Karlsruhe Institute of Technology

Engineering Distributed Graph Clustering using MapReduce

Master Thesis of

Tim Zeitz

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Henning Meyerhenke
Advisors:	Michael Hamann, M.Sc.
	Dipl.-Inform. Ben Strasser

Time Period: 1st December 2016 – 31st May 2017

Acknowledgments

First and foremost I want to thank my advisors Michael Haman and Ben Strasser. Through their feedback I learned a lot of lessons about writing efficient C++ and scientific texts. Both this work and the implementation benefited greatly from their support.

I also want to thank Timo Bingmann for his support on all question and issues I had with Thrill.

Special thanks Johanna Splieth for her countless language corrections and improvements.

Finally, I want to thank Prof. Wagner for the opportunity to write this thesis at her department.

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 31st May 2017

Abstract

Detecting community structures in networks is an important problem in graph analytics. With the recent BigData trends network sizes are growing tremendously. Oftentimes the networks are now too big for the RAM of a single computer. This leads to the need for distributed graph clustering algorithms. We present two MapReduce based algorithms which build upon the well-known Louvain algorithm. We implement them in Thrill, a new experimental BigData batch processing framework. Our evaluation shows that our algorithms can cluster graphs with tens of billions of edges in a few hours while still delivering quality similar to the original Louvain algorithm.

Zusammenfassung

Die Identifizierung von Communitystrukturen in (sozialen) Netzwerken ist ein wichtiges Problem in der Analyse von Graphen. Bedingt durch Trends wie BigData, sind die zu analysierenden Netzwerke in den vergangenen Jahren immer weiter gewachsen. Die Netzwerke sind mittlerweile oft zu groß für den RAM eines einzelnen Computers. Somit werden verteilte Clustering Algorithmen zu einer Notwendigkeit. In dieser Arbeit entwickeln wir auf Basis des bekannten Louvain-Algorithmus zwei verteilte Clustering Algorithmen für das MapReduce Paradigma. Wir implementieren die Algorithmen in Thrill, einem neuen experimentellen BigData Framework. Unsere Evaluation zeigt, dass unsere Algorithmen Graphen mit mehreren Milliarden Kanten in wenigen Stunden mit einer ähnlichen Qualität wie der ursprüngliche Louvain-Algorithmus clustern können.

Contents

1. Introduction

Networks are an almost omnipresent phenomenon in our world. With the popularity of modern communication networks like the Internet – and even more so with the advent of the so-called "Internet of Things" – this may seem like an obvious statement. But actually networks are a much broader phenomenon than the Internet and have much broader applications than the technological advances of the past few decades. Today the term social network refers to web services like Twitter or Facebook, but actually in a more literal sense social networks have been studied by sociologists since the 1930s. Citations in scientific publications form a network of references as do links between websites. A biological example would be the brain which is made up of neurons and connections between them – which leads back to neural networks in artificial intelligence. The list goes on. The study of networks goes back as far as 1736 when Euler solved the puzzle of the Königsberg bridges [Eul41] and its applications range among others through biology, physics, economics, engineering, ecology, marketing, social and political sciences and of course computer science.

While it is possible to study these networks in their respective context, it is also of great interest to analyze them purely on the basis of their structural properties. To this end networks are often represented as a *graph* – an abstract model of networks as a set of entities called *nodes* and the connections between them denoted as *edges* or *links* – and often some additional data like the strength of the links or some weight attached to the nodes. From these more abstract representations the goal is to extract structural information about the original network.

One key problem in this is often to identify so-called *communities* or *clusters* – groups of nodes which are strongly connected among each other and much less so to the rest of the graph. This is of course no precise definition and as it has turned out, finding a good general formal definition of what a cluster is, is rather difficult. The "correct" definition is also dependent on the kind of data one is analyzing. For example clusters in social network graphs may often overlap since most people have connections into several communities (e.g. students have connections to their fellow students but also to their dorm mates) while in web graphs pages from one domain often form non overlapping clusters – though on a higher level several domains might also form a cluster. These are only two of many characteristics clusters in certain contexts may or may not exhibit. Thus, in literature there is not one commonly accepted clustering definition but many competing with different strengths and weaknesses.

While the study of networks has been an important problem for more than a century, what has changed over the past decades is the size of the networks in question – which steadily increases. The recent trend of "Big Data" is an additional amplifier to that tendency. The growing size of the graph instances leads to the necessity of algorithms which are able to handle the enormous amounts of data within reasonable time and memory limits. While there have been efficient sequential algorithms to find clusterings for quite some time [BGLL08], the growth in graph size makes available main memory much more of an issue. For example Facebook – currently the worlds largest social network – has approximately 1.86 Billion users[1] at the time of writing. In 2011, [UKBM11] found that every user had an average of 190 "friends". If that number had stayed the same since then (which is unclear) that would mean that the network has now about 350 Billion connections. A simple edge-list representation of this graph would take roughly 1400 GB of memory[2] – well beyond the capabilities of commodity hardware. This leads to the need for distributed clustering algorithms[3] – which are the subject of this work.

Networks are of course not the only datasets growing tremendously. In the past years a lot of engineering effort has gone into developing techniques and frameworks to be able to cope with such amounts of data. One particular successful approach has been the *MapReduce* paradigm which was originally introduced by Google [DG08]. In MapReduce, algorithms are expressed as a set of transformations on list elements distributed among several computation nodes. For algorithms in MapReduce, the transformations must be executable independently on each element of the list. This then allows almost trivial parallelization and distribution over several workers of a compute network. The programmer does not have to worry about parallelization, synchronization or network communication of his program but only about the actual algorithm. That said, MapReduce is very simple to use and still easily scales almost arbitrarily if one can express his algorithm as a set of transformations on elements of a list where each element can be transformed independently. For graphs this is not a trivial assumption since their structural key component are in fact connections.

In this work, we are going to implement and evaluate two graph clustering algorithms in MapReduce to determine whether or not MapReduce can be a feasible approach to create scalable graph algorithms scalable for the massive graphs available today.

Related Work

Community detection has been a very important topic in the recent study of networks. An in-depth comparative overview over the different approaches and algorithms available can be found in [FH16, LF09, For10]. One method that gained particular popularity is the optimization of *modularity* – a graph clustering quality measurement introduced in 2004 by Newman and Girvan [NG04]. Even though finding a clustering with maximum modularity was proven to be $\mathcal{NP}$-complete [BDG+08], several algorithms successfully employed greedy modularity optimization heuristics – most notably the so-called Louvain algorithm [BGLL08] which is still widely used today and also foundational to this thesis. The Louvain algorithm has two phases. In the first one (called *local moving*) nodes are moved around between clusters so each node ends up in the cluster which yields the best modularity. This can be done very efficiently since the change of modularity for moving a node from one cluster to another can be computed in constant time. After a local maximum is reached a so-called meta graph is built. All nodes in each cluster are merged together to

[1] `http://newsroom.fb.com/company-info/`

[2] 1.86 Billion users ×190 Friends ×8 B per edge (two 32 bit node ids) /2 because we store each edge only in one direction. It is of course possible to compress such a graph significantly but for the algorithms to run efficiently it is usually needed in some expanded form.

[3] External memory algorithms would also an option but are not the topic of this work [ASS14]

one meta node and the new edges between the meta nodes have the accumulated weight of edges between nodes of the respective clusters. This is the second phase, called *contraction*. Both phases are then repeated recursively on the meta graph until no further improvement is possible. We will cover this algorithm in greater depth in Section 2.2. The algorithm has been parallelized successfully in [SM16]. The parallel implementation is publicly available as part of the Networkit toolkit [SSM16].

Another successful approach to clustering which is of particular importance to this work, is *label propagation*. It was first introduced by Raghavan et al. in [RAK07]. The algorithm initializes each node with its own label. Then each node propagates its label to all neighbors. Now each node selects the label which he received most often as his new label. This process is repeated until the labels stabilize. This algorithm can be performed both sequentially by traversing nodes successively and selecting the label most prominent among their neighbors or in parallel by performing this in parallel for all nodes at the same time. The parallel version is suited very well for MapReduce implementations. This simple version of the algorithm has some issues with some labels getting too strong but there are many more sophisticated versions of it which address this problem [ŠB11], extend it to overlapping clusters [XS12] or apply the algorithm to other problems. In the KAHiP [SS13] implementation[4], it is used to partition social network or web graphs which proved difficult for traditional partitioning approaches [MSS14]. In this case the algorithm was extended by a size constraint for the labels. Also, it has been demonstrated that label propagation can actually also be seen as the optimization of a certain objective function [BC09]. This opens up the possibility to optimize other objective functions – like modularity – through label propagation under specific constraints. As it turns out, the local moving phase of the Louvain algorithm can actually be formulated as a label propagation algorithm.

Work on distributed clustering algorithms is only a relatively recent endeavor. One of the first algorithms was proposed by Riedy et al. [RBM12] and is based on finding edge weight maximal matchings between clusters and then merging matched communities in parallel. This approach is in fact an extension to one of the earliest modularity based methods, the Clauset-Newman-Moore algorithm [New03].

EgoLP, another distributed algorithm to find overlapping clusterings in social networks, was proposed in [BKA+14]. The implementation is based on Apache Spark and makes use of label propagation.

Several attempts were made to develop distributed versions of the Louvain algorithm. Cheong et al. proposed an GPU based algorithm in [CHLG13]. They use parallelism both in internal steps of the Louvain algorithm like the calculation of the best cluster for each node but also on the level of splitting the graph into partitions, distributing each partition to one processor and than clustering each partition independently. The interconnecting edges are ignored. This can lead to a loss in clustering quality as nodes may be merged into wrong communities if many of the nodes of their actual community are in other partitions. Even though this independent clustering of graph partitions can yield significant speed-ups, it is also problematic since it is highly dependent on the partitioning schema. Cheong et al. use a simple node id range chunking schema. This works well for graphs with a good input order like the web graphs they use in their evaluation. But if the input order were randomized or not as good as for web graphs this can have problematic consequences for the algorithms output quality. Zeng and Yu [ZY15] attempt to solve this problem by keeping the nodes directly neighbored to the nodes in each partition along with each partition. This solves the problem in part. However, it also creates new ones regarding the effectiveness of the parallelization. As they continue to use node id range chunking – with a little improvement for better load balancing – the question how to find partitions in

[4] `http://algo2.iti.kit.edu/kahip/`

a fast yet more robust and input independent way is left open. The partitioned Louvain algorithm is also used in [WFSP14]. This implementation is also publicly available – both in a MPI[5] and a Hadoop[6] based variant. Here, a proper partitioning schema based on PMETIS is used but the evaluation treats the partitioning as preprocessing and does not include it in the overall running time of the algorithm.

There are also algorithms which perform a local moving like step for all nodes in parallel like in label propagation. An open source implementation based on GraphX is available on Github[7]. Another GraphX based implementation was proposed in [LYW+16] but the evaluation includes only very small graphs. GossipMap [BH15] is an algorithm based on another objective function to be optimized – the *Map equation* [RAB09]. Since for the map equation computing the best move for a node is more expensive than for modularity, the algorithm makes use of a heuristic. The evaluations of GossipMap and EgoLP were, to the best of our knowledge, the only ones to include graphs with more than one Billion edges.

Contribution

In this work we present two modularity based distributed clustering algorithms for the MapReduce paradigm. We implement the algorithms in C++ using the experimental Thrill framework [BAJ+16]. We extend Thrill with several new operations optimized for graph processing. We thoroughly evaluate the algorithms and their performance both in terms of the running time and the quality of the clusterings they obtain. We continue the work of [ZY15] by implementing the algorithm with a much more robust partitioning schema. Additionally, our evaluation yields valuable insights on how the relationship between partition structure and clustering structure influences the parallelization of this algorithm. We evaluate all our algorithms on real world graphs and on synthetically generated benchmark graphs up to a size of about 25 Billion edges.

Outline

The remainder of this work is organized as follows: Chapter 2 begins by introducing some basic notation. Then, we cover the foundational sequential clustering algorithms we later build upon. In Chapter 3, we then will introduce our distributed algorithms. Chapter 4 contains an in-depth evaluation of the algorithms and their internals. Finally, Chapter 5 concludes this work.

[5] `https://github.com/usc-cloud/parallel-louvain-modularity`
[6] `https://github.com/usc-cloud/hadoop-louvain-community`
[7] `https://github.com/Sotera/distributed-graph-analytics`

2. Foundations

In this chapter, we introduce the concepts and algorithms on which we build our distributed algorithms. We start by introducing the necessary formal notation in Section 2.1. In Section 2.2 we then discuss the Louvain algorithm which provides the schema for our distributed algorithms. Finally, in Section 2.3 we cover label propagation, another clustering algorithm essential to our work.

2.1 Preliminaries

A *graph* $G = (V, E, w)$ is a tuple of a set of *nodes* V, a set of undirected *edges* E, and an edge weight function w. Each edge connects two nodes. The edge weight function $w : E \to \mathbb{N} \cup \{0\}$ assigns a weight to each edge. E may include self-loops but no multi-edges. We extend w to non-existing edges by setting their weight to zero. In the case of an unweighted graph, we assign a weight of 1 to each existing edge.

For conciseness, we allow $w(u, v)$ as a shorthand notation for $w(\{u, v\})$. In pseudo code, the use of subscript (for example w_{uv} rather than $w(u, v)$) indicates that the variable contains a precomputed value.

The degree of a node is the sum of the weights of the incident edges: $deg(u) := \sum_{v \in V} w(u, v)$. We denote the number of nodes $|V|$ by n, the number of edges $|E|$ as m, and the total weight of all edges $\sum_{e \in E} w(e)$ as M. Our input graphs are not necessarily connected.

We define a *clustering* as a disjoint partition of the node set. Formally, a *clustering* $\mathcal{C}$ of a graph consists of several clusters $C \in \mathcal{C}$ containing nodes of the graph so that $\cup_{C \in \mathcal{C}} C = V$ and $\forall C, D \in \mathcal{C} : C \neq D \implies C \cap D = \emptyset$. In a slight abuse of notation, we define $\mathcal{C}(v), v \in V$ to be the partition of v in $\mathcal{C}$.

This definition excludes overlapping clusters, but other than that it allows arbitrary node partitionings. But many of these clusterings do not fit into our informal understanding of clusters as groups of nodes strongly connected among each other and weakly connected to the rest of the graph. To define what actually is a good clustering, we need to introduce a quality function.

2.1.1 Modularity

We first introduce two concepts of weights of sets of nodes.

The first one is the weight between two node sets C and D:

$$E(C, D) := \sum_{u \in C, v \in D} w(u, v) + \sum_{u \in C \cap D} w(u, u) \tag{2.1}$$

Edges between nodes which are in both sets will be counted twice. The second term ensures that loops are also counted twice. The weight between a cluster and itself $E(C, C)$ is called *intra-cluster weight*. Trivially, we conclude that $E(V, V) = 2M$.

Further, there is the *total weight* of a node set or a cluster. This is the sum of the weights of all edges incident to nodes in C – again, counting loops twice.

$$A(C) := \sum_{u \in C} \left(\sum_{v \in V} w(u, v) \right) + w(u, u) \tag{2.2}$$

The total weight of a cluster could also be expressed by means of the weight between clusters:

$$A(C) = \sum_{D \in \mathcal{C}} E(C, D). \tag{2.3}$$

For brevity we define $e(C, D) := \frac{E(C,D)}{2M}$ and $a(C) := \frac{A(C)}{2M}$ which put these weights in relation to the total weight of the graph.

We can now define the *modularity* for a given clustering:

Definition 2.1. *Modularity is a function $Q(\mathcal{C})$ which maps a clustering to a rational number.*

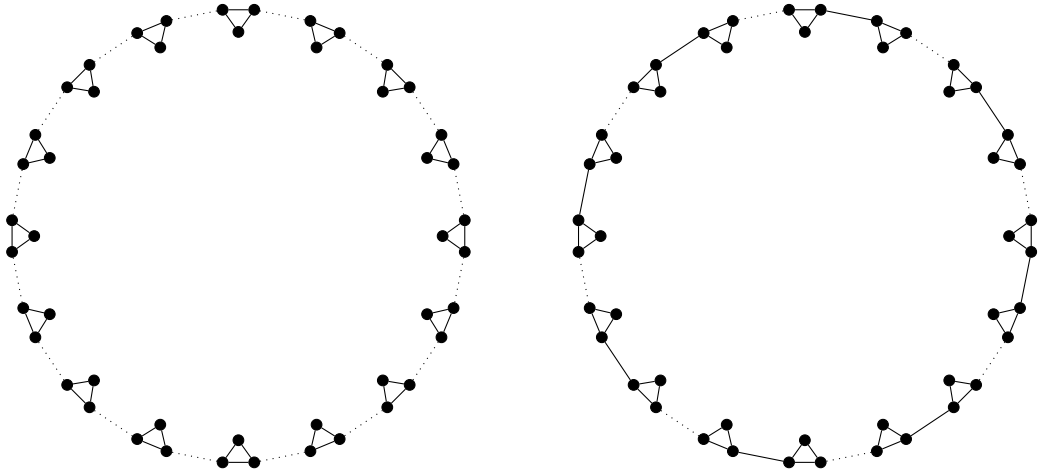
$$Q(\mathcal{C}) := \sum_{C \in \mathcal{C}} \Big(e(C, C) - a(C)^2 \Big) \tag{2.4}$$

The interpretation is that the first term is the observed fraction of weight within the cluster while the second term is the expected weight fraction in a corresponding random graph where each node has the same number of incident edges, but the target nodes of each edge are chosen at random. Thus, higher values of modularity indicate a "better" clustering. To optimize the modularity one needs to maximize the first term and minimize the second. The first term $e(C, C)$ grows when there are more edges inside the clusters. The second term $a(C)^2$ shrinks when each cluster has small total weight.

Properties

Modularity has been studied intensively from a theoretical perspective. In [BDG+08] several fundamental observations were made: For undirected and unweighted graphs modularity values are in the range of $[-\frac{1}{2}, 1]$. Isolated nodes do not contribute to the overall score of a clustering. Finding the clustering with maximum modularity for a given graph is $\mathcal{NP}$-complete.

In some cases, Modularity exhibits counterintuitive behavior: For example when doubling a graph (the doubled graph contains two unconnected subgraphs which each are isomorph to the original graph) the optimal clustering will be something completely different than the original optimal clustering on each subgraph. Another problem is the resolution limit of modularity [FB07, GdMC10]. Modularity fails to recognize clusterings below a certain total weight which depends only on the total weight M of the graph. Figure 2.1 shows the example of a ring of cliques. The intuitive clustering where each clique forms a cluster has a worse modularity score than when each two cliques are grouped in one cluster. With bigger graphs this example works also with larger cliques.



(a) $\mathcal{Q} = 0.79464$ (b) $\mathcal{Q} = 0.80357$

Figure 2.1: Resolution limit of modularity. Normal lines indicate edges within a cluster, dotted lines edges between different clusters.

2.1.2 Graph Partitioning

Related to clusterings are *graph partitions*, which also split the nodes of a graph into disjoint subsets. The difference is that the number and the size of the node subsets are restricted through parameters given as input to the problem.

Formally, a *ϵ-k-graph-partition* $\mathcal{P}$ splits the graph nodes into a fixed number of disjoint subsets k which is given as part of the problem input. Additionally, an imbalance parameter ϵ is given which induces a balance constraint on the sizes of partition parts:

$$|P| \leq (1+\epsilon)\left\lceil \frac{n}{k} \right\rceil : \forall P \in \mathcal{P} \tag{2.5}$$

In contrast to clusterings, for partitions it is easier to introduce a quality function reflecting the intuitive understanding. One widely used quality function is the total cut: The less the weight of the edges in between the different parts of the partition the better the partition.

$$\sum_{O,P \in \mathcal{P}, O \neq P} E(O,P) \tag{2.6}$$

Whether this quality function is a good one, is of course strongly dependent on the problem it is applied to and there are plenty of different quality functions available. For this work, the total cut is only used as a means to compare the quality of different partitions.

2.2 Louvain Algorithm

The Louvain algorithm was proposed in [BGLL08]. It is a greedy modularity optimization algorithm. Despite its age, it is still a popular graph clustering algorithm. It is one of the fastest known algorithms and delivers good results – though there are other algorithms which yield better clusterings at the cost of a higher runtime, for example the smart local moving algorithm [WvE13]. There is also a variety of techniques which build upon the schema of the Louvain algorithm and extend it with so-called refinements to improve the clustering quality.

The Louvain algorithm consists of two phases: A local moving phase and a contraction phase. These are repeated recursively until no further improvement is possible.

Algorithm 2.1: Louvain Algorithm

Input: Graph $G = (V, E, w)$
Output: Clustering $\mathcal{C}$

```
// Initialization
```
$\mathcal{C} \leftarrow \text{SingletonClustering}(G)$
```
// Local Moving
```
while $Q(\mathcal{C})$ *improved* **do**
  forall $v \in V$ *in random order* **do**
    $N_v \leftarrow \{\mathcal{C}(u) \mid \{u, v\} \in E\} \cup \{\mathcal{C}(v)\}$
    $C^* \leftarrow \arg\max_{D \in N_v} \Delta Q_{\mathcal{C}(v) \to v \to D}$ `// Resolve ties randomly`
    $\mathcal{C}(v) \leftarrow C^*$
```
// Contraction and Recursion
```
if $\mathcal{C}$ *changed* **then**
  $\mathcal{C}' \leftarrow \text{Louvain}(\text{BuildMetaGraph}(G, \mathcal{C}))$
  $\mathcal{C} \leftarrow \{\bigcup_{C \in C'} C \mid C' \in \mathcal{C}'\}$
return $\mathcal{C}$

Algorithm 2.1 depicts the algorithm. The algorithm begins by initializing the clustering to a so-called singleton clustering. That means that each node is in its own unique cluster. Then, in the local moving phase the algorithm iterates over all nodes – possibly several times.

Each time the algorithm gets to a node, it considers all clusters of nodes adjacent to the current node. For each cluster, the change of modularity, if the node was moved into that cluster is calculated. The cluster with the best resulting modularity gets selected and the node will be moved into it, but only if the resulting modularity is strictly better than the current one. We resolve ties uniformly at random, but other strategies are possible as well[1]. The order in which the nodes are visited, is randomized after each iteration.

In the base algorithm, this process is repeated until no further improvement is possible. For most graphs the algorithm converges relatively fast and in later rounds only very few nodes are moved. Exploiting that, one often uses a fixed number of iterations or a threshold for the modularity improvements to terminate the process earlier.

Finally, in the contraction phase, a meta graph is built from the clustering. In this meta graph each cluster from the clustering retrieved by the local moving becomes a new node. Edges between the meta nodes have the accumulated weight of edges between the nodes of the clusters. Edges between nodes of one cluster will become loops with the summed weight. A property of this meta graph is that its singleton clustering has the same modularity as the result of the local moving phase. Formally:

Definition 2.2. *For a graph $G = (V, E, w)$ and a clustering $\mathcal{C}$ the meta graph $G' = (V', E', w')$ is defined as*

$$\begin{aligned} V' &= \mathcal{C} \\ E' &= \{\{C, D\} \mid \exists u \in C, \exists v \in D : \{u, v\} \in E\} \\ w'(C, D) &= E(C, D) \end{aligned} \tag{2.7}$$

[1]The original publication mentions that several strategies are possible, but does not specify which one is used in the implementation.

The algorithm is then recursively applied to that meta graph until a local moving phase does not perform any changes and returns the singleton clustering. Recursively unpacking the meta clusterings yields the final result.

One of the most performance critical points in this algorithm is the modularity difference calculation. We can derive the equation to do so from the modularity definition. First, we derive the change of modularity when two clusters are merged. Since we defined modularity as a sum over all clusters, we only need to consider the two clusters in question.

$$\begin{aligned}\Delta Q_{C\cup D} &= \Big(e(C\cup D, C\cup D) - a(C\cup D)^2\Big) - \\ &\quad \Big(e(C,C) - a(C)^2\Big) - \Big(e(D,D) - a(D)^2\Big) \\ &= e(C,C) + e(D,D) + 2\cdot e(C,D) - \Big(a(C)+a(D)\Big)^2 - \\ &\quad \Big(e(C,C) - a(C)^2\Big) - \Big(e(D,D) - a(D)^2\Big) \\ &= 2\cdot\Big(e(C,D) - a(C)\cdot a(D)\Big)\end{aligned} \tag{2.8}$$

Now moving a node can be considered as moving it out of its current cluster – the inverse of merging it with this cluster – and then merging it with the target cluster.

$$\begin{aligned}\Delta Q_{C\to v\to D} = 2\cdot\Big(&e(\{v\}, D\backslash\{v\}) - e(\{v\}, C\backslash\{v\}) - \\ &a(\{v\})\cdot\Big(a(D\backslash\{v\}) - a(C\backslash\{v\})\Big)\Big)\end{aligned} \tag{2.9}$$

To be able to efficiently compute this value, we need to keep track of the total weights of each cluster. We maintain a list containing these weights which is updated whenever a node is moved between clusters. $a(\{v\})$ is equivalent to $deg(v)$, so it is trivially computable from our graph data structure. Calculating the weights between the node and all neighboring clusters can be done by iterating once over all outgoing edges. This needs to be done in any case, to aggregate the neighboring clusters. With all that, we can then compute the modularity difference for each cluster in constant time.

Despite its popularity and effectiveness, relatively little is known about the algorithm from a theoretical standpoint. It is possible to derive some bounds, but in comparison to the practical results they are much worse than what the algorithm usually delivers. One round of local moving takes $\mathcal{O}(m)$ time: iterating over n nodes and for each node iterating over all outgoing edges. Since the modularity strictly increases in each move taken by the algorithm, a limit for the number of moves can be implied by the number of possible modularity values which is in $\mathcal{O}(m^2)$. The number of recursions is trivially bound by n since the algorithm would terminate if all nodes stayed in their original singleton cluster. However, practical experience suggests the total runtime is indeed quasi-linear in the number of edges and that the runtime of a full round of local moving is the dominating element in the total runtime.

Interestingly, due to its recursive nature the Louvain algorithm can sometimes circumvent certain issues of modularity, like the resolution limit. For example, in some cases a clustering found on a more fine-grained level might still contain clusters with sizes below the resolution limit. These will, of course, be merged on a more coarse level because it improves the modularity score. But when one makes use of the clustering from the more fine-grained levels, this information can be retained.

2.3 Label Propagation

Label Propagation, as introduced in [XS12], is another community detection approach. The term *label* refers to the same concept we previously defined as cluster. Both terms are used interchangeably in this work, but for consistency with the literature we will use the term label when referring to the label propagation algorithm.

The algorithm is conceptually relatively simple: Initialize each node with its own unique label, iterate over all nodes and update each node's label to the label appearing most often among the neighboring nodes (resolving ties randomly) and repeat this until the labels have stabilized. This process is actually conceptually very similar to the local moving algorithm. Only the criterion to determine the best label is much simpler.

There are two conceptual variants of this algorithm. In the first one, the updating of the node labels is performed asynchronously (see Algorithm 2.2). That means, a node can already know the updated labels of other nodes which were updated before in the same iteration. In the other variant, the updating is executed synchronously (see Algorithm 2.3). In this case, all node labels are updated at the same time.

Algorithm 2.2: Asynchronous Label Propagation

Input: Graph $G = (V, E)$
Output: Labels $\mathcal{L}$

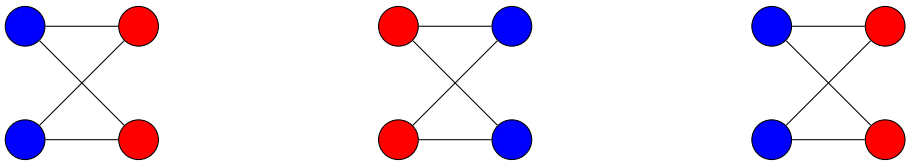
`// Initialization`
$\mathcal{L}_0 \leftarrow$ SingletonLabels
while $\mathcal{L}$ *changed* **do**
  forall $v \in V$ *in random order* **do**
    $N_v \leftarrow \{\mathcal{L}(u) \,|\, \{u, v\} \in E\}$
    $L^* \leftarrow \arg\max_{L \in N_v} |\{u \,|\, \{u, v\} \in E, \mathcal{L}(u) = L\}|$ `// Resolve ties randomly`
    $\mathcal{L}(v) \leftarrow L^*$
return $\mathcal{L}$

Algorithm 2.3: Synchronous Label Propagation

Input: Graph $G = (V, E)$
Output: Labels $\mathcal{L}$

`// Initialization`
$\mathcal{L}_0 \leftarrow$ SingletonLabels
while $\mathcal{L}$ *changed* **do**
  forall $v \in V$ *in parallel* **do**
    $N_v \leftarrow \{\mathcal{L}_t(u) \,|\, \{u, v\} \in E\}$
    $L^* \leftarrow \arg\max_{L \in N_v} |\{u \,|\, \{u, v\} \in E, \mathcal{L}_t(u) = L\}|$ `// Resolve ties randomly`
    $\mathcal{L}_{t+1}(v) \leftarrow L^*$
return $\mathcal{L}_t$

The synchronous variant has one great advantage: it is very well suited for parallelism. On the other hand it also introduces the problem of label oscillation – see Figure 2.2 for an example. If there are two connected nodes which currently have different labels but belong to the same community, they will switch their labels in each iteration. More general, this is always the case when there is a bipartite network where the two parts each have exactly one label.



(a) At $t-1$ (b) At t (c) At $t+1$

Figure 2.2: Example of label oscillation in a bipartite graph

The usually observed runtime of label propagation scales linearly with the number of edges, quite similar to what one usually sees with the Louvain algorithm. This also corresponds to the complexity of one full iteration, which is in $\mathcal{O}(m)$: iterating over all nodes and for each node over all its neighbors.

One problem of label propagation is that oftentimes one label starts to grow too strong and successively consumes more and more other labels. This corresponds to an observation made in [BC09]. There, it has been shown that label propagation also has a quality function which it optimizes. This function actually has its global maximum in the trivial state where all nodes share the same label. The algorithm only delivers reasonable results because it gets stuck in some local maximum beforehand.

Size Constrained Label Propagation

Beside community detection, label propagation can also been applied to graph partitioning. In this work, we use the size constrained label propagation approach from [MSS14] to compute partitions for a partitioned Louvain algorithm. To fulfill the partitioning balance constraint, size constrained label propagation prohibits the propagation of labels containing too many nodes. If a label already contains more or as many nodes as the balance constraint allows, no more nodes are added to the label. This also solves the problem mentioned above of a single label becoming too strong and consuming all others. Algorithm 2.4 shows this variant of the algorithm.

Algorithm 2.4: Size constrained Label Propagation

Input: Graph $G = (V, E)$
Output: Labels $\mathcal{L}$

`// Initialization`
$\mathcal{L}_0 \leftarrow$ SingletonLabels
while $\mathcal{L}$ *changed* **do**
  forall $v \in V$ *in random order* **do**
    $N_v \leftarrow \{\mathcal{L}(u) \,|\, \{u, v\} \in E, |\mathcal{L}(u)| < (1+\epsilon)\lceil \frac{n}{k} \rceil\}$
    $L^* \leftarrow \arg\max_{L \in N_v} |\{u \,|\, \{u, v\} \in E, \mathcal{L}(u) = L\}|$ `// Resolve ties randomly`
    $\mathcal{L}(v) \leftarrow L^*$
return $\mathcal{L}$

3. Distributed Algorithms

This chapter begins with an introduction to the MapReduce paradigm in Section 3.1. We continue with an overview about Thrill, the MapReduce framework our implementation uses. In Section 3.2 we then introduce our distributed clustering algorithms.

3.1 The MapReduce Paradigm

MapReduce is a programming model for scalable computations on large datasets. It was introduced by Google [DG08] in 2004. It allows the programmer to formulate his algorithms without much concern about the technical details of parallelization, data distribution, fault tolerance, scheduling, and networking. The original proprietary implementation from Google was never published but luckily several open source frameworks implemented the model over the years – for example Apache Hadoop[1] and its successors.

MapReduce programs operate on lists of items. Each item in these lists is a key value pair. MapReduce programs are expressed by defining a *Map* function and a *Reduce* function. The *Map* function takes one key value pair and maps it zero or more new key value pairs. This corresponds to a *FlatMap* function in functional programming. The *Reduce* function takes a key and all values associated with that key and combines these values into a single value.

The MapReduce framework applies these functions to the items in the list. The *Map* function is applied to each item. The *Reduce* function is applied to each key together with all values associated with that key. Everything which has to happen in between the *Map* and *Reduce* operations is done by the framework. The execution of a MapReduce program can be grouped into five phases:

- Input: An input reader reads data from a distributed file system, generates the key value pairs and distributes them among the available workers. Oftentimes the input might already be partitioned into appropriate subfiles.
- Map: The map function is applied to each key value pair. This can happen completely independently on each worker. Additionally, each worker could further parallelize this step using multiple threads.

[1] `http://hadoop.apache.org/`

- Shuffle: The key value pairs are redistributed among all workers so that in the end key value pairs with the same key all end up on the same worker. It is crucial to the performance of the program that this step redistributes as few elements as possible. Oftentimes the network communication time is the dominating factor in the overall running time.
- Reduce: The reduce function is applied to all groups of elements with the same key. This again can be done completely in parallel.
- Output: Each worker writes its items to the file system. The resulting files can be stored or used as input to another MapReduce program.

An additional combine phase can be inserted after the map phase which locally combines all elements of the same key on each worker before transmitting them to their target worker. This can reduce the network traffic and thus improve the performance.

Data in MapReduce does not have any specific ordering. The original Google implementation guaranteed that in the output the elements of each worker would be sorted by key but this is no requirement of the programming model. Also, there is no order defined among the workers.

This programming model in itself does not contain any means to perform iterative or recursive algorithms. Nevertheless, it is possible to achieve this by chaining multiple MapReduce programs – or the same program for several times. In fact, all of the MapReduce algorithms in this work are chains of several MapReduce programs. Thrill, the framework we use, allows to construct arbitrary data flows using map, reduce and quite a few other functional operations. However, most of them can be expressed in terms of the original map and reduce operations.

MapReduce has also been subject to theoretical analysis. In [KSV10], a theoretical computation model for MapReduce was proposed and compared to the PRAM model.

3.1.1 Visualization of MapReduce Algorithms

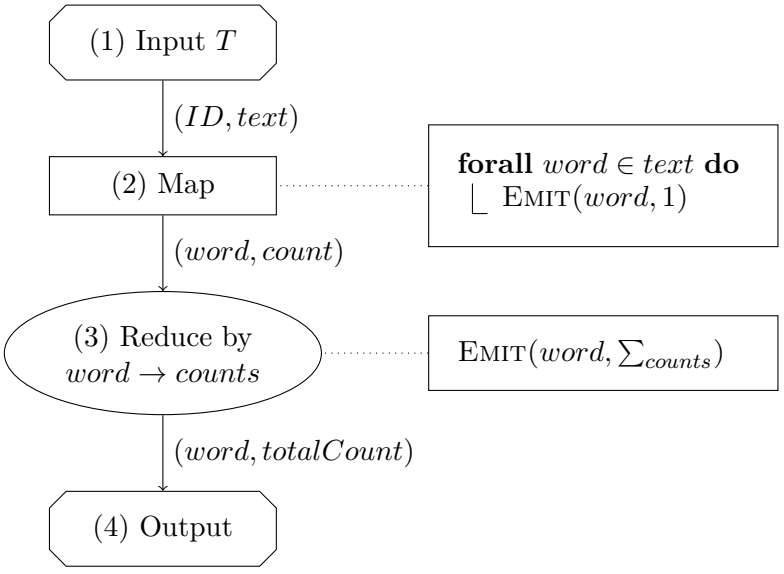


Figure 3.1: The Word Count Algorithm in MapReduce

We will depict MapReduce algorithms as data flow graphs. Figure 3.1 shows the standard example of counting words in a list of texts. Data flows from input (1) through the map (2) and reduce (3) operations to output (4). Non-trivial operations are annotated with pseudo

code. The pseudo code specifies the algorithm applied to each item during the operation. The EMIT function is used to generate the output of one operation and pass it to the next.

The map operation (2) iterates over all words in each text, and emits each word as key together with a 1 as the value. Then, in the reduce operation (3) all items with the same word as key are combined and the occurrence counts are summed up. The $word \rightarrow counts$ notation in the reduce operation node means that all list items with the same *word* will be grouped and the variable *counts* will contain a list of all the associated *count* values.

The connections between the operations indicating the data flow are annotated with a schema[2] of the data flowing through it. The names from the schema are used in the following operation to reference the respective parts of the data. Values in square brackets (for example $[v]$) indicate a list of values.

In contrast to the original MapReduce programming model, we do not require the data to be in the form of key value pairs but allow tuples. Operations relying on keys will explicitly specify the key they use. We omit map operations which only select a subset of the data in a tuple or reorder the data – they are reflected by the schema.

Nodes with an octagonal shape indicate special operations related to control flow or interacting with the runtime environment. Operations represented by a rectangular shape are local operations. That means that no communication between the workers is required. Nodes with an oval shape represent distributed operations which require communication.

For reduce operations, like (3), which sum up the values for each key, we will in the following omit the pseudo code and denote them by *Reduce by* key *and sum* values. Furthermore, for key value pairs where the value is a collection, we also define a shorthand operation called *expand*. This operation is a (flat-)map operation which iterates over all elements in the value collection and emits them together with a key. An example for *expand* would be to turn an adjacency list graph where the items are pairs of a node id and a list of neighbor ids into an edge list graph where the items are pairs of node ids representing an edge. We also define *aggregate* as a shorthand for the inverse operation.

3.1.2 Thrill

Thrill[3] is an experimental C++ framework for distributed data processing [BAJ+16]. The code is available on Github[4]. It features an interface very similar to the MapReduce programming model.

Thrill can be compared to other Big Data frameworks like Apache Spark but there is one big conceptual difference: Data in Thrill is represented as an array like data structure – thus, it always has an inherent order. The order might be arbitrary at times but there always is one. These data structures are called *Distributed Immutable Arrays* or *DIAs*.

DIAs can not only contain primitive data, but also complex types. Specifically, even data where the serialized items might be encoded using a variable number of bits. Thrill supports the C++ STL vectors out of the box and the programmer can define custom serialization methods for other types.

The data in the DIAs cannot be accessed directly by the user. Instead, one needs to define a data flow by means of distributed operations including the traditional map and reduce operations. To define this data flow, the programmer calls the function associated with the operation on the DIA, for example *map*, and passes a C++ lambda as an argument

[2] The data can be thought of as a SQL style table. Each item would be one row and the schema would specify the column names.

[3] `http://project-thrill.org/`

[4] `https://github.com/thrill/thrill`

to the function. This lambda takes one item of the list as an argument and returns the transformed item. The framework then invokes the lambda for every item of the DIA. This interface is very similar to how lists are often processed in functional programming languages.

A technical difference to the original MapReduce programming model is that Thrill does not require its DIA items to be key value pairs. Instead, when invoking a key dependent operation the user supplies an additional key extractor lambda which maps each element to its respective key.

Another significant difference between MapReduce and Thrill is that Thrill does not limit the user to a single data flow but allows operating on an arbitrary number of DIAs. DIAs can almost be treated like regular variables. They can be passed around in and out of functions, be used iteratively and even recursively. Thrill data flows can branch out, so their data is used in multiple subsequent operations. And on the other hand multiple data flows can also be merged together and combined back into a single one. In fact, the flow of data in Thrill can assume any form of a directed acyclic graph.

Due to these two key differences, Thrill offers more operations than the traditional MapReduce programming model. In the following, we give a brief overview of the operations relevant to this work. Some of these are conceptually equivalent but differ in the implementation. This has important implications for performance.

- *FlatMap*: Thrill's *FlatMap* operations corresponds to the *map* operation in the traditional MapReduce programming model. It takes one item and emits zero or more new items. Each worker can carry out this operation locally on all his items without any communication to other workers.
- *Map*: The *Map* operation is a specialization of *FlatMap* where each element is mapped to exactly one new item.
- *Filter*: *Filter* removes all items for which a user-supplied predicate is false.
- *GroupByKey*: Thrill's *GroupByKey* operation has the closest resemblance to the original *reduce* operation. Items are grouped by their key and transmitted to one specific worker responsible for their key. There, they get sorted and delivered to the user's function through an iterator. That means that before the user operation can be executed, all items must have been transferred to their destination and only once everything is in place, the execution of the user function can begin. The assignment of keys to workers is done through hashing.
- *GroupToIndex*: This operation works very similar to *GroupByKey*. The difference is the handling of the keys. In this operation the keys are considered to be indices in a range from 0 to x – the user has to supply x as an argument to the call. The index range gets partitioned and distributed among the workers. After the execution of the operation, the resulting DIA contains the elements ordered by their indices. If there are any holes in the range (indices for which there were no elements), they will be filled with a neutral element.
- *ReduceByKey*: For Thrill's *ReduceByKey* operation, the user has to supply a function which combines two list items with equal key into one item of the same type. Similar to *GroupByKey*, elements will also be assigned to one worker where all elements with the same key will be reduced together. But in addition to that the reduce function is also applied to all elements with equal keys on each worker before transmitting them over the network. Thrill does so by using hash tables. This also allows the reduction of elements as soon as they arrive rather than first receiving all items, like

in *GroupByKey*. This is in most cases a huge advantage and Thrill advises its users to favor reduce operations over group operations. But reduce has the limitation that input and output have to have the same type.

- *ReduceToIndex*: This operation works like *ReduceByKey*, only that the keys are again treated as indices similar to *GroupToIndex*.
- *Sort*: As the name suggests the *Sort* operation sorts a DIA into the order defined by a user-supplied comparator. It also re-balances the data among the workers, so that each worker has a similar number of items.
- *Zip*: The *Zip* operation combines two (or more) DIAs of the same size into a single new DIA by combining elements at the same position through a user-supplied function.
- *ZipWithIndex*: This operation works like *Zip*, only that the elements are not combined with elements from another DIA but with their indices.
- *InnerJoin*: Recently a *Join* operation was implemented in Thrill as part of a master's thesis [NB17]. *Join* is related to *Zip*, but it combines elements based on keys rather than order. The two DIAs do not need to have equal lengths but each pair of elements in the Cartesian product will be passed into the user-supplied combination function. This is similar to an SQL style inner join. During development, we made heavy use of this operation but more detailed benchmarking showed that usually *Join* causes more performance problems than it solves. In most cases, there are more efficient alternatives to a full-blown *InnerJoin*, so in our final algorithms this operation is used seldom.

Besides these operations which take data from one DIA and deliver it into another DIA, there are also so-called *actions*. Actions perform a calculation on the DIA whose result is a single value which then gets delivered onto all the workers.

- *Size*: delivers the total number of elements in the DIA to each worker.
- *Sum*: calculates the sum of all elements in the DIA.
- *Gather*: Transfers the content of the entire DIA to one worker where it is made accessible as a STL vector.

These lists are not comprehensive, but cover only the operations most relevant to this work. During the process of implementation, we also implemented a few more operations necessary to solve specific (performance) problems we encountered.

- *CollectLocal*: This is an action which makes the local DIA content of each worker accessible as a STL vector.
- *FoldByKey*: This operation is another variant for the reduce step from MapReduce. Similar to *ReduceByKey* elements can be combined (or rather folded) immediately when they arrive on their target worker by making use of a hash table, but the restriction that the output type has to be the same as the input type is avoided. Formally speaking, for a DIA with items of type A, reduce requires its function to have the signature $A \times A \to A$. Fold allows $B \times A \to B$. This enables significant improvements when aggregating all elements of one key into a single item.
- *Unique*: This is a utility operation which is mapped to a reduce operation. The elements themselves are the keys. That means all equal elements will be grouped. The reduction function just picks one of the equal elements.

In the pseudo code examples given throughout this thesis, we use a generic *Reduce* (or *ReduceToIndex*) operation rather than the concrete *FoldBy*, *ReduceByKey* or *GroupByKey* as this is a performance related implementation detail.

3.2 Distributed Louvain

In this section we present two Thrill/MapReduce algorithms for distributed graph clustering. Both are based on the Louvain algorithm. The first one is a hybrid approach which uses MapReduce to partition, distribute, and contract the graph but still performs the local moving step using the sequential local moving algorithm on subgraphs. It is described in Section 3.2.1. The second algorithm, introduced in Section 3.2.2, is completely MapReduce based and can be parallelized on the level of individual nodes. Both algorithms use our distributed graph contraction algorithm which is covered in the final section (3.2.3).

3.2.1 Distributed Louvain with Partitioned Local Moving (DLPLM)

This first algorithm is based on and extending the work of [ZY15]. The basic idea behind the algorithm is to split the graph in as many partitions as there are workers available and let each worker perform the local moving phase independently on its subgraph. The clusterings from each worker are then combined, the graph is contracted and the whole process is repeated recursively. Once the graph shrank below a certain size, we switch back to the sequential Louvain algorithm on a single worker.

We will start by covering the partitioning phase. In previous work, this step is either treated as preprocessing and performed by external tools or implemented rather simplistically, based on chunking the node id range. Both approaches have problematic consequences. If the partitioning is part of the preprocessing, only the first level of the algorithm can be distributed among the available workers. So if the coarsened graph is still too big for a single compute node this approach will fail.

Id chunking approaches, on the other hand, are strongly dependent on the input order of the graph. See 4.2.2 on how randomizing the input order renders partitioned local moving based on id range chunking completely ineffective.

To circumvent these issues, the partitioning is an integral part of our algorithm. That also means the graph can, if necessary, be partitioned on any level of the algorithm, not just the first. Our requirements regarding the partitioning quality are relatively loose. The most important goal is to cut through as few clusters as possible, while still achieving a reasonable balance. That makes label propagation a good fit, since the algorithm has been successfully used in both partitioning and clustering.

Figure 3.2 depicts the data flow of a single iteration of label propagation. The algorithm has two inputs: The graph as a DIA of node ids with an array of neighbor node ids, and the current labels for each node as a list of pairs of a node id and a label id. Before the first iteration the labels are initialized to singleton labels, so every node has its own unique label. Both inputs have to be sorted by the node id, so they can be combined without any further communication in operation (3).

The next step is to transfer the label of each node to all neighbors. This is done in step (4) by emitting the label once for each adjacent node – together with the id of that neighbor. Reducing by this target id, we then get all incoming labels for each node. We aggregate them and select the one with most occurrences. Ties are resolved randomly, similar to the original algorithm.

As we use a *ToIndex* operation for this reduction, the new labels will be ordered by the node id. That allows us to use the output of the reduction step (5) directly as the label input for the next iteration. This process can be repeated until the labels have completely stabilized – or until only very few nodes still move between labels.

Each iteration can be divided into three phases (zip, expand, reduce), as indicated by the coloring. We will encounter these three phases again in later algorithms.

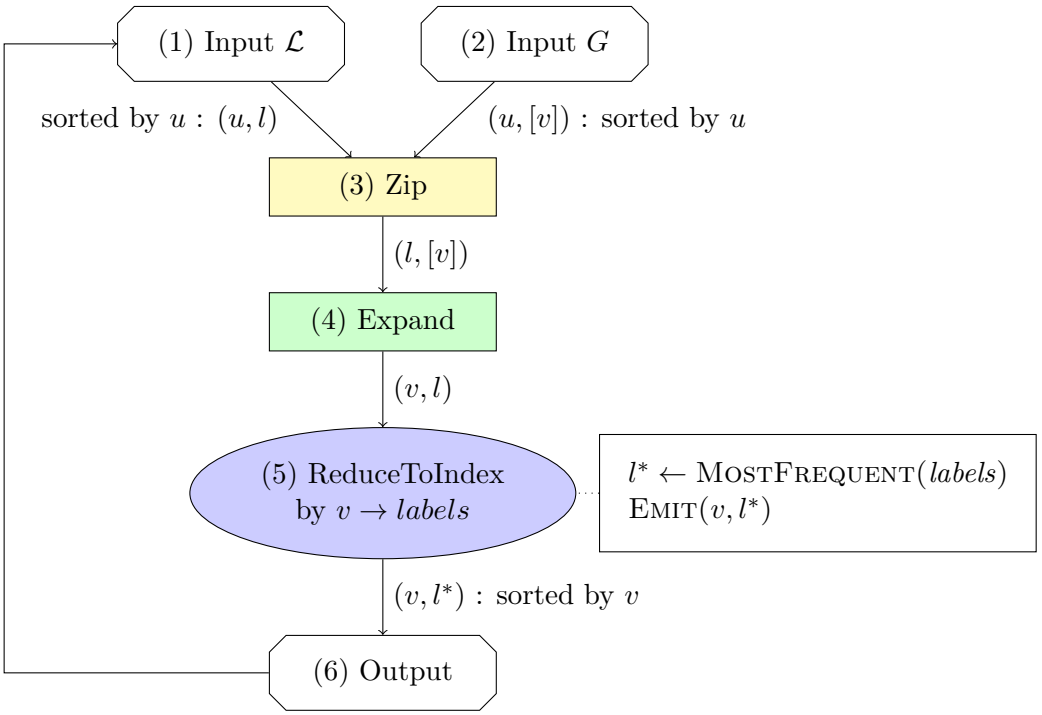


Figure 3.2: Distributed Label Propagation

This algorithm can produce reasonable clusterings but we need a graph partitioning. And since the clusters can have arbitrary sizes, the output of this algorithm is not yet suited for our purposes. Thus, we extend the algorithm with size constraints in analogy to Algorithm 2.4.

For this we need to limit the propagation of labels which already contain too many nodes. Figure 3.3 shows how to achieve this in MapReduce. We introduce a fourth phase to the label propagation schema which consists of the operations (4) and (5).

Operation (4) aggregates all nodes of one cluster into a single MapReduce item. If labels grew arbitrarily the size of such an item could even become too big for the RAM of a single worker. But as this algorithm is actually limiting the size of each label, this is not a problem.

Operation (5) continues by filtering out all labels containing too many nodes, so no outgoing labels will be emitted for them. The operations (6) and (7) correspond to operation (4) from the original algorithm. We need to expand twice since we first need to expand all nodes out of each label and then each neighbor node from each node. The rest of the algorithm is the same as in the original algorithm.

There is one edge case omitted in this depiction: A node might only have incident labels which are already too big. In this case there will be no incoming labels and the node will be missed in the reduction. This can be prevented by emitting each node together with its own label, regardless of the cluster size. Or it could be mitigated by zipping the new labels with the old labels and retrieving the missing information from there.

This algorithm now yields a clustering sufficient for our use case. Individual clusters might still exceed the maximum size depending on how many nodes join the cluster in the iteration where it grows beyond the maximum size. But in practice the results are good enough.

We still might have many more small clusters than we need for the partitioning. To get an actual partitioning with as many parts as we have workers we need to combine clusters until we have the right number of partitions.

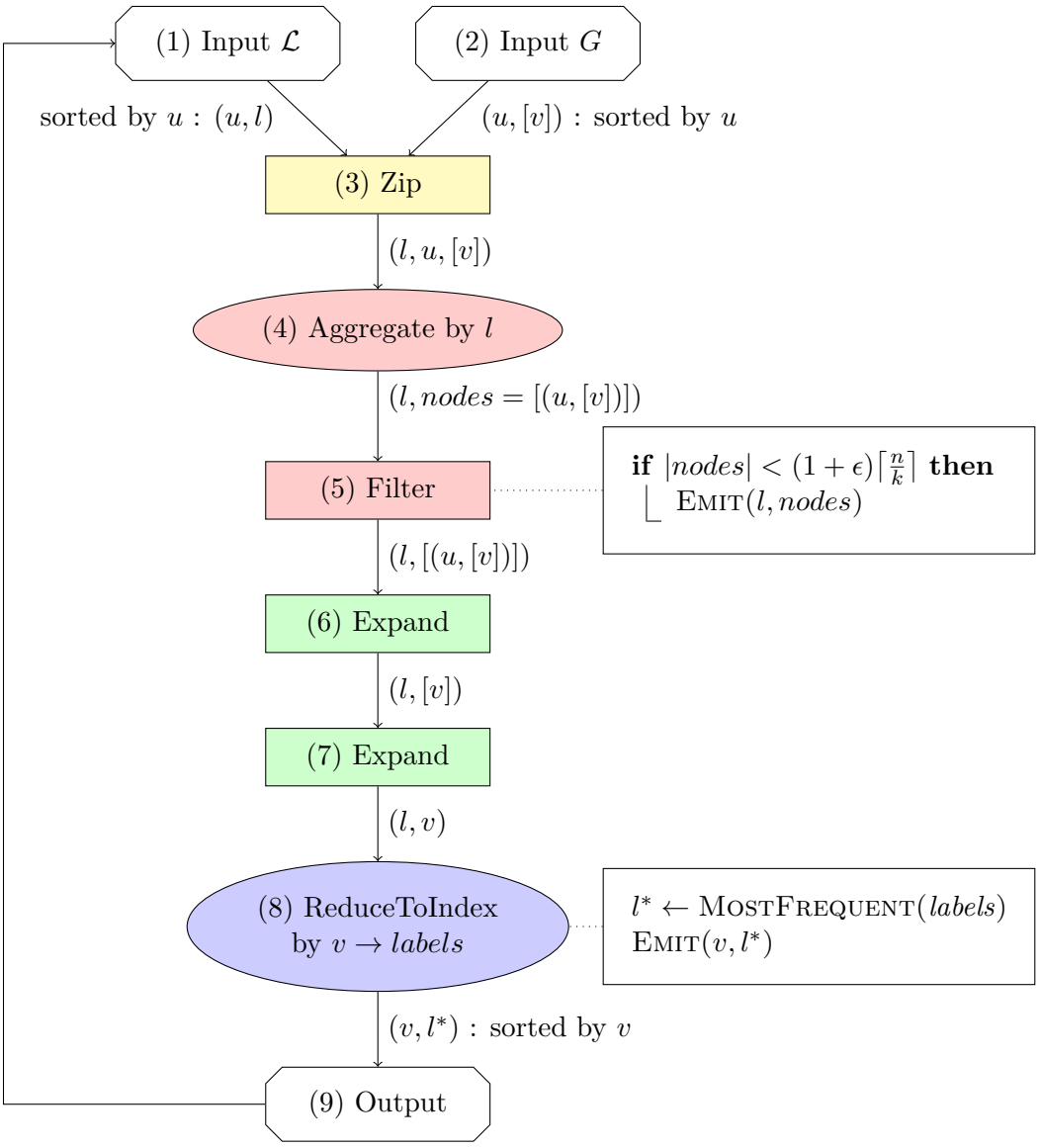


Figure 3.3: Distributed Size Constrained Label Propagation

This problem is related to the bin packing problem. Because of that, we use one of the well-known approximation strategies: the *best fit decreasing* strategy. That means we sort our clusters by decreasing size and then successively assign each of them to the partition where it fits with the fewest space left. We assume that the number of clusters is small enough to perform this algorithm sequentially on the master worker. That finally yields a graph partitioning with which we can proceed to the actual clustering algorithm.

We need to zip each node with its partition and then distribute each node to the worker tasked with clustering its partition. Then we can execute the original sequential local moving algorithm on each subgraph.

There is one issue left open and that are the interconnecting edges. It would be possible to just drop them. But that would reduce the quality of the final clustering. Nodes belonging to clusters from a different partition would likely be merged into a local cluster.

To avoid this problem we implement another improvement from [ZY15]: *ghost nodes*. Ghost nodes are nodes which are directly adjacent to nodes from the partition but not part of the partition. We keep these nodes and thus the interconnecting edges along with each partition. Additionally, we store the degrees of all ghost nodes.

During local moving, ghost nodes start in their own unique cluster, just like any other node. The difference is that ghost nodes will never be moved during the local moving. But it is still possible for regular nodes to be moved into clusters of ghost nodes.

When passing the resulting clustering on to the contraction, ghost nodes are just ignored. They will be in the cluster they were assigned to in their own partition. If the cluster a ghost node was assigned to in its own partition and the local cluster of the ghost node need to be merged, then this will happen in any case during the next local moving on the contracted graph.

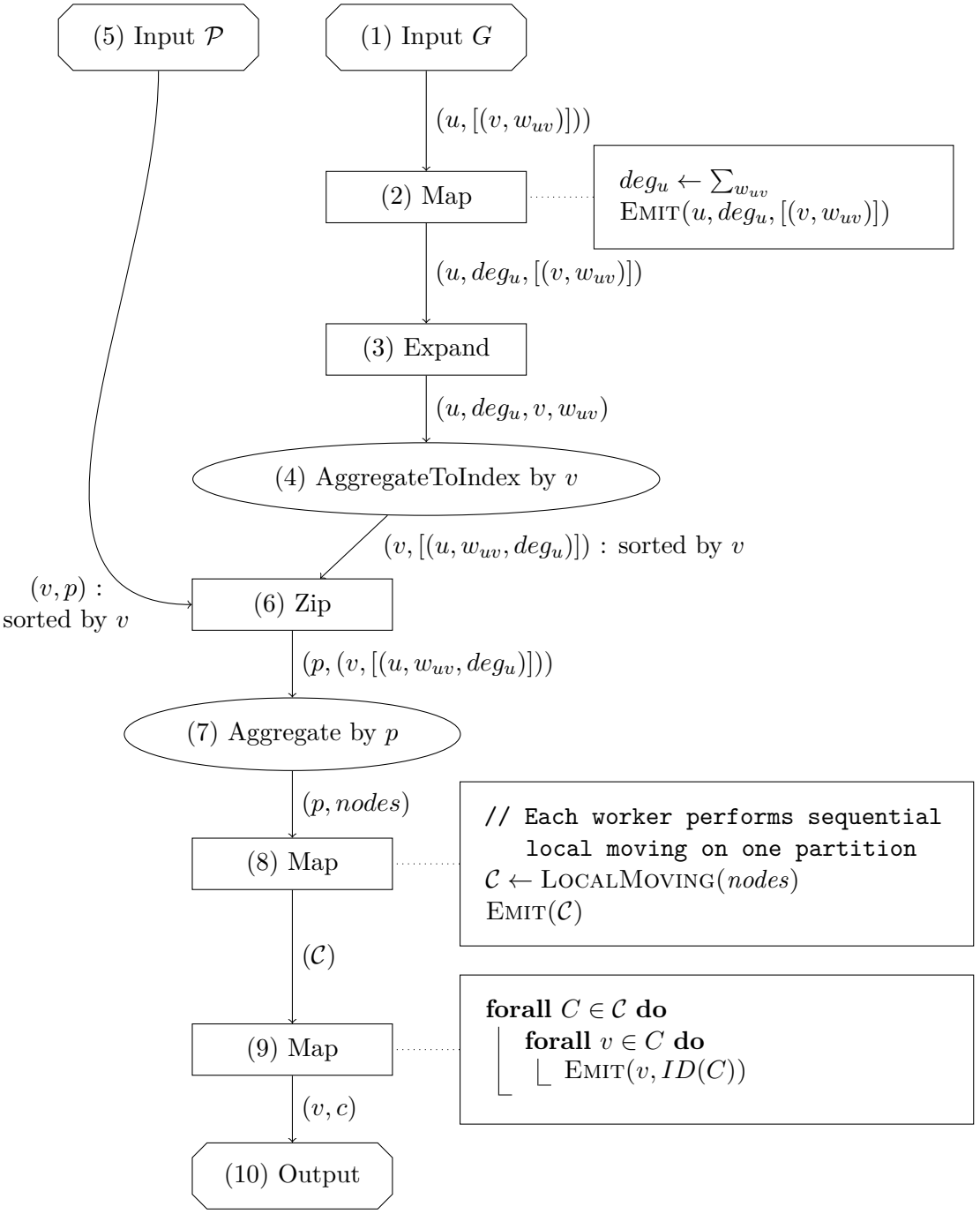


Figure 3.4: Partitioned Local Moving

Figure 3.4 depicts the data flow of the implementation of the described algorithm. As input, we take the graph (1) as a DIA, where each item consists of a node id and a list of

adjacent neighbor node ids together with the weight of the edge. The second input is a DIA containing the partition of each node (5). Both DIAs have to be ordered by the node id. In operation (6) we zip the graph and the partitioning. When aggregating these in (7), we get a local adjacency list for each partition.

The adjacency arrays already include the ghost nodes. But without the operations (2), (3) and (4) a crucial information would be missing: the degree of the ghost nodes. Operation (2) calculates the degree of each node by summing up the weights of the incident edges. Then operation (3) expands the graph into an edge list, now annotated with the degree of the source node of the edge. We collapse this into an adjacency list graph by aggregating by the target node of each edge in (4). Now the adjacency lists contain the degree of each neighboring node.

With that, operation (8) can perform the sequential local moving with ghost nodes for each partition. Finally, operation (9) expands the local clusterings into a DIA, which is then passed on to the distributed contraction algorithm.

But before covering this algorithm we will first introduce another algorithm to perform distributed local moving purely in MapReduce without the need for explicit partitioning.

3.2.2 Distributed Louvain with Synchronous Local Moving (DLSLM)

The local moving and the label propagation algorithm have a similar structure. In both algorithms we iterate over all nodes, look at each nodes' neighbors and their labels or clusters and select the best one of them according to some objective function. Thus, it should be possible to formulate the local moving in a label propagation style MapReduce algorithm. The challenge lies within calculating the change of modularity. Therefore, we need more information than just to which cluster the neighbors of each node belong. In this section, we present an MapReduce algorithm achieving this with only two distributed operations per iteration – and a structure analogue to the distributed size constrained label propagation algorithm.

According to equation 2.9 to calculate the change in modularity, we need the degree of a node, the weight of each cluster – but excluding the node – and the weight between the node and the neighboring cluster. And we would need all that for both the current cluster of each node and each neighboring cluster. It is possible to calculate all this information and join it onto the appropriate parts of the graph representation. However, this would be extremely expensive in terms of necessary distributed operations and the communication overhead they introduce.

Luckily, actually not all of the information is strictly necessary. And the parts which are indeed necessary can be retrieved efficiently without using joins. The first thing to note is that to select the best neighboring cluster we do not need to calculate the precise change of modularity, but only a value proportional to it. As the parts of the equation referring to the current cluster will remain constant for any neighboring cluster, we can just drop them.

$$\Delta Q_{v \to D} \propto 2M \cdot e(\{v\}, D \backslash \{v\}) - a(\{v\}) \cdot a(D \backslash \{v\}) \tag{3.1}$$

But now, in this new equation, the baseline for a positive change of modularity is not zero anymore. Rather, the baseline is the value of this function for the node with its current cluster. That means we need to make sure to always include the current cluster in the list of possible clusters for a node. Luckily, this is much easier and cheaper in terms of computation and communication than calculating and distributing these values to each node.

We still need each nodes' degree and all the values for each neighboring cluster. We can obtain the degree similarly to how we did in the partitioned local moving algorithm, by annotating the adjacency list with the degree of each neighbor node.

The other two remaining values are actually quite similar to what we already had in size constrained label propagation. $a(D\backslash\{v\})$ is the weighted equivalent of the label size we used to restrict propagation of labels which were too big. Only now, we need to exclude the weight of the node, if it is part of that cluster, and pass the value on to the operation actually selecting the best cluster.

The weight between a node and a neighboring cluster also has an equivalent in label propagation. There, we counted the occurrences of incoming labels. This is equal to the number of edges to nodes with that label. And the weight between a node and a neighboring cluster is the sum over the weights of these edges. So this value can be calculated the same way. With that, we are able to formulate our algorithm as shown in Figure 3.5.

The algorithm starts by annotating the adjacency list with the neighbor node degrees through operations (2), (3) and (4), as explained before in Figure 3.4. There, we can see the four familiar phases of distributed label propagation. First, we zip the extended graph data structure with the current clustering. Then, we aggregate nodes by their cluster in operation (7)[5].

This time, we do not filter by the cluster size but rather calculate the information and pass it to the next operations. There are several noteworthy details in the expansion operation (8). First, when iterating over the outgoing edges we exclude loops. That is because we always need the weight between a node and the cluster excluding the node. Secondly, we additionally emit the cluster information for each node, but with a total weight decreased by the degree of the node and a weight between the cluster and the node of zero. We could achieve the same by adding a loop with weight zero to each node (and not ignoring it in the iteration over the outgoing edges).

In the operations (9) and (10), we now aggregate this information on incident clusters for each node. There will be as many incoming elements as the node has incoming edges – excluding regular loops and adding one for the imaginary loop. For each of the clusters represented in this information we can now sum up the weights. This will yield the weight between the node and the cluster. For the total weight of the cluster, we take the minimum of all the total weights for this cluster. If the cluster is the nodes current cluster, there will be one total weight (the one from the imaginary loop) which contains the correct reduced total weight, which excludes the node. Together with the node degree, which was also passed along, this is everything that is required to calculate ΔQ. So the algorithm calculates the modularity change values, and selects the best of them, resolving ties randomly. If no improvement is possible, this will always be the current cluster or at least one equally good.

However, this has one problematic consequence: the convergence of the algorithm is not ensured anymore. For example, if one node were placed in two communities with equal modularity values, the algorithm could oscillate endlessly between these two states. To avoid this, we implement oscillation countermeasures.

Cluster Oscillation

In Section 2.3. we touched on the issue of label/cluster oscillation. This is an issue which this algorithm needs to mitigate. Our solution is to move only a subset of the nodes in each iteration.

[5]This has, again, the implication that for each cluster we can fit all its nodes into the RAM of a single worker. In contrast to size constrained label propagation, there is no limit for the size of a cluster. Nevertheless, we did not encounter any issues with this assumption in our experiments.

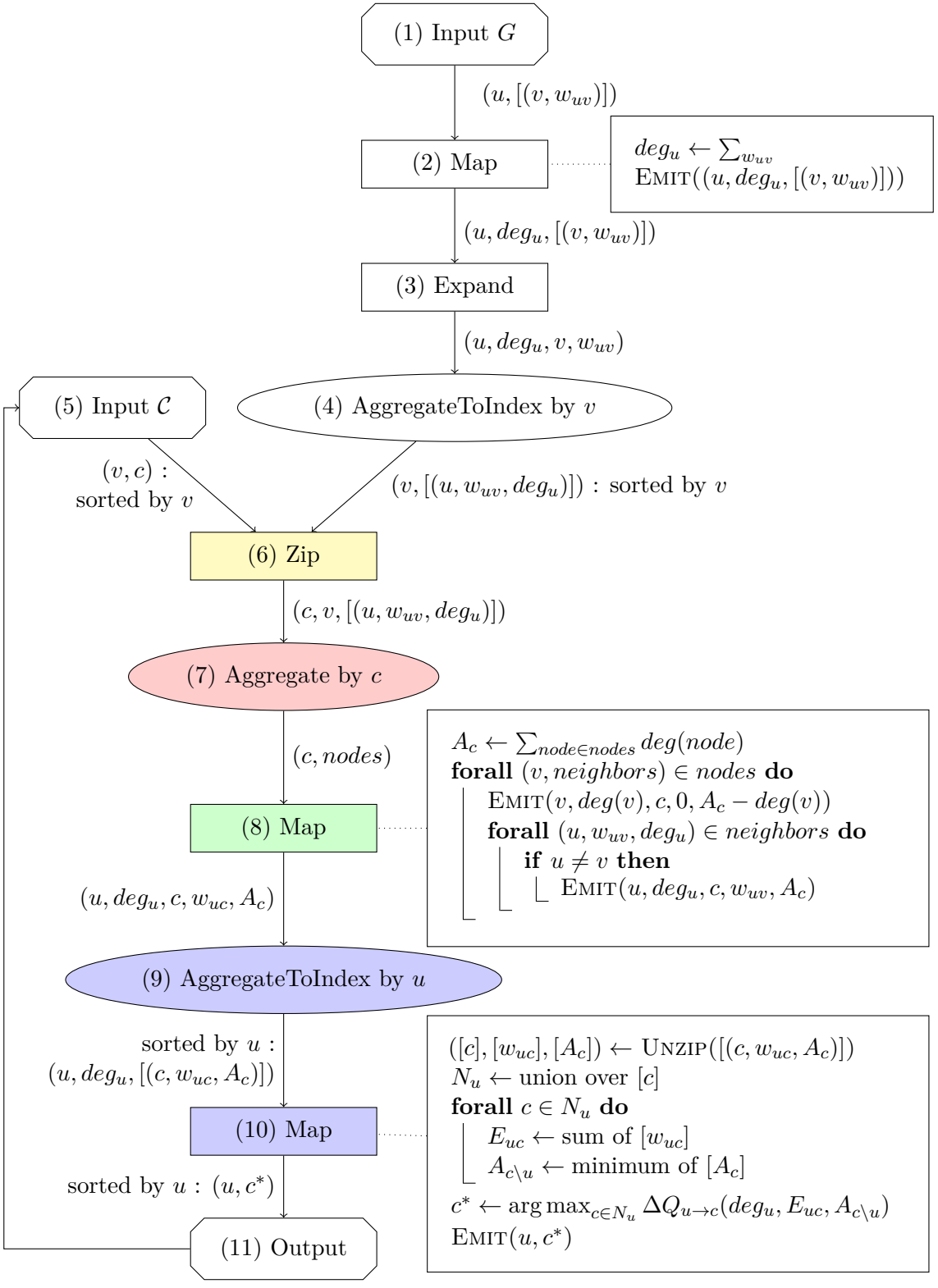


Figure 3.5: Synchronous Local Moving

We use a hash of the node id and the iteration counter to select a pseudo random subset of nodes to be possibly moved in each iteration. This selection is done in (8) when iterating over the adjacent nodes. Rather than emitting the cluster information for all neighbors, only the ones included in the current iteration are used.

We implemented two variants to determine the amount of nodes in each iteration. The first possibility is to use a fixed amount of the nodes in each iteration. The alternative is to vary this amount over the course of the local moving phase. In that case, we start with a small amount and, as the clustering stabilizes we include more nodes in each iteration. We can calculate the rate for the next iteration out of the ratio of nodes moved in the previous round. Given that iteration t had a ratio of R_t and x_t nodes were moved into a different cluster, the ratio for the next round can be calculated like this:

$$R_{t+1} = 1 - \frac{x_t}{n * R_t} \tag{3.2}$$

Stopping Criteria

We implement and evaluate two stopping criteria for this algorithm. The first one is to terminate the local moving as soon as the number of nodes being moved between clusters during one iterations falls below a threshold. The threshold is defined in relation to the total number of nodes n.

The alternative is based on the number of clusters. We expect that, despite our countermeasures oscillation is not completely avoidable. Also, due to the resolution limit of modularity, the meaningfulness of the modularity changes for moving individual nodes in a mostly stabilized clustering of a very large graph might be limited. Thus, we implement a stopping criterion which terminates the local moving phase once the decrease of the number of clusters in one iteration falls beyond a certain threshold. This threshold is also defined in relation to n.

In addition to that, we limit the number of iterations. In the case of a dynamic ratio, the limit is 32. When using a fixed ratio R, the maximum number of iterations is $8 \cdot R^{-1}$.

Optimizations

When implementing this algorithm, there are several things one can do to improve the performance further. First, calculating E_{uc} and $A_{c\setminus u}$ (the two lines in the **forall** $c \in N_u$ loop at (10)) can actually be performed in (8). Here, we just aggregated all nodes of each cluster, so we can precompute these values for each node in the adjacency lists. Then, at (9), there will be one incoming element for each neighboring cluster, rather than for each neighboring node.

That allows a second optimization. Rather than aggregating all the information about neighboring clusters and then selecting the best cluster, we can now use Thrills *ReduceToIndex* operation and reduce these elements pairwise as they come in.

There is another optimization, which allows us to omit the target degree annotation in the adjacency list. This is very important since the annotated adjacency list is very costly in terms of memory. Memory consumption is the crucial factor when trying to cluster graphs where the (unannotated) adjacency list in memory representation takes several hundreds of gigabytes of RAM. We can drop this annotation because with the *ReduceToIndex* operation we know on which worker operation (10) will take place for each node. So rather than passing the degree information around with the regular data flow, we distribute it once in the beginning to each nodes' corresponding worker. This optimization is not strictly within the MapReduce model anymore, but a crucial step in making this algorithm efficient.

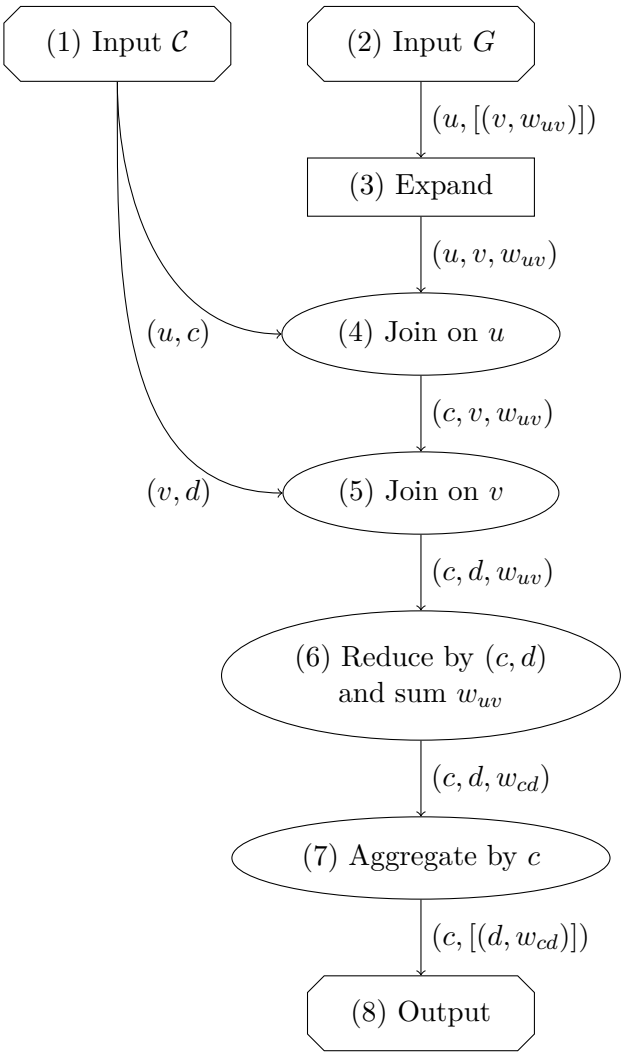


Figure 3.6: Simple Distributed Contraction

3.2.3 Distributed Graph Contraction

In previous work on distributed variants of the Louvain algorithm, the question how to contract a graph which is too big for the RAM of a single worker is left open. In the case of partitioned local moving algorithms, each worker can contract its subgraph independently. But that still leaves out the interconnecting edges which in extreme cases might still be too many for the RAM of a single worker.

We implement this operation in a fully distributed manner. Figure 3.6 outlines a simple version of this algorithm. In operation (3), we expand the given adjacency list into an edge list representation. Then, in operation (4) and (5), we join the clustering by node successively onto both sides of each edge. The algorithm then (6) combines all edges which now share the same source cluster and the same target cluster into one edge with the combined weight. The final step (7) is to aggregate by the source node of each edge which yields an adjacency list representation of the contracted graph. This is the input for the next level of the algorithm.

As mentioned before, joins in Thrill are usually not the very best option. Especially in this case where the joins operate on the edge list representation. This algorithm can be implemented significantly faster by exploiting the fact that in Thrill data can be ordered. Figure 3.7 depicts this improved algorithm.

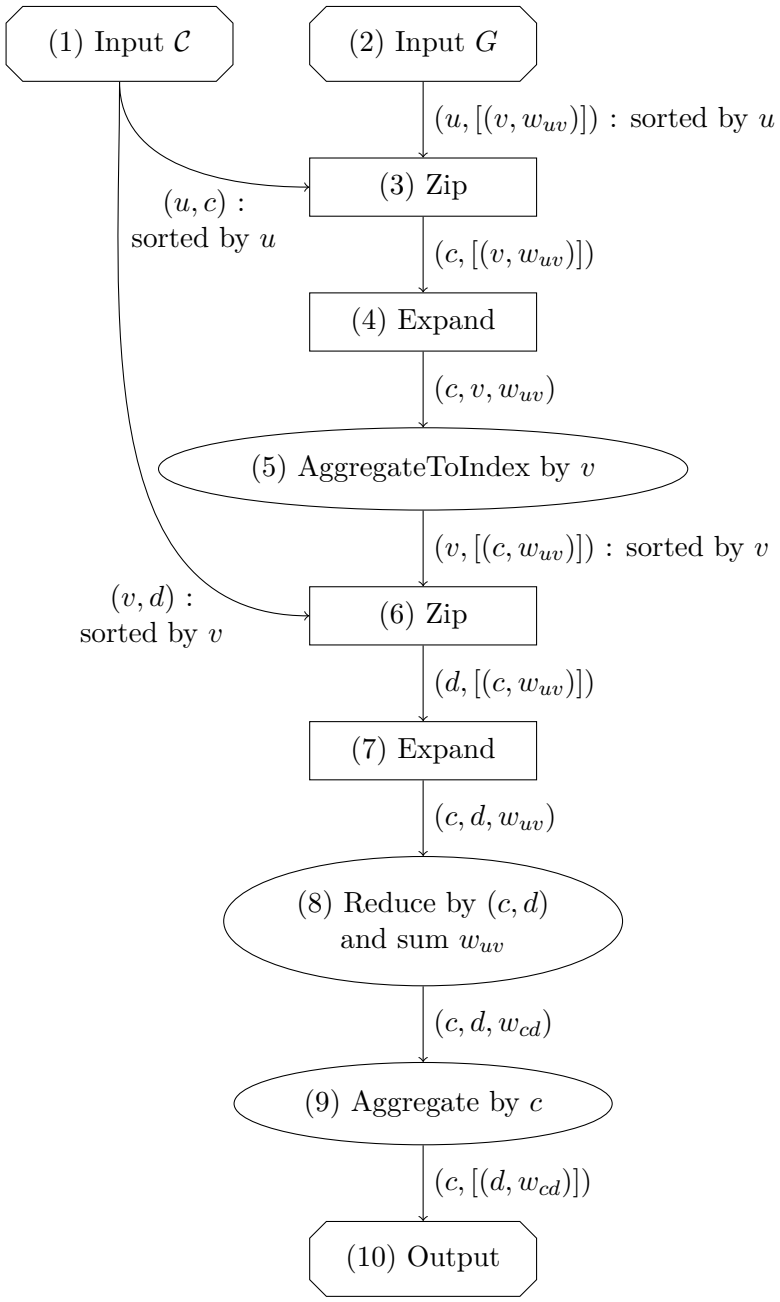


Figure 3.7: Distributed Contraction without Joins

By having the adjacency list and the clustering representation both ordered by node ids, we can combine them without any network communication. That allows us to immediately translate the source node of each edge in a very efficient way. This is done in operation (3). We then expand (4) the graph and immediately aggregate by the edge target (5). This yields the inverted adjacency list. By zipping again with the clustering (6), we can now translate the target node of each edge. After expanding again (4), we can reduce (8) and sum up the weights between the clusters. With the final aggregate (9), we get the contracted adjacency list.

This algorithm is conceptually very similar to the one actually implemented in our code base but it still makes some non-trivial assumption on the input. Most notably on the clustering: for Thrill's *ReduceToIndex* operation to work properly, it expects that the id space ranges from 0 to n without any holes or other irregularities. And our local moving algorithms rely heavily on this ordering feature of *ReduceToIndex*. Since it is very likely

that the ids of the clusters returned from the local moving are only a very small and non-consecutive subset of the id range, the output of our local moving algorithms is not yet suited for building the meta graph.

We would first have to assign new ids from the range $[0, |\mathcal{C}|)$ to our clusters. This could be done in a separate step by taking the clustering, extracting the unique cluster ids, generating a cluster id mapping to new indices and then translating the clustering to the fixed id range. But actually it is also possible to integrate that step into the graph contraction.

Figure 3.8 shows the algorithm actually used in our implementation. After zipping (3) the nodes with their cluster, we aggregate all nodes of each cluster (4). Now each cluster is represented as one item in our collection. That allows us, to use Thrill's *ZipWithIndex* operation (6) to assign a new id to each cluster.

We can now expand this into both the half translated edge list (9) and the clustering DIA with cleaned up ids. After sorting (8) the cleaned clustering and aggregating by edge target (10), we can now translate the second half of our representation (11). This allows us, to then proceed as in the previous algorithm.

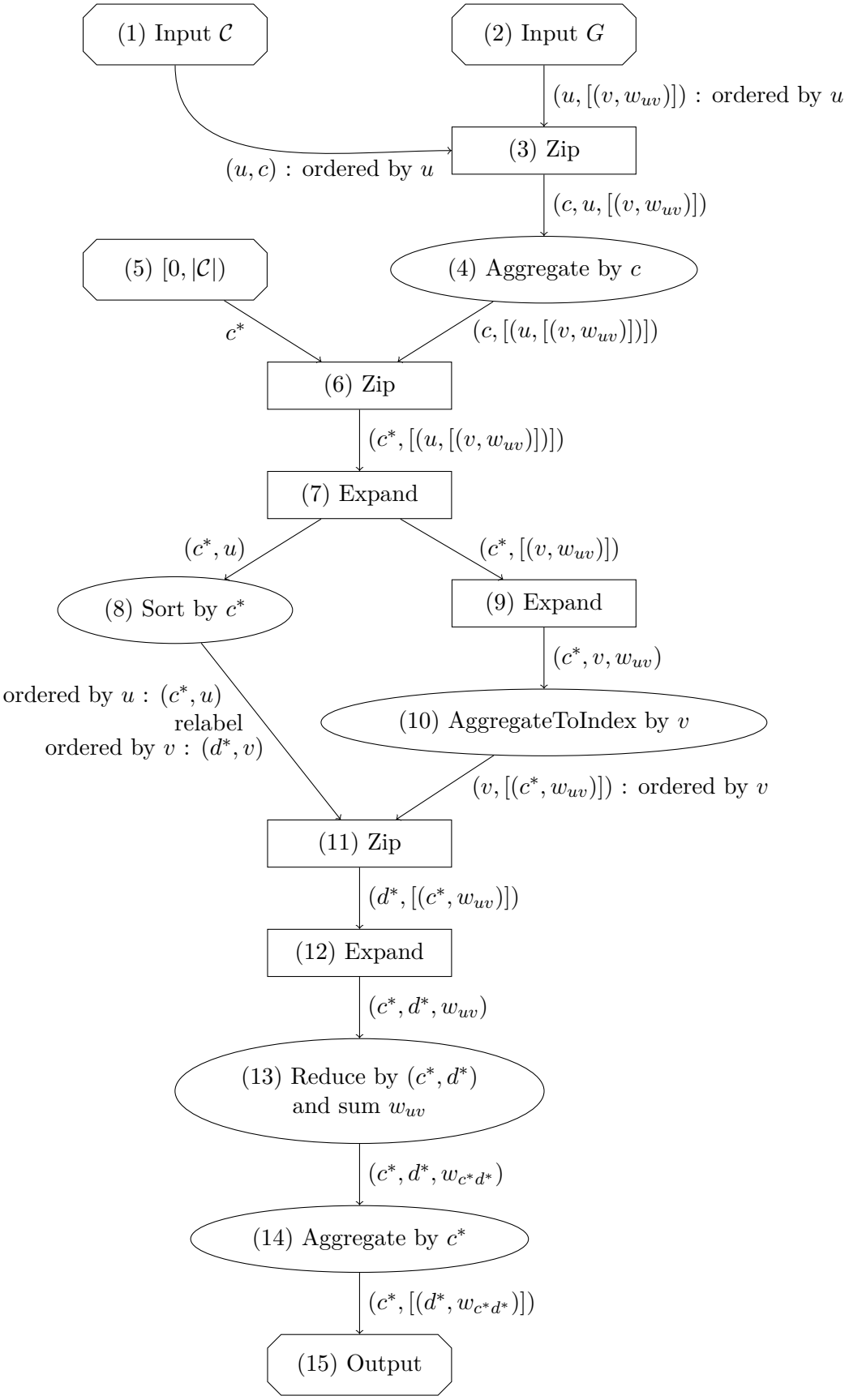


Figure 3.8: Distributed Contraction With Consecutive Output Id Range

4. Evaluation

In this chapter, we present the evaluation of our algorithms. Section 4.1 gives an overview about the environment our experiments were performed in and the graphs used. Section 4.2 contains the actual experiments.

4.1 Setup

4.1.1 Implementation

We implement our algorithms in C++. The distributed algorithms were built using the Thrill framework. We also implement the sequential Louvain algorithm for comparison. The code will be published on Github[1] after the submission of this thesis.

Thrill requires a modern C++ standard, at least C++14. Our code was compiled with GCC 5.2. Network communication in Thrill is performed with MPI. We use OpenMPI 2.0 for this.

To implement our custom operations, we created our own fork of Thrill[2]. Several of our changes to Thrill were merged back into the main repository, and for others there are, at the time of writing, open pull requests.

As Thrill is an experimental framework and still in active development, we encountered some stability issues. Specifically, we had several issues with Thrills memory management. We encountered some of these issues also in our final experiments. Thrill has external memory functionality to swap out data if the available RAM is not sufficient. However, these mechanisms seem not to work well in combination with the memory provisioning of the cluster we performed our experiments on. We had several crashes where Thrill allocated more memory than the resource management of the cluster allowed.

4.1.2 Test Machines

All performance relevant experiments were conducted on bwUniCluster[3][4]. Our experiments use between 1 and 32 28-way Intel Xeon compute nodes. Each of these nodes contains two Fourteen-core Intel Xeon processors E5-2660 v4 (Broadwell). The processor frequency

[1] `https://github.com/SDEagle/ma_code`
[2] `https://github.com/SDEagle/thrill`
[3] `http://www.scc.kit.edu/dienste/bwUniCluster.php`
[4] `https://www.bwhpc-c5.de/wiki/index.php/BwUniCluster_Hardware_and_Architecture`

instance	n	m	source
com-amazon	334 863	925 872	SNAP [YL15]
com-youtube	1 134 890	2 987 624	
com-lj	3 997 962	34 681 189	
com-orkut	3 072 441	117 185 083	
com-friendster	65 608 366	1 806 067 135	
in-2004	1 382 867	13 591 473	DIMACS 10 [BMS+13, -di11, BCSV04]
uk-2002	18 483 186	261 787 258	
uk-2007-05	105 153 952	3 301 876 564	
LFR 100K	100 000	13 215 438	generated
LFR 1M	1 000 000	132 018 381	
LFR 10M	10 000 000	1 320 354 525	
LFR 100M	100 000 000	13 203 573 605	
LFR 200M	200 000 000	26 406 659 441	
hypercube	8 388 608	96 468 992	

Table 4.1: Test instances

is 2.0 GHz with normal workloads, and can clock up 3.2 GHz when using only one or two cores per socket. Each node has 128 GB of main memory and a local 480 GB SSD. The nodes are connected with InfiniBand 4X FDR interconnect.

4.1.3 Graphs

We evaluate our algorithms on both real world instances and artificially generated benchmark graphs. The real world networks are taken from SNAP [YL15] and the 10th DIMACS implementation challenge [BMS+13, -di11, BCSV04]. We focus on web graphs and social networks. Our artificial benchmark graphs are generated using the LFR model [LFR08].

As our algorithms are intended to be used in a distributed setting, we need graphs for which this setting actually makes sense. Since the traditional Louvain algorithms is given enough time and RAM well capable of clustering graphs with a billion edges, we need graphs too big for the RAM of a single host. To generate graphs in that order of magnitude, we use the external memory LFR graph generator introduced in [HMPW17]. Table 4.1 shows an overview of the graphs used in our experiments. We also include a 23 dimensional hypercube graph to evaluate the performance of our algorithms on graphs they were not designed to handle.

LFR Graph Generation Parameters

We choose the parameters for the LFR graph generation by trying to mimic the properties of a social network. We use the properties reported in [UKBM11] as an orientation. The node degree distribution is a power-law-distribution with $\gamma = -2$. The degrees range from 50 to 10 000. This results in an average degree of around 264. Community sizes are also taken from a power-law-distribution. The minimum size is also 50, the maximum size is 12 000. The power-law-exponent β for the community sizes is -1. The mixing parameter μ is 0.4. This means that for each node 40 % of the edges will lead to nodes from a different cluster. These parameters remain the same through all graph sizes.

For our quality experiments (see Section 4.2.6), we generate an additional set of LFR benchmark graphs with a fixed number of nodes but varying generation parameters. For this, we use the LFR graph generator from the NetworKit toolkit [SSM16].

Preprocessing

Before the actual experiments, we preprocess all graphs. Preprocessing includes removing degree zero nodes and making the id space consecutive. We also randomize the node order to ensure that our algorithms are independent of a certain input order. Another reason for the node order randomization is load balancing. In our algorithms the nodes are sometimes distributed among the workers based on id ranges. If one range had significantly more edges than the others this would degrade the overall performance[5]. The preprocessed graph is emitted in a binary format, split into several files of fixed size.

4.1.4 Methodology

When possible, we report averaged results over several runs of the same experiment. Due to limited time and resources on bwUniCluster this was unfortunately not possible for all experiments. The experiments from Section 4.2.1 and 4.2.2 are averaged – the rest were run only once.

Our running times always exclude the time necessary to read the graph from the file system and to write back any results. When an algorithm is run in parallel across multiple hosts, we always take the time from when the first host starts the operation until when the last host finishes it.

Clustering Comparison

To quantify the similarity of two clusterings, we use two well-known measures: The *Normalized Mutual Information* (NMI) and the *Adjusted Rand Index* (ARI). For a precise formal definition of these values see Section 3.2.2 and 5.2 in [WW07]. We use the Networkit implementation for the actual calculation of these measurements [SSM16].

The NMI is most sensitive to changes where nodes are put in a different cluster. Changes where entire clusters are split in parts or merged together are reflected less intensively. The ARI is more sensitive to this kind of changes.

4.2 Experiments

This section contains our experimental results. We start with two preliminary experiments regarding the oscillation countermeasures and the effectiveness of different partitioning strategies for the DLPLM algorithm. We continue with several experiments on the scalability and clustering quality of our distributed algorithms. We conclude this chapter with a comparative overview of running times and clustering results of the two distributed algorithms and the sequential Louvain on all our test graphs.

4.2.1 Oscillation Countermeasures

This section contains the experiments on the performance of the oscillation countermeasures introduced in Section 3.2.2. We evaluate the different strategies in terms of running time and quality of the clusterings measured by their modularity score. We use three different types of graphs, one web graph, one social network and one of our LFR graphs. The experiments were executed on 4 hosts with 16 worker threads each. Table 4.2 contains the results.

The configurations where all nodes were moved concurrently are very fast but the clusterings they find are of very bad quality. So in some cases, oscillation does not prevent convergence,

[5] Our LFR Graphs are a particular extreme example for this. The generator emits the nodes sorted by their degree.

	stopping crit.	cluster count					moved count				
	ratio	1	1/2	1/4	1/8	dynamic	1	1/2	1/4	1/8	dynamic
graph											
LFR 10M	modularity	0.587117	0.595163	0.588008	**0.598188**	0.574811	0.593185	0.596367	0.596907	0.587475	0.584092
	running time [s]	**145.0**	198.4	283.8	912.7	204.4	202.0	**174.2**	190.0	202.8	220.9
friendster	modularity	0.505257	0.574823	0.577839	0.591440	0.607118	x	0.595428	0.611349	0.613112	**0.624124**
	running time [s]	**845.8**	**948.1**	1319.6	2920.2	1413.7	x	1967.538978	2620.5	2298.9	2741.2
uk-2002	modularity	0.348436	0.988985	0.989275	0.989550	0.989080	x	0.989374	0.989609	**0.989660**	0.989579
	running time [s]	85.2	124.2	159.0	450.1	**71.0**	x	149.557490	132.5	114.8	123.7

Table 4.2: Performance and quality of different oscillation countermeasures. The configurations marked with x did not converge and crashed at some point with a memory management related failure.

order	size k / algorithm	4	16	64	256	1024
original	chunk	0.989281	0.989132	0.989080	0.989069	0.989076
	stream	0.989288	0.989179	0.989247	0.989213	0.989205
shuffled	chunk	0.989548	0.989467	0.989673	0.989710	0.989733
	stream	0.989644	0.989706	0.989725	0.989698	0.989591
	LP	0.989740	0.989739	0.989777	0.989773	0.989770

Table 4.3: Modularity results for partitioning with chunking, streaming graph partitioning and label propagation (LP) into different partition sizes k on the original and shuffled uk-2002 graph.

but rather leads to convergence on bad clusterings. Moving all nodes concurrently in combination with the stopping criterion based on the number of nodes moved, works surprisingly well on the LFR 10M graph, but on the other graphs the algorithm does not converge at all and eventually crashes.

The data shows no configuration that performs best for all graphs. Apparently, for different graphs different strategies perform best. We settle for the constant 1/4 ratio in combination with the moved stopping criterion and use this configuration for all following experiments. This is one of the few configurations delivering stable quality and running time on all three graph types. It also offers a reasonable trade-off between running time and clustering quality. We also apply this strategy to our label propagation algorithm.

4.2.2 Effectiveness of Partitioned Local Moving

In this experiment we investigate the effectiveness of the partitioning approaches used in [ZY15, CHLG13]. We compare the chunking approach on both the original graph and our preprocessed graph with shuffled node order with the results of our label propagation approach. As label propagation is independent of the input order, we run it only on the shuffled graph. We also include a streaming graph partitioning technique from [SK12] we evaluated in early development.

For the streaming approach we use a strategy referred to as *deterministic greedy with linear penalty*. As nodes arrive, they are immediately assigned to the partition they have the most edges to. A linear penalty function is applied to the weight, based on the remaining capacity of the partition. In the evaluation from [SK12], this was the strategy performing best.

We perform our experiment on the uk-2002 instance. The algorithm used is a modified version of our sequential Louvain implementation where on the first level the local moving is performed in partitions. After the first contraction, the regular Louvain algorithm is applied. This is a sequential implementation of the algorithm proposed in [ZY15].

Table 4.3 shows the modularity results. Modularity generally only varies very slightly across all strategies and sizes. Label propagation delivers the best modularity results across all sizes. The streaming approach is, in most cases, superior to the chunking. There is no obvious relation between partition size and modularity score. The shuffling improves the modularity scores for both the chunking and the streaming approach. Overall, the partitioning strategy and size have surprisingly little influence on the modularity score.

These observations stand in contrast to the quality of the partitionings measured by their total cut, as in shown in Table 4.4 As expected, the cut sizes grow with the number of partition elements. Chunking the id range on the original graph delivers the best cuts.

order	size k / algorithm	4	16	64	256	1024
original	chunk	9 404 346	11 754 668	12 642 552	13 792 340	19 288 568
	stream	53 014 526	68 323 728	73 633 832	78 585 883	85 501 755
shuffled	chunk	198 339 260	246 944 129	258 509 423	261 113 135	261 578 869
	stream	25 095 327	31 906 359	34 145 074	35 090 278	43 401 203
	LP	17 376 628	23 864 982	25 929 781	26 655 192	26 960 313

Table 4.4: Total cut sizes for partitioning with chunking, streaming graph partitioning and label propagation (LP) into different partition sizes k on the original and shuffled uk-2002 graph.

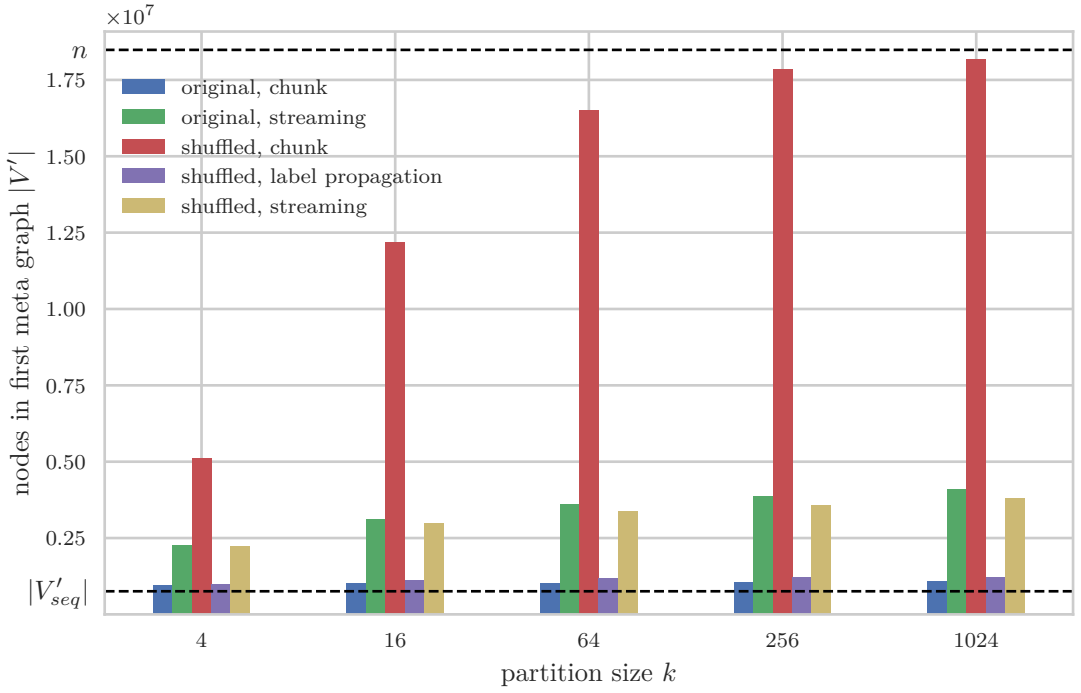


Figure 4.1: Size of the first meta graph with different partition strategies and sizes.

Chunking the id range on the shuffled graph yields extremely heavy cuts, though. The total weight of the uk-2002 graph is 261 787 258. With four partition elements, the cut includes already three quarters of the edges.

In contrast, the streaming approach benefits from the randomization. On the randomized graph, it performs much better than the chunking, but not as good as label propagation. Label propagation generally delivers relatively good results. It also delivers very stable total cut weights, even for partitions with a large k. This poses the question why the modularity scores do not reflect these big differences.

The reason for this is that for bad partitionings the partitioned local moving will not generate a bad clustering, but rather no clustering at all. Figure 4.1 illustrates this behavior. It shows the number of nodes in the meta graph after the partitioned local moving. The dashed lines indicate the size of the meta graph after sequential local moving and the original graph size. For partitions with $k > 64$ created by chunking on the shuffled graph, the meta graph stays around the size of the original graph. That means the partitioned local moving did not do anything.

This also explains why the modularity scores stay roughly the same. If the partitioning has a bad quality, nothing will happen in the partitioned phase, and afterwards the sequential algorithm will start with a graph very similar to the original graph. That leads to the conclusion that a good partitioning is essential for the partitioned local moving algorithm. Otherwise, the parallelization will be completely ineffective.

Label propagation is almost equally effective to chunking on the original graph. Since a good input ordering is not always available, we choose label propagation as the partitioning strategy for our DLPLM algorithm. In the next section we evaluate its performance.

4.2.3 Label Propagation Scalability

We apply the label propagation partitioning algorithm to three of our LFR graphs – with 1, 10, and 100 million nodes – and compare running time, speedup, and efficiency of different threading configurations. Figure 4.2 depicts the results.

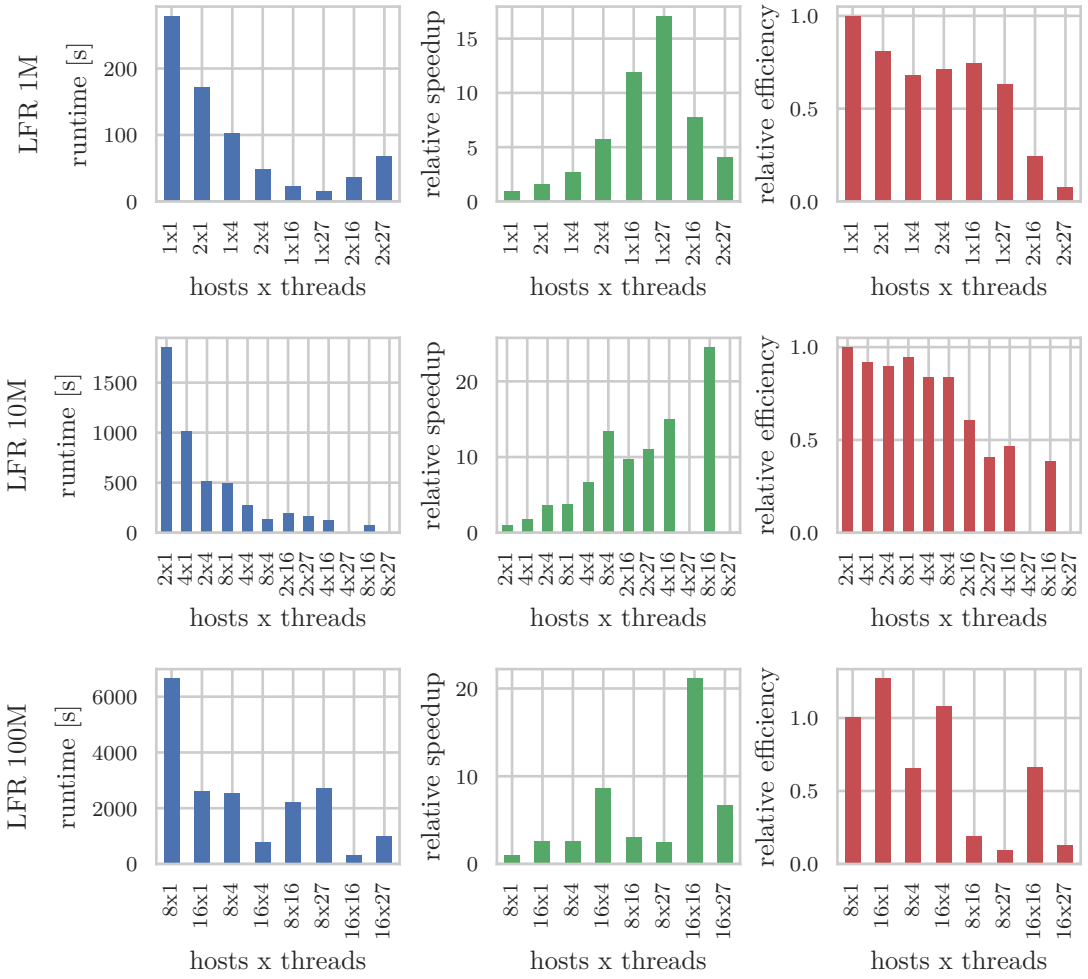


Figure 4.2: Label propagation scaling for three LFR graphs with different host and thread per host configurations. The configurations are ordered by the total thread count. Speedup and efficiency are calculated in relation to the configuration with the fewest total threads.

For brevity, we reference configurations in a short form. For example, the configuration 16x4 means that the algorithm is run on 16 hosts with each 4 worker threads. We use different numbers of hosts depending on the graph size since for example the LFR 100M graph requires more RAM than a single worker has. For each host count we run the

algorithm with 1, 4, 16 and 27 worker threads. 27 threads is the maximum number, since we have 28 cores and Thrill requires one extra worker for MPI communication.

The algorithm exhibits almost linear scaling in the input size. With the 2x1 configuration the LFR 1M graphs needs around 180 seconds and the LFR 10M 1800 seconds. With the 8x1 configuration the LFR 10M graphs needs around 500 seconds and the LFR 10M 6500 seconds.

Regarding scaling with the number of threads, the results depend on both the graph size and the specific configurations. For the LFR 1M graph with 32 or more total threads, the performance is actually worse than with fewer threads. So there is a point where adding more threads adds more communication overhead than gained by parallelization. For larger graphs, for example the LFR 10M graph, we can use more threads and still gain improvements.

The efficiency plots show, that even though large numbers of threads can yield significant speedups, using more hosts with fewer threads is more efficient. Using too many threads per host can even degrade the performance. When using the maximum number of threads per host on the LFR 10M graph, we even get memory management related crashes. In all other cases, except with exactly one host, the configurations with 16 threads per host deliver better running times than those with 27. This indicates that RAM access during local operations might actually be the bottleneck of this algorithm.

On the LFR 100M graph, both the 16x1 and the 16x4 configurations have super-scalar speedups over 8x1. Also, 16x1 performs as good as 8x4 which actually has more total threads. 8x16 still yields a slight improvement over 8x4. But 16x4 is much better even though it uses only half the number of threads as 8x16. We conclude that for this setup the RAM usage of the local operations is the greater bottleneck than the network communication for the distributed operations.

4.2.4 DLPLM Scalability

In this section, we evaluate the performance of the DLPLM clustering algorithm. We expect the algorithm to exhibit similar scaling properties as label propagation. DLPLM primarily adds the partitioned local moving phase, which can be run completely in parallel without any communication between the partitions.

For this algorithm, the number of worker threads defines the size of the partitioning and thus also influences the quality of the final clustering. To investigate this, we also report the influence of the number of threads on the modularity. We also use different graphs to investigate if the graph properties have implications for the algorithm's performance. We cluster each graph on 4, 8, and 16 hosts. Due to its bad performance in the previous experiment, we do not use configurations with 27 threads per host. Figure 4.3 shows the scaling results.

The graph structure has a strong impact on performance, much stronger than expected by just scaling the running time with the number of edges. Clustering com-friendster with 4x1 took more than 5 hours. Scaling with the number of threads performs analogue to what we have seen for label propagation. Comparing the 16x4 and the 4x16 configurations shows that more hosts lead to greater improvements than more threads per host. Clustering the uk-2002 graph with more than 4 threads per host does not improve the performance much further.

The efficiency plots show clearly that adding hosts is much more effective, than adding threads per host. Scaling just the number of hosts with one thread per host yields almost perfect linear scaling. In some cases, we even get super-scalar speedups. This leads to the

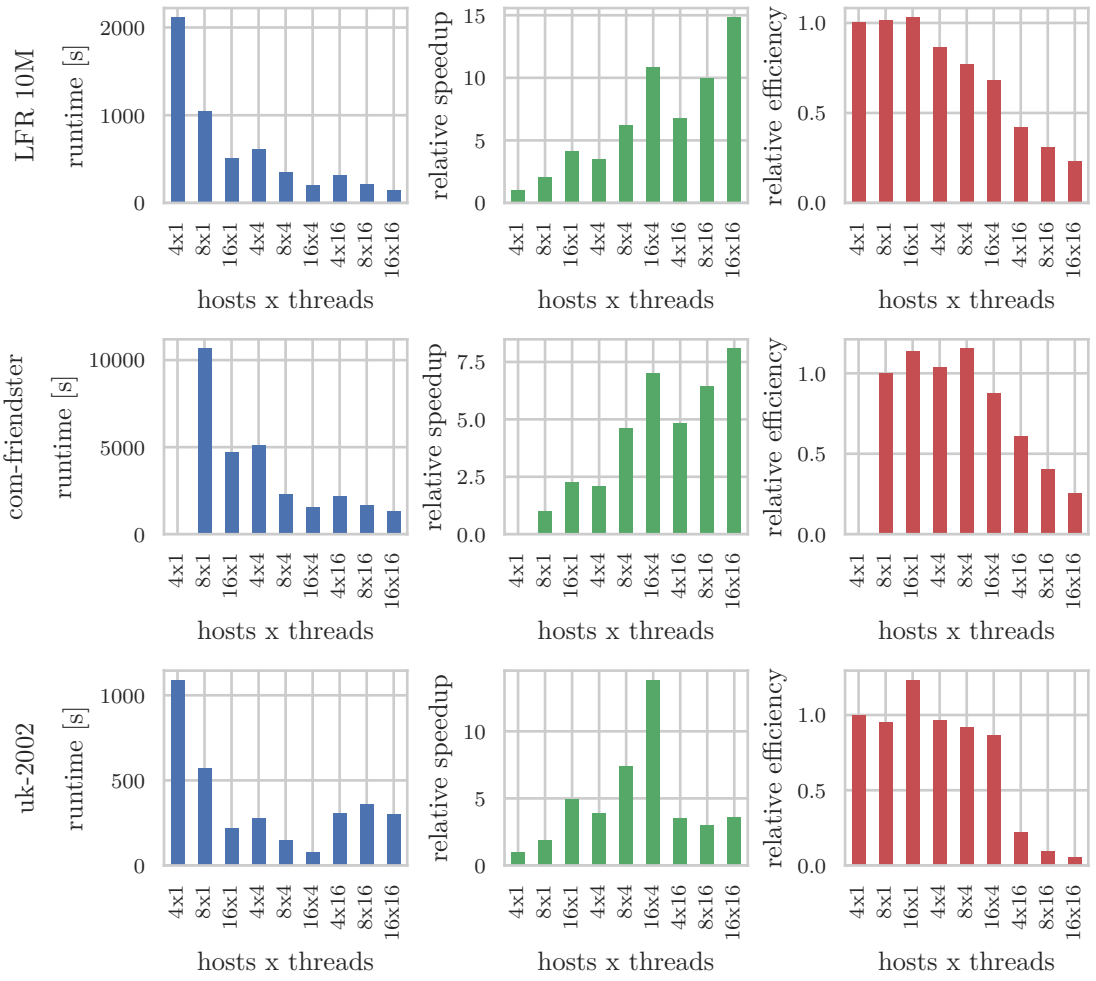


Figure 4.3: DLPLM scaling for three different types of graphs with different host and thread per host configurations. The configurations are ordered by the total thread count. Speedup and efficiency are calculated in relation to the configuration with the fewest total threads.

conclusion that this algorithm can scale linearly with the number of hosts. Adding more threads can improve the performance further, but not as efficiently as adding hosts.

Figure 4.4 depicts the modularity results for different total thread numbers and thus, partition sizes. For uk-2002 and LFR 10M, the modularity remains roughly constant through all configurations. With a growing number of total threads, the modularity scores improve slightly. For com-friendster, this effect is much more significant. The reasons for this are unclear. Nonetheless, we conclude that adding more threads does not worsen the clustering quality.

Resource Usage

Figure 4.6 shows an example of resources used during a DLPLM run. The graph in this case is LFR 10M. The charts show the resource usage of one of eight hosts.

During the initial label propagation, the CPU utilization is relatively good. The local minima in the beginning of each iteration correspond to the network communication performed at these points. At the end of the label propagation phase, the already high RAM usage (the plain graph as an adjacency list should take about 8 GB in total) increases

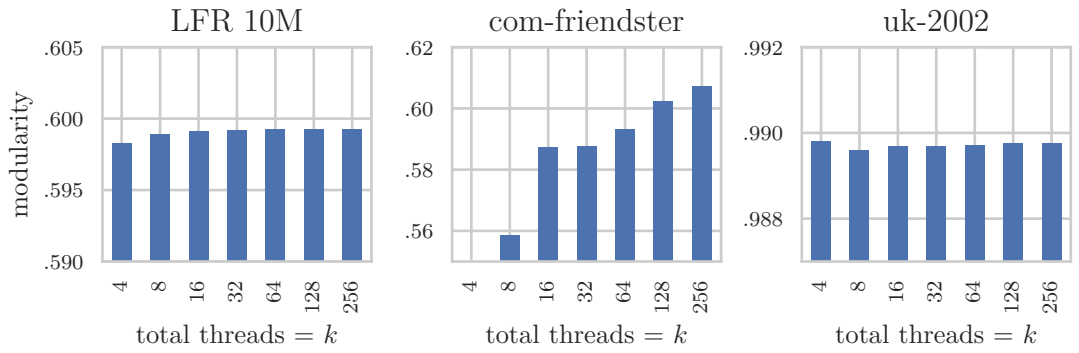


Figure 4.4: DLPLM modularity results for different total thread counts and thus partition sizes k.

significantly. This is where the graph representation containing the target degrees of each edge is built.

In the actual partitioned local moving, the CPU utilization is very good in the beginning, but decreases later on. That indicates that a better load balancing by the number of edges in each partition could improve the performance of this algorithm further. After the contraction, the higher levels of the algorithm take place with relatively low resource utilization. This also indicates potential for further improvements. The resource usage also confirms that in this configuration the network is not a limiting factor.

4.2.5 DLSLM Scalability

This section contains the scaling experiments for the DLSLM algorithm. Since the data flow is similar to label propagation, the expectation is that the scaling behavior will also be similar. We use the same graphs and configurations as for the label propagation experiment. Nonetheless, the results are not completely comparable to the results from Section 4.2.3 since the DLSLM algorithm also contains the contraction and recursion on the contracted graph. Figure 4.5 depicts the results.

The results confirm the similarity of the two algorithms. Despite the higher complexity of the modularity difference calculation and the additional contraction and recursion, the running times are very similar. The algorithm also scales slightly super-linear with both the number of threads and the graph size. Clustering LFR 1M with 2x1 takes around 160 seconds and clustering LFR 10M with 2x1 takes approximately 1900 seconds. Clustering LFR 10M with 8x1 takes 500 seconds and clustering LFR 100M with 8x1 takes around 7000 seconds.

Configurations with 27 threads per host also perform bad or lead to memory management related crashes. Configurations with 16 threads achieve significant speedups but are relatively inefficient. On the LFR 100M graph, 16x1 achieves a super-scalar speedup over 8x1 and speedups are almost perfectly linear with up to 4 threads per hosts. Once again, this leads to the conclusion that the RAM usage of local operations is the bottleneck of this algorithm and not the network usage of distributed operations.

When comparing these numbers to the DLPLM algorithm, we note that the DLSLM algorithm does not scale as good, but performs better in absolute running times. It actually takes only slightly more time to perform the entire clustering than the label propagation takes in DLPLM. This leads to the question if the DLSLM algorithm could also be used to build a graph partition, and what quality this partition would exhibit.

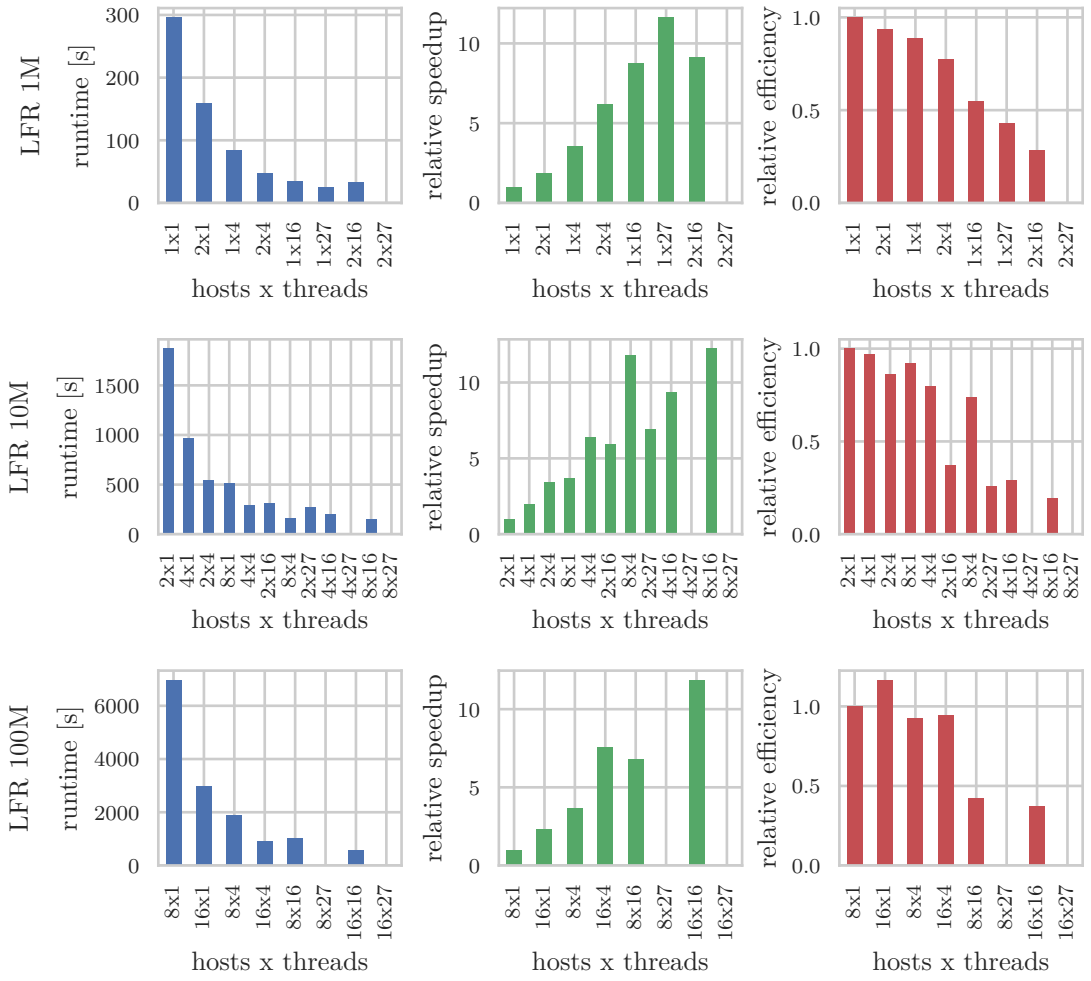


Figure 4.5: DLSLM scaling for three different types of graphs with different host and thread per host configurations. The configurations are ordered by the total thread count. Speedup and efficiency are calculated in relation to the configuration with the fewest total threads.

Resource Usage

Figure 4.7 shows an exemplary resource usage of the DLSLM algorithm. The graph and configuration are the same as in Figure 4.6 – LFR 10M on eight hosts.

The DLSLM algorithms uses less RAM and less CPU than the DLPLM algorithm. Peak RAM usage is almost a third less. The reason for this is that DLSLM can avoid the expansive graph representation and stick to a simple adjacency list.

During the synchronous local moving, network utilization is higher than during label propagation, as more data needs to be communicated to calculate the modularity differences. Interestingly, the CPU utilization during synchronous local moving is worse than during label propagation, even though more computations need to be performed. Also, the lows in utilization do not correspond to peaks in network communication. We follow that in these periods the workers are either limited by RAM, or they are waiting on other workers to finish their work. In the latter case, the times could be optimized with better load balancing.

The later levels of the algorithm show low utilization of both CPU and the network. We conclude that switching to the sequential Louvain algorithm once the graph has shrank

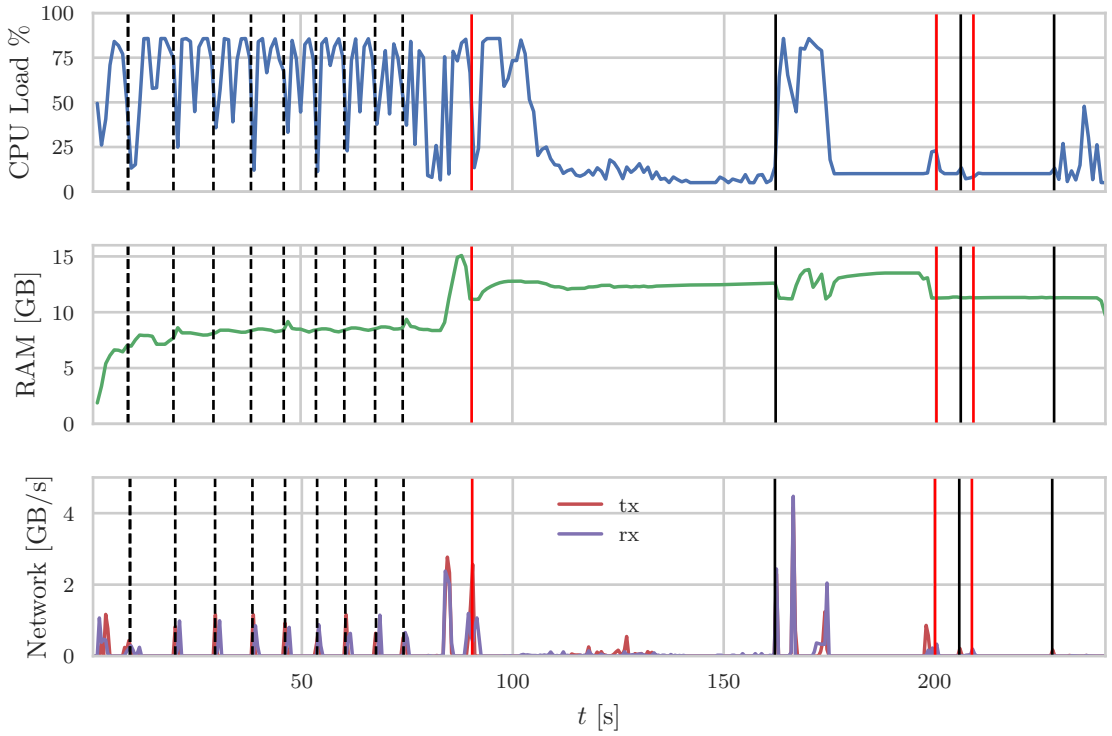


Figure 4.6: Resource usage on one of eight hosts performing DLPLM on LFR 10M. Dashed lines indicate the start of a label propagation iteration, but only on the first level. Red lines mark the start of the partitioned local moving. The continuous black lines indicate the contraction phase.

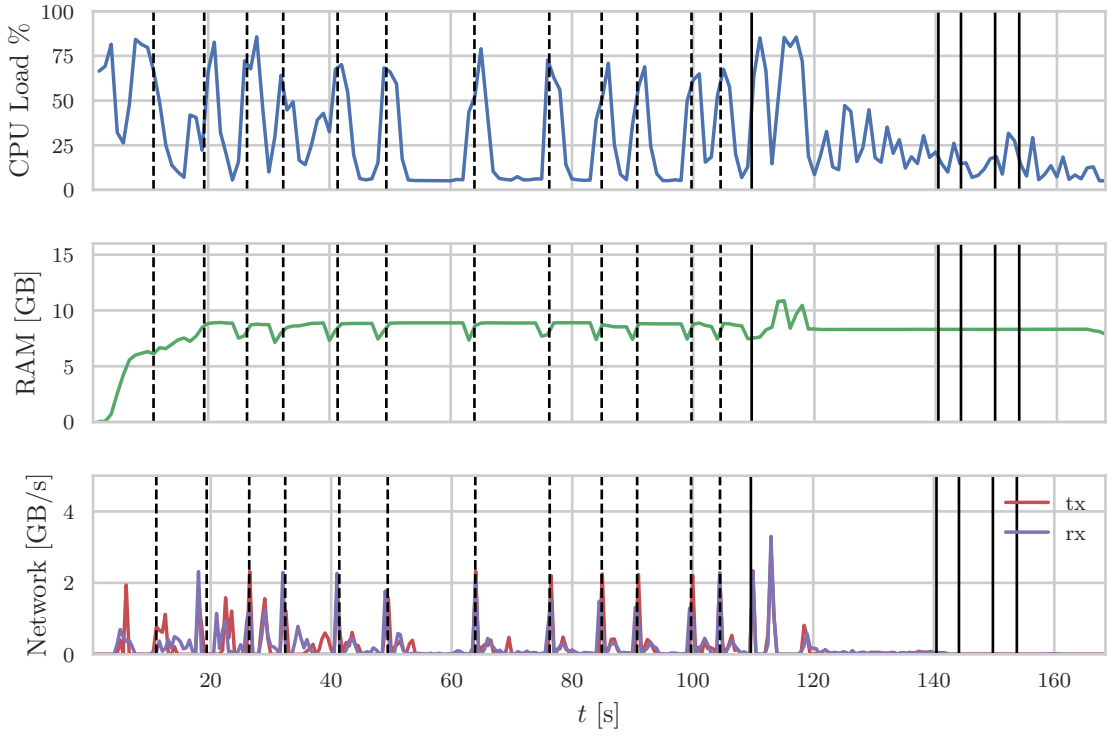


Figure 4.7: Resource usage on one of eight hosts performing DLSLM on LFR 10M. Dashed lines indicate the start of a synchronous local moving iteration, but only on the first level. The continuous lines indicate the contraction phase.

to a certain size would be an optimization worth exploring. Concluding, this analysis demonstrates that the DLSLM algorithm is more efficient than DLPLM, both in terms of time and space. In the next section, we will investigate how the algorithms compete in terms of the quality of the clusterings found by them.

4.2.6 Quality

In this section we evaluate the quality of the clusterings found by our algorithms. We conduct two experiments. First, we evaluate the similarity of our clusterings to the ground truth of a different set of LFR graphs. These graphs are smaller and cover a greater variance of LFR parameters. Second, we compare our clustering results to the ground truths of the LFR graphs we used throughout the rest of our experiments.

The LFR graphs generated for this set of experiments all have 100 000 nodes. Both the node degree distribution and the cluster size distribution are power-law-distributions. We use an average of 50 and a maximum of 1000 as parameters for the node degree distribution. For the community size distribution the minimum is 30 and the maximum 1200. Our degree distribution exponent γ is -2 or -3. The cluster size distribution exponent β is -1 or -2. The mixing parameter μ ranges from 0.1 to 0.9. We generate a graph for each combination of these values.

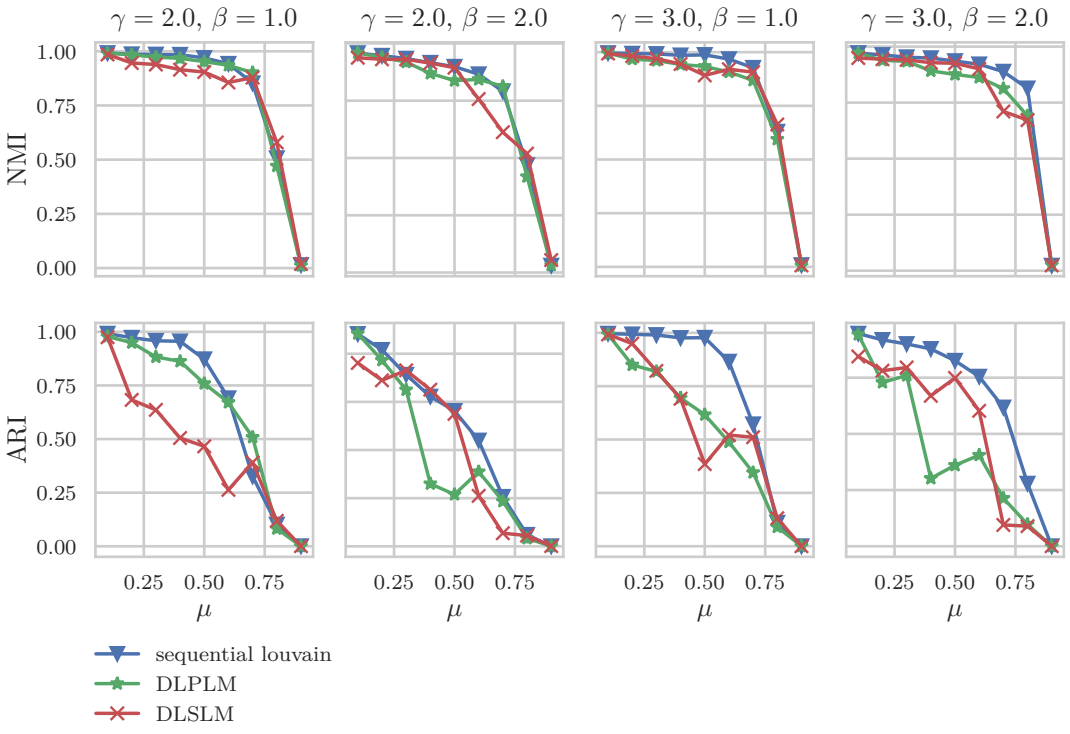


Figure 4.8: Similarity of clustering results to ground truth with varying LFR parameters by algorithm.

Figure 4.8 depicts the comparison results between our algorithms and the ground truth for each graph. In terms of NMI, our algorithms perform as good as or only slightly worse than the sequential Louvain algorithm. Whether DLSLM or DLPLM perform better, depends on the community size and degree distribution exponents.

When measured by ARI, the differences to the ground truth are bigger. This also applies to the sequential Louvain algorithm, but the distributed algorithms perform even worse.

Which distributed algorithm performs better depends on the community size and degree distribution exponents.

The NMI results indicate that up to a μ of 0.6, our algorithms generally put nodes into the correct cluster. The worse ARI results demonstrate that while the nodes may be in the correct cluster, our algorithms often merge several ground truth clusters together. We can see this behavior even better in the next figure.

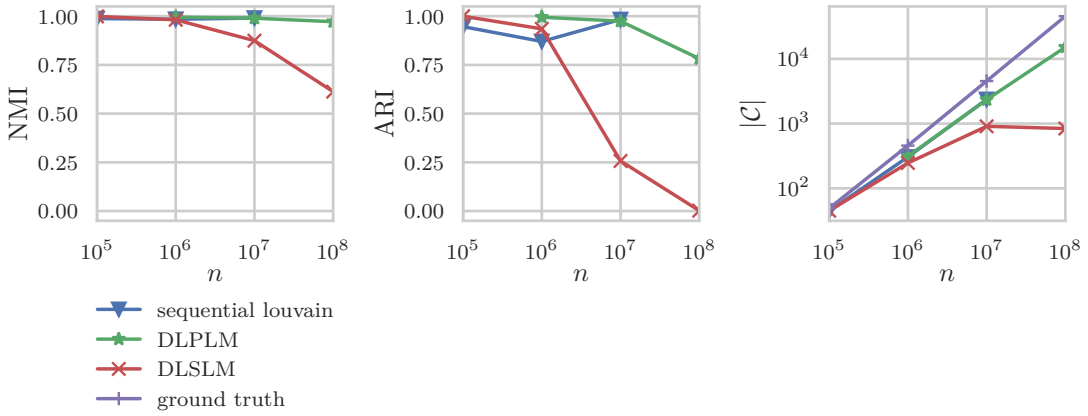


Figure 4.9: Similarity of clustering results to ground truth of our LFR graphs by graph size and algorithm. The third chart shows the number of clusters in the clustering retrieved by each method.

Figure 4.9 shows the comparison results between our algorithms and the ground truth for the LFR graphs used in the other experiments. We also include the number of clusters in the clustering found by each algorithm. With the RAM available, the sequential Louvain algorithm is not able to cluster the LFR 100M and LFR 200M graphs. We do not apply the DLPLM algorithm to graphs with less than a million nodes because the algorithm would switch immediately to the sequential Louvain algorithm.

The DLPLM approach delivers very good similarity across all graph sizes. The only exception is the ARI for the LFR 100M graph. This indicates that for larger graphs the algorithm is prone to merging clusters which do not belong together. But for graphs of this size, this is likely not to be an issue of the schema of the algorithm, but of modularity and its resolution limit.

The DLSLM algorithm depicts a much stronger decline in quality for larger graphs, especially in terms of ARI. In fact, an ARI of zero is also the expected value for a random clustering. We conclude that on very big graphs too many clusters are eliminated during the first rounds or our oscillation countermeasures are not yet performing well enough.

4.2.7 Clustering all Graphs

Finally, we cluster all test instances with all our implemented algorithms. All runs were performed with 16 threads per host. Table 4.5 shows the resulting running times and modularity scores.

We observe that DLSLM is with two exceptions (in-2004, com-friendster) significantly faster than DLPLM. This advantage in running time is complemented by worse modularity scores, again, with two exceptions (com-youtube, uk-2007-05).

Both distributed algorithms introduce a significant overhead. For sufficiently large graphs and enough threads, they are still able to outperform the sequential algorithm. Also, they are able to cluster graphs which the sequential algorithm cannot due to RAM limitations.

RAM is still an issue though, especially for the DLPLM algorithm. Clustering the LFR 200M graph with 32 hosts fails because the RAM is not sufficient. The edge list annotated with degrees is very expensive in terms of memory. The DLSLM algorithm can avoid this representation and is much more memory efficient.

The hypercube graph causes noteworthy behavior. Even though the structure of the graph, by its construction, contains no clustering at all, both the sequential Louvain algorithm and the DLSLM algorithm find one, which is reasonable according to the modularity score. This is another case, demonstrating the limited meaning of absolute modularity values [GdMC10].

The DLPLM algorithm fails to cluster the graph within the time limit of ten hours. Label propagation is unable to generate a meaningful partitioning for this type of graph. In each contraction the graph size shrinks only by a handful of nodes. This process would theoretically converge at some point, but it did not within the time limit.

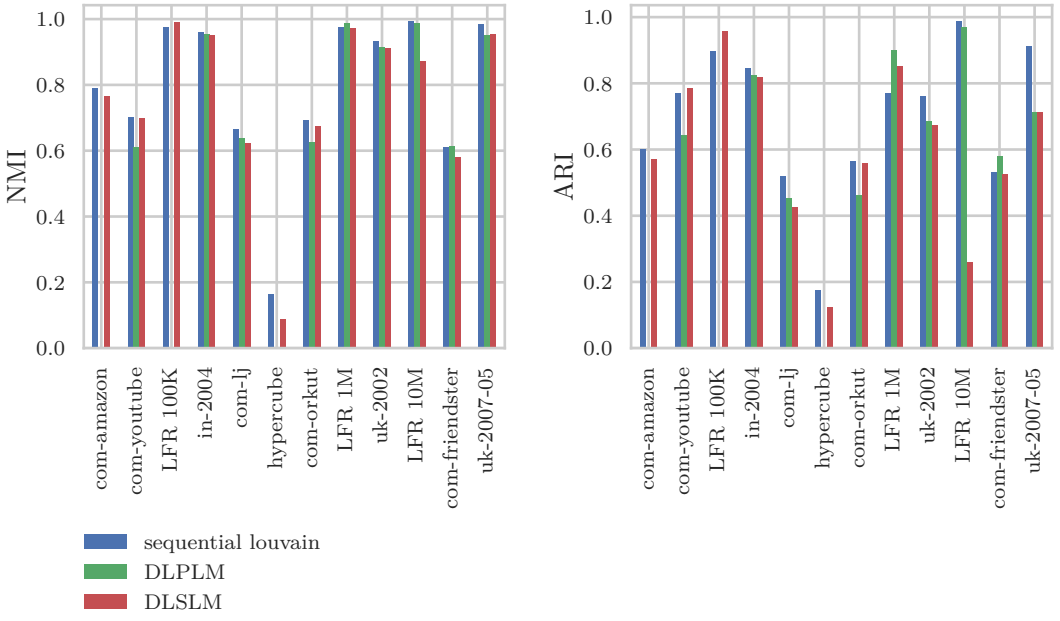


Figure 4.10: Similarity of clustering results to sequential Louvain clustering results for all graphs by algorithm.

We also compare the outputs of the algorithms with the results of a base run of the sequential algorithm. Figure 4.10 depicts the similarity of the results in terms of NMI and ARI. The similarity between the sequential algorithm and the base run indicates how strong two runs of the sequential algorithm with different seeds differ from one other. For the most part, the deviation of the distributed algorithms from the base run is very similar to the deviation between two sequential runs. One strong exception is DLSLM on the LFR 10M graph. On the uk-2007-05 graph, both distributed algorithms tend to merge clusters which the sequential algorithm keeps separated. Also, the hypercube graph causes extremely low similarities. Since it has no inherent clustering structure, this is expected. We conclude that our distributed algorithms produce clustering results with a stability comparable to the original Louvain algorithm. The results also indicate that on real world graphs the decline in clustering quality of the DLSLM algorithm is not as bad as in our LFR graphs.

algorithm	ground truth	sequential louvain			DLPLM		DLSLM	
	modularity	running time [s]	modularity	hosts	running time [s]	modularity	running time [s]	modularity
graph								
com-amazon		1.1	0.925990	2			35.5	0.925443
com-youtube		6.3	0.719774	2	150.6	0.713472	41.2	0.720974
LFR 100K	0.524983	1.5	0.521162	2			9.6	0.524983
in-2004		14.6	0.980155	4	68.6	0.980307	98.2	0.980120
com-lj		102.3	0.748221	8	222.1	0.744079	167.7	0.730967
hypercube		1534.8	0.572928	8			115.4	0.539654
com-orkut		173.7	0.667512	8	217.5	0.664707	132.8	0.657185
LFR 1M	0.592803	29.2	0.591490	4	49.1	0.592719	27.2	0.592534
uk-2002		587.3	0.989749	8	324.4	0.989725	222.0	0.989648
LFR 10M	0.599280	1021.0	0.599284	8	225.2	0.599273	151.7	0.596983
com-friendster		5437.9	0.611236	16	1443.0	0.616655	1597.7	0.612749
uk-2007-05		7295.1	0.996053	16	1185.7	0.996166	496.9	0.996190
LFR 100M	0.599928			16	3084.7	0.599885	640.9	0.564620
LFR 200M	0.599963			32			860.8	0.574363

Table 4.5: Clustering results of all algorithms and graphs in comparison.

5. Summary

We present two distributed clustering algorithms in the MapReduce programming model. The first one, DLPLM, builds and improves on existing distributed clustering algorithms [ZY15, WFSP14] by turning the partitioning phase into an integral part of the algorithm. This makes our algorithm much more generally applicable than its predecessors. The second one, DLSLM, combines the Louvain method and synchronous label propagation into a highly efficient clustering algorithm. It is both faster and requires less RAM than the partitioned approach.

We implement both algorithms using the experimental BigData framework Thrill. Both algorithms are able to cluster graphs with tens of billions of edges in a matter of hours. This demonstrates that Thrill is not only an efficient batch processing tool, but also capable of distributed graph processing.

We present a detailed evaluation of our implementation. We find that both algorithms scale very well. Similar to the original Louvain method, the running time scales almost linearly with the number of edges. Also, given a big enough problem, both algorithms scale linearly with the number of available hosts.

DLPLM offers quality very similar to the original Louvain algorithm. On real world graphs, DLSLM delivers also qualitatively good results. However, evaluating the algorithms performance on our artificial benchmark graphs uncovers some qualitative weaknesses.

Our work demonstrates that distributed graph clustering in MapReduce and Thrill is a feasible approach. Both our algorithms can cluster graphs with several billions of edges in a few hours. With that, they are a valuable tool to apply community detection to more and bigger networks than previously possible.

Future Work

We plan to continue the work on the DLSLM algorithm to further improve the quality of its output. One approach would be to evaluate other clustering quality measures, for example the map equation [RAB09]. Developing more sophisticated and generally applicable oscillation countermeasures is another.

Following our results from Section 4.2.5, we would also like to develop a modularity-based partitioning approach and evaluate, if it can outperform approaches based on label propagation. If so, it would also be an interesting approach to combine the two distributed algorithms and use the synchronous local moving to partition the graph for the partitioned

local moving. Another promising approach to compensate the qualitative downsides of DLSLM would be to apply refinements. One option would be to locally recluster all clusters. This would likely split up clusters which should not have been merged, similar to the Smart Local Moving algorithm [WvE13].

Bibliography

[-di11] 10th DIMACS Implementation Challenge – Graph Partitioning and Graph Clustering, 2011. http://www.cc.gatech.edu/dimacs10/.

[ASS14] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. (semi-) external algorithms for graph partitioning and clustering. In *2015 Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 33–43. SIAM, 2014.

[BAJ+16] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. Thrill: High-performance algorithmic distributed batch data processing with c++. *arXiv preprint arXiv:1608.05634*, 2016.

[BC09] Michael J Barber and John W Clark. Detecting network communities by propagating labels under constraints. *Physical Review E*, 80(2):026129, 2009.

[BCSV04] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software - Practice and Experience*, 34(8):711–726, 2004.

[BDG+08] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Höfer, Zoran Nikoloski, and Dorothea Wagner. On Modularity Clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):172–188, February 2008.

[BGLL08] Vincent Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10), 2008.

[BH15] Seung-Hee Bae and Bill Howe. Gossipmap: A distributed community detection algorithm for billion-edge directed graphs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 27. ACM, 2015.

[BKA+14] Nazar Buzun, Anton Korshunov, Valeriy Avanesov, Ilya Filonenko, Ilya Kozlov, Denis Turdakov, and Hangkyu Kim. Egolp: Fast and distributed community detection in billion-node social networks. In *Data Mining Workshop (ICDMW), 2014 IEEE International Conference on*, pages 533–540. IEEE, 2014.

[BMS+13] David A. Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Schumm, and Dorothea Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*. Springer, 2013. to appear.

[CHLG13] Chun Yew Cheong, Huynh Phung Huynh, David Lo, and Rick Siow Mong Goh. Hierarchical parallel algorithm for modularity-based community detection

using gpus. In *European Conference on Parallel Processing*, pages 775–787. Springer, 2013.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[Eul41] Leonhard Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, 8:128–140, 1741.

[FB07] Santo Fortunato and Marc Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Science of the United States of America*, 104(1):36–41, 2007.

[FH16] Santo Fortunato and Darko Hric. Community detection in networks: A user guide. *Physics Reports*, 659:1–44, 2016.

[For10] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3–5):75–174, 2010.

[GdMC10] Benjamin H. Good, Yves-Alexandre de Montjoye, and Aaron Clauset. Performance of modularity maximization in practical contexts. *Physical Review E*, 81(046106), 2010.

[HMPW17] Michael Hamann, Ulrich Meyer, Manuel Penschuck, and Dorothea Wagner. I/o-efficient generation of massive graphs following the lfr benchmark. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 58–72. SIAM, 2017.

[KSV10] Howard Karloff, Siddharth Suri, and Virginia Vassilevska. A Model of Computation for MapReduce. In *Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'10)*, pages 938–948. SIAM, 2010.

[LF09] Andrea Lancichinetti and Santo Fortunato. Community detection algorithms: A comparative analysis. *Physical Review E*, 80(5), November 2009.

[LFR08] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical review E*, 78(4):046110, 2008.

[LYW+16] Xiao Ling, Jiahai Yang, Dan Wang, Jinfeng Chen, and Liyao Li. Fast community detection in large weighted networks using graphx in the cloud. In *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*, pages 1–8. IEEE, 2016.

[MSS14] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Partitioning complex networks via size-constrained clustering. In *International Symposium on Experimental Algorithms*, pages 351–363. Springer, 2014.

[NB17] Alexander Noe and Timo Bingmann. Bloom filters for reduceby, groupby and join in thrill. 2017.

[New03] Mark E. J. Newman. Fast algorithm for detecting community structure in networks. Phys. Rev. E 69, 066133 (2004), September 2003.

[NG04] Mark E. J. Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(026113):1–16, 2004.

[RAB09] Martin Rosvall, Daniel Axelsson, and Carl T Bergstrom. The map equation. *The European Physical Journal-Special Topics*, 178(1):13–23, 2009.

[RAK07] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3):036106, 2007.

[RBM12] Jason Riedy, David A Bader, and Henning Meyerhenke. Scalable multi-threaded community detection in social networks. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1619–1628. IEEE, 2012.

[ŠB11] Lovro Šubelj and Marko Bajec. Unfolding network communities by combining defensive and offensive label propagation. *arXiv preprint arXiv:1103.2596*, 2011.

[SK12] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2012.

[SM16] Christian L Staudt and Henning Meyerhenke. Engineering parallel algorithms for community detection in massive networks. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):171–184, 2016.

[SS13] Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2013.

[SSM16] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016.

[UKBM11] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[WFSP14] Charith Wickramaarachchi, Marc Frincu, Patrick Small, and Viktor K Prasanna. Fast parallel algorithm for unfolding of communities in large graphs. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6. IEEE, 2014.

[WvE13] Ludo Waltman and Nees Jan van Eck. A smart local moving algorithm for large-scale modularity-based community detection. *The European Physical Journal B*, 86(11):471, 2013.

[WW07] Silke Wagner and Dorothea Wagner. Comparing Clusterings – An Overview. Technical Report 2006-04, ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH), 2007.

[XS12] Jierui Xie and Boleslaw K Szymanski. Towards linear time overlapping community detection in social networks. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 25–36. Springer, 2012.

[YL15] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[ZY15] Jianping Zeng and Hongfeng Yu. Parallel modularity-based community detection on large-scale graphs. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 1–10. IEEE, 2015.