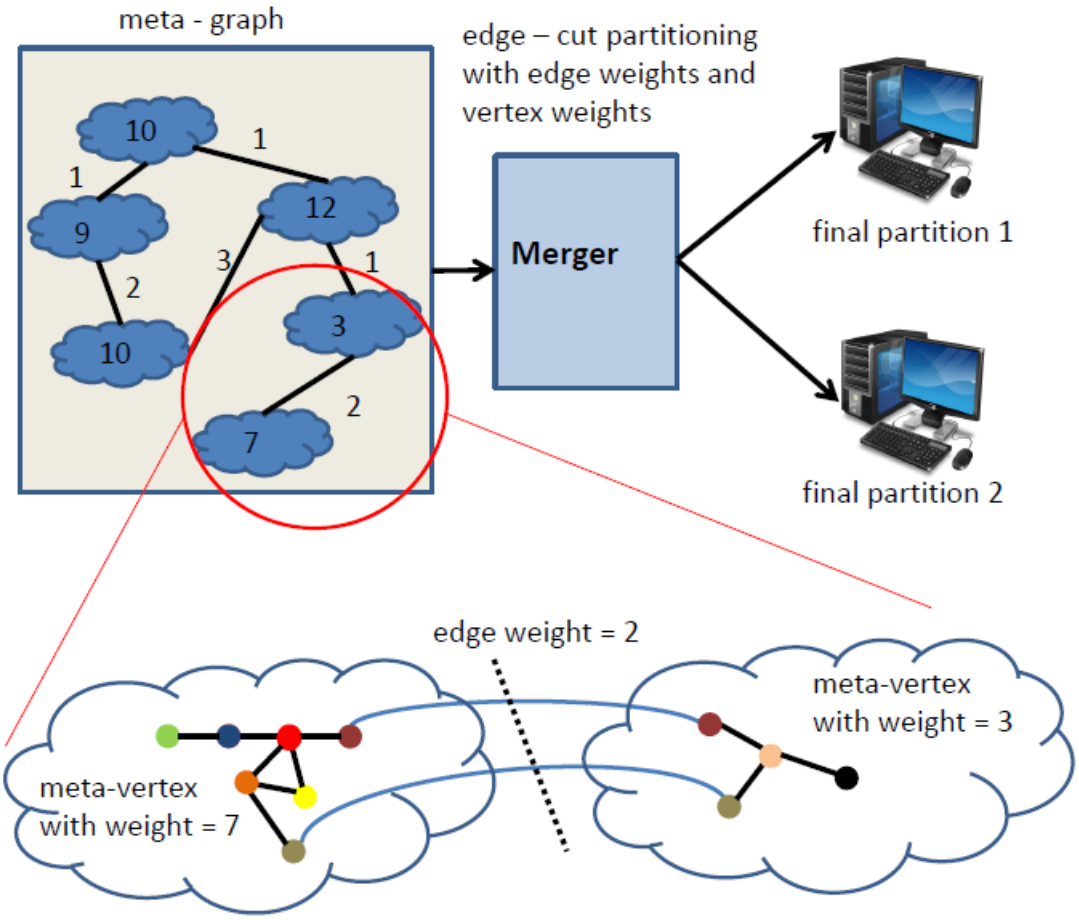

Figure 3.1: Meta-partitioning in the merger

The task of the merger can also be seen as an edge-cut on a graph with vertex-weights and edge-weights.

3.2 Graphical Overview of the System Model

The three steps of divide-and-conquer, as they are reflected in the divide-and-conquer based graph partitioning framework, are shown in figure 3.2.

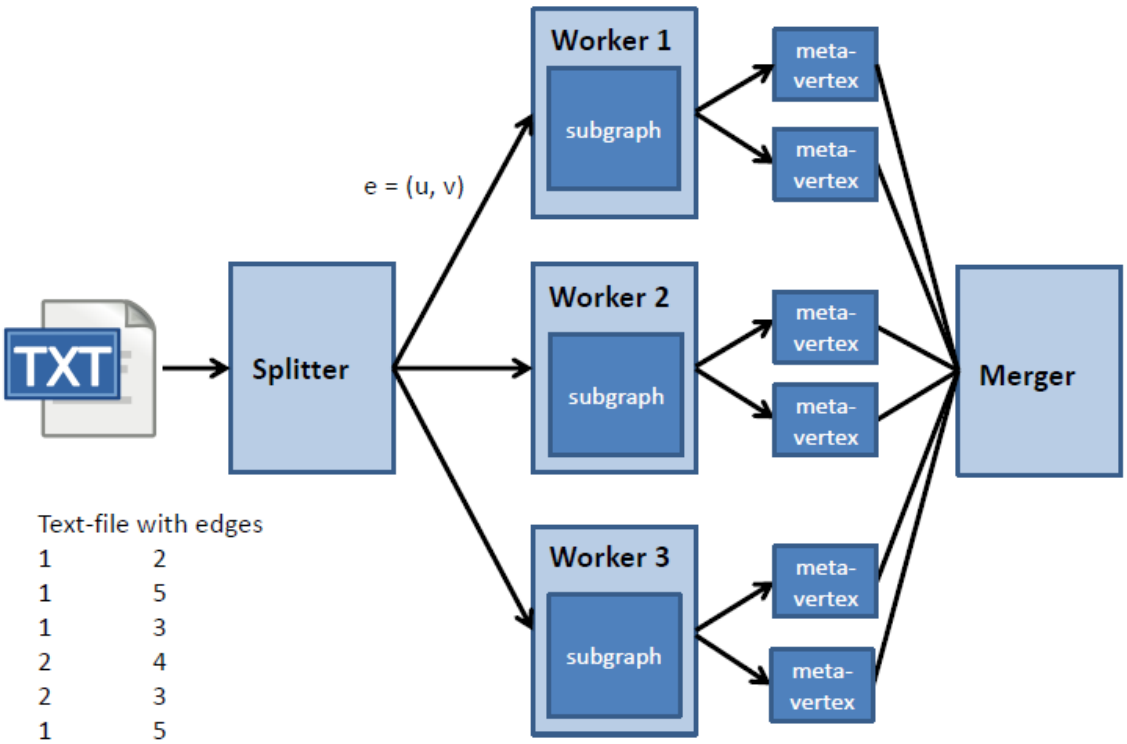

Figure 3.2: System Model

3.3 Approximating Linear Time

The underlying idea of the framework is to approximate linear time through several exponential time executions on smaller subproblems. Moreover this makes it possible to achieve better results without unreasonably high execution time. If the complete input graph was to be partitioned perfectly, the algorithm would have at least exponential runtime due to the NP-hardness. For any real world data this would take extreme amounts of time to find a solution. Through reduction of the problem size however, it is possible to use exponential time algorithms. Linear time algorithms for graph partitioning usually use some kind of a heuristic which means they trade partitioning quality for time. Therefore the divide-and-conquer based graph partitioning framework should also achieve better results than real linear time algorithms.

3.4 Interfaces

The framework is designed in a way that every step (splitter, worker, merger) can be implemented independently from the others. The three steps are implemented as modules. Each of those modules has an input and an output path from where it reads its data and writes its results after computation finished. To store data, txt-files are used. Reading and writing txt-files adds time to the total runtime but, for the sake of deeper investigation, the results of each step can be evaluated. Another advantage is that a module only has to be instantiated when it is needed. When its work is done, it can be deleted, thereby freeing memory resources. Listing 3.1 shows the implementation of the execute method of the framework.

Listing 3.1 Execute method of the framework

```
public void execute() {
      // Step 1: split graph in subgraphs
      Splitter splitter = new Splitter();
      numberEdgesInGraph = splitter.split();
      splitter = null;

      // Step 2: workers compute clusters on their subgraph
      WorkerArray workers = new WorkerArray();
      workers.computeClusters();
      workers = null;

      // Step 3: merger creates final partitioning
      Merger merger = new Merger();
      merger.merge();
      merger = null;
}
```

This makes it also very easy to introduce new steps. They can simply be implemented in a new module which must be called where it is supposed to be executed.

4 Splitter

The splitter represents the initialization step of the divide-and-conquer paradigm. It works on the whole input graph and is used to distribute it to the workers. The splitter processes the input graph quickly so that the actual work can be done by the workers. The splitter uses vertex-cut for the prepartitioning.

4.1 Task

The input graph has to be split in smaller work packages and distributed to the workers. Therefore the splitter performs a partitioning of the input graph but with other objectives than standard graph partitioning. Replication degree and balance only play a secondary role, whereas good clustering on the partitions is important. In addition the splitter must be able to cope with any possible order in which the input graph is read. If not, some workers may be left with no input at all which would not use well the available resources. Since the objective is to achieve a total runtime close to linear time and the whole input graph has to be distributed to the workers, the splitter has to use a linear time algorithm to perform its task.

4.2 Method and Implementation

Graph partitioning streaming algorithms exist for vertex-cut and edge-cut. In the case of vertex-cut they read edges from a stream and assign each edge once to a partition. Using a heuristic function they decide on which partition the current edge should be placed. There is no exchange of edges between partitions at a later time which means the algorithm's runtime is linear in the number of edges. Furthermore streaming algorithms keep the partitioning balanced an avoid vertex replications by trying to assign edges to partitions on which at least one of the edge's vertices already exists. This makes them a suitable choice for the splitter. The use of a streaming algorithm for the splitter implies that the whole divide-and-conquer process will never be faster than a simple streaming algorithm.

The simplest implementation of a streaming algorithm would be an algorithm that hashes the input from the stream to the partitions. This guarantees to achieve the best possible balance. Evaluations in literature show that this leads to very high replication degrees [PQD+15] which implies poor connectivity for the workers' input. A more intelligent implementation is the greedy assignment which takes existing vertex replications into account for its edge assignments. Therefore it is necessary to keep track of the vertices that have already been assigned to a partition.

In order to know which vertices exist on which partition, streaming algorithms need to remember all their previous edge assignments. Therefore they need a special data structure. This data structure was implemented as a HashMap. The HashMap's keys are vertex ids and the data is a HashSet with partition ids. So the function $A(v)$ which returns the partitions on which v is contained can be computed in $O(1)$. In order to know the current partition size there is a counter that counts how many edges were assigned to each partition.

In the current implementation of the framework, the input graph is read from a txt-file in which every edge is stored as two integers. These integers represent the vertex ids of the two vertices that are connected by the edge. The streaming algorithm assigns the incoming edges one by one to new txt-files which will be processed concurrently by the workers once the whole graph was read. The streaming algorithm has to know the number of partitions (representing the available machines) from the beginning of the process so that it can create one txt-file with data for each machine. It is not possible to compute this number dynamically depending on the graph's structure. The order in which the splitter receives the incoming edges depends on the application that reads the graph. Most of the times the graph's edges will be read in breadth first order or depth first order, for example in the case of a web-graph read by a web-crawler.

4.2.1 Greedy

An implementation of the greedy assignment [GLG+12] is shown in 4.1.

Algorithmus 4.1 Greedy Algorithm

Edge $e \leftarrow (v_i, v_j)$
if $A(v_i) = \emptyset \wedge A(v_j) = \emptyset$ **then**
    assign e to the least loaded partition in all partitions

else if $(A(v_i) \neq \emptyset \wedge A(v_j) = \emptyset) \vee (A(v_i) = \emptyset \wedge A(v_j) \neq \emptyset)$ **then**
    assign e to least loaded partition in the set that is not empty // $(*)$

else if $A(v_i) \cap A(v_j) \neq \emptyset$ **then**
    assign e to least loaded partition in $A(v_i) \cap A(v_j)$ // $(+)$

else if $A(v_i) \neq \emptyset \wedge A(v_j) \neq \emptyset \wedge A(v_i) \cap A(v_j) = \emptyset$ **then**
    assign e to the least loaded partition in $A(v_i) \cup A(v_j)$
    a new vertex replica is created accordingly

end if

This heuristic yields much better results than hashing does. However there are two problems with the algorithm. First of all, it does not take into account the vertex degrees (see chapter 2). The second problem is that it needs a uniformly randomized order of edges which is rarely to be expected in real world applications. If the edges in the stream arrive in breadth-first-order the algorithm assigns all edges to the same partition. This is due to the fact that starting from the second edge the algorithm will always end up in case 2 $(*)$ or in case 3 $(+)$ because there is always at least one replica of a vertex. For this reason, the Greedy algorithm is not robust enough to be used in the splitter. The HDRF algorithm tries to tackle those two problems.

4.2.2 HDRF

High Degree vertices Replicated First (HDRF) has an additional data structure where it stores a degree estimate for each vertex it has seen. The vertex degree is used as a tiebreak if a new replica has to be created. To avoid very imbalanced partitionings HDRF introduces a parameter λ which weights the importance of the balance. The effects of λ are given [PQD+15] as:

$$\begin{cases} \lambda = 0, & \text{agnostic of the load balance} \\ 0 < \lambda \leq 1, & \text{balance used to break the symmetry} \\ \lambda > 1, & \text{balance importance proportional to } \lambda \\ \lambda \to \infty, & \text{random edge assigment} \end{cases}$$

This is achieved by computing a score for the current edge on each partition. The total score for an edge on a partition consists of a score for vertex replications (0,1,2) and a score for balance. This balance score is multiplied by λ. If λ is set to 1 the HDRF algorithm works as shown in algorithm 4.2. If the edges in the stream arrive in random order $\lambda = 1$ is the best option. As said before the assumption that the edge stream is fully randomized does not fit well with real world applications. Therefore it is necessary to set λ to a value bigger than 1.0. This results in a higher replication degree but the imbalance can be kept low.

Algorithmus 4.2 HDRF Algorithm

Edge $e \leftarrow (v_i, v_j)$
if $A(v_i) = \emptyset \wedge A(v_j) = \emptyset$ **then**
    assign e to the least loaded partition in all partitions

else if $(A(v_i) \neq \emptyset \wedge A(v_j) = \emptyset) \vee (A(v_i) = \emptyset \wedge A(v_j) \neq \emptyset)$ **then**
    assign e to least loaded partition in the set that is not empty // $(*)$

else if $A(v_i) \cap A(v_j) \neq \emptyset$ **then**
    assign e to least loaded partition in $A(v_i) \cap A(v_j)$ // $(+)$

else if $A(v_i) \neq \emptyset \wedge A(v_j) \neq \emptyset \wedge A(v_i) \cap A(v_j) = \emptyset$ **then**
  if $degree(v_i) < degree(v_j)$ **then**
    assign e to the least loaded partition in $A(v_i)$, creating a new replica of v_j
  else
    assign e to the least loaded partition in $A(v_j)$, creating a new replica of v_i
  end if
end if

HDRF is the strongest known streaming algorithm, therefore it is the first choice. Of course, there are other streaming algorithms which are also robust enough to be used as splitter algorithms, e.g., SPowerGraph Degree. HDRF has the lowest replication degree on most graphs, which means the subgraphs on the partitions are better connected than in the results of other algorithms.

4.2.3 PSHDRF

Probabilistically Stabilized HDRF (PSHDRF) is an adaption of the HDRF algorithm to be used in the splitter. It uses the HDRF algorithm with $\lambda = 1$ (see algorithm 4.2) and works as shown in algorithm 4.3. The idea behind this modification is to make use of breadth-first and depth-first-ordered edge streams and trade some balance for a lower replication degree. λ is set to 1.0 so that the HDRF decisions do not take balance into account.

Algorithmus 4.3 PSHDRF Algorithm

```
Edge e ← (v_i, v_j)
float r ← randomNumber()

if r < threshold then
    assign e to the least loaded partition in all partitions
else
    assign e with HDRF
end if
```

Depending on the threshold parameter the random assignments happen more or less often. The bigger the input graph, the lower this threshold can be. For normal HDRF behavior (with $\lambda = 1$) the threshold can simply be set to 0. If the threshold is bigger than 0 and the edges in the stream arrive in breadth-first-order PSHDRF assigns all the incoming edges to a partition where there is already at least one replica of the edge's vertices. This goes on until a random assignment to the least loaded partition happens. The following edges will most likely all be assigned to this least loaded partition until the next random assignment happens. This behavior is supposed to lead to bigger and better connected subgraphs on the partitions.

4.2.4 Alternatives

Instead of using a streaming algorithm, a new algorithm could be designed. Through streaming the input several times, additional knowledge can be used in the algorithm. For instance, starting from the second streaming iteration, the total number of edges in the graph is known. Therefore the average number of edges that should be assigned to a partition can also be computed. Using a breadth-first-order for the edge stream, the algorithm can assign edges to the same partition as long as the edge limit per partition is not exceeded. Before the algorithm assigns an edge to a partition it should also check if at least one replica of the edge's vertices is contained on the partition. Due to the

breadth-first -order this will be the case most of the times. If not, the current edge is assigned to an empty partition. If no more empty partitions are left, the edge is assigned to a default partition. Once a partition reaches upon its limit of edges, the following edges are also assigned to another empty partition. As long as there are edges on the default partition, the streaming is repeated.
This algorithm guarantees that every worker receives exactly one connected component. If the final results will also be better, compared to a streaming algorithm in the splitter, is not sure. It would probably require a more sophisticated assignment of the edges so that the components on the workers are not only connected but also dense. The big advantage of such an algorithm would be that the workers could start their algorithm immediately. Currently they have to identify the connected components first, if not a clustering or minimal cut algorithm yields useless results.

4.3 Analysis

In the current implementation the splitter uses a streaming algorithm. Most streaming algorithms, like HDRF, compute a score for each edge on each partition and subsequently assign the edge to the partition where it achieved the highest score. If p is the set of partitions, the complexity of PSHDRF is $O(|p|)$ when the edge is hashed to the least loaded partition. Since the number of partitions is a constant which is independent of the input size it can be said that for the hashing case the complexity is $O(1)$. For the HDRF case the complexity of the algorithm is $|p| * |E| * c$, where $|E|$ is the number of edges in the input graph. c is a small constant that describes the computation time of the evaluation function of an edge on a partition. So the total complexity of the current splitter implementation is:

$$O(1) + O(|p| * |E| * c) \in LINEAR\ TIME \tag{4.1}$$

4.4 Conclusion

The splitter algorithm is crucial for the overall performance of the implementation. If the subgraphs computed by the splitter are not well connected, the worker step does not have any effect. It would simply identify small clusters that are not connected to the rest of the subgraph. Figure 4.1 visualizes this problem.

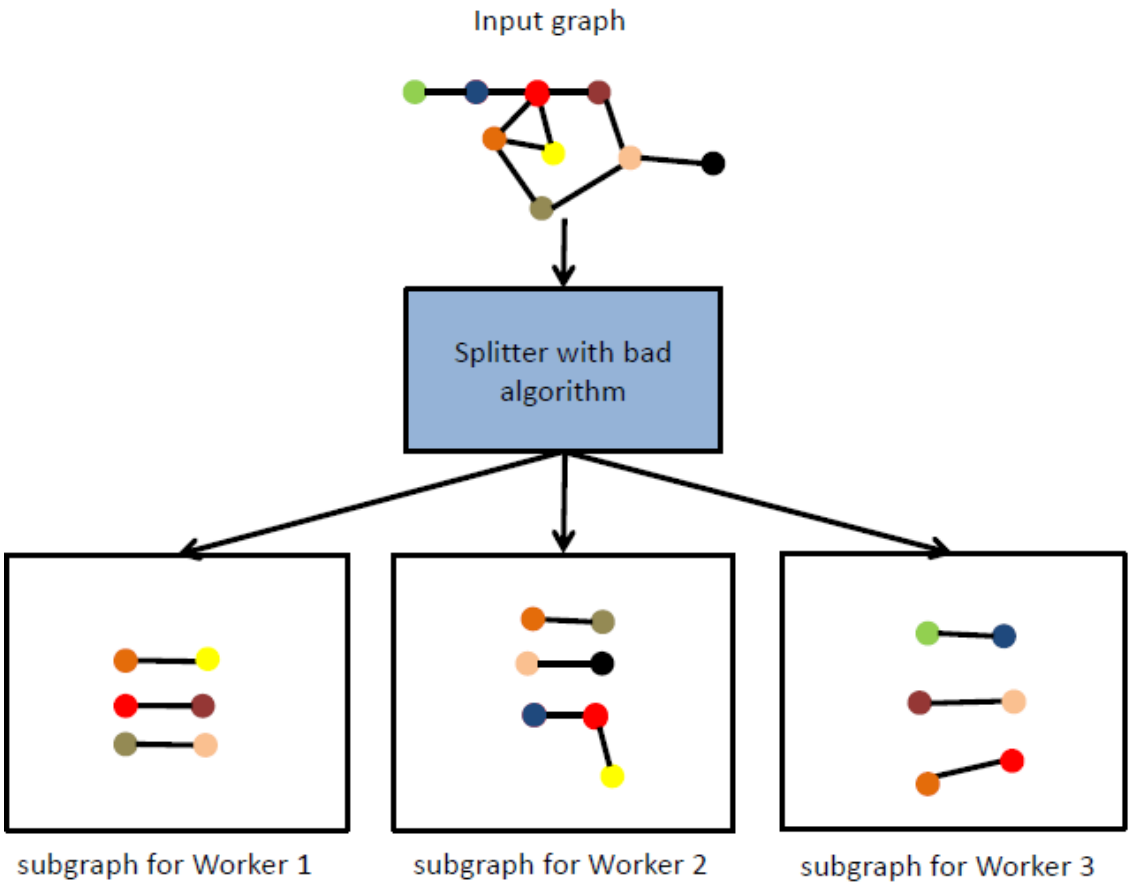

Figure 4.1: Result of a bad algorithm in the splitter

Concerning the balance in this step, it is only important that each worker has a part of the graph, of course it should not be too big in order to make good use of the available resources. Yet balance can be traded so that stronger connected clusters can be achieved.

5 Worker

Once the splitter has read the whole input graph and assigned a subgraph of it to each worker the recursive step of the divide-and-conquer paradigm starts. The workers can either be implemented as threads or as separate applications on a remote machine. The workers' algorithm does not have to be recursive, it can either compute a partitioning directly on its input or recursively partition its input in subgraphs until a certain criteria is met.

5.1 Task

The workers' task is to identify dense clusters on their subgraph. First they have to check if their subgraph is connected or dissociates into several independent components. In the second step they run their algorithm on each component independently. Since the workers' input is rather small compared to the original graph, they can use more sophisticated algorithms, even with exponential runtime behavior. It is also possible to use knowledge about the whole input graph. The goal of this is to find dense clusters and cuts with low replication degree, using relatively little time. The size of those clusters is not important, though they should not be too different in size, since this could lead to imbalanced results in the merger step.

5.2 Method and Implementation

In the current version of the framework the workers are implemented as threads. Every worker reads the edges of its subgraph from its txt-file and stores the whole subgraph in a special data structure. This data structure is designed to allow algorithms quick access to the data. The workers use Karger's Algorithm to further partition their subgraphs. Karger's Algorithm is a probabilistic algorithm that computes a minimal edge-cut on a graph. In this way clusters are identified and better cuts can be found than a streaming algorithm could compute. Since the workers are implemented as threads, at this point of time the whole input graph is stored in the RAM of the machine.

5.2.1 Data Structure

To store the entire input graph of a worker a HashMap is used. The keys of the HashMap are vertex ids (int), the values are HashSets of Integers. For each new edge one access to the HashMap is necessary for each of the edge's vertices. If an entry exists for the vertex id, the other vertex's id is added to the HashSet, if not a new HashSet is created and the other vertex's id is added. For the second vertex respectively. In the end the HashMap contains a list of all neighbors for each vertex so that they can be accessed in $O(1)$ for any vertex. The following listing 5.1 shows how the graph's edges can be extracted from this data structure. It takes $O(|V| + |E|)$ steps to extract all edges.

Listing 5.1 Algorithm to extract all edges of undirected graph from data structure

```
public static HashSet<Edge> extractEdges(HashMap<Integer, HashSet<Integer>>
    vertexToNeighbours){
      HashSet<Edge> edges = new HashSet<Edge>();

      for(Entry<Integer, HashSet<Integer>> entry : vertexToNeighbours.entrySet()){
            HashSet<Integer> neighbours = entry.getValue();
            int vertex = entry.getKey();

            if (neighbours != null) {
                  for (Integer neighbour : neighbours) {
                        if (neighbour < vertex) {
                              edges.add(new Edge(neighbour, vertex));
                        }
                  }
            }
      }
      return edges;
}
```

5.2.2 Karger's Algorithm

Karger's Algorithm is a probabilistic algorithm for edge-cut minimal cuts. It partitions a graph into two parts by cutting the least number of edges. This is done through contraction of randomly chosen edges. When an edge (v_i, v_j) is contracted, the two vertices of the edge become on new vertex. The newly created vertex has the neighbors:

$$neighbors(v_{ij}) = neighbors(v_i) \cup neighbors(v_j) \setminus \{v_i, v_j\} \tag{5.1}$$

This step is repeated iteratively until there are only two vertices left. The resulting cut's weight is given by the number of edges that connect those two remaining vertices. The algorithm has to be repeated so that the error probability becomes low enough. In

literature the number of necessary repetitions on a graph with $n = |V|$ vertices is given as:

$$T = n^2 \log n \tag{5.2}$$

The probability of not finding the minimal cut after those repetitions is:

$$p_{err} = \left[1 - \binom{n}{2}^{-1}\right]^T \leq \frac{1}{e^{\ln n}} = \frac{1}{n} \tag{5.3}$$

The execution time for the algorithm on a graph with n vertices and m edges is:

$$O(n^2 * m * \log n) \tag{5.4}$$

Therefore if one wants to be sure to find the minimal cut, the total execution time of the algorithm is bigger than $O(n^4)$. Compared to other minimal cut algorithms, Karger's Algorithm is very slow. Nevertheless it is a sufficiently good choice for the workers since it is not necessary to find the minimal cut but just to find a cut with low weight. In many cases real world graphs have at least one vertex that is connected to the rest of the graph by just one edge. So the real minimal cut would be through that edge, cutting off one single vertex. Such cuts would not be useful in the current setting. Therefore the number of repetitions can be reduced drastically and the algorithm still serves its purpose.

5.2.3 Karger's Algorithm Adapted for Vertex-Cut

As Karger's Algorithm is designed to compute an edge-cut partitioning, it has to be adapted to vertex-cut in order to use it in the workers. This adaption is not difficult: after Karger's Algorithm was executed, the cut is shifted to one end of the cut edges. The shift is always performed in the way that the cut edges fully belong to the half of the partitioning with less edges. This method only works if the vertices and edges do not have weights. Figure 5.1 visualizes the shift of the edge-cut to vertex-cut.

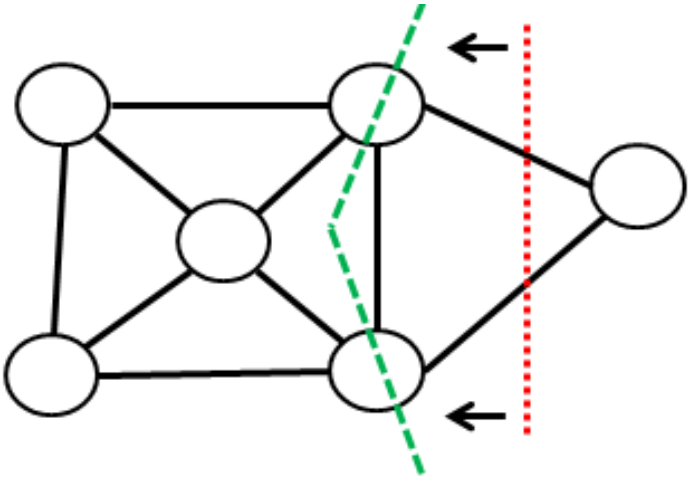

Figure 5.1: Shift of edge-cut to vertex-cut

In the current implementation this operation can be performed in $O(|E_{bigger}|)$, where E_{bigger} is the set of edges of the bigger partition.

5.3 Analysis

In Karger's algorithm the edge contraction step can be implemented in $O(n^2)$ [Aro10]. Since the workers repeat the step only a small constant number of times the worker's algorithm takes $O(cn^2) \in O(n^2)$. In the faster version of the algorithm (Karger-Stein) the contraction step still takes $O(n^2)$ and the total runtime is only $O(2n^2 \log n)$ because of less contraction steps. Another algorithm to find a minimal cut is the Stoer-Wagner algorithm. Its total runtime is $O(nm + n^2 \log n)$ [SW97]. If a Fibonacci-Heap is used as data structure, the Stoer-Wagner-algorithm can be as fast as $O(m + n \log n)$ which would make it an interesting alternative. For the sake of simplicity, and due to the fact that only 10 - 25 repetitions of the the contraction step are necessary, it is a fair choice.

Figure 5.2 and figure 5.3 visualize the idea of the divide-and-conquer approach. Instead of running an $O(n^2)$ algorithm on the whole input, smaller problem instances are solved in concurrent executions.

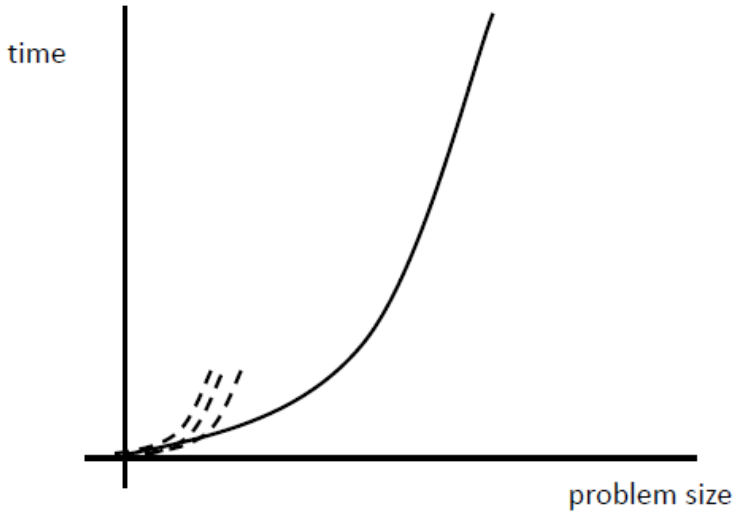

Figure 5.2: Runtime on one big problem instance and several concurrent executions on smaller instances

If the concurrent executions were to be run sequentially, linear time can be approximated.

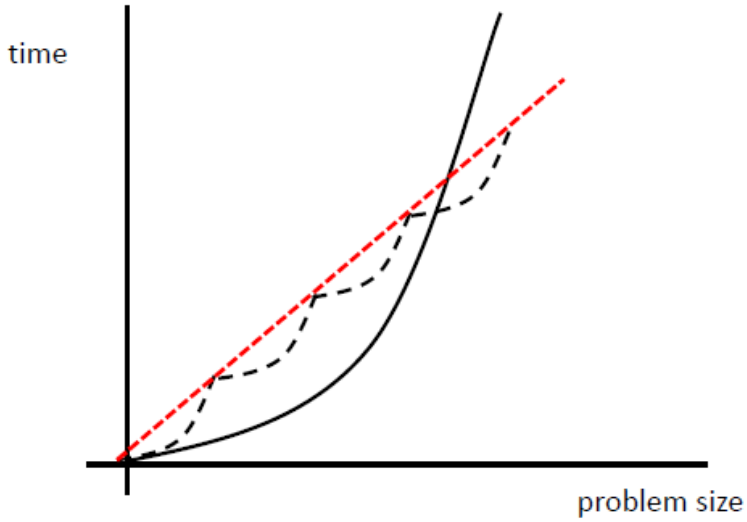

Figure 5.3: Approximating linear time

5.4 Conclusion

In the current implementation the workers do not exchange information nor do they use knowledge about the original input graph. Clearly, for partitioning a small graph the same rules hold true as for partitioning a big graph. But in the case of the workers their subgraphs represent more or less a random sample of the original graph. Therefore it could be that it is not possible to compute a reasonable cut on the subgraph without taking into account the original graph. This point has to be investigated in depth in the evaluation chapter. Figure 5.4 visualizes how a good partitioning decision on a subgraph can be a bad decision in relation to the original graph.

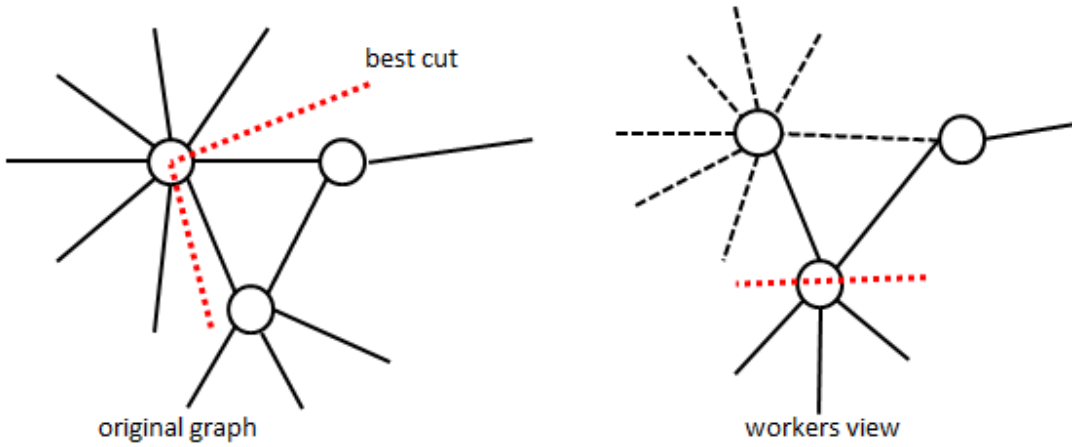

Figure 5.4: Cut on worker's view of graph

Karger's algorithm is a good choice because it can be used either recursively, dividing its input in two until it reaches a predefined size, or just once. Furthermore its runtime and preciseness can be easily influenced by the number of repetitions.

If more workers than final partitions are used, additional vertex replications will be introduced. These vertex replications would not be necessary if the input was directly assigned to a lower number of final partitions.

6 Merger

The merger implements the recombination step of the divide-and-conquer paradigm. As input it takes the clusters which were computed by the workers in the previous step. Those clusters can be seen as meta-vertices with a weight, representing the number of edges contained in them. If a vertex from the input graph is contained in two meta-vertices, there is an edge between those two meta-vertices. Since it is possible that this is the case for several vertices from the original input graph, those edges can have a weight bigger than one. Therefore the merger's input can be seen as a graph with edge and vertex weights (see figure 6.1).

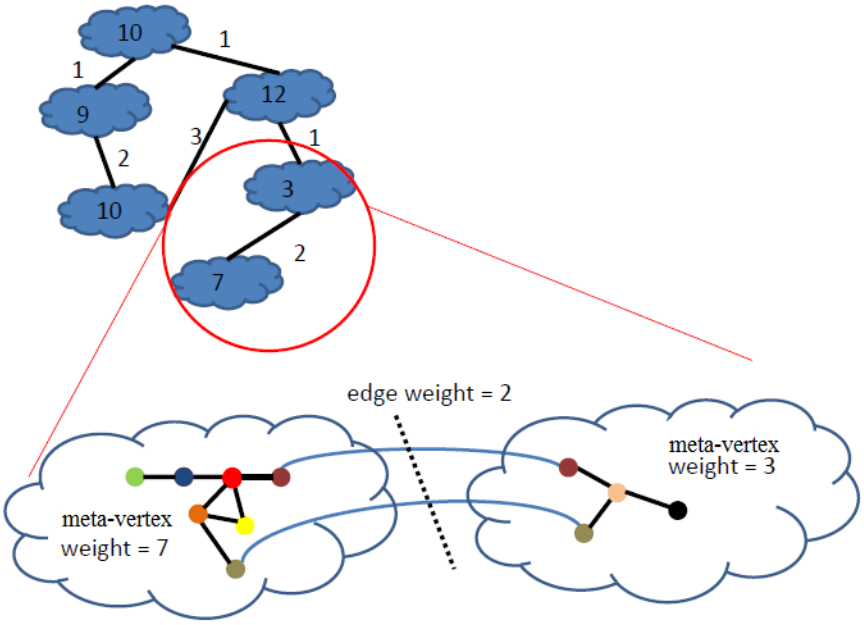

Figure 6.1: Meta-graph for the merger

By assigning meta-vertices to partitions they are merged together. The more common vertices two meta-vertices contain, the better they fit together and the more vertex

replications are eliminated. This assignment of meta-vertices to partitions leads to an edge-cut on the meta-graph. Edge-cut is especially well fit for sparse graphs which makes it a good choice for the meta-graph. Only if the average vertex degree of the input graph is very high (e.g., fully connected graphs), the meta-graph will also be dense.

6.1 Task

The merger computes a minimized edge-cut partitioning on the meta-graph such that the sums of weights of the meta-vertices in each partition are as equal as possible. To achieve this, meta-vertices are merged together by assigning and reassigning them to different partitions. Finally this results in a balanced k-way vertex-cut partitioning of the input graph.

6.2 Method and Implementation

In the current implementation the merger uses iterated local search (ILS) to compute an assignment of the meta-vertices to the final paritions. By placing connected meta-vertices on different partitions, the edge between them is cut. At first the meta-vertices are distributed over the partitions so that a well balanced partitioning is produced. This rather random partitioning is iteratively improved through reassignments of meta-vertices to different partitions. Those reassignments reduce the number or weight of cut edges and is bound to a balance constraint. To avoid that the search gets stuck in a local optimum a mutation function is applied to the currently best solution. The mutation is realized as some random exchanges of small meta-vertices and a transfer of some meta-vertices from the biggest partition to the smallest partition. This transfer of meta-vertices from the biggest to the smallest partition is important because it introduces a bias via balanced partitionings.
In order to know how well a meta-vertex fits with other meta-vertices, a fit value is used. If there is a meta-vertex mv1 and a set of meta-vertices (mvs), the fit value is computed as shown in equation 6.1.

$$fitValue = \frac{|mv1.verticeSet \cap mvs.verticeSet|}{|mv1.verticeSet|} \tag{6.1}$$

For the random exchange two partitions are chosen randomly and the worst fitting meta-vertex from each partition is exchanged. The fit value is the ratio of common

vertices to the total number of vertices in the meta-vertex (see equation 6.1), which avoids higher fit values for meta-vertices that simply contain more vertices.

The ILS reassigns meta-vertices based on the difference between their inner connections and their outer connections. Inner connections are defined as the number of common vertices contained in the other meta-vertices on a partition and the meta-vertex to exchange. The outer connections are defined as the number of common vertices with the meta-vertices of another partition. If there are more outer connections than inner connections for a meta-vertex on a certain partition, it has to be assigned to the other partition. Thereby transforming the outer connections to inner connections and vice versa. From the viewpoint of the meta-graph it can be said that the cut edge's weight between the two meta-vertices has been reduced and the overall partitioning improved (see figures 6.2 and 6.3).

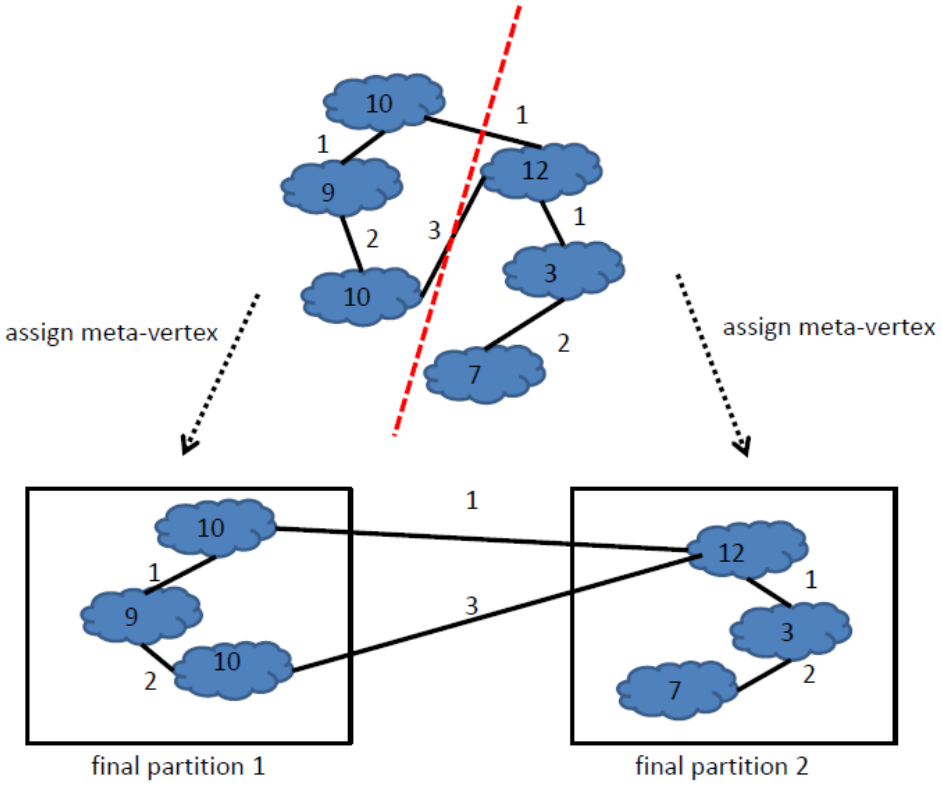

Figure 6.2: Edge-cut through meta-vertex assignment

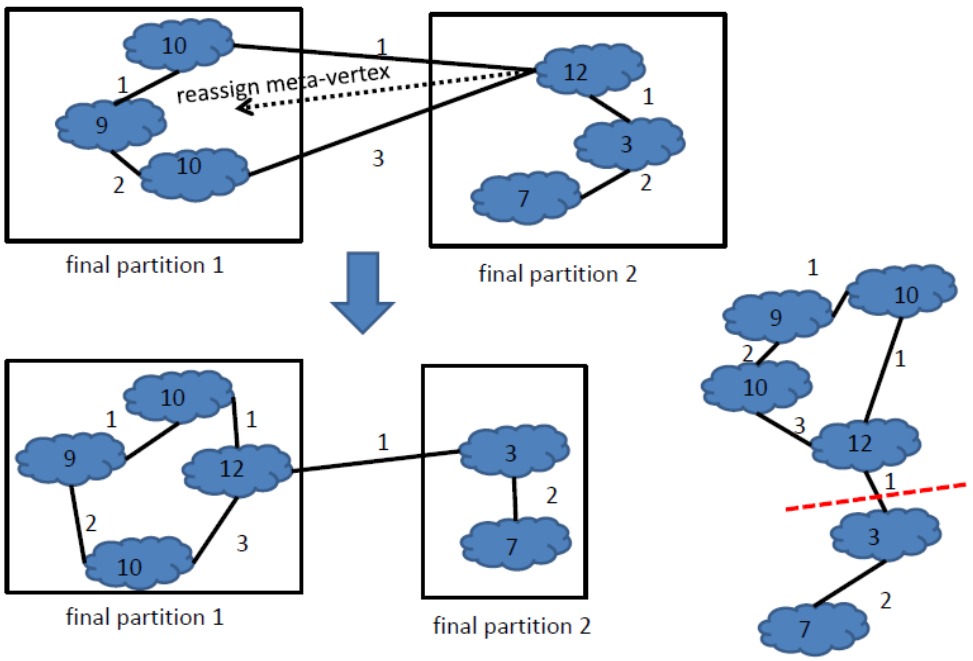

Figure 6.3: Cut weight reduced through reassignment of meta-vertex

Any exchange can only be performed if the imbalance of the partitioning does not get too high. This is important to consider, otherwise all meta-vertices end up on the same partition. This can easily be explained: The more meta-vertices there are already on a partition, the more common vertices there are between the partition and any other meta-vertex.

In the current implementation the local search algorithm iterates over all meta-vertices on all partitions. It checks greedily for every meta-vertex if an assignment to one of the other partitions would reduce the number of outer connections. As soon as one possibility is found, the meta-vertex is reassigned. The process continues until no more exchanges can be found. This is defined as a local optimum so one execution of the local search is complete.

6.2.1 Iterated Local Search

ILS is a simple but powerful technique to find high quality approximate solutions for combinatorial problems. The basic idea of ILS is the repeated application of a local search algorithm and a mutation function to the current search point. Furthermore an acceptance criterion is used to decide whether to take the new solution or to go back to the old search point (see algorithm 6.3).

Algorithmus 6.1 Iterated Local Search

$s_0 \leftarrow$ generateInitialSolution
$s \leftarrow$LocalSearch(s_0)
repeat
    $s^{'} \leftarrow$ Mutate(s)
    $s^{''} \leftarrow$ LocalSearch($s^{'}$)
    $s \leftarrow$ AcceptanceCriterion($s, s^{'}$)

until termination condition met

The start solution can be generated either randomly or with the help of a heuristic function. Using the start solution the result is iteratively improved through the local search algorithm. Each application of the local search algorithm leads to a local optimum in the solution space. In order to improve the results and get close to the global optimum, the currently best solution is mutated and local search is executed again. If the result of the local search has improved, the new result is set as the currently best solution, if not the search goes back to the last best solution. The acceptance criterion decides whether one solution is better than another one. Since it has been shown [Stü06] that better solutions lie closer to the global optimum, this procedure leads the local search algorithm closer to the global optimum. The mutation is a decisive step in the process and has to be designed carefully. If it is too weak, the search gets stuck in a local optimum because the local search algorithm returns to the last local optimum after only a few steps. If the mutation is too strong, the search cannot converge. Figure 6.4 [Stu03] shows how a solution s^* is mutated into $s^{'}$ and becomes an improved solution $s^{*'}$ after the application of the local search algorithm.

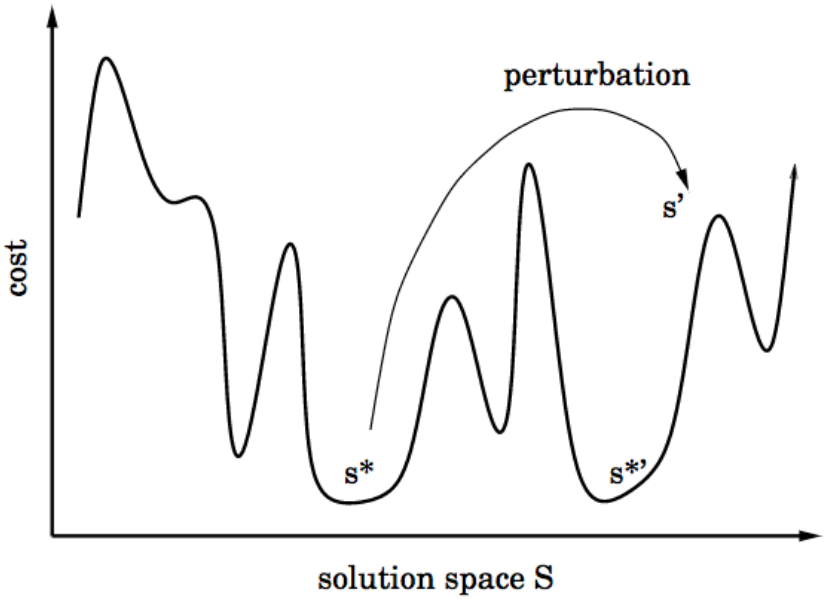

Figure 6.4: Searchspace and mutation of the solution in ILS

6.2.2 Fast Iterated Local Search

Since the strict implementation of ILS that was presented above (algorithm 6.3) is rather slow and strongly dependent on the number of meta-vertices, a faster implementation of the merger will also be evaluated. Strictly speaking it is not an ILS but it is directly derived from it. The local search algorithm does no longer have to compute a better solution from its input. Instead a simulated annealing value is introduced which defines how well a meta-vertex has to fit with the other meta-vertices on its partition. It is used as a threshold. In every iteration all meta-vertices that have worse fit values than the threshold defines, are eliminated from their partition. Then they are reassigned to the partitions optimizing only the partitioning's balance. Then the threshold value is decreased. Once the threshold is so low that no meta-vertices are eliminated any more, the iteration terminates. Algorithm 6.2 shows this procedure.

Algorithmus 6.2 Fast Local Search

```
repeat
    empty(badmeta-vertices)
    for all partitions : currentState do
        badmeta-vertices .add(currentPartition.eliminateBadmeta-vertices(threshold))
    end for
    if badmeta-vertices.size > 0 then
        reassignmeta-vertices(badmeta-vertices)
        decrement(threshold)
    end if
until badmeta-vertices.size = 0
```

Algorithmus 6.3 Fast Iterated Local Search

```
s_0 ← generateInitialSolution
s ←FastLocalSearch(s_0)
repeat
    s' ← Mutate(s)
    s'' ← Fast LocalSearch(s')
    s ← AcceptanceCriterion(s, s')

until termination condition met
```

The acceptance criterion only accepts better solutions so that the series of solutions improves over time. As termination condition a number of iterations can be used. The elimination of badly fitting meta-vertices could even be done concurrently for big amounts of meta-vertices.

6.3 Analysis

The ILS can be implemented in many different ways, however its runtime will always depend on the number of meta-vertices. What appears to be contradictory is that more workers will lead to more meta-vertices and therefore to a longer runtime for the ILS. This means that the total runtime will not go down simply by increasing the number of workers. However the number of workers can not be decreased too much either since this leads to too few meta-vertices. The fewer meta-vertices there are, the less degrees of freedom there are for the ILS. A faster implementation of the ILS reduces time cost, though this will produce worse results.

Since streaming algorithms lack complete knowledge about the graph and can see each stream element only once, there will be suboptimal decisions. Most of the times streaming algorithms do not compute connected subgraphs but rather several connected components on each partition. Only if the average vertex degree of the input graph is very high, the subgraphs on the partitions are likely to be connected. Through the reassignment of such connected components between the partitions, some vertex replications can be deleted by the ILS which leads to a lower replication degree. Those connected components might sometimes comprise just one single edge. Figure 6.5 visualizes this process of reassignment.

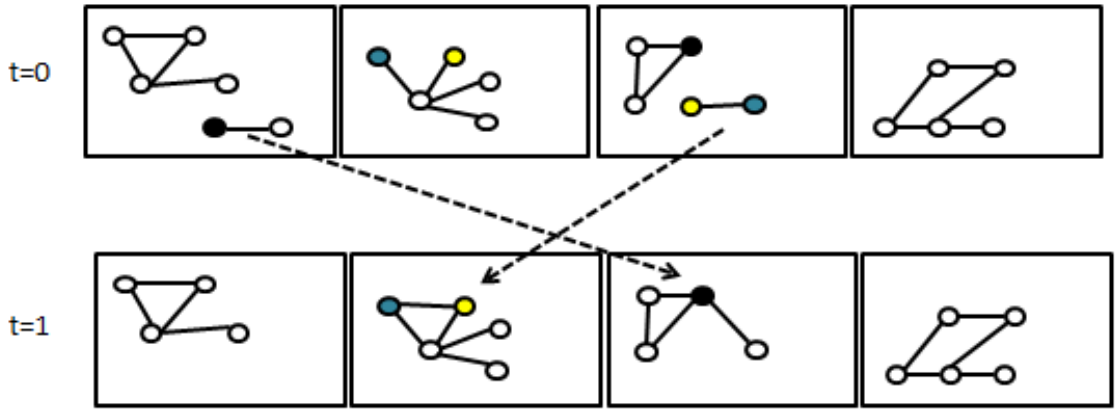

Figure 6.5: ILS improves streaming algorithm results

After the reassignment of the two disconnected edges to other partitions, three replications could be deleted without worsening the partitioning's balance. Therefore ILS is a good choice to be used in the merger.

A streaming algorithm's result can not become worse by applying ILS to it. In general the number of workers will be set higher than the number of final partitions which can have a negative influence to the replication degree in the end of the partitioning. To make the implementation of the framework provably better than a normal streaming algorithm, if the number of workers is set to be the same as the number of final partitions, it is necessary to analyze the ILS's initial solution in more depth. When the workers subpartition their components further, it is possible that new, unnecessary vertex replications are introduced. If a given component is cut by a worker in a bad way it might be possible that it can not be reunited because of the balance constraint in the ILS. The balance constraint is important, without it all meta-vertices would be assigned to the same partition thereby leading to a replication degree of 1.0 and complete imbalance. The problem of unnecessary vertex-replications introduced by worker subpartitioning and a random initial solution for the ILS is visualized in figures 6.6 to 6.8.

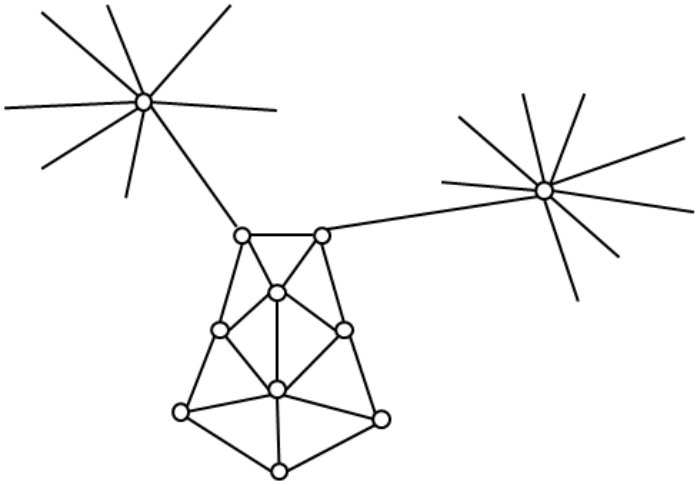

Figure 6.6: Part of a dense graph

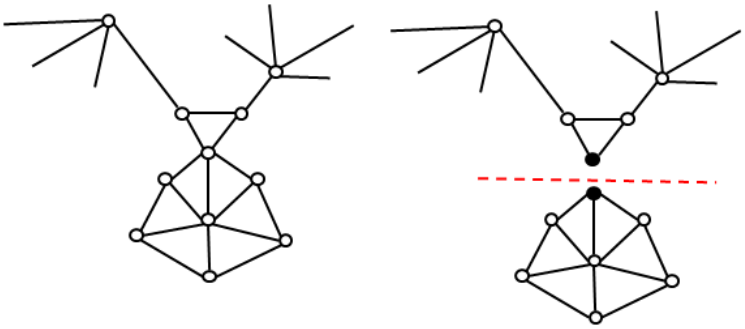

Figure 6.7: A worker's view on one part of the graph

Figure 6.8 shows how the ILS reassigns the two small components to the partitions where they fit better. Because of the balance restriction no further reassignments of the big components are possible and unnecessary vertex replications are introduced. The small components are not reassigned either since the inner connections of them are equal to their outer connections.

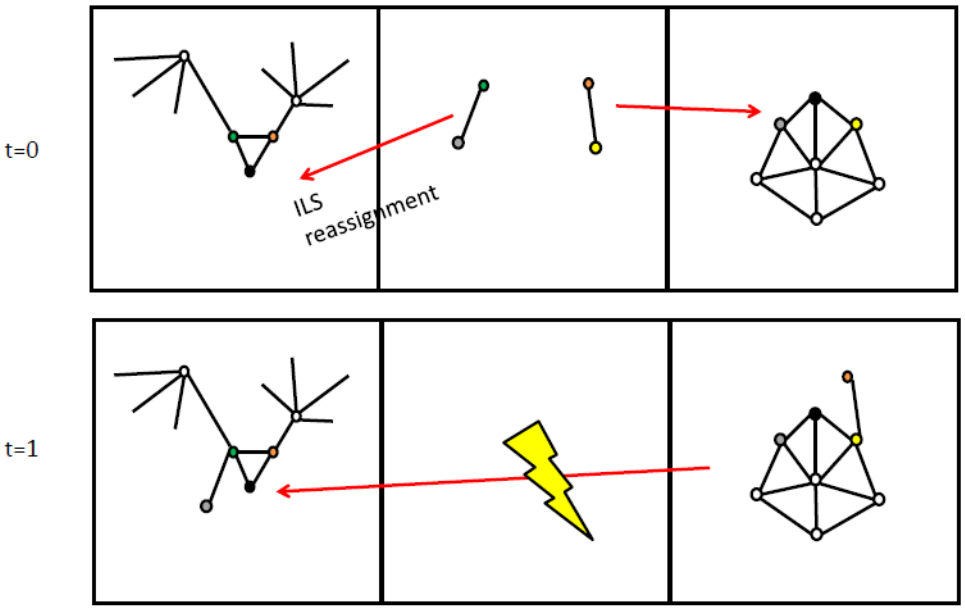

Figure 6.8: ILS with additional replications

Tagging tags all clusters that belonged to the same component on a worker. In the ILS's initial solution all clusters (=meta-vertices) with the same tag are assigned to the same meta-partition thus avoiding the problem presented above.

6.4 Conclusion

If tagging is used and the number of workers is equal to the number of final partitions, the refinement by ILS using subpartitioning in the workers will never lead to worse results but will most likely improve the solution's quality. Furthermore it would be desirable to be able to define an upper bound for the ILS's runtime since the number of meta-vertices influences the runtime the most. However it is difficult to assure a maximal number of meta-vertices since it depends on several factors such as number of used workers and the input's structure. The solution to this problem will be left for further investigation.

Facebook

7 Evaluation

This chapter presents the experimental findings based on the results for different parameter combinations. The presented implementation of the divide-and-conquer based framework will be called multistep algorithm (MSA). First the effects of different splitter algorithms are evaluated, then the improvements of an ILS applied to streaming graph partitioning algorithm results are investigated. After that additional degrees of freedom are introduced by using more workers than final partitions. In the next step the workers partition their subgraphs further using Karger's algorithm and the effects of tagging subclusters are evaluated. Ultimately the results of the multistep algorithm are compared to other graph partitioning algorithms. $w = x \times fp$ indicates the ratio of workers to final partitions.

7.1 Input Graphs

As input for the evaluations the following graphs were used.

graph	#vertices	#edges	avg vertex degree
Facebook	4,039	88,234	21.846
Twitter	81,306	1,944,439	21.747
World-Wide-Web	325,729	1,497,134	4.596
Wiki-Vote	7115	103,689	14.573
email-EuAll	265,214	420,045	1.583
movielens100k	1682	100,000	59.453

Table 7.1: Graphs used for evaluation

If the extension "random" is added to a graph's name, the edges were read in random order and the extension "combined" means that edges were read in breadth-first-order.

7.2 Variables

The following table shows which settings can be adjusted in order to influence the system's performance.

step	variable	explanation
splitter	algorithm	defines how the input graph is distributed to the workers
workers	cluster split size	minimal cluster size for Karger start
workers	number	how many workers
workers	Karger repetitions	repetitions of the edge contraction step
merger	algorithm	how to perform the ILS
merger	epsilon	allowed imbalance in ILS
merger	iterations	how many times the local search is iterated

Table 7.2: Variables in the system

7.3 Evaluation of Splitter

In this section the splitter is evaluated. Depending on the input graph and the algorithm for the splitter the results differ strongly from each other.

7.3.1 HDRF on Social Graphs

Social graphs have some highly connected vertices (hubs) so that the subgraphs on the workers do not dissociate into too many independent connected components. The average vertex degree is the decisive factor that determines how strongly the workers' subgraphs are connected. On "movlielens100k" every worker receives exactly one connected component for any algorithm and up to 256 workers. This is due to the high average vertex degree of the graph. Figures 7.1 - 7.4 visualize how the total number of connected components goes up depending on the number of workers and the used algorithm in the splitter. As input the "Facebook" and the "Twitter" graph were used with random edge order and breadth-first edge order. In one case the λ parameter of HDRF was incremented in the other case λ was set to 1.0 but some edges were hashed. λ was set as low as possible without incrementing it further.

Facebook

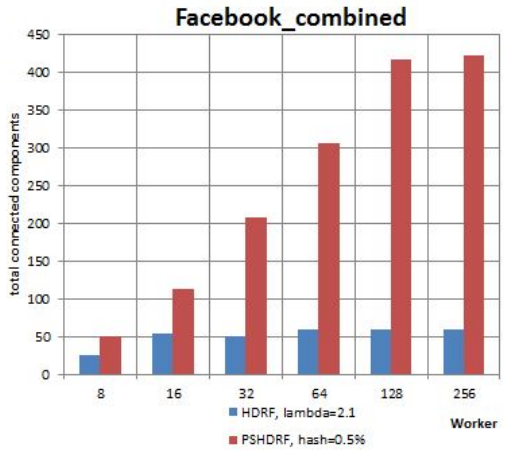

Figure 7.1: Total number of connected components depending on HDRF variant on input: Facebook-combined

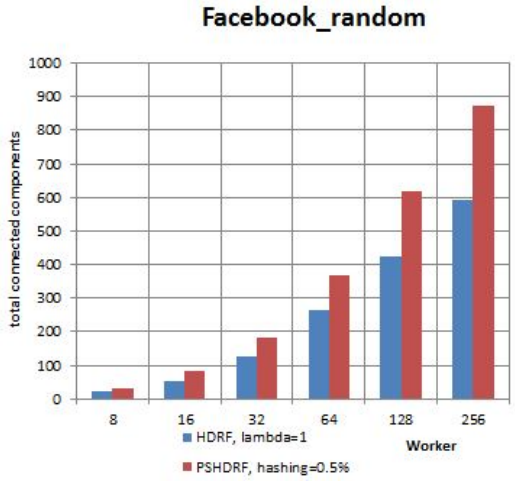

Figure 7.2: Total number of connected components depending on HDRF variant on input: Facebook-random

As the number of workers goes up, the total number of connected components also goes up. The increase can be described as proportional to the number of workers. Higher numbers of partitions require λ or the hashing probability to be increased, if not, some workers do not receive any input. In figure 7.1 this scaling problem can be seen, for higher numbers of workers (128, 256) the number of connected components does not change because some workers did not receive any input. It can also be seen that random edge orders produce higher numbers of independent components on the workers.

Twitter

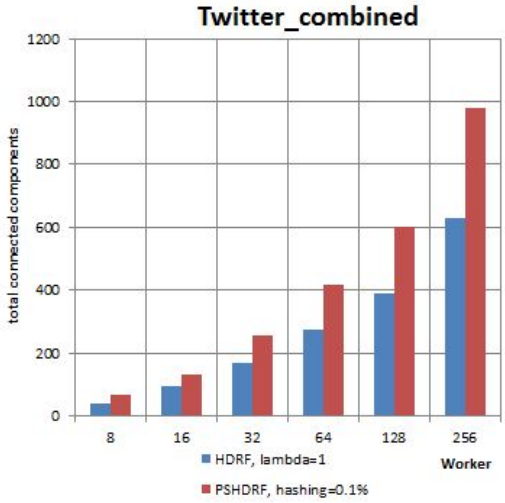

Figure 7.3: Total number of connected components depending on HDRF variant on input: Twitter-combined

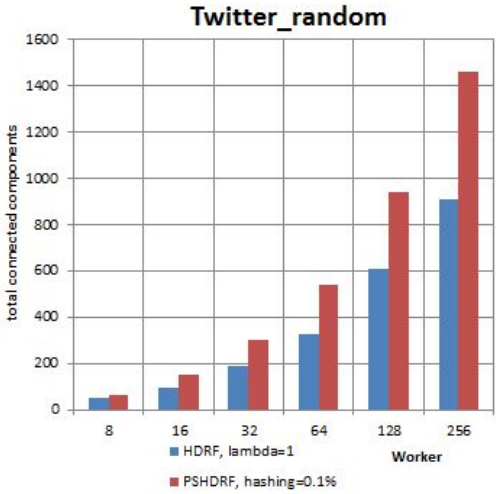

Figure 7.4: Total number of connected components depending on HDRF variant on input: Twitter-random

On the "Twitter" graph the same results as on "Facebook" can be observed. It is interesting to observe that HDRF is able to process "Twitter-combined" even with λ set to 1.0. How low λ can be set, can only be determined through various repetitions of the algorithm. A perfectly calibrated λ-parameter minimizes the number of independent components.

On both graphs the number of resulting independent components was lower when the edges arrived in breadth-first-order. This is an important finding that confirms the idea for a specialized splitter algorithm presented in chapter 4. Furthermore it is interesting because streaming algorithms normally perform best on randomized edge streams, whereas in the context of the splitter they perform better on sorted edge streams.

7.3.2 Other Algorithms and Different Input Graph Types

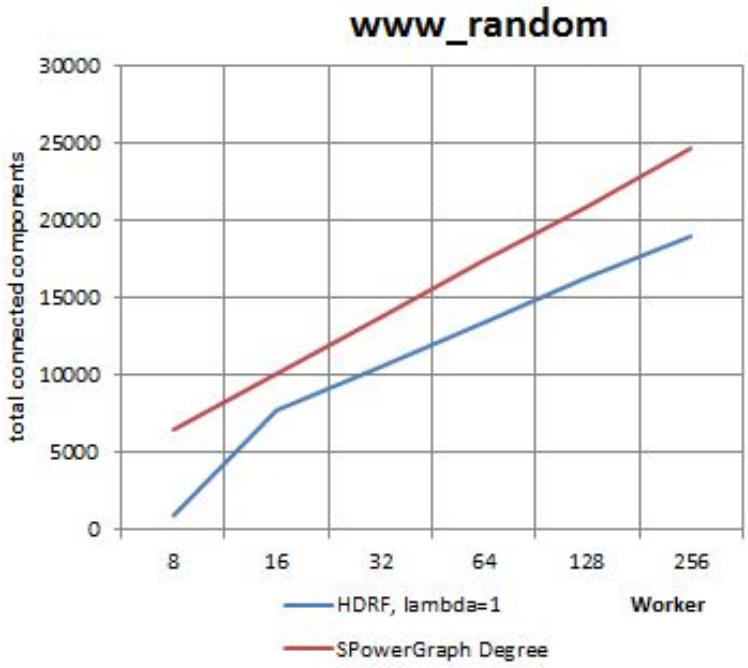

Figure 7.5: Total number of connected components of HDRF on input: www-random

Figure 7.5 shows the results of HDRF and another streaming algorithm, SPowerGraph Degree, on the "World-Wide-Web" graph. It is a web graph with lower average vertex degree than social graphs. The lower average vertex degree leads to stronger dissociation of the worker's subgraphs into more independent components. On "movielens100k" all subgraphs on the workers were just one connected component. Both results are not optimal for the multistep algorithm. If the input graph dissociates too much, the ILS will take extremely long time. If the input graph is too densely connected, the splitter algorithm influences the balance of the final partitioning through few very big components and degrees of freedom are lost. Using HDRF with $\lambda = 1$ the number of components could be reduced but still it is much higher than it was on the power-law graphs, Facebook and Twitter.

Figure 7.6 shows the result of SPowerGraph Degree in the splitter with Facebook as input. SPowerGraph Degree has a heuristic function that assigns edges based on the replica of the edge's vertices and a balance score. Once the partitioning gets imbalanced by more than 10% it assigns edges only based on a balance score, until the partitioning's balance is reestablished. Therefore SPowerGraph Degree can process any edge-order without adjusting parameters.

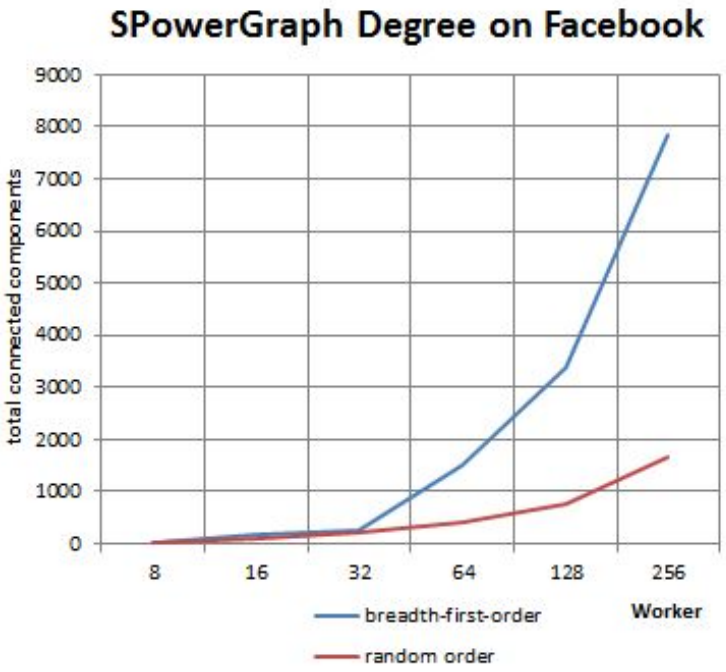

Figure 7.6: Total number of connected components of degree on input: Facebook

The results are the inverse of the results of both HDRF variants. SPowerGraph Degree's subgraphs on the workers dissociate in less independent connected components if the input graph is read in random-edge-order. If the edges arrive in breadth-first-order, the total number of connected components gets higher. This can be explained with the heuristic function it uses, since it takes most of its edge assignment decisions based on balance optimization for breadth-first edge orders, neglecting existing replications of an edge's vertices. Anyways, HDRF is more suitable for the task because SPowerGraph Degree's subgraphs on the partitions always dissociate more than the ones of HDRF.

7.4 Meta-graphs

The inputs's structure and average vertex degree influence the meta-graphs decisively. The higher the average vertex degree, the higher the probability that the meta-graph is fully connected. If, in addition to that, the edges in the meta-graph have very similar weights, the ILS becomes obsolete. All possible balanced cuts will have the same weight, a balanced random assignment of blobs to partitions achieves the same result in much less time.
Low average vertex degrees lead to sparse meta graphs with many light weight meta-vertices. This is also a suboptimal input for ILS. First of all the high number of meta-vertices leads to long execution time and the light weight of the meta-vertices allows for many balanced reassignments which adds also to the execution time. Moreover the improvements of ILS in comparison to a direct streaming of edges to final partitions is low. Since there are no dense clusters which would be expressed as meta-vertices with high weight, reassignments delete only few vertex replications.

A good meta-graph for ILS looks like shown in figure 7.7 and could be the result of HDRF on a social graph.

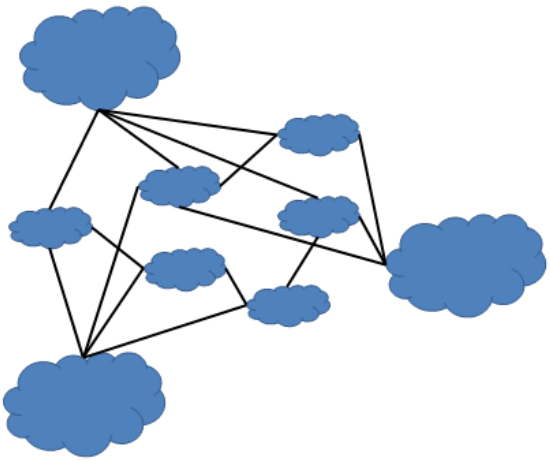

Figure 7.7: Good meta-graph for ILS

Every worker has got one big component and some small ones that are not connected to it. The start solution of the ILS assigns one big meta-vertex to each final partition and distributes the small meta-vertices to the final partitions based on good balance. In the ILS steps the small meta-vertices can be reassigned and many of their replications can be deleted. Furthermore the final cut will be located somewhere between the small meta-vertices which have lighter edges than bigger meta-vertices. This can be explained by the fact that bigger meta-vertices contain more edges and thus most of the times

also more vertices which increments the probability that there is an edge to another meta-vertex.

7.5 Improving Streaming Algorithms Using ILS

The partitioning of a streaming algorithm can be improved by applying ILS to the result. Only if the average vertex degree of the input graph is very high, like in the case of "movielens100k", it is not possible. Since every worker's subgraph is one connected component, no improvements can be achieved through reassignment of connected components. Any reassignment would lead to extreme imbalance. Social graphs are best suited because they tend to form one big cluster and several small clusters on each worker. Sparse graphs, like web graphs, can also be improved easily but ILS takes longer due to the higher number of independent components. Figures 7.8 and 7.9 show the improvements of the replication degree achieved by ILS on the Facebook graph and the Twitter graph. The ILS iterated 10 times and the number of workers was set to be equal to the number of final partitions.

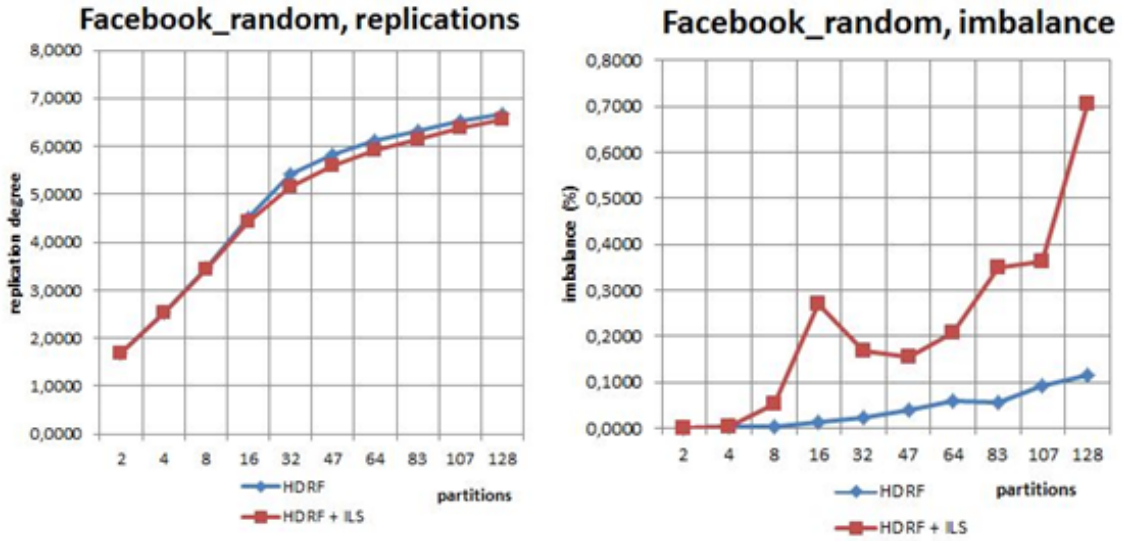

Figure 7.8: Replication degree improved through ILS on Facebook

In most cases the reassignments worsen the balance. If no balance constraint was implemented, all meta-vertices would end up on the same partition. For low numbers

of partitions no improvement can be achieved because there are not enough degrees of freedom. As soon as there are more than 16 partitions small improvements can be achieved. The improvements are small since the reassignments can only delete some vertex replications. The imbalance worsens but is still clearly below 1.0%. On the Twitter graph small improvements can already be achieved for lower numbers of partitions.

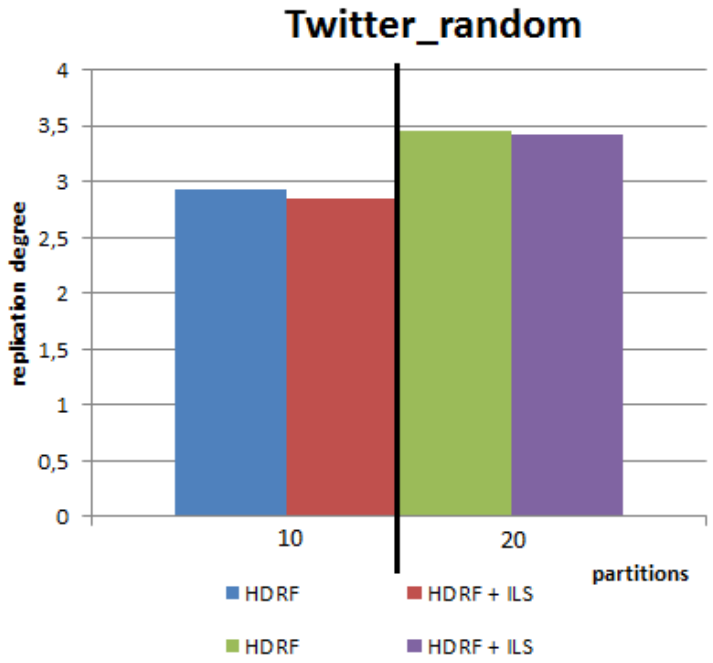

Figure 7.9: Replication degrees on Twitter, with and without application of ILS

If the acceptance criterion of the ILS focuses exclusively on the replication degree, the application of ILS will result in an improvement of the replication degree or make no changes at all. In order to gain additional degrees of freedom in the next evaluations the number of workers is set higher than the number of final partitions. This means that the ILS is not simply applied to the result of the streaming algorithm but uses many clusters to iteratively build up the final assignment.

7.6 Additional Degrees of Freedom

The evaluations in this section show the impact of higher numbers of workers. Higher numbers of workers lead to more independent components than there would be if only the final partitioning was to be improved by ILS. Therefore there are more degrees of freedom for the merger. However, this comes with the cost of additional vertex

replications. Only if the merger is able to eliminate those additional replications, the higher number of degrees of freedom can be used to improve the partitioning result. The Evaluations show that the results depend on the input graph's structure. On good inputs great improvements can be achieved in comparison to a linear time streaming algorithm. The number of workers must always be a multiple (>1) of the final number of partitions, else the partitioning result will be imbalanced.

7.6.1 Power-law Graphs

Power-law graphs are the best input for the system. As explained before they tend to form meta-graphs which are well fit for the ILS. This means, they have one big cluster on each worker and several small clusters. Figure 7.10 shows the replication degree depending on the number of workers for a partitioning in 8 partitions of the Facebook graph. Different algorithms for the splitter were used and the results are measured against the result of plain HDRF. All imbalances were clearly below 1%.

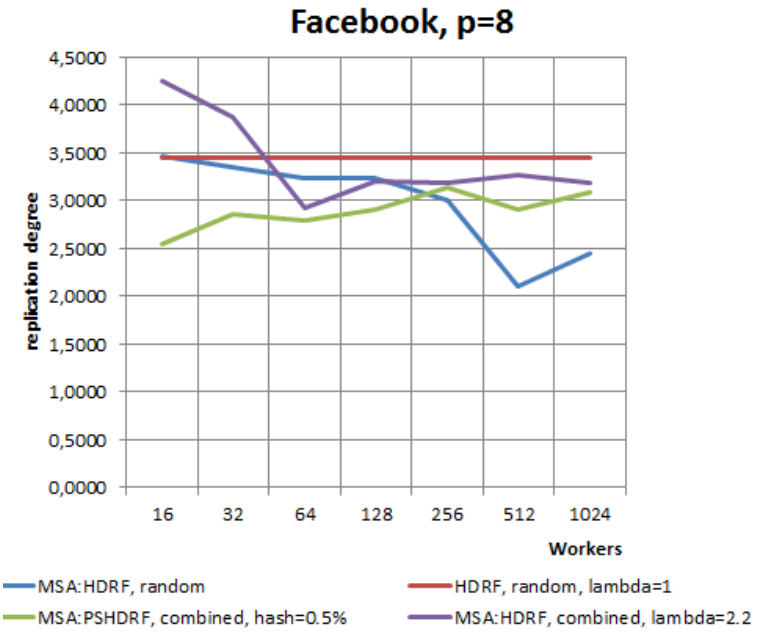

Figure 7.10: Replication degree depending on number of workers

For lower numbers of workers a breadth-first-order of the input yielded better results, for higher numbers of workers randomized edge orders led to better results. PSHDRF was always better as plain HDRF with $\lambda = 2.2$. This justifies the adaption of HDRF to PSHDRF (see chapter 4) for combined input. For 512 workers and randomized edge

input the result is much better than the one of plain HDRF. The replication degree could be improved from 3.45 to 2.09 and the imbalance was still below 1.0%. This result shows the power of the multistep algorithm with ILS on well fit input.

The results of a partitioning of the Twitter graph into 4 partitions are shown in figure 7.11. For higher numbers of partitions the results looked similar. Though the improvement of the replication degree is only small, the imbalance could also be lowered. Therefore the results are truly better than those of HDRF.

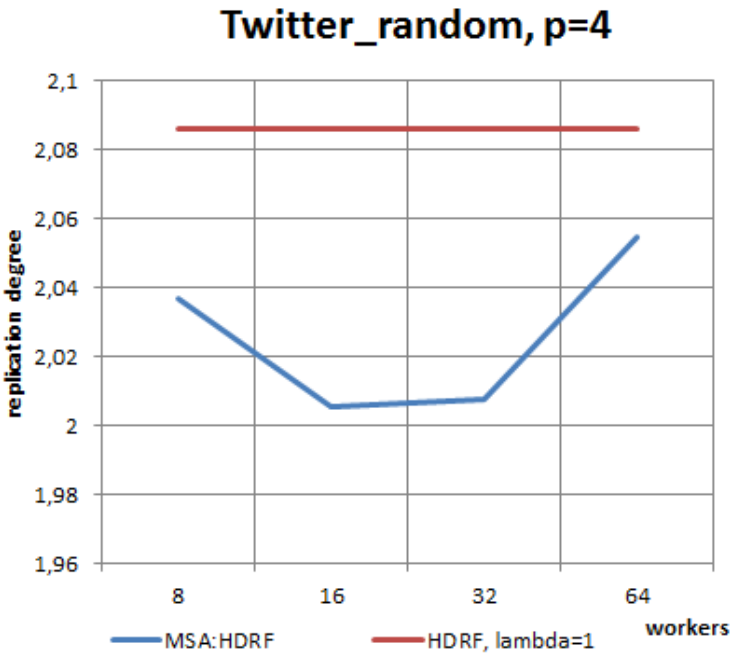

Figure 7.11: Replication degree depending on number of workers

On the Twitter graph the same effect can be observed: an intermediate number of workers yields the best results. If the number is too low, there are not enough degrees of freedom, if the number is too high, too many additional vertex replications are introduced. If the edges were read in breadth-first-order, no improvements could be achieved. The Twitter graph's order is different from the one of Facebook. It can be processed in any edge order with $\lambda = 1$, for the Facebook graph this is not possible. This might be why the improvements on the Facebook graph are higher. However, the partitionings of both graphs could be improved compared to HDRF.

7.6.2 Dense Graphs

Dense graphs turn out to be the worst input for the system. If the the ILS is applied to the final partitioning, no reassignments of components are possible since there is only one component on each partition. If the number of workers is higher than the number of final partitions, so that there are several independent components to assign to the final partitions, the problem is that too many additional vertex replications are created. The ILS is not able to reduce these additional replications so far that an overall improvement of the final result can be achieved. Figure 7.12 illustrates this for several numbers of workers.

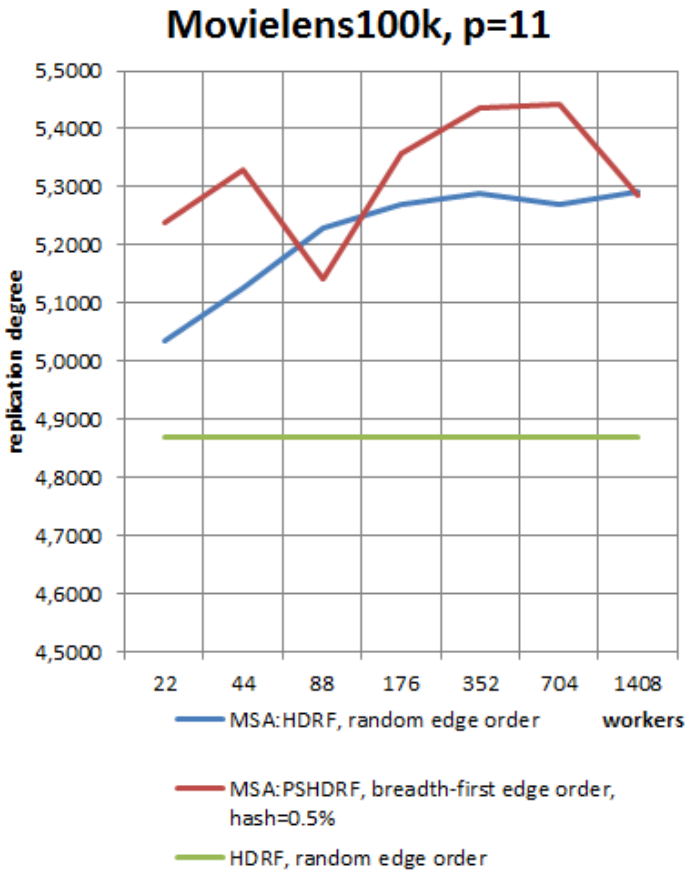

Figure 7.12: Replication degree depending on number of workers

For random edge input the replication degree grows steadily as the number of workers goes up. If the edges arrive in breadth-first-order, the replication degree is even higher than in the case of random input. The use of hashing with a certain probability leads to non-deterministic results which can be seen in the small oscillations of the results. For

higher numbers of workers both input variants create more vertex replications as the number of workers goes up. It is not possible to beat plain HDRF on the Movielens100k graph. The system's behavior shown in figure 7.12 is typical for dense input graphs.

7.6.3 Sparse Graphs

On sparse graphs slightly better partitioning results can be achieved but in general they are not well fit as input. On the World-Wide-Web graph plain HDRF achieves a replication degree of 1.294 for 4 partitions. Using 8 workers and ILS with 25 iterations, the replication degree could be lowered to 1.246, both imbalances were below 1%. For the Email-EuAll graph plain HDRF achieves a replication degree of below 1.03 for 4 partitions and SPowerGraph Degree a replication degree of 1.28 for 144 partitions. There is no point in inverting additional time to improve such results further. In addition the problem with sparse graphs is that they dissociate into many independent components. Therefore the ILS takes orders of magnitude more time compared to a streaming algorithm and the improvements are minimal.

7.6.4 Using Subpartitioning in Workers

The following evaluations investigate the effects of Karger's algorithm being used in the workers. If a component exceeds a certain threshold of edges, the worker splits it. As input a power-law graph, Facebook, and a dense graph, Movielens100k, were used. The ration of workers to final partitions was chosen as indicated in the diagrams headlines.

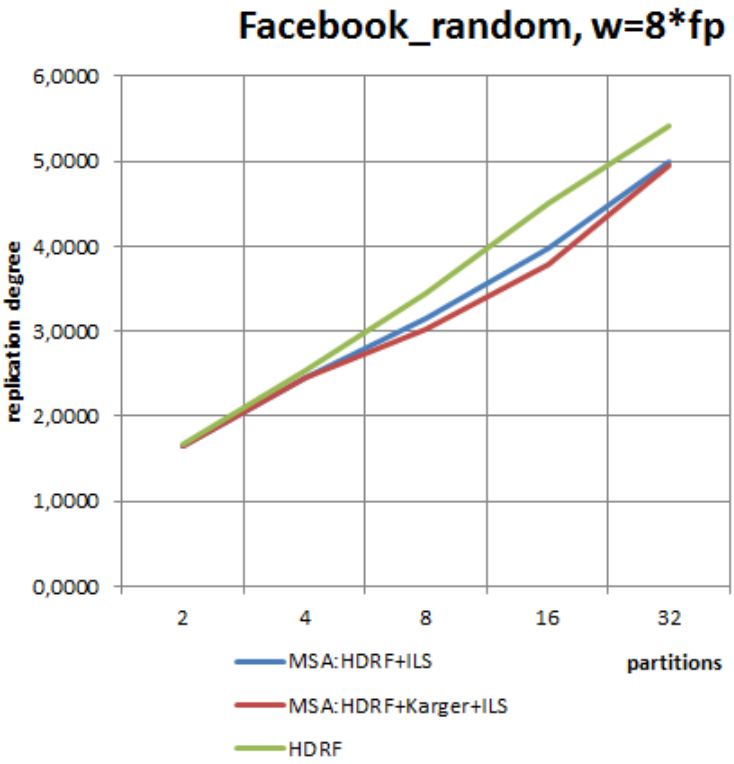

Figure 7.13: Replication degree with and without using Karger's algorithm in the workers on Facebook

Figure 7.13 shows the replication degree of the final partitioning depending on the number of partitions. As the number of final partitions is increased the multistep algorithm's result get better in comparison to HDRF's results. Using workers' subpartitioning, the results are improved even more.

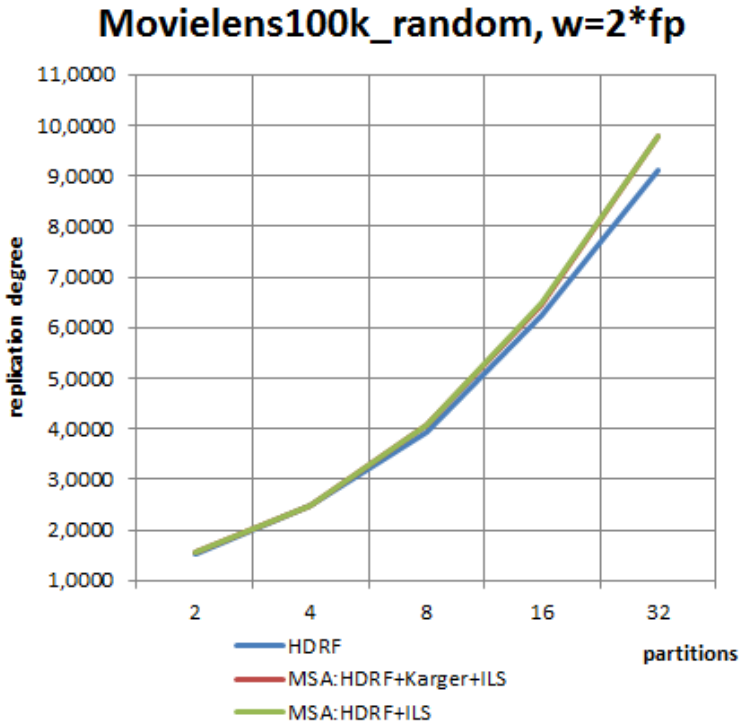

Figure 7.14: Replication degree with and without using Karger's algorithm in the workers on Movielens

Figure 7.14 shows the same evaluation on the Movielens graph. The results in this case are the inverse of those seen on Facebook. As the number of final partition increases the results of the multistep algorithm become worse compared to HDRF's results. Using Karger's algorithm or not makes almost no difference, sometimes however it improves the replication degree by around 1%.

The following evaluations show the effects of higher numbers of workers, again on Facebook and on Movielens100k. Tagging is not yet used. Using 1024 workers a real improvement could be achieved: a replication degree of 1.99 and an imbalance of below 0.3% for 8 partitions on Facebook. HDRF's partitioning for the same input has a replication degree of $3,45$ and an imbalance of 0.0039%.

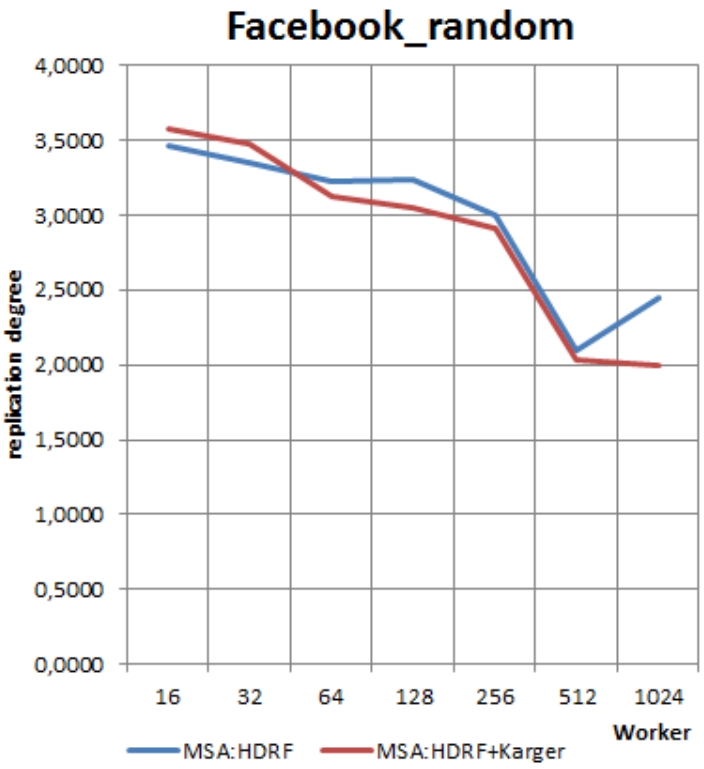

Figure 7.15: Replication degree with and without using Karger's algorithm in the workers on Facebook

Unfortunately the same results can not be observed for the Movielens graph. Here the results are the other way round. For low numbers of workers the results were better when Karger's algorithm was used. The more workers there were the worse the results became compared to the results achieved without Karger's algorithm.

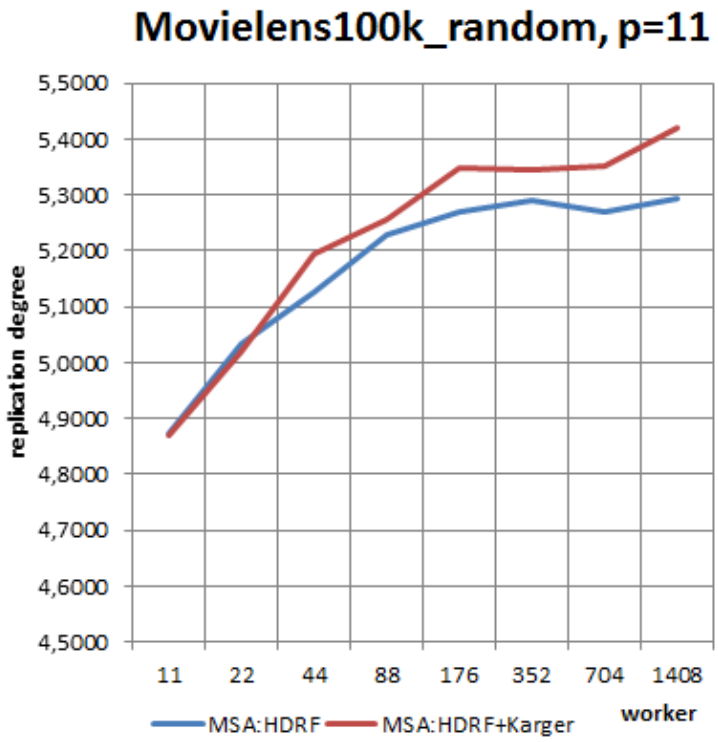

Figure 7.16: Replication degree with and without using Karger's algorithm in the workers on Movielens

From the evaluations in this sections it can be deduced that lower numbers of workers are better for dense graphs and higher numbers of workers are better for power-law graphs. For sparse graphs low numbers of workers are better. The ratio of workers to final partitions can not be fixed, it depends on the input.

7.7 ILS Initial Solution with Tagging

In the evaluations presented so far, it can be seen (e.g., figure 7.16) that subpartitioning in the workers can ultimately lead to a higher replication degree than without using it. If the number of workers is set to be higher than the number of final partitions it is not possible to guarantee a lower replication degree than a streaming algorithm would achieve by streaming edges directly to final partitions. When using worker subpartitioning however it is possible to guarantee at least the same replication degree as would be achieved without using worker subpartitioning. In chapter 5 it was analyzed how subpartitioning in the workers can introduce more vertex replications. Using

tagging of subclusters it is possible to guarantee that the replication degree does not worsen. Figure 7.17 demonstrates the effects of tagging.

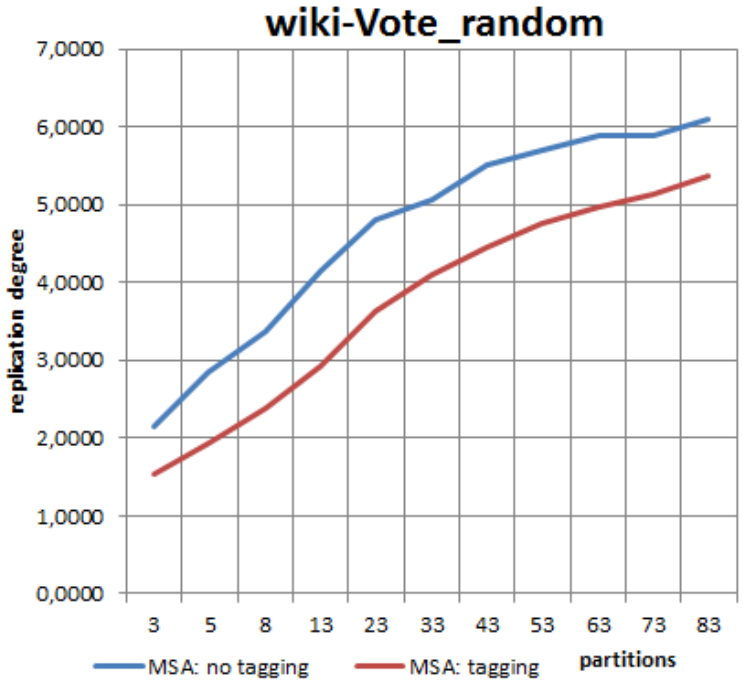

Figure 7.17: Replication degree with and without tagging of subclusters

If only HDRF is used, the results are very close to those of the run using tagging, most of the times being a bit worse. This result confirms the theory presented in chapter 6. Naive subpartitioning and a totally random initial solution for the ILS introduce additional vertex replications. In the presented instance the replication degree increased by around 0.8 when no tagging was used.

7.8 Comparison to Linear Time Streaming Algorithms

Based on the insights earned from the previous evaluations, this section aims to demonstrate the potential of the multistep algorithm with ILS. Therefore the results of it are shown in comparison to results of other graph partitioning algorithms. The settings for the evaluations were not always the same but were adapted to the number of final partitions. For instance, the hashing probability has to be higher if the number of workers is incremented. The best results were achieved on the Facebook graph where the multistep algorithm performed better than other state-of-the-art streaming partitioning algorithms.

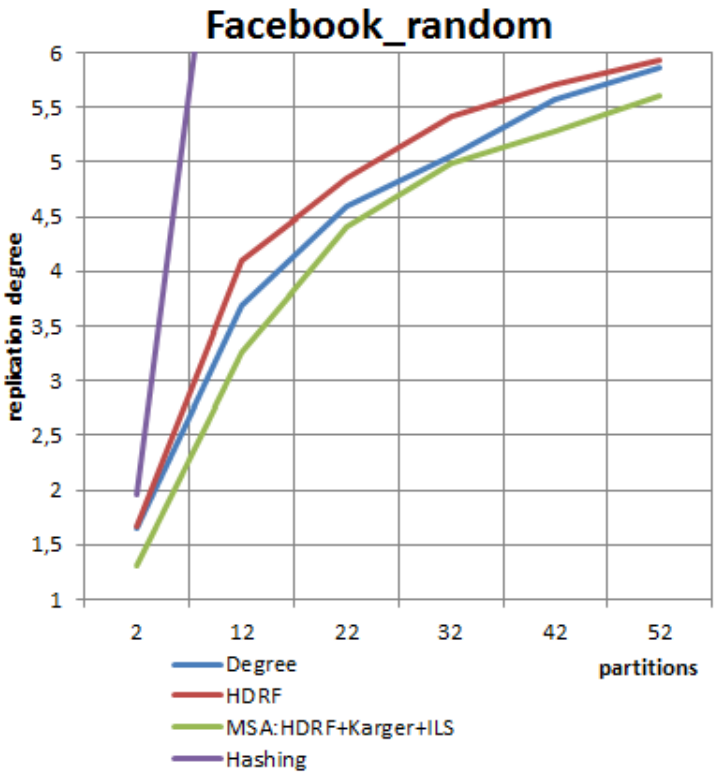

Figure 7.18: Results of different algorithms on Facebook

On the Twitter graph the multistep algorithm still produced better partitionings than the other algorithms but the improvement was rather small.

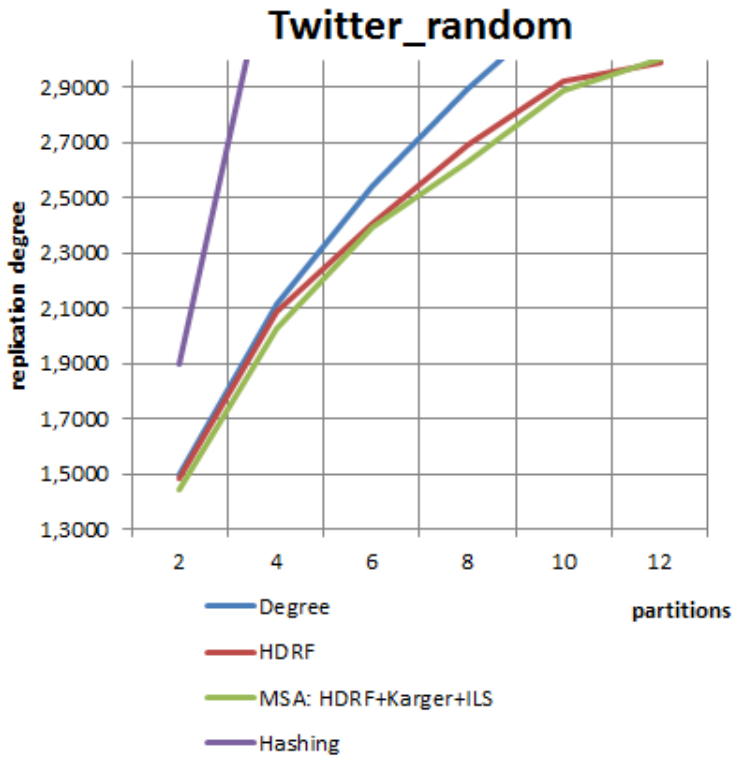

Figure 7.19: Results of different algorithms on Twitter

The Wiki-Vote graph, even though a social network and power-law graph, was best partitioned by the SPowerGraph Degree algorithm which had the best results for all numbers of partitions. Hashing led to extremely high replication degrees. HDRF and the multistep algorithm had similar results that are shown in figure 7.20. For lower numbers of partitions the multistep algorithm was better but for numbers bigger than 20 HDRF yields better results. Of course it would be possible to set the number of workers equal to the number of final partitions and use tagging, this would guarantee that the results are not worse than those of HDRF. But at the same time it would not be possible to improve the results decisively.

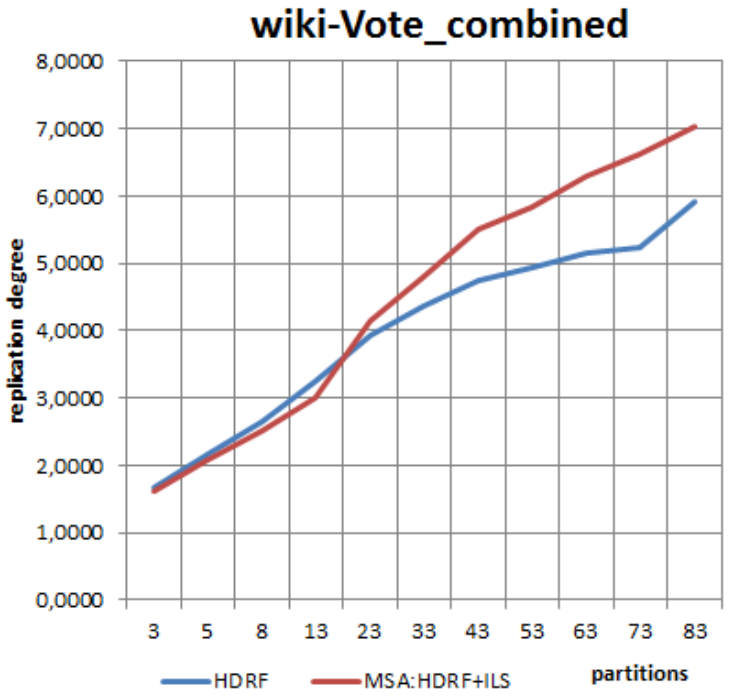

Figure 7.20: Results on Wiki-Vote

Especially on power-law graphs the multistep algorithm yields good and competitive results. In order to achieve real improvements of the partitioning quality it is necessary to set the number of workers higher than the number of final partitions. As shown this can also lead to worse results because of additional vertex replications, introduced because of a higher number of partitions in the splitter.

7.9 Comparison to an Offline Approach

The comparison to offline approaches is not easy. Most offline algorithms have to be purchased or are edge-cut algorithms. In order to provide some reference data the evaluated graphs were transformed to hypergraphs and processed by HMetis, an offline algorithm. Edges become vertices and vertices become hyperedges, therefore the edge-cut on the hypergraph leads to a vertex-cut. Several methods have been proposed to get an estimate of the replication degree based on the edge-cut weight. In the following evaluation the estimated replication degree of Metis is computed by $\frac{3 \times cutweight + |V|}{|V|}$. Since hyperedges are cut one can not say if they are cut in several pieces or just once in two halves. Therefore the factor three represents an average estimate of the cuts for each

hyperedge. Clearly, there can not be said a lot about the absolute values. However the growth of the replication degree can be interpreted.

7.9.1 Facebook

partitions	#meta-edges cut	estimated rep deg	multistep rep deg
2	108	1.08	1.16
4	455	1.34	1.56
6	1144	1.85	1.86
8	1485	2.10	1.99
10	1850	2.37	3,00
12	2262	2.68	3.72
14	2299	2.71	4.36
16	2437	2.81	4.43
18	2556	2.90	4.44
20	2604	2.93	4.44

Table 7.3: Metis results on Facebook

7.9.2 Movielens

partitions	#meta-edges cut	estimated rep deg	multistep rep deg
2	1153	3.06	1.54
4	1344	3.40	2.49
6	1390	3.48	3.3
8	1421	3.53	4.08
10	1426	3.54	4.75
12	1443	3.57	5.35
14	1451	3.59	5.94
16	1450	3.59	6.46
18	1470	3.62	6.94
20	1473	3.63	7.43

Table 7.4: Metis results on Movielens

HMetis leads to better results than the multistep algorithm, in addition, on the evaluated graphs it had also faster runtime.

7.9.3 Runtime

In general the runtime of the system is difficult to evaluate because it does not only depend on the input's size, but also on the input graph's structure. First, the input's size is composed of two factors, edges and vertices, both of them with different influences on the runtime. Higher numbers of edges lead to longer runtime in the splitter, more vertices influence the runtime of the worker's partitioning algorithm. Different input graph structures lead to different numbers of meta-vertices so that this influences directly the runtime of the ILS. Finally, the number of used workers and of final partitions also influences the runtime. Figure 7.21 visualizes the runtime behavior for different numbers of final partitions, a fixed ratio of workers to final partitions and a fixed input.

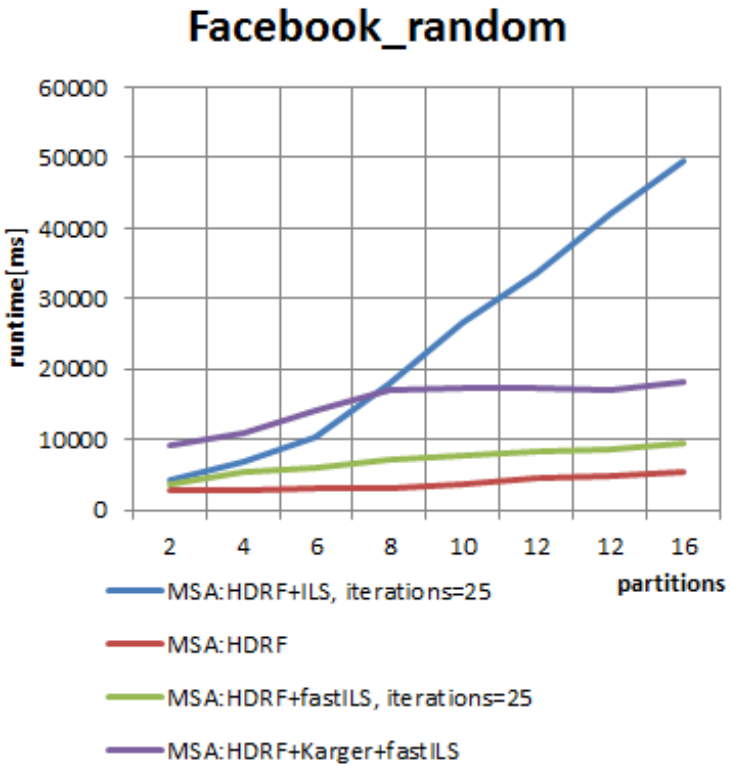

Figure 7.21: Runtime of the system with different settings

The lowest runtime was achieved when the workers only identified independent components and no ILS was performed, which is obvious. Using 25 iterations of the faster version of ILS (see chapter 6) a constant is added to the runtime. When the workers perform a Karger minimal-cut on all components bigger than 50 edges, the runtime goes up a little more. So far all additional work adds a constant amount of time. Once the standard implementation of ILS is used with 25 iterations and no worker algorithm,

the runtime goes up drastically. Even though for higher numbers of final partitions the runtime behavior looks linear, it is far too slow when the standard ILS is used. Linear time streaming algorithms have much lower runtime and perform almost as well on power-law and sparse graphs, on dense graphs they perform even better. Using fastILS the runtime can be reduced, but still, it is relatively slow. The results of fastILS are worse on most instances, in some rare cases even better than those of ILS. The influence of the number of meta-vertices processed by the ILS is shown in figure 7.22. The graph "movielens100k-random" was partitioned into two partitions, the workers only identified the independent components and the ILS used 25 iterations.

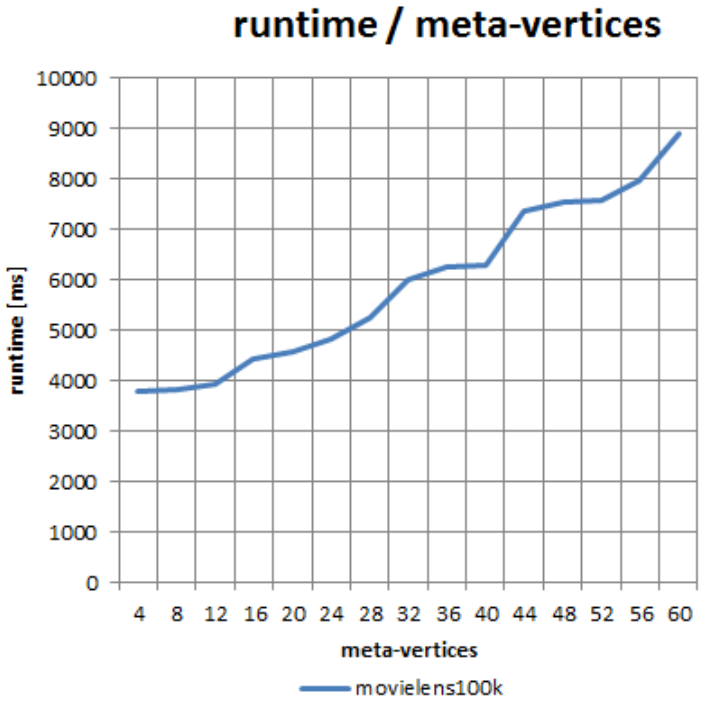

Figure 7.22: Runtime of the system depending on the number of meta-vertices

HDRF takes 2250 ms for the same partitioning but achieves better results as shown before.

7.10 Conclusion

The best results were achieved on power-law graphs. On average the results are better in comparison to real linear time algorithms. On the Facebook graph, great improvements were achieved. For further investigation and improvement of the multistep algorithm,

especially the structure of the Facebook graph should be understood in more depth in order to see why it yields such good results. Sparse graph lead to long runtime but still the multistep algorithm is able to compute good partitionings. The worst results were achieved on dense graphs.

8 Related Work

Graph partitioning has been researched in the last decades. Different approaches have been investigated which deal with two different ways of graph partitioning, vertex-cut and edge-cut. In the theoretical study of graph partitioning most of the research traditionally concentrated on edge-cut partitioning of planar graphs, meshs and VLSI graphs. Only recently research started to focus on power-law graphs and in this context also on vertex-cut partitioning. Natural data such as web graphs or social graphs are most of the times power-law graphs which makes it interesting to investigate vertex-cut graph partitioning in that context.

One of the first applications of graph partitioning was Google's Pregel [MAB+10] which uses edge-cut for graph partitioning. These partitionings suffer from large communication cost as each vertex communicates with its neighbors on other partitions. Since then other graph partitioning frameworks have been developed [AG08]. Edge-cut graph partitioning can be either performed offline or online. An example for offline graph partitioning is Metis[KK95] which uses a multilevel approach for graph partitioning. In order to split a graph, the input is coarsened by contracting vertices. Thereby a sequence of graphs is generated until a predefined number of vertices is reached. Then a partitioning is computed on the smallest graph. In the last step the partitioning is projected back through the sequence of graphs refining the partitioning at each level until the original graph is reestablished. Metis is a state-of-the-art algorithm that is often used as a benchmark to be compared to other algorithms. Despite its good results it has a disadvantage. Being an offline algorithm it needs to keep the whole input in memory which is not possible for big data. Furthermore it is not fast enough once the input becomes very big.
A new approach, called streaming graph partitioning, was proposed by Stanton and Kliot [SK12]. Streaming algorithms address the problem of memory capacity and speed. Instead of reading the whole graph and partitioning it afterwards, they receive the input as a stream of vertices. Each vertex is assigned to a partition where it fits best. Using this method a graph is partitioned on the fly while it is read. Therefore it does not have to be kept in memory. Furthermore simple objective functions can be used to compute the best partition for an incoming vertex. FENNEL [TGRV12] is an example of a streaming algorithm for edge-cut.

As mentioned before, edge-cut is not optimal for power-law graphs. Since most natural graphs are power-law graphs it is more interesting to focus on vertex-cut approaches in the context of graph processing for natural data. A first simple step would be to transform edges into vertices and vertices into edges. This creates a hypergraph which can be partitioned by edge-cut again. Cutting through hyperedges translates into a vertex-cut of the original graph. HMetis is an example of a hypergraph partitioning algorithm. Clearly the graph transformation costs additional time making such approaches slow. Therefore, starting in 2010, pure vertex-cut approaches were proposed such as PowerGraph [LBG+12]. At first these approaches were still offline algorithms. Recent research started to focus on streaming vertex-cut algorithms. These algorithms receive the input as a stream of edges. Each edge gets assigned to a partition using some heuristic function. The heuristic aims to assign the edges so that new vertex replications are avoided and no partition has many more edges than the others. One of the strongest state-of-the-art vertex-cut streaming algorithms is HDRF [PQD+15]. It is designed especially for power-law graphs and makes use of their characteristics.

The divide-and-conquer paradigm was used before to solve other graph problems, for example the common connected problem [XHP04] or a divide-and-conquer framework for finding clusters in a graph [YX15]. Combining local search with simulated annealing was investigated in [MO96] before. In [HL97] local search was also applied to the graph partitioning problem though focusing on low-degree graphs.

9 Discussion

By implementing the theoretically defined framework, the runtime behavior could be measured and partitioning results evaluated. The total amount of time needed to partition a given graph resulted to be highly dependent on the input's structure. However the data and theoretical results suggest that through all the defined steps linear time can be approximated (see figures 7.21 and 7.22). In addition to that it was possible to improve the quality of the partitioning of some problem instances in comparison to other linear time algorithms. Thus the basic idea is confirmed by this work. However the ILS is still too slow and the current implementation of the divide-and-conquer framework is not competitive in terms of runtime.

Most importantly a deeper understanding of the whole framework, especially the multi-step algorithm, could be gained. Streaming graph partitioning algorithms do not create fully connected subgraphs on the partitions but rather a set of independent components, except for very dense graphs. If the subgraphs contained on the partitions are to be partitioned further in this setting, those independent components have to be identified first so that a minimal cut or clustering algorithm can be applied. Nevertheless this characteristics also make it possible to improve most results of streaming algorithms using iterated local search. If small independent components are exchanged between partitions, it is possible to delete vertex replications without producing additional imbalance.

The idea to use breadth-first-ordered edge streams as input to create denser clusters on the partitions through hashing combined with HDRF (PSHDRF) worked only for small numbers of partitions. The advantage of PSHDRF with $\lambda = 1$ is that, as long as there are less than approximately 10 partitions, a very low percentage of hashing can be used. If HDRF is used with $\lambda > 1.0$, every edge assignment is influenced by the higher weight for balance. On the contrary, PSHDRF hashes a very low percentage of edges so that no partition is left empty. All other edge assignments can be done by HDRF as usually with an equilibrated weight for vertex replications and balance which enables better clustering of edges for breadth-first-ordered edge streams. Neither hashing nor an increased $\lambda > 1.0$ scale automatically so that they have to be adjusted according to the number of partitions. If the k-way partitioning is to be performed for k > 10 the hash percentage has to be set higher. Ultimately too many edges will be hashed and the result is worse than if λ was increased. Currently the hashing percentage or λ has to be

set in advance. To be fully independent of the order of arriving edges, the parameters should be adapted automatically. An algorithm exclusively for the splitter should be designed as sketched in chapter 4 since connected components on the partitions are more important than perfect balance. Moreover this algorithm should also focus on making the system more independent from the structure of the input graph, maybe by computing the number of workers dynamically.

At the point of time when the workers perform their subpartitioning the whole input graph was read thus it would be possible to use a global view of the graph in the workers instead of only using the subgraph view. This theory of better results through more relevant information about the problem instance can be formulated as a theorem:
Given an instance P_1 of a problem P and an algorithm A that solves P. Furthermore two sets I_1, I_2 containing relevant information about the structure of P_1 and a function $q(s)$ which attributes a quality value to a solution. A solution of P_x can be computed by $A(I_x) = s$. Then follows:

$$I_2 \subset I_1 \Rightarrow q(A(I_2)) \leq q(A(I_1))$$

Using a global view would therefore make it possible to improve the workers performance and thus the whole algorithms performance. This is also supported by the results shown in section 7.9 of chapter 7.

ILS is a powerful tool but it needs enough degrees of freedom to perform up to its potential. More degrees of freedom in the setting of graph partitioning means more independent components on the workers. However more independent components also lead to more vertex replications. This is what makes the system so dependent on the structure of the input. If the input is dense and does not automatically dissociate into enough independent components, the ratio of workers to final partitions has to be increased or the workers have to perform more cuts with their algorithm. In any case partitioning a graph into more subgraphs always introduces additional vertex replications. Not partitioning the workers' subgraphs further is not possible either because the ILS would not have enough degrees of freedom. Normally the ILS eliminates additional vertex replications and gains performance by deleting even more than there were initially.
Another comment should be made about the initial solution of the ILS: For good performance the initial solution must be balanced or at least not imbalanced due to one big component.
Furthermore it is difficult to think of a good definition of a local optimum in graph partitioning for local search. Currently a local optimum is defined as a situation where no more components can be reassigned to another partition without introducing new vertex

replications. Though this might seem a logical definition of a local optimum it makes the local search too slow. The acceptance criterion has a similar problem, currently it is a combination of low replication degree reward and imbalance punishment. However, to compute the current replication degree all vertices on all partitions have to be counted and summed up. This seems also too slow and dependent on the number of vertices in the input graph.

Finally it has to be mentioned that the system suffers from a contradiction. Increasing the number of workers (threads or machines) increases the runtime because more independent components will be created which lead to a longer runtime of the ILS.
With the presented fastILS it was shown that the concept of ILS can be made clearly faster and more scalable by relaxing the definitions of ILS without loosing much solution quality. In the setting of distributed graph processing it would also be interesting to think of multithreaded ILS.
Furthermore this work shows that there is potential to improve the results of existing algorithms and think, out-of-the-box, of completely new approaches.

10 Conclusion

First applications of graph partitioning were described and the general graph partitioning problem was introduced. Different types of graph partitioning algorithms and cut variants were presented. In chapter 2 the problem was formulated mathematically for edge-cut and vertex-cut. In addition it was extended to graphs with edge weights and vertex weights.
Starting from chapter 3 a new graph partitioning approach based on the divide-and-conquer paradigm was presented. After a general overview of the system, every single step (splitter, worker, merger) was described. Therefore for each step a task was formulated. Then a method and implementation to solve the task was shown and analyzed. Finally the presented implementation of the framework was evaluated on several input graphs.
For power-law graphs and sparse graphs competitive results were achieved whereas on dense graphs the approach does not work well. Even though linear time could be approximated, the ILS in its strict implementation is too slow for large-scale graph analysis. In its current set up the framework can be used for graph partitioning of up to middle sized power-law graphs. For the application to large-scale graphs further investigation and refinement is necessary.

The divide-and-conquer based framework was implemented with different algorithms for each step. A schematic overview can be seen in figure 10.1.

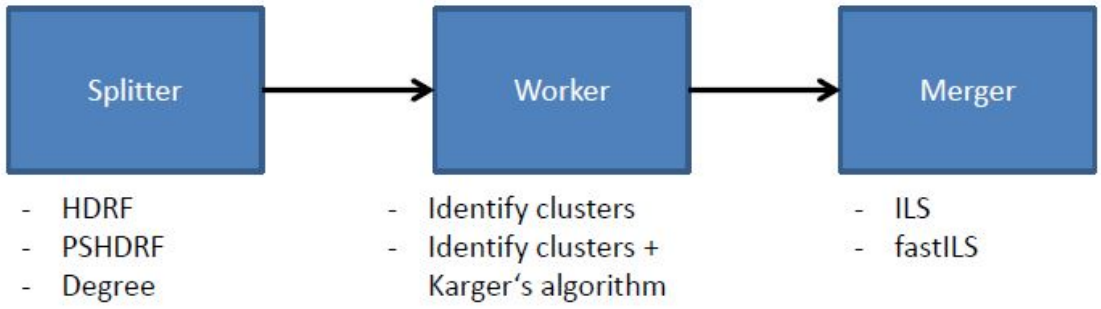

Figure 10.1: Possible settings for the multistep algorithm

Recapitulating the findings of the evaluation section, figure 10.2 shows which settings are best on average for certain inputs. There can still be instances of graphs that do not fit the scheme.

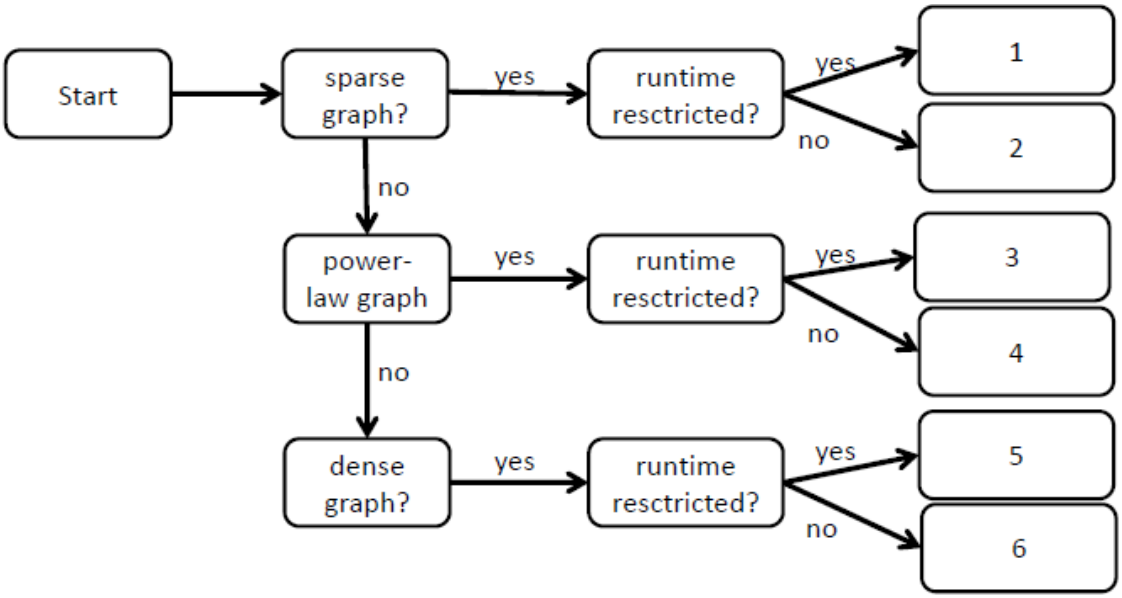

Figure 10.2: Decision tree for settings

For all inputs HDRF with $\lambda = 1.0$ should be used for random edge orders. If the edges arrive in breadth-first-order and there are less than 10 workers, PSHDRF should be used with a very low hash probability. If there are 10 partitions or more, HDRF with $\lambda > 1.0$ should be used.

1. Since sparse graphs tend to dissociate into many independent components the number of workers should be set equal to the number of final partitions. This reduces the number of meta-vertices for the ILS (see figure **??**). The workers should not use any subpartitioning algorithm and identify only the connected components. Most of the times it is not possible to achieve better results than HDRF on sparse graphs, therefore fastILS is good enough to find possible, small improvements without taking much time. The iterations can be set to 25 or lower depending on the available runtime.

2. If the runtime is not important, the same settings as in 1 can be used but the classic implementation of ILS can be used instead of fastILS.

3. For power-law graphs a ratio of 2 to 128 between workers and final partitions should be used (see figure 7.13). Since there are enough degrees of freedom it is not necessary to use subpartitioning algorithms in the workers. Depending on

the graph's structure different amounts of meta-vertices will ocure. If the number of merta-vertices is lower than 100 up to 75 iterations can be performed by ILS. More than 75 iterations will not lead to improvements in general.

4. Similarly to 3 a ratio of 2 to 128 between workers and final partitions should be used. In addition a subpartitioning algorithm can be used in the workers and the iterations for the ILS can be kept at 75 even for more than 100 meta-vertices.
5. For dense graphs it is important to have a ratio of 1 between workers and final partitions (see figure 7.12). If there are more workers, the additional vertex-replications will lead to a bad partitioning result. However it is important to use a subpartitioning algorithm in the workers in order to have enough degrees of freedom. Normally dense graphs produce just one connected component on each worker and thus only few meta-vertices for the ILS. Therefore the iterations for the ILS can be set as low as 10.
6. If more runtime is available, the iterations of the ILS can be set a bit higher (around 25) than in 5.

abbreviation	definition
s.t.	subject to
ILS	iterated local search
MSA	multistep algorithm, implementation of the divide-and-conquer framework

Bibliography

[AG08] R. Angles, C. Gutierrez. “Survey of Graph Database Models.” In: *ACM Comput. Surv.* 40.1 (Feb. 2008), 1:1–1:39. ISSN: 0360-0300. DOI: 10.1145/1322432.1322433. URL: http://doi.acm.org/10.1145/1322432.1322433 (cit. on p. 81).

[AHB00] R. Albert, H.Jeong, A.-L. Barabasi. “Error and attack tolerance of complex networks.” In: *Nature* (2000) (cit. on p. 16).

[Aro10] S. Arora. *Karger’s Min-Cut Algorithm*. 2010. URL: https://www.cs.princeton.edu/courses/archive/fall13/cos521/lecnotes/lec2final.pdf (visited on 10/15/2016) (cit. on p. 42).

[CEK+15] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, S. Muthukrishnan. “One Trillion Edges: Graph Processing at Facebook-scale.” In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 1804–1815. ISSN: 2150-8097. DOI: 10.14778/2824032.2824077. URL: http://dx.doi.org/10.14778/2824032.2824077 (cit. on p. 15).

[Cia15] C. Ciacca. *Facebook by the Numbers - Here’s How Big It Really Is*. 2015. URL: https://www.thestreet.com/story/13352739/1/facebook-by-the-numbers-here-s-how-big-it-really-is.html (visited on 10/13/2016) (cit. on p. 15).

[EDD+12] C. Eaton, D. Deroos, T. Deutsch, G. Lapis, P. Zikopoulos. *Understanding Big Data*. Mc Graw Hill, 2012 (cit. on p. 15).

[FFF99] M. Faloutsos, P. Faloutsos, C. Faloutsos. “On Power-law Relationships of the Internet Topology.” In: *SIGCOMM Comput. Commun. Rev.* 29.4 (Aug. 1999), pp. 251–262. ISSN: 0146-4833. DOI: 10.1145/316194.316229. URL: http://doi.acm.org/10.1145/316194.316229 (cit. on pp. 15, 22).

[GLG+12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin. “PowerGraph: Distributed Graph-parallel Computation on Natural Graphs.” In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI’12. Hollywood, CA, USA: USENIX Association, 2012, pp. 17–30. ISBN: 978-1-931971-96-6. URL: http://dl.acm.org/citation.cfm?id=2387880.2387883 (cit. on pp. 23, 32).

[HL97] M. M. Halldorsson, H. C. Lau. “Low-degree Graph Partitioning via Local Search with Applications to Constraint Satisfaction, Max Cut, and Coloring.” In: *Journal of Graph Algorithms and Applications* (1997) (cit. on p. 82).

[IBM16] IBM. *Bringing big data to the enterprise*. 2016. URL: https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html (visited on 10/13/2016) (cit. on p. 15).

[KHKM11] J. Kim, I. Hwang, Y.-H. Kim, B.-R. Moon. “Genetic Approaches for Graph Partitioning: A Survey.” In: *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*. GECCO ’11. Dublin, Ireland: ACM, 2011, pp. 473–480. ISBN: 978-1-4503-0557-0. DOI: 10.1145/2001576.2001642. URL: http://doi.acm.org/10.1145/2001576.2001642 (cit. on p. 16).

[KK95] G. Karypis, V. Kumar. “Multilevel graph partitioning schemes.” In: *ICPP (3)*. 1995, pp. 113–122 (cit. on pp. 22, 81).

[KLPM10] H. Kwak, C. Lee, H. Park, S. Moon. “What is Twitter, a Social Network or a News Media?” In: *Proceedings of the 19th International Conference on World Wide Web*. WWW ’10. Raleigh, North Carolina, USA: ACM, 2010, pp. 591–600. ISBN: 978-1-60558-799-8. URL: http://doi.acm.org/10.1145/1772690.1772751 (cit. on p. 22).

[LBG+12] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J. M. Hellerstein. “Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud.” In: *Proc. VLDB Endow.* 5.8 (Apr. 2012), pp. 716–727. ISSN: 2150-8097. DOI: 10.14778/2212351.2212354. URL: http://dx.doi.org/10.14778/2212351.2212354 (cit. on p. 82).

[MAB+10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski. “Pregel: A System for Large-scale Graph Processing.” In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’10. Indianapolis, Indiana, USA: ACM, 2010, pp. 135–146. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807184. URL: http://doi.acm.org/10.1145/1807167.1807184 (cit. on pp. 16, 81).

[MMTR16] R. Mayer, C. Mayer, M. A. Tariq, K. Rothermel. “GraphCEP: Real-time Data Analytics Using Parallel Complex Event and Graph Processing.” In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. DEBS ’16. Irvine, California: ACM, 2016, pp. 309–316. ISBN: 978-1-4503-4021-2. DOI: 10.1145/2933267.2933509. URL: http://doi.acm.org/10.1145/2933267.2933509 (cit. on p. 16).

[MO96] O. C. Martin, S. W. Otto. “Combining simulated annealing with local search heuristics.” In: *Annals of Operations Research* 63.1 (1996), pp. 57–75 (cit. on p. 82).

[MTLR16] C. Mayer, M. A. Tariq, C. Li, K. Rothermel. “GrapH: Heterogeneity-Aware Graph Computation with Adaptive Partitioning.” In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. June 2016, pp. 118–128. DOI: 10.1109/ICDCS.2016.92 (cit. on p. 16).

[PQD+15] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, G. Iacoboni. “HDRF: Stream-Based Partitioning for Power-Law Graphs.” In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. CIKM ’15. Melbourne, Australia: ACM, 2015, pp. 243–252. ISBN: 978-1-4503-3794-6. DOI: 10.1145/2806416.2806424. URL: http://doi.acm.org/10.1145/2806416.2806424 (cit. on pp. 16, 32, 33, 82).

[RPG+13] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, S. Haridi. “Ja-be-ja: A distributed algorithm for balanced graph partitioning.” In: *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*. IEEE. 2013 (cit. on p. 16).

[SK12] I. Stanton, G. Kliot. “Streaming Graph Partitioning for Large Distributed Graphs.” In: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’12. Beijing, China: ACM, 2012, pp. 1222–1230. ISBN: 978-1-4503-1462-6. URL: http://doi.acm.org/10.1145/2339530.2339722 (cit. on pp. 22, 81).

[Stu03] T. Stuetzle. *Iterated Local Search and Variable Neighborhood Search*. 2003. URL: http://www.sls-book.net/Slides/sls-ils+vns.pdf (visited on 10/17/2016) (cit. on p. 49).

[Stü06] T. Stützle. “Iterated local search for the quadratic assignment problem.” In: *European Journal of Operational Research* 174.3 (2006), pp. 1519–1539 (cit. on p. 49).

[SW97] M. Stoer, F. Wagner. “A simple min-cut algorithm.” In: *Journal of the ACM (JACM)* 44.4 (1997), pp. 585–591 (cit. on p. 42).

[SZCH15] R. Sun, L. Zhang, Z. Chen, Z. Hao. “A Balanced Vertex Cut Partition Method in Distributed Graph Computing.” In: *Revised Selected Papers, Part II, of the 5th International Conference on Intelligence Science and Big Data Engineering. Big Data and Machine Learning Techniques - Volume 9243*. IScIDE 2015. Suzhou, China: Springer-Verlag New York, Inc., 2015, pp. 43–54. ISBN: 978-3-319-23861-6. DOI: 10.1007/978-3-319-23862-3_5. URL: http://dx.doi.org/10.1007/978-3-319-23862-3_5 (cit. on p. 16).

[TGRV12] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, M. Vojnovic. *Fennel: Streaming Graph Partitioning for Massive Scale Graphs*. Tech. rep. Nov. 2012. URL: https://www.microsoft.com/en-us/research/publication/fennel-streaming-graph-partitioning-for-massive-scale-graphs/ (cit. on pp. 16, 81).

[XHP04] B. Xuan, M. Habib, C. Paul. "Divide and Conquer Revisited. Application to Graph Algorithms." In: (2004) (cit. on p. 82).

[XLZ15] C. Xie, W. Li, Z. Zhang. "S-PowerGraph: Streaming Graph Partitioning for Natural Graphs by Vertex-Cut." In: *CoRR* abs/1511.02586 (2015). URL: http://arxiv.org/abs/1511.02586 (cit. on pp. 17, 22).

[YX15] W. Yang, H. Xu. "A Divide and Conquer Framework for Distributed Graph Clustering." In: *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 2015, pp. 504–513 (cit. on p. 82).

All links were last followed on November 10, 2016.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature