Master Computer Science — Vrije Universiteit Amsterdam

Master Computer Science

Par-CD: Flexible Framework for Parallel Community Detection in Large Networks

Stijn Heldens
s.j.heldens@student.vu.nl

July 31, 2015

Supervisor
prof. dr. ir. Henri E. Bal
Dept. of Computer Science
VU University Amsterdam
bal@cs.vu.nl

Daily supervisor
dr. Ana Lucia Varbanescu
Informatics Institute
University of Amsterdam
a.l.varbanescu@uva.nl

Second Reader
dr. ir. Raphael Poss
Informatics Institute
University of Amsterdam
r.poss@uva.nl

Par-CD: Flexible Framework for Parallel Community Detection in Large Networks

By

Stijn Heldens

Under the supervision of

prof. dr. ir. Henri E. Bal

and

dr. Ana Lucia Varbanescu

Thesis submitted in partial fulfilment of
the requirements for the degree of

Master of Science

in

Computer Science

at the

Vrije Universiteit Amsterdam.

July 2015

Acknowledgements

I would like to express my sincerest gratitude to my daily supervisor, dr. Ana Lucia Varbanescu, for her tremendous support and guidance throughout my thesis. Without her knowledge, motivation, and patience, this project would not have been possible. Our weekly meetings motivated me to continue my research and write this thesis. I could not have wished for a better supervisor.

I would also like to thank dr. ir. Raphael 'kena' Poss who acted as the second reader of my thesis. My thanks also goes to prof. dr. ir. Henri E. Bal who agreed to be the supervisor of my thesis.

Abstract

Community detection is the problem of partitioning a network into *communities*, i.e., densely connected clusters which are sparsely connected to each other. Much research has gone into understanding the communities in networks, designing communities detection algorithms and designing metrics to measure the quality of these algorithms. Despite these efforts, no single algorithm or metrics has become universally accepted. Additionally, very little effort has been invested in performance, which becomes a problem for massive networks consisting of millions or billions of edges.

In this work, we explore a different approach towards community detection. We consider community detection to be a generic optimization problem where the goal is to partition a network into communities by optimizing for any chosen quality metric. Additionally, to enable the processing of large networks, we focus on massive parallelism to achieve high performance.

We have integrated these ideas into *Par-CD*: a flexible framework for parallel community detection in large networks. Our framework is designed to be user-friendly: the user needs to provide a quality metric, select a number of optimization techniques, provide the network to analyze, and the framework will automatically find the communities by optimizing the chosen metric.

We have implemented Par-CD for two modern parallel platforms: multi-core architectures (multi-core CPUs) and many-core architectures (GPUs). We demonstrate that Par-CD is highly scalable, obtains excellent performance on both platforms and produces results which are comparable to state-of-the-art community detection algorithms.

Contents

1 Introduction **7**

2 The Basics of Community Detection **11**
2.1 The Basics 11
2.2 Applications 13
2.3 Detection Methods 14
2.3.1 Graph Partitioning 14
2.3.2 Hierarchical Methods 14
2.3.3 Modularity Optimization 15
2.3.4 Label Propagation Methods 16
2.3.5 Alternative Methods 16
2.4 Evaluation Methods 17
2.4.1 Metrics 17
2.4.2 Synthetic Benchmarks 17
2.4.3 Real-World Networks 18

3 Comparison of Community Quality Metrics **21**
3.1 The Basics 21
3.2 Quality Metrics 22
3.2.1 Clique-based Metrics 23
3.2.2 Density-based Metrics 23
3.2.3 Cut-based Metrics 23
3.2.4 Degree-based Metrics 25
3.2.5 Triangle-based Metrics 27
3.2.6 Potts Model 28
3.3 Comparison 31
3.3.1 Properties 31
3.3.2 Optimization 33
3.3.3 Accuracy 35
3.3.4 Popularity 35
3.4 Summary 35

4 Parallel Community Detection using Label Propagation 37
4.1 Background . . . 37
4.1.1 The Label Propagation Algorithm (LPA) . . . 37
4.1.2 The Louvain Method . . . 38
4.1.3 Scalable Community Detection (SCD) . . . 39
4.2 Label Propagation . . . 40
4.2.1 Generalization . . . 40
4.2.2 Parallelization . . . 41
4.3 Metrics . . . 42
4.3.1 Modularity . . . 43
4.3.2 Constant Potts Model (CPM) . . . 43
4.3.3 Weighted Community Clustering (WCC) . . . 44
4.4 Evaluation . . . 46
4.5 Summary . . . 51

5 Parallel Communities Detection Using Merging 53
5.1 Background . . . 53
5.2 Parallel Merging . . . 55
5.2.1 Generalization . . . 55
5.2.2 Parallelization . . . 56
5.3 Metrics . . . 57
5.3.1 Modularity . . . 57
5.3.2 Constant Potts Model (CPM) . . . 58
5.3.3 Weighted Community Clustering (WCC) . . . 59
5.4 Evaluation . . . 59
5.5 Summary . . . 63

6 Design of Par-CD 65
6.1 Motivation . . . 65
6.2 Requirements . . . 66
6.3 Design . . . 67
6.3.1 Initial Partitioning Phase . . . 67
6.3.2 Refinement Phase . . . 67
6.4 Summary . . . 69

7 Implementation of Par-CD for Multi-Core Architectures 71
7.1 Background . . . 71
7.2 Implementation . . . 72
7.2.1 Data Structures . . . 72
7.2.2 Sequential Implementation . . . 74
7.3 Parallelization . . . 78
7.4 Evaluation . . . 78
7.4.1 Sparse Histogram . . . 79

7.4.2 Sequential Implementation 80
7.4.3 Parallel Implementation 81
7.5 Summary 83

8 Implementation of Par-CD for Many-core Architectures 85
8.1 Background 85
8.1.1 Programming Models 86
8.1.2 Architecture 86
8.2 Implementation 88
8.2.1 Collecting Statistics 90
8.2.2 Calculating Metric 91
8.2.3 Semi-Parallel Label Propagation 91
8.2.4 Best-Neighbor Parallel Merging 93
8.3 Evaluation 93
8.3.1 Performance 94
8.3.2 Memory Footprint 97
8.3.3 End-To-End Performance 98
8.4 Summary 99

9 Evaluation of Par-CD 101
9.1 Case-studies 101
9.1.1 Modularity 102
9.1.2 WCC 106
9.1.3 CPM 109

10 Conclusions & Future Work 115
10.1 Summary 115
10.2 Contributions 115
10.3 Future Research 116
10.4 Final Remarks 117

11 Bibliography 119

A Graph Theory 127

B Approximation Model for WCC when Merging Communities 129

CHAPTER 1

Introduction

A *network* (or *graph* [1]) is data representation consisting of a set of *vertices* (or *nodes*) and a set of *edges* (or *links*). The vertices represent *entities* (e.g., people, cities or products) and the edges represent the *relations* or *interactions* between pairs of entities (e.g., friendships, roads or co-purchase).

Networks play a fundamental role in many fields of research since empirical data can often be represented in this format. For example, a computer network [ACLY00] is a network consisting of computers and routers which are connected by links. The World Wide Web [AJB99] can be seen as a network consisting of web pages which are linked together by hyper links. A social network is a type of network commonly used in sociology where vertices represent people and edges represent the relations between them. Examples of large social networks can be found on social media websites such as Facebook [Fac] and LinkedIn [Lin]. Networks also play an important role in biology and chemistry, for example to model chemical reactions between molecules [MV07] or to describe interactions between DNA segments [Bal85].

These real-world networks are commonly referred to as *complex networks* [KW08], since their topology has been found to more "complex" than that of randomly generated networks. Understanding the characteristics of complex networks has been the subject of much research in recent years [BLM+06, New03, Str01]. Some properties of complex networks are, for example, that they are *sparse*, they have a small diameter (small-world property [Mil67]) and the degree distribution follows a power law (scale-free property [BA99]). Another characteristic is that the edges are not uniformly distributed over the network, but instead, vertices tend to cluster together into tightly packed *communities* [GN02]. Many connections exist within one community, while there are only few connections between different communities. Fig. 1.1 shows an example of the communities in a network.

[1] Throughout this thesis, the terms *network* and *graph* will be used interchangeably.

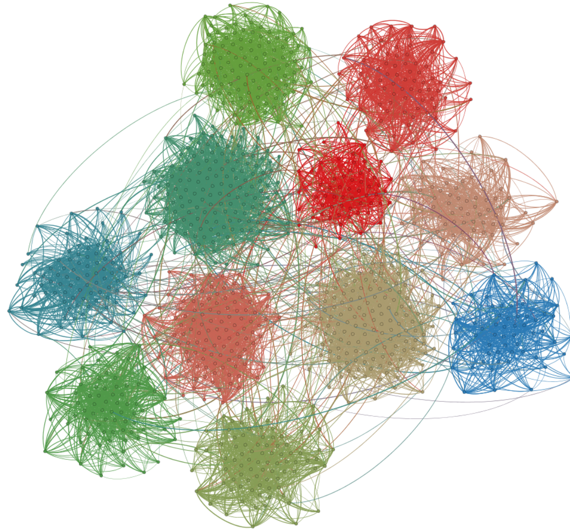


Figure 1.1: Communities detected by the Louvian method [BGLL08] on an synthetic network. Reprinted from `http://perso.uclouvain.be/arnaud.browet/` with permission.

Finding these communities in networks has been an active field of research since the community structure gives insight into the structure and topology of the network. For example, communities on the World Wide Web correspond to web-pages on the same topic. Communities in social networks represent individuals with similar interests, lifestyles or occupations. Communities in protein-protein interaction networks help us understand which proteins have similar functions.

However, detecting communities in a network is a challenging problem. The concept of a community is intuitive and informal. Therefore, designing an algorithm for community detection is not straightforward. Research into community detection algorithms has been very active and many algorithms have been proposed based on different methods. Examples of these methods are spectral analysis [New06], hierarchical clustering [NG04], simulated annealing [GA05], extremal optimization [DA05], spin model [RB04], random walks [PL05] and label propagation [RAK07]. Additionally, a large number of *metrics* have been proposed for measuring the quality of the communities discovered these algorithms. This allows for comparison of the “goodness” of different algorithms. Metrics can be based on different criteria, such as number of triangles, cliques or density.

However, despite much research, no community quality metric or community detection algorithm has become universally accepted. Additionally, very little effort has been placed on scalability and performance, making most algorithms unsuitable for massive networks consisting millions or billions of edges. Most algorithms also use a fixed produce for finding the communities and do not expose parameters to control the algorithm, making them rigid and inflexible.

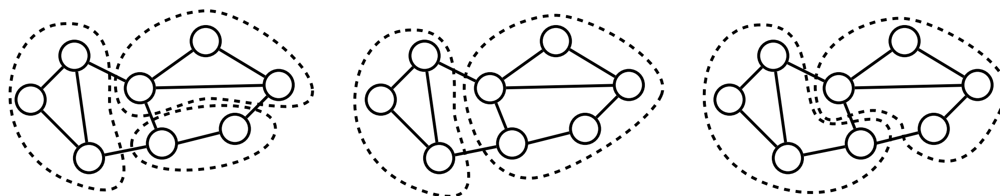


Figure 1.2: Single network partitioned in three different ways

Most existing algorithms are based on intuition about how communities arise and how they can be found. Quality metrics are used to measure the quality of the communities found by these algorithms. However, since different metrics capture the quality of communities in different ways, different communities can be found depending on how one measures their quality.

This problem is illustrated in Fig. 1.2 which shows a single network partitioned in three different ways. Each one of these three partitionings could be considered a valid partitioning of the network since all communities are densely connected and sparsely connected to each other. However, which one of these three partitionings is the best, depends on metric used to evaluate these communities. Metrics based on density might prefer the communities in the first partitioning since those have a density. Metrics based on cut size might prefer the second partitioning since those communities have a low number of boundary edges. Metrics based on number of vertices might prefer the third partitioning since those communities have equal size.

In this thesis, we explore a different approach towards solving the problem of community detection. We believe that there is not single "best" partitioning of a network into communities, but the best partitioning differ per application and depends on the metric chosen. Instead of designing a community detection algorithm around an intuition, we consider community detection to be a optimization problem where the goal is to partition a network into communities by optimizing for a chosen quality metric. The main advantage of this approach is *flexibility*: It is possible to optimize for any chosen quality metric.

Additionally, we focus on *performance*. In recent years and in the future, massive networks consisting of millions or even billions of edges are become more common. Modern computer architectures rely on parallelism to achieve high performance. To enable processing of these networks, algorithms must expose massive amounts of *parallelism*.

This brings us to the research question of this thesis:

> *Can we design a framework for community detection which is flexible, fast and highly parallel?*

The remainder of this thesis is structured as follows:

Community Detection Algorithms & Metrics

Chapter 2 presents background information on community detection. It presents the basic notations of communities, categorizes some of the most well known community detection algorithms and discuss how to evaluate the quality of community detection algorithms. One of these evaluation methods is using a community quality metric. Chapter 3 presents a number of different metrics and makes a comparison between them.

Label Propagation & Merging

Chapters 4 and 5 focus on two community detection methods that are commonly used in practice: Chapter 4 focuses on label propagation and Chapter 5 focuses on merging. In both chapters, we compare community detection algorithms which are based on one of these techniques and show how these techniques can be seen as greedy optimization procedures. For both techniques, we propose a new generalized algorithm which can optimize for any chosen metric and offers massive parallelism.

Design of Framework

Chapter 6 continues the findings from Chapter 4 and 5 by presenting Par-CD: a flexible framework for community detection that focuses on flexibility, high performance and parallelism. This framework only requires the use to select a metric, select a number of optimization techniques and provide a network to analyze. Given this input, the framework will automatically find the community in the given network by optimizing the selected metric using the selected optimization techniques.

Implementation of Framework

Chapters 7 and 8 presents two different implementations of Par-CD for two modern parallel architectures: Chapter 7 focuses on multi-core architectures by presenting an implementation for multi-core CPUs and Chapter 8 focuses on many-core architectures by presenting an implementation for GPUs. We demonstrate that the Par-CD obtains excellent performance on both platforms, although each platforms requires its own approach in order to achieve this performance.

Evaluation of Framework

Chapter 9 compares Par-CD against existing community detection algorithms. We show that Par-CD obtains results that are comparable to state-of-the-art algorithms, while offering much better performance.

Conclusions & Future Work

Finally, Chapter 10 presents conclusions and future work.

CHAPTER 2

The Basics of Community Detection

This chapter gives background information on community detection. It is assumed that the reader is familiar with the basic definitions from graph theory (see Appendix A otherwise).

2.1 The Basics

In a real-world network, vertices typically correspond to *entities* and the edges describe the *relations* or *interactions* between these entities. For example, a friendship network is a network where vertices correspond to people and the edges indicate friendships between people.

Since real-world networks are based on real-world processes, the probability that an edge exists between two vertices is not uniform and depends on many factors. For instance, in a friendship network, it is more likely that two people are friends if they have the same interests, have the same occupation or live in the same area. In practice, vertices have been found to "group" together into tightly-knit clusters [GN02]. In a friendship network, this might happen because all people who attend the same school or work for the same company are friends with each other. It is also common that a friend of a friend is again your friend, resulting in many friendships between people who belong to the same "cluster". This results in groups of vertices which are densely interconnected and only sparsely connected to the rest of the network.

These clusters are typically referred to as *communities*, *clusters*, *classes* or *modules* [For10]. The goal of community detection is to locate these communities. For example, Fig. 2.1 shows a network which can clearly be divided into three communities. We only need a quick visual inspection of this network to locate these three communities. However, formalizing this intuition into a strict mathematical

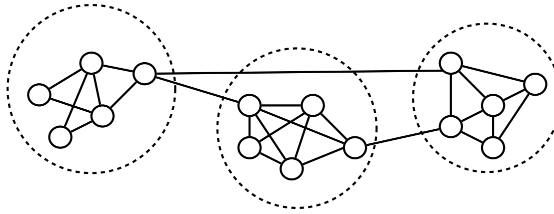


Figure 2.1: Example of a network which can be divided into three communities

definition has turned out to be a non-trivial task. Many researchers have looked at the concept of a community from different perspectives and different definitions have been proposed (See [For10] for an extensive survey), yet no definition is universally accepted.

Despite the lack of a formal definition, consensus has been reached on a number of requirements that any community must fulfil.

- The number of edges inside a community must be high and the number of edges between the community and the rest of the network must be low. In other words, all vertices belonging to the same community must have strong relations with each other and weak relations with the rest of the network.

- Communities must be connected, i.e., for any two vertices that belong to the same community, there must exist a path between these vertices that does not cross the community boundary. This requirement can be seen as a consequence of the first requirement: If a community is not connected, it consists of two or more separate clusters and these clusters do not have a strong relation with each other, violating the first requirement.

- There are no restrictions on the size of a single community or the total number of communities found within a network. The goal of community detection is *not* to partition a network into a fixed number of communities or communities of fixed size. Communities should arise naturally from the topology of the network.

Communities in a network are often defined as a *partitioning* of the network, i.e., the vertices are placed in non-overlapping sets which together cover the entire network. This means that every vertex is assigned to one and only one community. This is obviously a simplification of the real world since one can think of scenarios where a vertex intuitively belongs to multiple communities. For example, in a social network, a person might belong two different communities such as friends, family, co-workers, etc. There also exists community detection algorithms [PDFV05] which allow communities to *overlap*, i.e., a vertex can be assigned to multiple communities. However, along this thesis, we shall only consider non-overlapping communities.

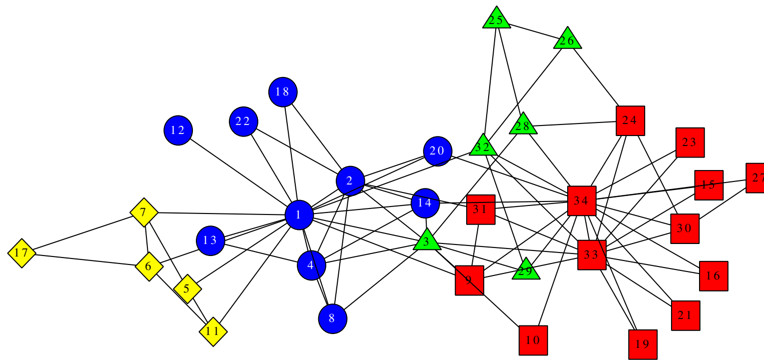


Figure 2.2: Communities in Zachary's network found by the NG algorithm [NG04].

2.2 Applications

There are many applications of community detection. For example, identifying the communities in a network is useful for network analysis. To illustrate this point, see Fig. 2.2 which shows Zachary's network of karate club members [Zac77]. Over a period of three years, Zachary observed the interactions between members of a karate club in the United States. He constructed a network where every vertex represents a club member and two members are connected if they frequently interacted with each other outside the club. At some moment in time, a conflict arose between the club leader and one of the instructors, which led to a split of the karate club into two smaller clubs. Some members joined one club, while others joined the other club.

In this example, community detection can be useful for predicting which members will join which club. Each person will most likely join the club which most of its friends will join. Fig. 2.2 shows the four communities found for this network by the Newman-Girvan algorithm [NG04]. Vertex 1 is the instructor, so all "circle" vertices will most likely join the instructors club. Vertex 34 is the leader, so the "rectangle" vertices will most likely join the leader's club. The "triangle" vertices are indecisive; it is unclear which club they will most likely support. This network is obviously a toy example since the network is small and the communities can easily found by hand. However, for larger networks, finding the communities by hand is no longer feasible.

There are more applications for community detection besides network analysis. One example is implementing a product recommendation system [KASM05] by finding the communities in a product co-purchase network. If a user purchases a product, the system can recommend another product from the same community. Another application is graph visualization where vertices belonging to the same community are visually grouped together [DGDGL07]. A completely different application is to increase the data locality of graph operations by placing vertices from the same community adjacent in memory [PPDSLP11].

2.3 Detection Methods

Many methods for community detection have been proposed over the last decade. This section categorizes some of the most well-known algorithms. For a comprehensive overview, see the surveys by Fortunato [For10] and Papadopoulos et al. [PKVS12].

2.3.1 Graph Partitioning

The problem of community detection is very closely related to the problem of *graph partitioning*. The goal of the graph partitioning problem is to partition a network into k equal-sized components such that number of edges "cut" by the partitioning is minimal. Well-known solutions for graph partitioning are the Kernighan-Lin algorithm [KL70], the Fiduccia-Mattheyses algorithm [FM82] and METIS [KK98]. However, these algorithms are not suitable for community detection since the number of components must be given beforehand, which goes against the principles of community detection. Graph partitioning algorithms do not yield high-quality communities.

2.3.2 Hierarchical Methods

Hierarchical methods find communities in a hierarchical fashion, either from top-down or from bottom-up.

Divisive Methods

Algorithms which apply the top-down approach are known as *divisive* algorithm. These algorithms initially consider the entire network to be a single community and repeatedly split communities until some predefined criterion is met. The result of this procedure is a hierarchy of communities.

One of the most well-known divisive algorithms is the algorithm by Girvan & Newman [GN02, NG04]. This algorithm marks an historically important milestone since it sets the beginning of the popularity of community detection [For10]. The algorithm works by repeatedly deleting the edge from the network with the highest *betweenness* score until all edges have been removed. The *betweenness* of an edge measures how much an edge is *between* two communities. After removing an edge, each connected component is considered to be a community in the original network and the *modularity* (see Chapter 3) of this partitioning into communities is calculated. The partitioning with the highest modularity is considered to be the best partitioning.

Different metrics to measure the *betweenness* of edges have been proposed. Girvan & Newman originally propose *shortest-path betweenness* as a measure for betweenness of edges. Radicchi et al. [RCC+04] proposed to use the *edge clustering coefficient*, Fortunato et al. [FLM04] introduced a measure known as *information centrality* and Vragović & Louis [VL06] proposed a metric based on the cycles in the newtork.

Newman also proposed a divisive method based on spectral analysis. Newman [New06] showed how the first few eigenvectors of a special matrix, the *modularity matrix*, of a graph can be used to split a graph into two partitions with maximal modularity. Since the procedure only yields a bipartition, one needs to recursively apply the same procedure to the two resulting partitions. The result is a hierarchy of communities which is shaped like a binary tree.

Agglomerative Methods

Algorithms which apply the bottom-up approach are known as *agglomerative* algorithms. In these algorithms, each vertex is initially considered to be a community and pairs of communities are repeatedly merged until some predefined criterion is met.

The most popular agglomerative algorithm is Newman's greedy algorithm [New04] which uses the increase in modularity as a criterion for which two communities to merge in each iteration. Specifically, in every iteration, the two communities that lead to the maximum increase (or minimum decrease) in modularity are merged. Clauset et al. [CNM04] and Wakita & Tsurumi [WT07] later improved the runtime of the algorithm by designing more efficient data structures.

Zhou & Lipowsky [ZL04] and Latapy & Pons [PL05] proposed two agglomerative algorithms based on random walks which are called *NetWalk* and *WalkTrap*, respectively. These algorithms are both based on the intuition that, since communities have a high internal density and a low external density, the probability that one would "escape" from a community when performing a random walk is small. Netwalk and WalkTrap both merge communities based on the probability p^k_{uv}, i.e., the probability that if one would start a walk of length k at vertex u then one would arrive at vertex v.

2.3.3 Modularity Optimization

Newman & Girvan originally introduced the concept of modularity to determine the best division of the network for the Girvan-Newman algorithm. After its introduction, the metric quickly gained a lot of popularity and has become the most used and best known quality metric for communities [For10]. Using a metric to measure the quality of communities turns the problem of community detection

into an optimization problem. Specifically, if one assume that a partitioning of the network with high modularity consists of communities of high quality, then community detection is equivalent to maximizing modularity.

Newman's bisection algorithm and Newman's greedy algorithm, both discussed in Section 2.3.2, are essentially greedy produces to maximize modularity. Blondel et al. [BGLL08] proposed a different procedure based on modularity optimization. Initially, every vertex is assigned to a unique community. Next, one vertex is placed in the community of one its neighbors which leads to the largest positive increase in modularity. This step is repeated until a local optimum is reached.

Yet another approach is to use a generic optimization procedure, and apply it to modularity. Guimerà & Ameral [GA05] proposed a community detection based on *simulated annealing*, a probabilistic procedure for global optimization. Duch & Aeranas [DA05] showed how *extremal optimization*, a heuristic search procedure, could be used for finding a partitioning having high modularity. However, both procedures are computationally expensive and can only be applied to very small networks.

2.3.4 Label Propagation Methods

Label propagation algorithms work by assigning a label to every vertex which indicates to which community the vertex belongs. Next, a number of iterations are performed to propagate the labels through the network. In every iteration, the vertices of the network are updated. When a vertex gets updated, it can decide to keep its current label or change it to the label of one of its neighbors.

The Label Propagation Algorithm (LPA) by Raghavan et al. [RAK07] is the simplest label propagation algorithm. For this algorithm, every vertex must change its label to common label amongst its neighbors. The Louvain method [BGLL08] can be seen as label propagation algorithm based on modularity maximization. SCD [PPDSLP14] is a label propagation algorithm based on *WCC* [PPDSBLP12] maximization.

2.3.5 Alternative Methods

Community detection is a broad field of research and many proposed algorithms do not fit into any of the above categories. For example, Agarwal & Kempe [AK08] showed how the problem of finding the partitioning with optimal modularity can be turned into a linear or quadratic mathematical programming problem and thus can be solved by any linear/quadratic solver. Van Dongen [VD00] proposed an algorithm called *Markov Cluster Algorithm* which works by repeatedly *expanding* and *inflating* the *transfer matrix* of a graph. Arenas et al. [ADGPV06] intro-

duced the idea of using Kuramoto oscillators [Kur03] for detecting communities. Newman & Leicht [NL07] proposed a community detection algorithm based on an *expectation-maximization* technique. There are also exotic algorithms which do not focus on partitioning a network but focus on finding overlapping communities [PDFV05] or finding hierarchical communities [LFK09].

2.4 Evaluation Methods

A fundamental problem of community detection is that its evaluation is difficult. How does one determine whether the communities found by an algorithm are "good"? How does one compare the communities found by different algorithms on the same network in order to decide which algorithm is "best"? The problem of evaluating community detection algorithms is an open problem. In practice, three different methods are commonly used: community quality metrics, synthetic benchmarks, and comparison with ground-truth communities of real-world networks.

2.4.1 Metrics

A community quality metric $f(\mathcal{P})$ is a function which captures the quality of a partitioning $\mathcal{P}$ by assigning it a quality score. As an example, *modularity* [NG04] is a well-known quality metric which is defined as follows:

$$f(\mathcal{P}) = \sum_{C \in \mathcal{P}} \sum_{v \in C} \sum_{u \in C} \left(A_{uv} - \frac{d(u)d(v)}{2m} \right) \tag{2.1}$$

Where $d(u)$ is the degree of vertex u, m is the total number of edges and A the adjacency matrix of the network having $A_{uv} = 1$ if edge (u, v) exists and $A_{uv} = 0$ otherwise. We will discuss a number of quality metrics for community detection in Chapter 3.

2.4.2 Synthetic Benchmarks

Synthetic benchmarks are also commonly used to evaluate community detection algorithms. These benchmarks use a model to generate a synthetic network which has known ground-truth communities. Comparing the communities found by an algorithm to the ground-truth communities gives an indication of the quality of the algorithm. If the communities match, then the quality of the results is high. If all communities are different, the results are of low quality and therefore meaningless.

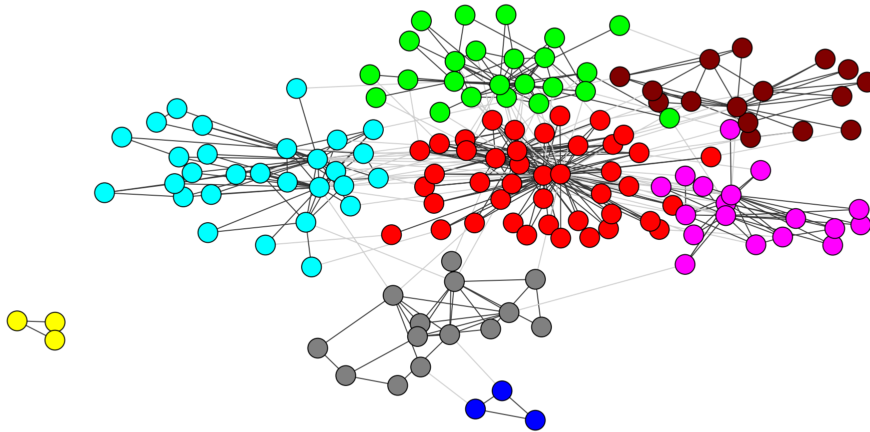


Figure 2.3: Example of an LFR network for $n = 150$, $\overline{d} = 15$ and $\mu = 0.2$. The colors indicate the eight different communities.

Generating realistic synthetic networks is not trivial since they must have the same topology and characteristics as real-world networks. Generating a network by simply randomly placing edges between vertices (Erdős-Rényi model) [ER61] does not yield a realistic network. Lancichinetti, Fortunato and Radicchi proposed a model (LFR model) based on the planted partition model [LFR08] for generating networks having ground-truth communities. This model is frequently used in practice [FLM04]. The most important parameter of this model is the mixing factor μ which determines how "entangled" the network is. For $\mu = 0$, the communities are completely isolated from each other. For $\mu = 1$, the communities are completely mixed. Figure 2.3 shows an example of an LFR network for $\mu = 0.2$.

To measure the similarity between the ground-truth communities and the communities found by the algorithm, the *normalized mutual information* (NMI) [DDGDA05] is commonly used. NMI is a metric from information theory which measures the similarity between two partitions A and B of a network. If A and B are identical then $\text{NMI}(A, B) = 1$. If A and B are completely independent then $\text{NMI}(A, B) = 0$.

2.4.3 Real-World Networks

Instead of evaluating algorithms on synthetic networks, it is also possible to use real-world networks. While evaluation on synthetic networks is more controlled, evaluation on real-world networks is more realistic.

To obtain a real-world network with ground-truth communities, one must place the vertices into meaningful communities. For example, for a social network, one could generate communities based on occupation or location of its users. For a product co-purchasing network, each product category can correspond to a community. For a scientific collaboration network, researchers working for the same institute can be assigned to the same community.

There exist many repositories that provide publicly available, real-world datasets having ground-truth communities. For example, SNAP [LK14] is a repository which provides a large number of real-world datasets.

CHAPTER 3

Comparison of Community Quality Metrics

This chapter revolves around community quality metrics. We discuss the concept behind such quality metrics, classify the most well-known metrics and make a comparison between them.

3.1 The Basics

Intuitively, communities should be densely internally connected and only sparsely connected to each other. However, this concept is an intuition and not a strict definition. Different community detection methods might find different communities in the same networks since they use different criteria to measure the "goodness" of a community.

To illustrate this point, consider Fig. 3.1. This figure shows how a single network can be partitioned in six different ways. Each partitioning consists of "valid" communities since they are densely interconnected and sparsely connected to each other. Now imagine that these six partitionings were produced by six different community detection algorithms. How does one determine which of these six algorithms is the best one? In other words, how do we measure the *quality* of these communities?

Community quality metrics attempt to solve this problem. A quality metric captures the "goodness" of a partitioning into a quality score. Formally, a metric f is a function which assigns a score to a partitioning $\mathcal{P}$ of a network into communities.

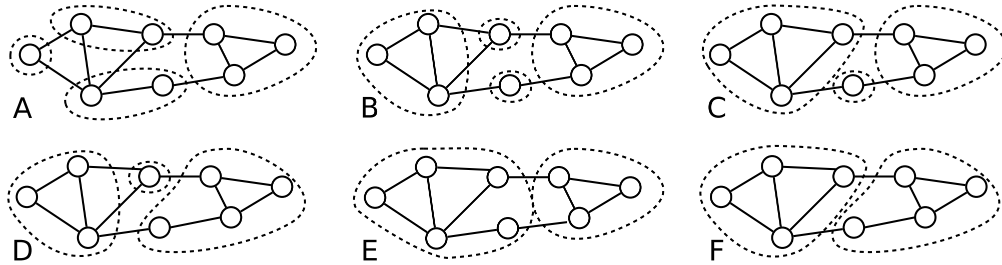


Figure 3.1: Example of networks partitioned in six different ways. Dashed circles indicates the different communities.

Clique-based	Density-based	Cut-based	Degree-based	Triangle-based	Model-based
Clique	Edges inside	Cut	Strong comm.	TPR	Modularity
n-clique	Density	Cut ratio	Weak comm.	LCC	RB
n-clan		Expansion	FOMD	WCC	CPM
K-core		Normalized cut	Avg-ODF		
K-plex		Conductance	Max-ODF		
P-quasi graph			Flake-ODF		
n-club					

Figure 3.2: Overview of community quality metrics.

3.2 Quality Metrics

Different metrics have been proposed based on different criteria. Fig. 3.2 shows an overview of the six classes of metrics which will be discussed in this chapter.

For the remainder of this chapter, we assume $G = (V, E)$ to be an arbitrary undirected, unweighted graph having $n = |V|$ vertices and $m = |E|$ edges. Furthermore, we assume $C \subseteq V$ to be a community of G having $n_C = |C|$ vertices, $m_C = |\{(u, v) \in E : u \in C, v \in C\}|$ *internal* edges and $b_C = |\{(u, v) \in E : u \in C, v \notin C\}|$ *boundary* edges. The degree $d(v)$ of vertex v is defined as the number of neighbors of v. The *internal* degree $d_{int}(v)$ and *external* degree $d_{ext}(v)$ are defined as the number of neighbors of v inside or outside its community, respectively.

Most quality metric do not assign a score to a partitioning of the network, but instead, assign a score to each individual community. In this case, the quality of a partitioning is defined as the weighted average over all communities.

$$f(\mathcal{P}) = \frac{\sum_{C \in \mathcal{P}} |C| \; f(C)}{\sum_{C \in \mathcal{P}} |C|} \tag{3.1}$$

3.2.1 Clique-based Metrics

A *clique* is a maximal subgraph where every vertex is connected to every other vertex and which cannot be extended by including one more adjacent vertex [BBPP99]. A clique is a high-quality community since the number of internal edges is maximal. Cliques can be used as a measure for community quality by defining $f(C) = 1$ if C is a clique and $f(C) = 0$ otherwise. However, such a metric is not useful in practice since large cliques are a rare occurrence in real-world networks.

Several relaxed definitions for a clique have been proposed. An n-clique [Alb73] is a subgraph such that the distance of the shortest path between any two vertices is less than or equal to n. A n-clan [Mok79] is an n-clique with the restriction that all shortest paths must only pass over other vertices of the subgraph. A k-core [Sei83] is a subgraph where each vertex must be adjacent to at least k other vertices of the subgraph. A k-plex [SF78] is a subgraph where each vertex must be adjacent to all except at most k other vertices of the subgraph. A p-quasi subgraph [MIH99] is a maximal subgraph where each vertex is adjacent to $p(k-1)$ other vertices of the subgraph where k is the order of the subgraph. A n-club [Mok79] is a maximal subgraph of diameter n.

In practice, none of these definitions are frequently used as quality metrics. They are very strict and removing only a single edge can cause a community to lose its clique-property. Additionally, finding cliques in networks is very computationally expensive. The problem of finding a maximum clique in a network is NP-complete [BBPP99].

3.2.2 Density-based Metrics

Since communities should have a high internal connectivity, a trivial quality metric would be the *number of internal edges* of a community. However, the number of internal edges is not a useful metric since larger communities also have more internal edges. We can solve by dividing the number of internal edges by the total number of *possible* internal edges. This metric is known as the *internal density* of a community [For10].

$$f(C) = \frac{m_C}{n_C(n_C - 1)/2} \tag{3.2}$$

3.2.3 Cut-based Metrics

The *cut* of a community is the number of boundary edges of a community. Intuitively, this corresponds to the number of edges required to "cut" away in order

to separate the community from the rest of the network. A low cut indicates a high-quality community since it is well separated from the rest of the network. However, the cut itself is not a good quality measure since larger communities commonly also have more boundary edges. Several metrics based on the cut have been proposed.

Cut Ratio

The *cut ratio* [WC89] is a metric originally introduced for the problem of graph partitioning. It is defined as the number of boundary edges divided by the total number of possible boundary edges.

$$f(C) = \frac{b_C}{n_C(n - n_C)} \tag{3.3}$$

Expansion

Expansion [RCC+04] is defined as the average number of boundary edges per vertex. This can be seen as the average "cut-per-vertex".

$$f(C) = \frac{b_C}{n_C} \tag{3.4}$$

Normalized Cut

Shi et al. defined the *normalized cut* [SM97] as a metric for bi-partitioning. It is defined as the cut divided by the volume of the two subgraphs that result from cutting the graph.

$$f(C) = \frac{b_C}{\sum_{v \in C} d(v)} + \frac{b_C}{\sum_{v \notin C} d(v)} \tag{3.5}$$

Conductance

Conductance [Bol98] is another popular metric used for graph partitioning, but it has been found to also be useful for community detection [For10]. The conductance

of a subgraph is generally denoted by Φ and is defined as follows.

$$\Phi(C) = \frac{b_C}{\sum_{v \in C} d(v)} \tag{3.6}$$

Intuitively, conductance can be seen as area-volume ratio of a community. The "border" of the community (i.e., number of boundary edges) is divided by the "area" of the community (i.e., the total degree of the vertices). This corresponds to our intuition that communities which have a larger "area" are allowed to have a larger "border".

3.2.4 Degree-based Metrics

A different approach for defining a metric is to compare the *internal* and *external* degree of every vertex of a community. Ideally, each vertex should have many neighbors inside the same community and few neighbors outside the community. In other words, vertices should have a high *internal* degree and a low *external* degree.

Strong & Weak Communities

A community is defined as a *LS-set* [LS69] or *strong* community [RCC+04] if the external degree of each vertex does not exceed its internal degree. In other words, each vertex has more edges towards vertices inside its community than to the rest of the network.

$$f(C) = \begin{cases} 1 & \text{if } d_{int}(v) \geq d_{ext}(v) \quad \forall v \in C \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

This definition is very strict and it can be relaxed into a weaker definition. A community is defined as a *weak* community [RCC+04] if the *sum* of the external degrees does not exceed the sum of the internal degrees.

$$f(C) = \begin{cases} 1 & \text{if } \sum_{v \in C} d_{int}(v) \geq \sum_{v \in C} d_{ext}(v) \\ 0 & \text{otherwise} \end{cases} \tag{3.8}$$

Note that any strong community is a weak community by definition, but the converse is not always true.

Fraction over median degree (FOMD)

Let d_m be the median value of $d(v)$ over all vertices v of C. The *fraction over median degree* (FOMD) [YL15] of a community is defined as the fraction of vertices which have an internal degree $d_{in}(v)$ greater than d_m.

$$f(C) = \frac{|\{v \in C : d_{in}(v) > d_m\}|}{n_C} \tag{3.9}$$

Out Degree Fraction (ODF)

Flake et al. [FLG00] proposed the *out degree fraction* (ODF) as a measure of community quality. The ODF of a vertex v is the ratio of the number of neighbors outside its community to the total number of neighbors. Vertices should have a low ODF.

$$ODF(v) = \frac{d_{ext}(v)}{d(v)} \tag{3.10}$$

Based on ODF, Flake et al. proposed three different metrics. The *avg-ODF* is simply the average ODF over all vertices in C.

$$f(C) = \frac{1}{|C|} \sum_{v \in C} ODF(v) \tag{3.11}$$

By taking the average ODF over all vertices, all *outliers* (i.e., vertices having an extremely high ODF) will be hidden. The *max-ODF* finds these outliers by finding the maximum ODF over all vertices.

$$f(C) = \max_{v \in C} ODF(v) \tag{3.12}$$

The *Flake-ODF* is the fraction of the vertices of a community having a higher internal degree than external degree. In other words, it is the fraction of vertices not satisfying the strong community definition from Section 3.2.4. Flake-ODF can be defined as follows:

$$f(C) = \frac{|\{v \in C : \ ODF(v) \leq \frac{1}{2}\}|}{|C|} \tag{3.13}$$

3.2.5 Triangle-based Metrics

Several studies have shown that triangles are more common in social networks than in random networks and that the presence of triangles is a good indicator for communities [NP03, BPSDGA04, SAS07, FCF11]. We will discuss three metrics based on triangle counting.

Triangle Participation Ratio (TPR)

The *triangle participation ratio* [YL15] is the fraction of vertices belonging to at least one internal triangle, where an internal triangle is defined as a triangle having all three endpoints in community C. Let $t(v, C)$ be defined as the number of triangles having v and two vertices from C as its endpoints. The TPR of C is now defined as follows:

$$f(C) = \frac{|\{v \in C : t(v, C) > 0\}|}{n_C} \tag{3.14}$$

Local Clustering Coefficient (LCC)

The *clustering coefficient* [WS98] was introduced by Watts & Strogatz to measure whether a network exhibits the properties of a small-world network. The clustering coefficient of a vertex is defined as the ratio of triangles closed by this vertex to the total number of possible triangles it could close with its neighbors. In other words, it measures how close a vertex and all its neighbors are to being a clique.

Basuchowdhuri & Chen [BC10] modified this metric to make it more suitable for community detection by only focusing on the neighbors of a vertex belonging to the same community. In other words, they measured how close a vertex and all its *internal* neighbors are to being a clique. The *local clustering coefficient* (LCC) of a vertex v is defined as follows:

$$LCC(v) = \begin{cases} \frac{t(v,C)}{(d_{in}(v)(d_{in}(v)-1)/2} & \text{if } d_{in}(v) \geq 2 \\ 0 & \text{otherwise} \end{cases} \tag{3.15}$$

Where $t(v, C)$ has the same definition as for the TPR. The local clustering coefficient of a community is defined as the average LCC over all its vertices.

$$f(C) = \frac{1}{n_C} \sum_{v \in C} LCC(v) \tag{3.16}$$

Weighted Community Clustering (WCC)

Prat et al. [PPDSBLP12] proposed a metric based on triangles called the *Weighted Community Clustering*. They defined the degree of cohesion $WCC(v, C)$ of a vertex v to its community C based on the number of triangles v closes with its neighbors and the size of C. The cohesion between a vertex and a community measures how well a vertex 'belongs' to that community.

Let $t(v, C)$ be the number of triangles that vertex v closes with vertices from C and let $vt(v, C)$ be the number of vertices from C that close at least one triangle with v and a second vertex from C. The $WCC(v, C)$ is now defined as follows.

$$WCC(v, C) = \begin{cases} \frac{t(v,C)}{t(v,V)} \cdot \frac{vt(v,V)}{|C\backslash\{v\}|+vt(v,V\backslash C)} & \text{if } t(v, V) \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.17}$$

The first case consists of a product of two fractions. The first fraction measures the ratio of the number of triangles that v closes with vertices in C to the total number of triangles that v closes with the entire network. The second fraction measures the ratio of the number of vertices that close at least one triangle with v to the size of C plus the number of vertices outside C that close at least one triangle with v. The first fraction rewards adding more vertices to C that create more triangles, while the second fraction rewards removing vertices from C which make the community more tightly connected.

The second case prevents a division by zero for the two fractions in the first case. It is easy to see that $t(v, V) \neq 0$ also implies $|C\backslash\{v\}| + vt(v, V\backslash C) \neq 0$. Assume that $|C\backslash\{v\}| + vt(v, V\backslash C) = 0$ then $|C\backslash\{v\}| = 0$ and $vt(v, V\backslash C) = 0$. The first condition gives $|C| \leq 1$ so v cannot close any triangles with vertices from C, i.e. $vt(V, C) = 0$. Since $vt(v, C) = 0$ and $vt(v, V\backslash C) = 0$ we have $vt(v, V) = 0$ and thus $t(v, V) = 0$.

The WCC of a community is defined as taking the average WCC over all vertices of C.

$$f(C) = \frac{1}{n_C} \sum_{v \in C} WCC(v, C) \tag{3.18}$$

3.2.6 Potts Model

There also exists a variety of metrics based on the so-called *Potts model* [RB06]. The Potts model is a model which originates from statistical mechanics, but has also been found to be useful for community detection. The idea behind the Potts

model is that a partitioning of the network has a high quality if it has many internal edges and few external edges. Based on this idea, Reichardt & Bornholdt [RB06] proposed the following quality function which punished external edges and missing internal edges and rewards internal edges and missing external edges:

$$\begin{aligned} f(\mathcal{P}) = & \underbrace{\sum_{u,v \in V} a_{uv} A_{uv} \delta(C_{i_u}, C_{i_v})}_{\text{Internal edges}} + \underbrace{\sum_{u,v \in V} -b_{uv}(1 - A_{uv}) \delta(C_{i_u}, C_{i_v})}_{\text{Internal missing edges}} \\ & + \underbrace{\sum_{u,v \in V} -c_{uv} A_{uv} (1 - \delta(C_{i_u}, C_{i_v}))}_{\text{External edges}} + \underbrace{\sum_{u,v \in V} d_{uv}(1 - A_{uv})(1 - \delta(C_{i_u}, C_{i_v}))}_{\text{External missing edges}} \end{aligned} \tag{3.19}$$

In this equation, A_{uv} is the adjacency matrix of the network with $A_{uv} = 1$ if edge (u, v) exists and $A_{uv} = 0$ otherwise, C_{i_v} is the community to which vertex v belongs and δ is the Kronecker delta function with $\delta(C_{i_u}, C_{i_v}) = 1$ if $C_{i_u} = C_{i_v}$ and $\delta(C_{i_u}, C_{i_v}) = 0$ otherwise. The weights a_{uv}, b_{uv}, c_{uv} and d_{uv} denote the weights of the contributions. Reichardt & Bornholdt argued that one should weight edges and missing edges equally, regardless whether they are external or internal. If we assume $a_{uv} = c_{uv}$ and $b_{uv} = d_{uv}$ then the equation can be simplified to:

$$f(\mathcal{P}) = \sum_{u,v \in V} 2(a_{uv} A_{uv} - b_{uv}(1 - A_{uv}))\delta(i_u, i_u) + \sum_{u,v \in V} (-a_{uv} A_{uv} + b_{uv} A_{uv}) \tag{3.20}$$

The first term can be divided by two and the second term can be removed since it is constant and does not depend on $\mathcal{P}$.

$$f(\mathcal{P}) = \sum_{u,v \in V} (a_{uv} A_{uv} - b_{uv}(1 - A_{uv}))\delta(i_u, i_u) \tag{3.21}$$

This only leaves the choice of the weights a_{uv} and b_{uv}. Several options have been proposed.

Modularity

Modularity [New04] is a metric proposed by Newman & Girvan. Modularity is by far the most popular and best known quality metric for community detection [For10]. The metric was originally introduced to find the best partition for

the Newman-Girvan algorithm [New04], but its use has quickly become widespread and it currently is the de-facto standard for community detection.

Modularity is based on the idea that one should use a model to determine the probability that an edge exists between two vertices if the network had the same characteristics as the original network, but without a community structure. The model is referred to as the *null* model. Let p_{uv} be the probability that edge (u, v) exists according to the null model and defined $a_{uv} = p_{uv}$ and $b_{uv} = 1 - p_{uv}$. By rewriting Eq. 3.21 we arrive at the definition for modularity.

$$f(\mathcal{P}) = \sum_{u,v \in V} (A_{uv} - p_{uv})\delta(C_{i_u}, C_{i_v}) \tag{3.22}$$

One possible option for the null model would be assume that the probability that an edge exists is constant, i.e. $p_{uv} = \frac{2m}{n^2}$. However, this is not an accurate model since it does not take the degrees of the vertices into account. The probability that an edge exists between vertices is higher if both vertices have a high degree than if they have a low degree. Based on this idea, Newman & Girvan proposed the following null model for modularity.

$$p_{uv} = \frac{d(u)d(v)}{2m} \tag{3.23}$$

This null model is implied whenever modularity is used.

Reichardt-Bornholdt

Reichardt & Bornholdt [RB06] extended the definition of modularity by introducing a *resolution parameter* γ:

$$f(\mathcal{P}) = \sum_{u,v \in V} (A_{uv} - \gamma p_{uv})\delta(C_{i_u}, C_{i_v}) \tag{3.24}$$

This resolution parameter allows one to consider the community structure at different *resolutions*. For $\gamma = 1$, the definition reverts back to the original definition of modularity. For $\gamma = 0$, the metric is equivalent to counting the number of internal edges and a trivial optimum can be obtained by placing all vertices in one single community. Varying the value of γ between 0 and 1 allows one to detect communities at different "levels", either from "up close" or from "far away".

Constant Potts Model

Traag & Dooren [TVDN11] proposed a function which includes a resolution parameter but which is not based on a null model. They defined this *Constant Potts model* (CPM) by choosing $a_{uv} = 1 - \gamma$ and $b_{uv} = \gamma$.

$$f(\mathcal{P}) = \sum_{u,v \in V} (A_{uv} - \gamma)\delta(C_{i_u}, C_{i_u}) \tag{3.25}$$

Note that this function has two trivial optima for $\gamma \leq 0$ and $\gamma \geq 1$. For $\gamma = 0$, the function becomes $\sum_{u,v} A_{uv}\delta(C_{i_u}, C_{i_u})$ and the global optimum is when all vertices are placed in the same community. For $\gamma = 1$, the function becomes $\sum_{u,v}(A_{uv} - 1)\delta(C_{i_u}, C_{i_u})$ and the global optimum is when each vertex is placed in singleton communities. These two optima also hold for $\gamma < 0$ and $\gamma > 1$. This means that only the interval $\gamma \in (0, 1)$ is interesting.

Ronhovde & Nussinov [RN10] proposed a similar function where $a_{uv} = 1$ and $b_{uv} = \gamma$. The CPM function is equivalent to the RN function for $\gamma_{CPM} = \frac{\gamma_{RN}}{\gamma_{RN}+1}$. However, the CPM is more intuitive and easier to work with due to the optima at 0 and 1 instead of at 0 and $+\infty$.

3.3 Comparison

In this section, we make a comparison between the metrics from the previous section. We compare the metrics based on what properties of a community they measure, their suitability for optimization, their accuracy, and their popularity. See Table 3.1.

3.3.1 Properties

Overall, community quality metrics can be classified into three categories: vertex-based, community-based and partition-based metrics (Table 3.1). Vertex-based metrics assign a quality score to each vertex of the network. The quality of a community is calculated by reducing these scores to a single value. Examples of metrics in this category are Max-ODF and WCC. Community-based metrics assign a score to a community based on the global properties of the entire community, such as its size or number of internal edges. Examples of metrics from this category are conductance and density. Partition-based metrics assign a score to a partitioning of a network into communities and thus they cannot be used to measure the quality of individual communities. The three metrics based on the Potts model belong to this category. In practice, vertex-based metrics are the easiest to use since it is

Table 3.1: Properties of community quality metrics. The letters in the Type column denote: v=vertex, c=community, p=partition.

		Criteria				Optimization			
Name	Type	Community size	Internal Connectivity	External Connectivity	Triangles	Goal	No trivial global optimum	Resolution-limit free	Popularity
Cliques	c		$\checkmark$			Max.	?	$\checkmark$	
Edges inside	c		$\checkmark$			Max.		$\checkmark$	
Density	c	$\checkmark$	$\checkmark$			Max.	?	$\checkmark$	+
Cut	c			$\checkmark$		Min.		$\checkmark$	
Cut Ratio	c	$\checkmark$		$\checkmark$		Min.	$\checkmark$		
Expansion	c	$\checkmark$		$\checkmark$		Min.		$\checkmark$	
Normalized Cut	c		$\checkmark$	$\checkmark$		Min.	$\checkmark$		
Conductance	c		$\checkmark$	$\checkmark$		Min.	$\checkmark$		++
Strong comm.	v		$\checkmark$	$\checkmark$		Max.		$\checkmark$	+
Weak comm.	c		$\checkmark$	$\checkmark$		Max.		$\checkmark$	+
FOMD	c	$\checkmark$	$\checkmark$			Max.	$\checkmark$	$\checkmark$	
Avg-ODF	v	$\checkmark$	$\checkmark$	$\checkmark$		Min.		$\checkmark$	
Max-ODF	v		$\checkmark$	$\checkmark$		Min.		$\checkmark$	
Flake-ODF	v	$\checkmark$	$\checkmark$	$\checkmark$		Min.		$\checkmark$	
TPR	v	$\checkmark$	$\checkmark$		$\checkmark$	Max.		$\checkmark$	+
LCC	v	$\checkmark$	$\checkmark$		$\checkmark$	Max.	?	$\checkmark$	
WCC	v	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	Max.	$\checkmark$	$\checkmark$	
Modularity	p		$\checkmark$	$\checkmark$		Max.	$\checkmark$		+++
RB	p		$\checkmark$	$\checkmark$		Max.	$\checkmark$		
CPM	p		$\checkmark$	$\checkmark$		Max.	$\checkmark$	$\checkmark$	

clear what the contribution of each individual vertex is to the overall quality of a partitioning.

Different metrics also measure different properties of a community. Intuitively, a community should have high internal connectivity and low external connectivity. A good metric should thus take both the internal and external edges of communities into account. If this is the case, then the quality of a community can be improved by either adding internal edges or removing external edges. Despite this, not all metrics are based on combining internal and external connectivity. Some metrics consider one type of connectivity. (Table 3.1).

It has been frequently shown that triangles are a lot more common in social networks than one would expect in a random network, and that triangles are a good indicator for communities [NP03, BPSDGA04, SAS07, FCF11]. A good metric should thus also incorporate triangles in some way. Yet, only three metrics are based on triangle counting: TPR, LCC and WCC.

Based on these observations, we can find that the WCC is the metrics with the best properties. It is a vertex-centric metric, it is based on triangle counting, and considers both external and internal connectivity of communities.

3.3.2 Optimization

Using metrics to measure the quality of a community turns the problem of community into an optimization problem. Better the score of a community for a chosen metric, better the quality of the community. In that sense, community detection can be seen as an optimization procedure for finding the partitioning of a network into communities by optimizing a quality metric. However, not all metrics are suitable for optimization.

First of all, some metrics need to be minimized while others need to be maximized (Table 3.1). For some metrics, such as conductance and expansion, a low score indicates a high-quality community. For other metrics, such as modularity and density, a high score indicates a high-quality community .

Second, a metric should have a global optimum which is interesting and not trivial to find. If a metric has a trivial global optimum then any optimization produce will either find this global optimum, which is not interesting, or a local optimum, which is not globally optimal.

For the following metrics, the global optimum can be obtained by assigning all vertices to the same community: the number of edges inside, the cut, the strong/weak communities, ODF-based metrics and TPR. For the following metrics, finding the global optimal partitioning is not trivial but finding a single community having the maximal score is trivial: internal density and LCC. The internal density is maxi-

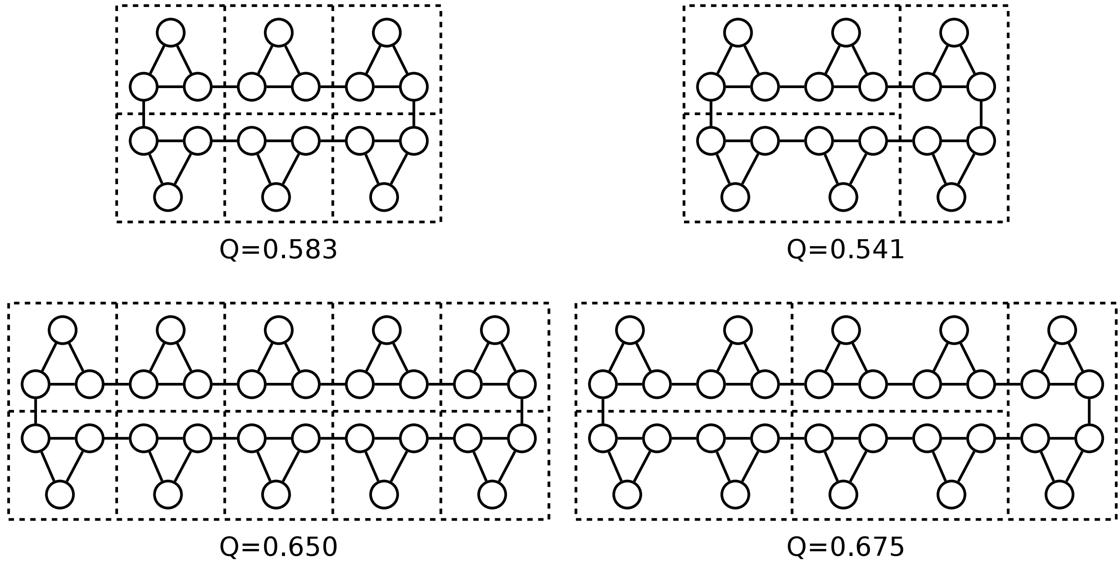


Figure 3.3: Example demonstrating the resolution limit. The number below each partitioning denotes its modularity.

mal for a community consisting of two connected vertices and LCC is maximal for a community consisting of three connected vertices.

Third, a metric should be *resolution-limit free.* The problem of the resolution limit in community detection was first discovered by Fortunato & Barthélemy [FB07] in the case of modularity. Intuitively, a metric which is resolution-limit free measures the quality of a community only based on the local neighborhood of the community. Not all metrics are resolution-limit free, see Table 3.1. Traag et al. [TVDN11] formalized the idea of the resolution-limit into a mathematical definition. Table 3.1 shows which metrics are resolution-limit free according to this definition.

To illustrate the problem of the resolution limit, consider the example shown in Fig. 3.3. This figure shows two networks which consists of six/ten triangles in a ring. For the first network, modularity is maximal if every triangle is seen as a community. For the second network, modularity is maximal when every two adjacent triangles are seen as a single community. This is counter intuitive since we would not expect that adding more triangles to the ring would affect the quality of the existing communities. From the perspective of one individual triangle, its local topology has not changed. Modularity is thus not resolution-limit free.

From these observations, we can conclude that the metrics most suitable for optimization are FOMD, WCC and CPM. These metrics do not have a trivial optimum and are resolution-limit free. CPM is particularly interesting since it include a resolution parameter, allowing one to choose the resolution of the communities.

3.3.3 Accuracy

Leskovec et al. [LLM10] performed an extensive empirical comparison of different community quality metrics. They showed that internal density, max-ODF, Flake-ODF and normalized cut are not good measures for community quality. They also found that conductance, expansion, normalized cut and avg-ODF are highly correlated and prefer the same partitioning.

Yang et al. [YL15] performed another empirical comparison where they measure the quality of real-world communities using different metrics. They compared the metrics based on four criteria: how much is a community separated from the rest of the network (*separability*), how dense is a community (*density*), how hard is it to split a community into two sub communities (*cohesiveness*) and how well clustered are the vertices (*clustering coefficient*). They concluded that conductance performs best in terms of separability. TPR scored the best on the other three criteria.

3.3.4 Popularity

Finally, we consider the popularity of the different metrics. Table 3.1 shows the popularity of each metric. We have rated the popularity of each metric based on its usage in recent literature on community detection.

Modularity is by far the most popular metric [NG04, CNM04, New06, BGLL08, DA05, HW08]. After its initial introduction in 2004 by Newman & Girvan [NG04], its popularity quickly skyrocketed and most studies on community detection included modularity. However, after Fortunato & Barthélemy [FB07] showed that modularity suffers from the resolution limit, the meaningfullness of the communities discovered by modularity maximization became questionable.

From our research, we found that modularity remains by far the most popular metric. Conductance, expansion, normalized cut, avg-ODF and TPR are also sometimes used. The other metrics are rarely mentioned in recent literature on community detection.

3.4 Summary

In this chapter, we have discussed a number of different metrics thate measure the quality of communities. We have seen metrics based on cliques, density, cut size, degree, triangles, and the Potts model. We have made a comparison between these metrics. From this comparison, we have concluded that there are three types of metrics, metrics consider different characteristics of communities, not all metrics are equally well suitable for optimization and not all metrics are equally popular.

Additionally, we have drawn the following conclusions. **WCC** considers the most criteria: it measures internal and external connectivity, takes the size of communities into account and considers triangles. **WCC**, **FOMD** and **CPM** are the metrics most suitable for optimization. **CPM** is particularly interesting since it is the only metric of these three having a resolution parameter. **Modularity** is by far the most popular metric, although the meaningfulness of the communities found by modularity maximization is questionable due to the resolution-limit.

CHAPTER 4

Parallel Community Detection using Label Propagation

This chapter focuses on community detection using label propagation. We analyze at three different existing label propagation algorithms, we show how label propagation can be generalized to use any quality metric and adapted to expose massive parallelism. Finally, we compare the performance of parallel and sequential label propagation.

4.1 Background

Over the last decade, several community detection algorithms based on label propagation have been proposed. The idea behind label propagation is that each vertex carries a label which indicates the community the vertex belongs to. These labels are propagated throughout the network using an iterative process. Fig. 4.1 shows an example of label propagation on a small graph.

In this section, we will describe three of the most popular label propagation algorithms: LPA by Raghavan et al. [RAK07], the Louvian method by Blondel et al. [BGLL08], and SCD by Prat et al. [PPDSLP14].

4.1.1 The Label Propagation Algorithm (LPA)

The Label Propagation Algorithm (LPA) by Raghavan et al. [RAK07] was one of the first community detection algorithm that used label propagation. The algorithm works using a very simple iterative process. Initially, each vertex is assigned a unique label. In each iteration, all vertices are updated, one-by-one, in a random

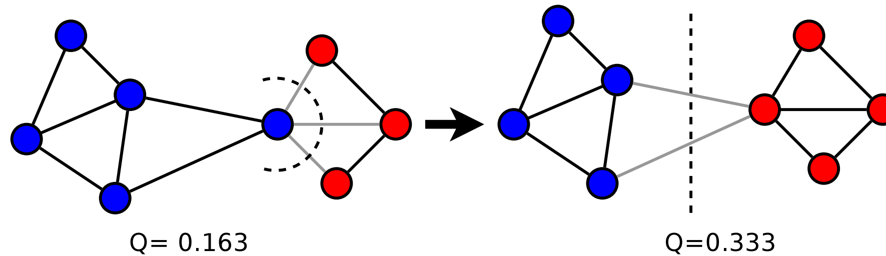


Figure 4.1: Example of label propagation. Transferring the middle vertex from the left community to the right community increases modularity.

order. This order needs to be randomly generated at the start of each iteration.

When a vertex gets updated, its label is changed to the label which is most common amongst its neighbors. If there are multiple labels to choose from, the vertex keeps its current label if it is one of the possible candidates, otherwise one candidate is chosen at random. Note that changing the label of a vertex affects the best label of its neighbors, so a vertex might need to change its label several times before it gets its final label.

The iterative process continues until a stable configuration is reached, i.e., an iteration occurs in which no vertex changes its label. The algorithm is guaranteed to terminate since the number of internal edges increases by at least one in each iteration and the total number of edges is finite. In practice, the algorithm converges very quickly: usually 95% of the vertices already have their final label after only 5 iterations [RAK07]. Its simplicity and fast convergence make this algorithm suitable for large networks.

4.1.2 The Louvain Method

The Louvain method by Blondel et al. [BGLL08] is a community detection algorithm based on modularity maximization. It consists of two phases which are alternated: the propagation phase and the aggregation phase. During the propagation phase, the labels are propagated throughout the network. During the aggregation phase, vertices having the same labels are merged into a single supervertex. The result of the algorithm is a hierarchy of communities. We will only describe the propagation phase, since this chapter does not focus on hierarchical communities.

Initially, each vertex gets a unique label. Similar to LPA, the propagation phase of the Louvain method consists of an iterative process in which vertices are updated, one-by-one, in a random order in every iteration. Every time a vertex gets updated, the gain in modularity when transferring the vertex to a neighboring community is evaluated for each neighboring community. The vertex is placed in the community for which the gain is maximal or, if no positive gain can be obtained, it stays in its

current community. This algorithm is slightly slower than LPA since its updating rule is more complex, but its results are better in terms of modularity.

4.1.3 Scalable Community Detection (SCD)

Prat et al. [PPDSLP14] presented a label propagation algorithm, named *Scalable Community Detection* (SCD), which is based on the maximization of the *weighted community clustering* (WCC) metric (Section 3.2.5). SCD consists of three phases: preprocessing, initial partitioning and refinement. During the first phase (preprocessing), all edges which do not close any triangles are removed. These triangles do not contribute to the WCC and removing them reduces memory consumption and increases performance.

During the second stage (initial partitioning), an initial partitioning of the network is generated by assigning a initial label to each vertex. This is done using the following heuristic. Initially, all vertices are unlabelled. Next, the vertices are visited in descending order of their clustering coefficient [1]. For each vertex which is still unlabelled, a new label is generated which gets assigned to the vertex and all its unlabelled neighbors. The intuition behind this heuristic is that if a vertex has a high clustering coefficient, the vertex and its neighbors are close to being a clique so it is likely that they belong to the same community.

Note that it not possible to simply assign a unique label to each vertex such as done by LPA and the Louvain method. WCC is a metric based on counting triangles. If each vertex has a unique label then improving the WCC using label propagation is useless since it is not possible to form a triangle by changing the label of a single vertex if all vertices have unique labels.

During the final phase (refinement), the labels are propagated using an iterative process. Each iteration consist of two steps: (1) select the best *action* for each vertex and (2) apply the selected best action for each vertex. There are three possible types of actions per vertex: keep the current label, copy the label of one of its neighbors or generate a fresh label. The "best" action is defined as the action which leads to the largest increase in WCC.

The algorithm terminates when the overall gain in WCC obtained in a single iteration is less than a given threshold. SCD is more computationally intensive than LPA and the Louvian method since calculating the WCC requires counting the triangles in the graph, which is an expensive operation.

[1] The clustering coefficient of a vertex is defined as the number of triangles the vertex closes with its neighbors divided by the total number of possible triangles it could close with its neighbors.

4.2 Label Propagation

In the previous sections, we have seen three different label propagation algorithms. Even though these algorithms might appear different on the surface, they use very similar approaches. In this section, we show how label propagation can be generalized to support any community quality metric. Next, we discuss parallelization and show how this is determined by the order in which vertices are updated.

4.2.1 Generalization

All three label propagation algorithms are based on two stages. During the first stage, an initial labelling of vertices is generated using a heuristic. LPA and the Louvain method simply assign a unique label to each vertex. SCD uses a heuristic based on the clustering coefficient. The second stage is an iterative process in which vertices get updated by applying one of the three following actions.

(1) **No action:** The vertex keeps its current label.

(2) **Transfer:** The vertex changes its label to the label of one of its neighboring communities.

(3) **Leave:** The vertex changes its label to a newly generated label.

For each action, a "benefit" needs to be defined and the action with the highest benefit is applied. For LPA, the benefit of action (2) is simply the number of edges towards the neighboring community, and the benefits of actions (1) and (3) are undefined. For the Louvain method, the benefit of action (1) is zero, the benefit of action (2) is the gain in modularity, and the benefit of action (3) is undefined. For SCD, the benefit of action (1) is zero and the benefit of action (2) and (3) is the gain in WCC.

In other words, the three label propagation algorithms can be seen as greedy optimization procedures which apply local actions to maximize a specific metric. This metric is defined as: the number of internal edges for LPA, modularity for the Louvain method, and WCC for SCD.

We can generalize this idea to any community quality metric f by defining the benefit for each type of action. Define $f_{no-action}(v)$ as the benefit of applying no action to vertex v, $f_{transfer}(v, C, C')$ as the benefit of transferring vertex v from community C to community C', and $f_{transfer}(v, C)$ as the benefit of transferring vertex v from community C to a singleton community $\{v\}$. Let $\mathcal{P}$ be a partitioning of a network. Formally, the benefit of these actions can be defined as follows:

$$
\begin{aligned}
f_{no-action}(v) &= 0 \\
f_{transfer}(v, C, C') &= f((\mathcal{P} \backslash \{C, C'\}) \cup \{C \backslash \{v\}, C' \cup \{v\}\}) - f(\mathcal{P}) \\
f_{leave}(v, C) &= f((\mathcal{P} \backslash \{C\}) \cup \{C \backslash \{v\}, \{v\}\}) - f(\mathcal{P})
\end{aligned}
$$

The last two functions can be simplified by defining these two actions in terms of a single operation. Let $f_{insert}(v, C)$ be the *insertion benefit* function: the gain in f when transferring v from singleton community $\{v\}$ to community C. The benefit of actions (2) and (3) can be simplified as follows.

$$
f_{transfer}(v, C, C') = f_{leave}(v, C) + f_{insert}(v, C') \tag{4.1}
$$

$$
f_{leave}(v, C) = -f_{insert}(v, C \backslash \{v\}) \tag{4.2}
$$

Eq. 4.2 holds since, intuitively, removing v from C is the inverse of inserting v into C. Eq. 4.1 holds since transferring v from C to C' is equivalent to removing v from C and then inserting v into C'. In Section 4.3, we will define the insertion benefit function for a number of metrics.

Additionally, we need to define tie breakers in case several different actions have the exact same benefit. We use the following rules.

1. Action (1) is always preferred over action (2). This prevents vertices from repeatedly changing their label if there are multiple communities to which they equally belong.
2. Action (2) is always preferred over action (3). This prevents many small singleton communities from being created.
3. If there are multiple actions (3) having the same benefit, one action is chosen at random.

4.2.2 Parallelization

There is the question of *how* the vertices are updated in every iteration. What strategy we use for updating the vertices determines how much parallelism the algorithm exposes and how quickly it converges.

LPA and the Louvain method update all vertices in every iteration one-by-one in a randomly generated order. We shall refer to this strategy as **sequential** updating. Eventually, a stable labelling will be reached in which every vertex has its final label. This strategy exposes little parallelism since only one vertex can be updated

at the time. It is not allowed to update the next vertex until the previous vertex has obtained a new label.

We will experiment with two strategies to parallelize label propagation. The first strategy is to, calculate in every iteration, the benefit of all possible actions for all vertices and only apply the most beneficial action over all possible actions. Only one action is applied during each iteration. We shall refer to this strategy as **greedy** updating. This strategy exposes a lot of parallelism since we can calculate the benefit of all possible actions in parallel and select the best one using a reduction operation. However, since only a single action is applied in every iteration, a huge number of iterations is required before a stable configuration is reached. This strategy is only feasible for small networks.

The second strategy is to, in every iteration, update all vertices in parallel, such done by SCD. We shall refer to this strategy as **parallel** updating. This strategy requires two stages: first select the most beneficial action for each vertex and then apply the selected action for each vertex. Parallel updating exposes massive amounts of parallelism compared to sequential updating.

However, a drawback of parallel updating is that it can lead to *oscillation*. This happens because each vertex decided the most beneficial action in isolation, but all vertices apply their actions simultaneously. For example, consider the scenario where vertex v wants to copy label x from vertex u and vertex u wants to copy label y from vertex v. If u and v both apply their actions simultaneously, they will swap their labels. Oscillation prevents a stable configuration from being reached and leads to poor quality communities compared to communities found by sequential updating. Oscillation does not occur with greedy or sequential updating since only one vertex at a time is updated.

4.3 Metrics

In the previous section, we showed how label propagation can be used for any community quality metric by defining the *insertion benefit* function $f_{insert}(v, C)$. In this section, we define the insertion benefit function for three different metrics: modularity [NP03], since it is the most popular metric, CPM [TVDN11], since it includes a resolution parameter, and WCC [PPDSBLP12], since it is a metric based on counting triangles instead of internal density.

4.3.1 Modularity

Recall from Section 3.2.6 that modularity is defined as in Eq. 4.3, where, for convenience, the total is normalized to the range -1 to $+1$ by dividing it by $2m$.

$$f(\mathcal{P}) = \frac{1}{2m} \sum_{u,v \in V} \left(A_{uv} - \frac{d(u)d(v)}{2m} \right) \delta(C_{i_u}, C_{i_v}) \tag{4.3}$$

The insertion benefit function can be defined as follows:

$$\begin{aligned}
f_{insert}(v, C) &= \frac{1}{2m} \sum_{u \in C} \left(A_{uv} - \frac{d(u)d(v)}{2m} \right) + \frac{1}{2m} \sum_{u \in C} \left(A_{vu} - \frac{d(v)d(u)}{2m} \right) && (4.4) \\
&= \frac{1}{m} \sum_{u \in C} \left(A_{uv} - \frac{d(u)d(v)}{2m} \right) && (4.5) \\
&= \frac{1}{m} \sum_{u \in C} A_{uv} - \frac{d(v)}{2m^2} \sum_{u \in C} d(u) && (4.6) \\
&= \frac{d_{in}(v, C)}{m} - \frac{d(v)d(C)}{2m^2} && (4.7)
\end{aligned}$$

Eq. 4.4 holds since only the edges between u and v are affected. The first rewrite step is possible since the graph is undirected, i.e. $A_{uv} = A_{vu}$. In Eq. 4.7, $d_{in}(v, C)$ is defined as the number of edges that vertex v has towards C and $d(C) = \sum_{u \in C} d(u)$. Note that from Eq. 4.7, we can conclude that it is only beneficial to insert v into C if $d_{in}(v, C) > d(v)d(C)/2m$. This equation was previously found by Newman [New04] and later again by Blondel et al. [BGLL08].

4.3.2 Constant Potts Model (CPM)

Recall from Section 3.2.6 that CPM is defined as in Eq. 4.8. Again, the total is divided by $\frac{1}{2m}$ to normalize the value.

$$f(\mathcal{P}) = \frac{1}{2m} \sum_{u,v \in V} (A_{uv} - \gamma) \delta(C_{i_u}, C_{i_v}) \tag{4.8}$$

The insertion benefit function for CPM is defined as:

$$
\begin{aligned}
f_{insert}(v, C) &= \frac{1}{2m}\sum_{u\in C}(A_{uv} - \gamma) + \frac{1}{2m}\sum_{u\in C}(A_{vu} - \gamma) && (4.9)\\
&= \frac{1}{m}\sum_{u\in C}(A_{uv} - \gamma) && (4.10)\\
&= \frac{1}{m}\sum_{u\in C}A_{uv} - \frac{|C|}{m}\gamma && (4.11)\\
&= \frac{d_{in}(v, C) - \gamma|C|}{m} && (4.12)
\end{aligned}
$$

The rewrite steps are similar to the ones used for modularity. From the final equation we can also conclude that insertion is only beneficial if $d_{in}(v, C)/|C| > \gamma$, i.e., if v is connected to at least a fraction γ of the vertices from C.

4.3.3 Weighted Community Clustering (WCC)

Recall from Section 3.2.5 that the WCC defines the cohesion of vertex v to community C as in Eq. 4.13.

$$
WCC(v, C) = \begin{cases} \frac{t(v,C)}{t(v,V)} \cdot \frac{vt(v,V)}{|C\setminus\{v\}|+vt(v,V\setminus C)} & \text{if } t(v, V) \neq 0 \\ 0 & \text{if } t(v, V) = 0 \end{cases} \quad (4.13)
$$

The quality of a partitioning is simply defined as the average cohesion over all vertices:

$$
f(\mathcal{P}) = \frac{1}{|V|}\sum_{v\in V} WCC(v, C_{i_v}) \quad (4.14)
$$

The insertion benefit function can now be defined as follows:

$$f_{insert}(v, C) = \frac{1}{|V|} WCC(v, C \cup \{v\}) + \frac{1}{|V|} \sum_{u \in C} (WCC(u, C \cup \{v\}) - WCC(u, C)) \tag{4.15}$$

$$= \frac{1}{|V|} \cdot \frac{t(v, C \cup \{v\})}{t(v, V)} \cdot \frac{vt(v, V)}{|C| + vt(v, V \backslash (C \cup \{v\}))} \tag{4.16}$$

$$+ \frac{1}{|V|} \sum_{u \in C} \left(\frac{t(u, C \cup \{v\})}{t(u, V)} \cdot \frac{vt(u, V)}{|C| - 1 + vt(u, V \backslash (C \cup \{v\}))} \right) \tag{4.17}$$

$$- \frac{1}{|V|} \sum_{u \in C} \left(\frac{t(u, C)}{t(u, V)} \cdot \frac{vt(u, V)}{|C| - 1 + vt(v, V \backslash C)} \right) \tag{4.18}$$

Unfortunately, the final equation cannot be simplified any further. This equation is computationally expensive since it requires one to count the number of triangles v closes with C and V and the number of triangles each $u \in C$ closes with C, $C \cup \{v\}$, and V. The complexity of this operation for a single vertex is $\mathcal{O}(d^2)$ where d is the degree of this vertex. Since this function will be called many times per iteration, the exact computation of this function is not computationally feasible.

Prat et al. encountered the same problem when they designed SCD. To solve this problem, they proposed a model which can estimate the improvement in WCC when inserting a vertex into a community using a constant time formula which only relies on statistics of the communities. This model is based on the following assumptions:

- Every edge closes at least one triangle.
- The internal density of C is homogeneous.
- The clustering coefficient is homogeneous for all the vertices outside C.

Next, they define the following parameters:

- ω: Average clustering coefficient of the graph
- n: Total number of vertices of the network.
- r: Number of vertices of C.
- b: Number of edges at the boundary of C.
- δ: Internal density of C.
- d_{in}, d_{out}: Internal/external degree of v.
- $q = (b - d_{in})/r$

Table 4.1: Networks used for evaluation.

Name	Vertices	Edges	Avg. Deg	Max. Deg.	Type
AstroPhysics	18,772	198,110	21.1	504	Collaboration Network
Email	36,692	183,831	10.0	1,383	Communication Network
Amazon	334,863	925,872	5.5	549	Co-purchasing Network
Stanford	281,903	2,312,497	16.4	38,625	Web Network
YouTube	1,134,890	2,987,624	5.3	28,754	Social Network

Using these definitions, Prat et al. [PPDSBLP12] approximated the value of $f_{insert}(v, C)$ as shown in Eq. 4.19. Conceptually, the approximation consists of three terms which represent the WCC improvement for vertices in C connected to v, vertices in C not connected to v and v itself respectively. We will use this approximation for the evaluation of our algorithm.

$$f_{insert}(v, C) \approx \tag{4.19}$$
$$\frac{d_{in}}{n} \cdot \frac{((r-1)\delta + 1 + q)(d_{in} - 1)\delta}{(r+q)((r-q)(r-2)\delta^3 + (d_{in}-1)\delta + q(q-1)\delta\omega + q(q-1)\omega + d_{out}\omega)} +$$
$$\frac{r - d_{in}}{n} \cdot \frac{(r-1)(r-2)\delta^3}{(r-1)(r-2)\delta^3 + q(q-1)\omega + q(r-1)\delta\omega} \cdot \frac{(r-1)\delta + q}{(r+q)(r-1+q)} +$$
$$\frac{1}{n} \cdot \frac{d_{in}(d_{in}-1)\delta}{d_{in}(d_{in}-1)\delta + d_{out}(d_{out}-1)\omega + d_{out}d_{in}\omega} \cdot \frac{d_{in} + d_{out}}{r + d_{out}}$$

4.4 Evaluation

We have evaluated label propagation for the three updating strategies from Section 4.2 and the three metrics from Section 4.3. Table 4.1 lists the graphs used for the evaluation. These datasets have been selected from the SNAP repository [LK14] because they have different properties in terms of size, density and origin. All results presented in this section are averaged over 5 runs since our algorithm involves randomness. However, error bars have been omitted since the errors are negligible.

Fig. 4.2, Fig. 4.3, Fig. 4.4, Fig. 4.5 show the scores obtained using label propagation versus the number of updates performed for modularity, CPM for $\gamma = 0.5$, CPM for $\gamma = 0.1$ and WCC. These figures only show the results for Email and Amazon. The results for the remaining datasets are similar. Fig. 4.6, Fig. 4.7, Fig. 4.8 and Fig. 4.9 show the scores obtained after convergence for all datasets. For parallel updating, the algorithm would rarely converge so the number of iterations was limited to 50. For WCC, the heuristic from SCD was used to create an initial partitioning, because otherwise label propagation would be ineffective.

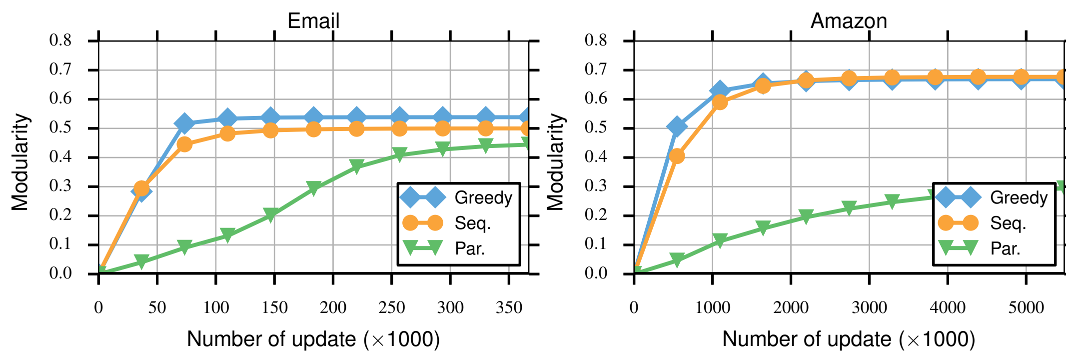


Figure 4.2: Modularity versus number of updates for Email and Amazon. Higher is better.

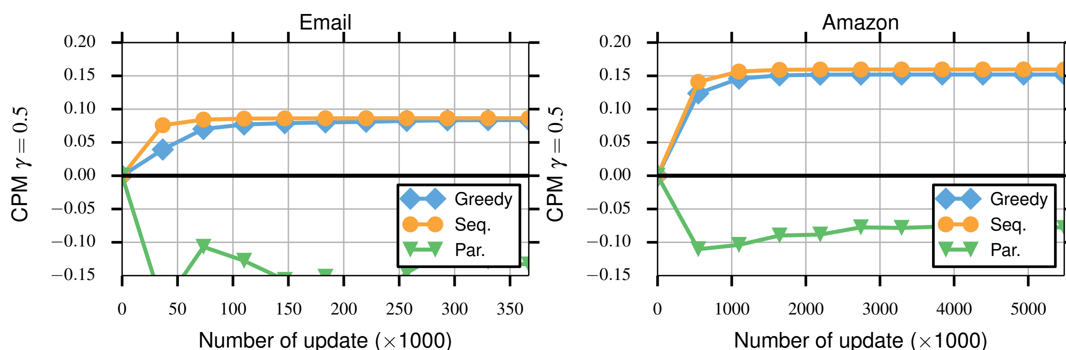


Figure 4.3: CPM for $\gamma = 0.5$ versus number of updates for Email and Amazon. Higher is better.

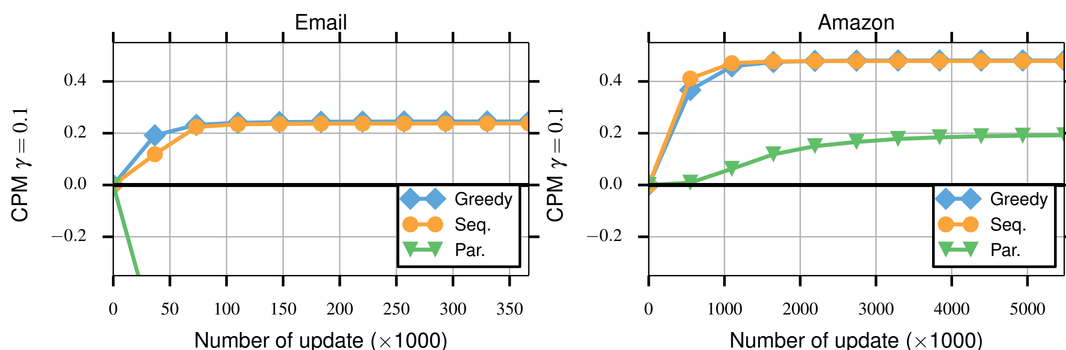


Figure 4.4: CPM for $\gamma = 0.1$ versus number of updates for Email and Amazon. Higher is better.

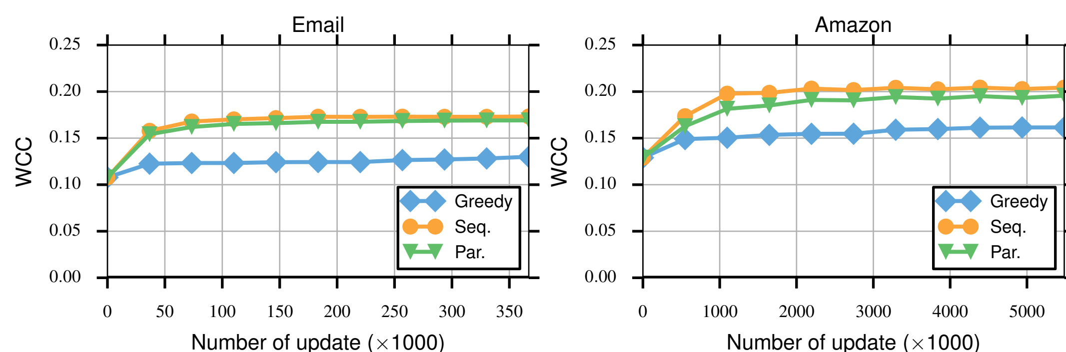


Figure 4.5: WCC versus number of updates for Email and Amazon. Higher is better.

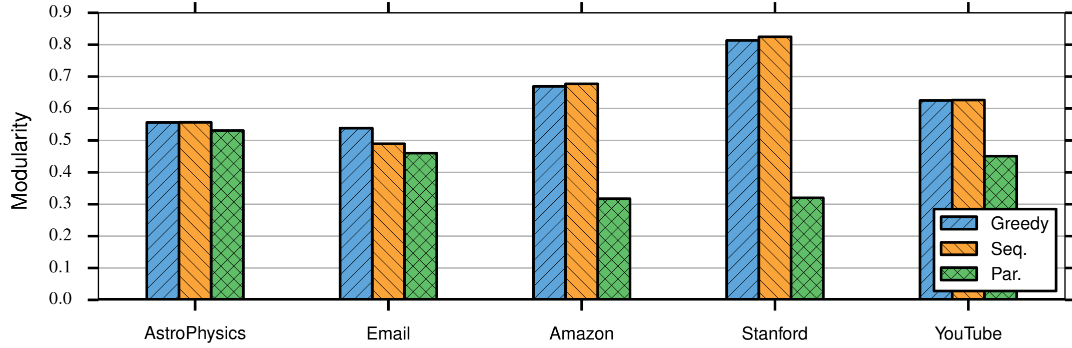


Figure 4.6: Highest modularity reached for different graphs. Higher is better.

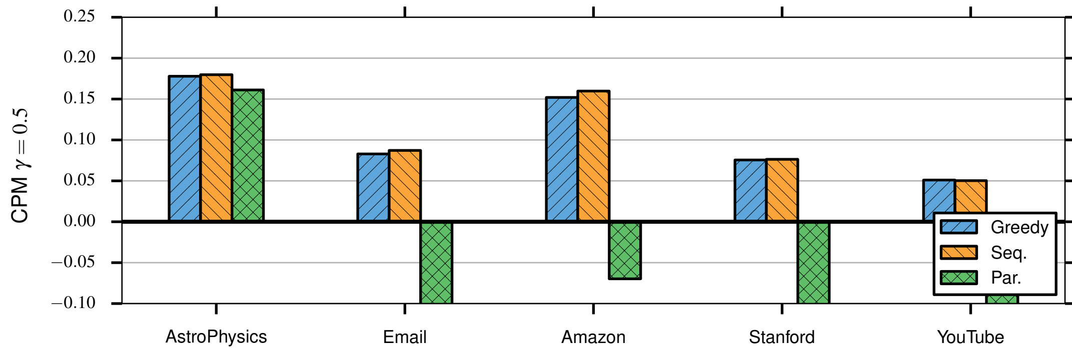


Figure 4.7: Highest CPM reached for $\gamma = 0.5$ for different graphs. Higher is better.

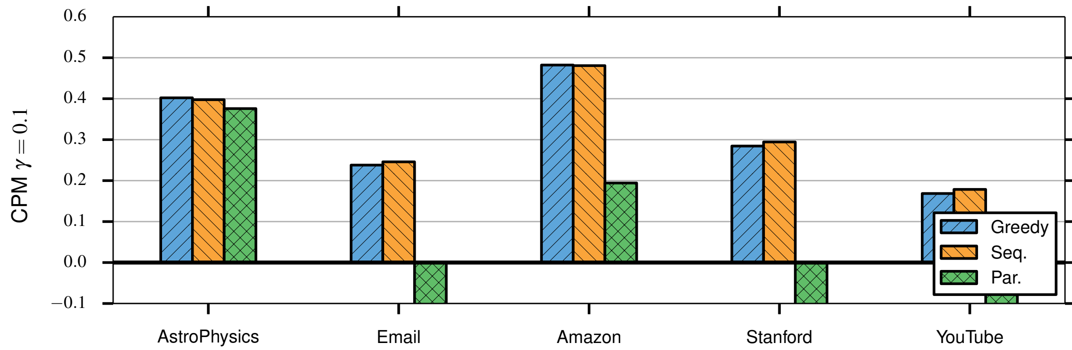


Figure 4.8: Highest CPM reached for $\gamma = 0.1$ for different graphs. Higher is better.

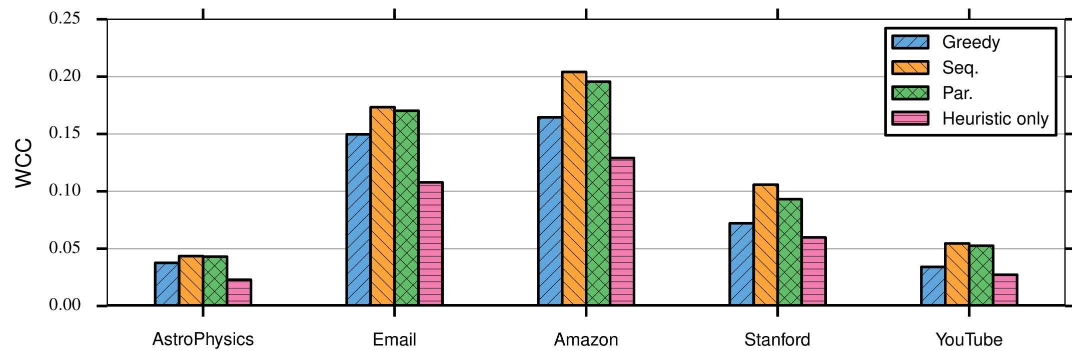


Figure 4.9: Highest WCC reached for different graphs. Higher is better.

We will first discuss modularity. Fig. 4.2 and Fig. 4.6 show that greedy and sequential updating perform equally well. Fig. 4.2 shows that they converge at the same rate and Fig. 4.6 shows that the difference in modularity after convergence is negligible (less than 5%). Parallel updating, on the other hand, performs poorly compared to greedy and sequential updating. Fig. 4.6 shows that parallel updating always obtained a lower modularity than greedy and sequential updating. In some cases, the difference is more than 50%.

Next, we consider CPM. Fig. 4.3, Fig. 4.4, Fig. 4.7 and Fig. 4.8 show, again, that greedy and sequential updating perform similarly for both $\gamma = 0.1$ and $\gamma = 0.5$. The parallel strategy, again, performs very poorly compared to the greedy and sequential strategy. In some cases, the CPM even becomes negative. This means that the partitioning found by label propagation is actually worse than a naive partitioning where every vertex has a unique label.

WCC, on the other hand, shows entirely different results. Fig. 4.5 and Fig. 4.9 show that sequential and parallel updating perform similarly. The greedy strategy gives poor results compared to the other two strategies. Fig. 4.9 also shows that only using the initial partition heuristic and omitting propagation already gives high WCC. Label propagation only increases the WCC by roughly 50%.

From these observations, we conclude that sequential updating is the best updating strategy for all three metrics. Greedy updating is also suitable for modularity and CPM, but sequential updating is preferred over greedy updating since it is less computationally expensive. Parallel updating can also be used for maximizing WCC, as the difference in WCC obtained using sequential or parallel updating is negligible.

As discussed in Section 4.2.2, oscillation is the reason why parallel updating performs poorly compared to greedy and sequential updating. To prevent oscillation, we must break the symmetry of the algorithm. In the scenario where u and v repeatedly swap each other's label, only updating one of the vertices will break the symmetric and stop oscillation. This can be achieved by only updating a *fraction* of the vertices in each iteration. This idea was previously attempted by Cordasco et al. [CG10]. We shall refer to this strategy as **semi-parallel** updating.

Semi-parallel updated is essentially a compromise between sequential and parallel updating. The trade-off between the two is defined by parameter p, the fraction of vertices which are updated in each iteration. Thus, semi-parallel updating with $p = 1/n$ is similar to sequential updating since only one vertex is updated at a time. Semi-parallel updating with $p = 1$ is equivalent to parallel updating since all vertices are updated in parallel. By varying the values of p between $1/n$ and 1, we can vary between sequential and parallel updating. Increasing the value of p increases the stability of the propagation process. Decreasing the value of p increases the amount of parallelism.

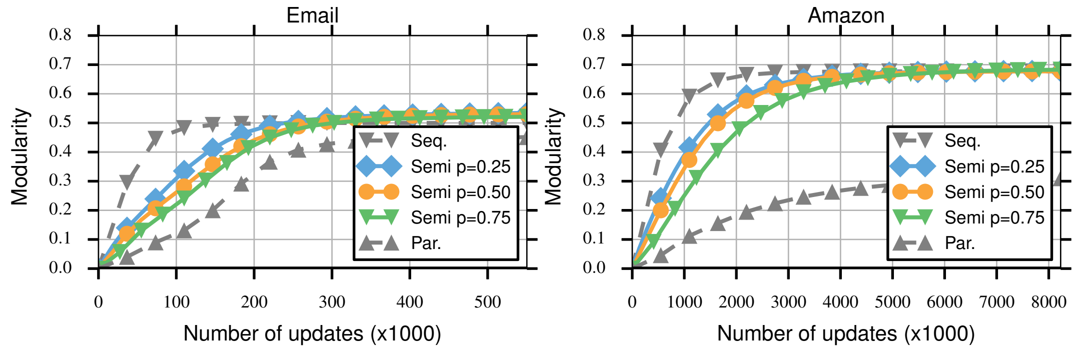


Figure 4.10: Modularity for different graphs. Higher is better.

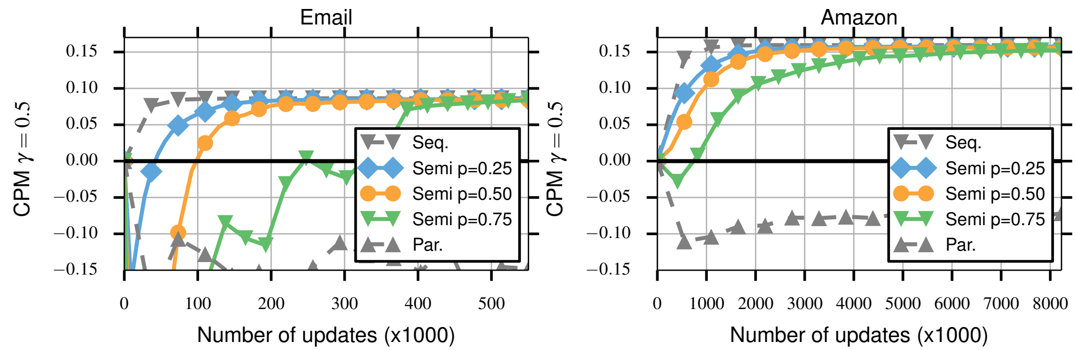


Figure 4.11: CPM for $\gamma = 0.5$ for different graphs. Higher is better.

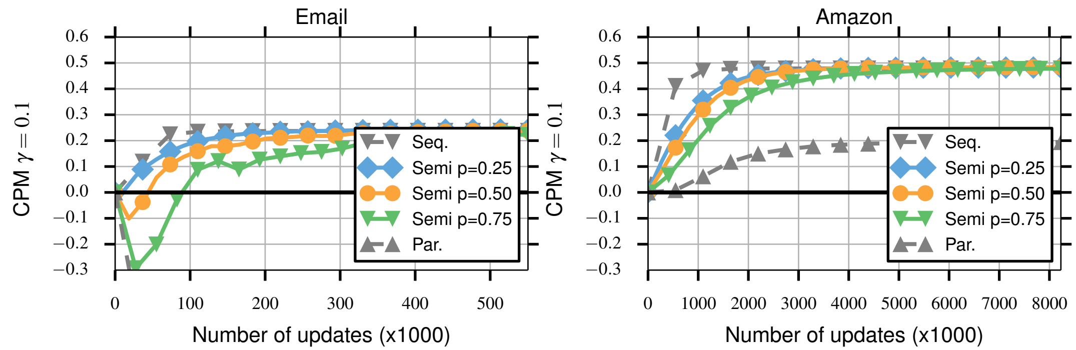


Figure 4.12: CPM for $\gamma = 0.1$ for different graphs. Higher is better.

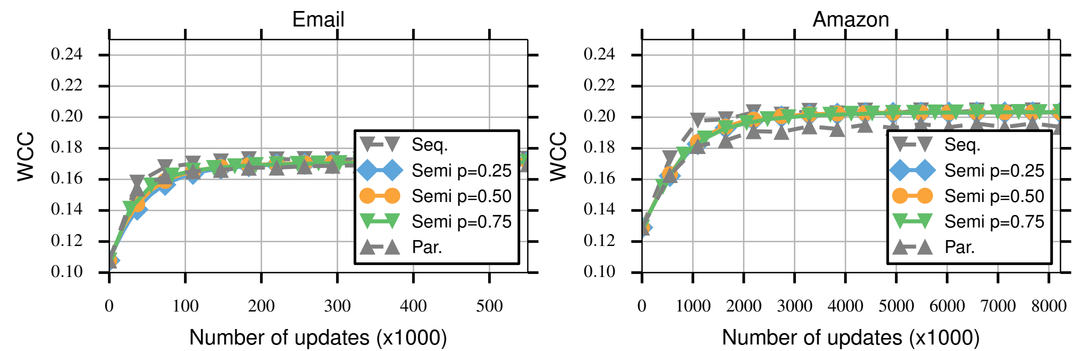


Figure 4.13: WCC for different graphs. Higher is better.

Fig. 4.10, Fig. 4.11, Fig. 4.12 and Fig. 4.13 show the effectiveness of semi-parallel updating for Email and Amazon for $p = 0.25$, $p = 0.5$ and $p = 0.75$ on the three metrics. These results show that the scores obtained by semi-parallel updating are similar to those obtained by sequential updating. In all cases, semi-parallel updating performs much better than parallel updating. Lowering the value of p seems to only cause the algorithm to converge quicker (i.e., less total updates are required to converge), but the final score obtained is not affected.

4.5 Summary

In this chapter, we discussed community detection using label propagation. We explored how label propagation can be generalized to be used for any community quality metric. We further compared three different strategies for updating the vertices in every iteration: **greedy**, **sequential** and **parallel** updating. Sequential updating is the best strategy of these three in terms of quality. Parallel updating suffers from oscillation, which can be mitigated by **semi-parallel** updating: a new strategy that only updates a fraction of the vertices in each iteration. Results show that this strategy gives the best of both worlds: the quality of sequential updating and the parallelism of parallel updating.

CHAPTER 5

Parallel Communities Detection Using Merging

This chapter focuses on community detection using merging. We look at one of the most well-known merging-based algorithms (Newman's greedy algorithm [New04]) and we propose a novel merging approach which exposes more parallelism than Newman's approach. We will evaluate our new approach by comparing it against Newman's approach.

5.1 Background

In the previous chapter, we discussed community detection using label propagation. Label propagation is a fast and efficient method for community detection since it only involves *local* actions, i.e., each vertex is updated by only inspecting the vertices in its local neighborhood. However, a major drawback of local actions is that is possible to get "stuck" in a local optimum where changing the label of any individual vertex is no longer beneficial even though the global optimum is not yet reached. Fig. 5.1 shows an example of a local optimum. In this example, changing the label of a single vertex in isolation does not increase modularity, however modularity can still be improved by *merging* the two communities at the bottom.

Merging is a well-known technique used for community detection and several merging-based algorithms have been proposed [New04, CNM04, ZL04, PL05, WT07]. The most well-known algorithm is the greedy algorithm by Newman [New04] which is based on modularity optimization. Initially, the algorithm considers each vertex to be a singleton community. Next, the improvement in modularity when merging two communities is evaluated for each pair of communities connected by at least one edge. The two communities having the highest increase are merged. This step

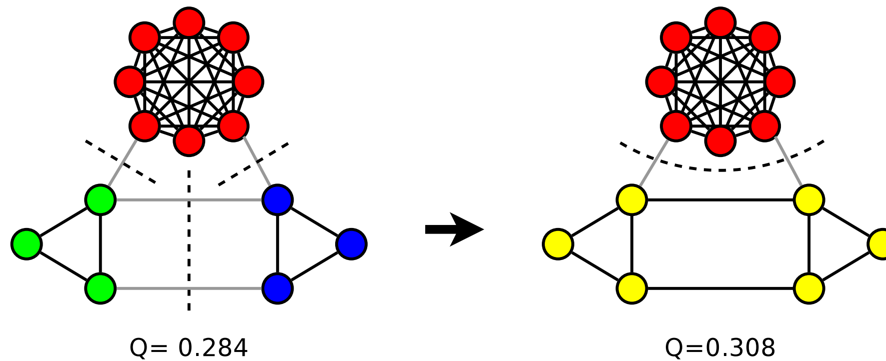


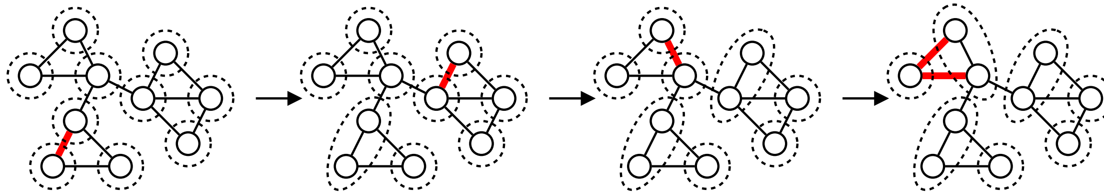
Figure 5.1: Example of local optimum for label propagation.

is repeated until merging is no longer beneficial.

Newman used two data structures for his algorithm: a vector D, where D_i is the total degree of community i, and a square matrix E, where E_{ij} is the number of edges between communities i and j. These data structures are initialized once at the start of the algorithm. In every iteration, the pairs of connected communities are found by iterating over the non-zero entries of E, the improvement in modularity when merging each pair is calculated, and the best pair is selected. Newman showed how the improvement in modularity when merging a pair of communities can be calculated in constant time using statistics from D and E. The selected communities i and j are merged and the data structures are updated. Newman argued that the total run-time is $\mathcal{O}(n(n+m))$, since each iteration takes $\mathcal{O}(n+m)$ time and the maximum number of iterations is $n-1$.

Clauset et al. [CNM04] later proposed a faster implementation of Newman's algorithm. They observed that when communities i and j are merged, the increase in modularity for all pairs of communities that do not involve i or j are not affected. Instead of storing the number of edges between every pair of communities in dense matrix E, they used a sparse matrix ΔQ to store the increase in modularity for every pair of *connected* communities. When communities i and j are merged into a new community k, *only* the values for ΔQ_{kl} need to be calculated for each community l, which was connected to i, j, or both. Additionally, a max-heap was added to select the pair of communities with the largest increase from ΔQ in logarithmic time. These modifications to the algorithm significantly reduced the amount of computation per iteration. Clauset et al. argued that the worst-case run-time of their algorithm is $\mathcal{O}(n(\log n)^2)$ for sparse networks.

Wakiti & Tsurumi [WT07] later presented an improved version of the implementation by Clauset et al. Wakiti & Tsurumi removed the sparse matrix ΔQ and, instead, stored the neighbors of each community as a linked list. Merging two communities comes down to merging the two linked lists for these communities. This improvement increased performance, although the complexity remains unchanged. The implementation by Wakiti & Tsurumi is considered to be the state-of-the-art implementation of Newman's algorithm [For10].



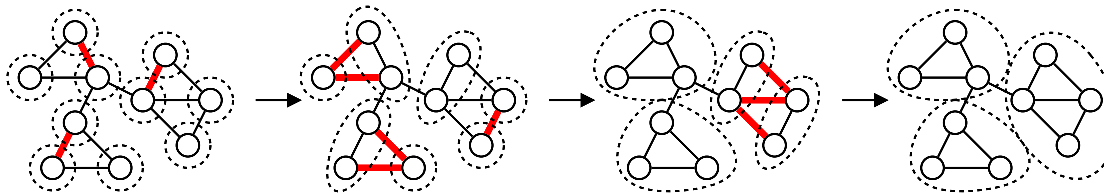


Figure 5.2: The difference between sequential and parallel merging. Solid circles indicates vertices, dashed circles indicate communities and thick lines indicate edges between communities that will be merged in the next iteration.

5.2 Parallel Merging

In this section, we will demonstrate how the algorithm by Newman can be generalized to support any quality metric and how it can be adapted to expose massive parallelism.

5.2.1 Generalization

The algorithm by Newman is essentially a greedy optimization process which attempts to maximize modularity by merging, in every iteration, the pair of communities which give the largest gain in modularity. It is possible to generalize this procedure for any quality metric. To be able to achieve this, we need to define a *merging benefit* function which gives the benefit for metric f when merging two communities. Let $\mathcal{P}$ be a partitioning of a network, $X, Y \in \mathcal{P}$ and $\mathcal{P}'$ the partitioning after merging X and Y. Now, $f_{merge}(X, Y)$ is defined as follows.

$$f_{merge}(X, Y) = f(\mathcal{P}') - f(\mathcal{P}) \tag{5.1}$$

In Section 5.3, we will show how the merging benefit function can be defined for a number of metrics.

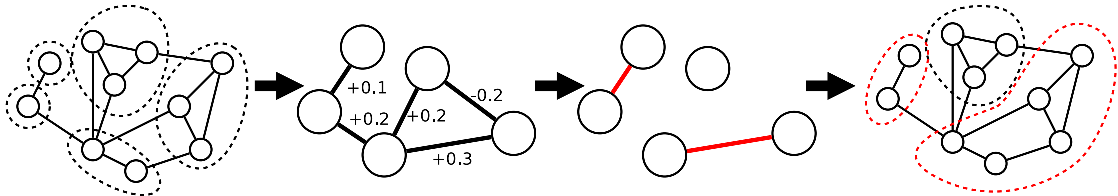


Figure 5.3: Step-by-step decomposition of one iteration of parallel merging.

5.2.2 Parallelization

The algorithm by Newman is computationally expensive. Parallelism can be used reduce the run-time of the algorithm. However, the fast implementations by Clauset et al. and Wakiti & Tsurumi are difficult to parallelize because these implementations use sparse data structures, such as binary trees, linked lists and heaps, which are inherently sequential and thus difficult to parallelize.

The original implementation by Newman is easier to parallelize since the benefit of merging two communities for every pair in the matrix E can be evaluated in parallel. However, the algorithm by Newman is naive since it recalculates the benefit for all possible pairs in every iteration, which is expensive. The implementations by Clauset et al. and Wakiti & Tsurumi only perform a tiny fraction of the amount of work Newman's implementation performs in every iteration.

To increase the amount parallelism, we will explore a different approach. Instead of reducing the amount of work in every iteration, we will attempt reduce the total number of iterations by merging more than one pair of communities in every iteration. A similar idea was previously explored by Riedy et al. [RMEB12]. We shall refer to this approach as **parallel** merging, as opposed to Newman's approach which we shall refer to as **sequential** merging. Fig. 5.2 demonstrates the difference between sequential and parallel merging.

Parallel merging works using an iterative process. In every iteration, a *super-graph* is created by collapsing all vertices which belong to the same community into a *super-vertex*. An *super-edge* exists between two super-vertices if there is at least one edge between the corresponding communities in the original network. Every super-edge is assigned a weight which correspond to the benefit or merging the two endpoints ($f_{merge}(X, Y)$). Super-edges having a negative weight are discarded.

Next, the pairs of communities which will be merged are selected. We only allow each community be merged with at most one adjacent community. This means that if community A has two adjacent communities B and C, then it can be merged with either community B or community C, but not both. Since each pair of communities corresponds to an edge in our super-graph, the problem of select the merging is equivalent to selecting an *matching*[1] of the super-graph.

[1] In graph theory, a *matching* or *independent edge set* of a graph is a set of edges such that no two edges share an endpoint.

We will consider two strategies for selecting this matching.

(1) **Greedy:** The edges of the super-graph are sorted in descending order of their weights and enumerate in this order. Each edge is added to the matching if none of its endpoints are already in the matching.

(2) **Best-neighbor:** Every community select the adjacent community as merging partner for which the merging benefit is maximal. If there are multiple candidates, one is chosen at random. If two communities select each other as partner, the pair is added to the matching.

We expect that the greedy strategy will most likely converge quicker than the best-neighbor strategy since the gain in every iteration is higher. However, the best-neighbor strategy will most likely find communities of higher quality than the greedy strategy since each community is only merged with its neighbor if this is the best option.

After selecting the matching, all pairs of communities in the matching are merged and the next iteration begins. The iterative process terminates when an iteration occurs in which no pairs are merged. The algorithm is guaranteed to converge since the number of communities decreases in every iteration and the number of communities is finite.

5.3 Metrics

In Section 5.2, we showed how the algorithm by Newman can be generalized for any quality metric by defined a *merging benefit* function. In this section, we shall define the merging benefit function for the same metrics as the previous section: modularity [NG04], CPM [TVDN11] and WCC [PPDSBLP12].

5.3.1 Modularity

Recall from Section 3.2.6 that modularity is defined as in Eq. 5.2.

$$f(\mathcal{P}) = \frac{1}{2m} \sum_{u,v \in V} \left(A_{uv} - \frac{d(u)d(v)}{2m} \right) \delta(C_{i_u}, C_{i_v}) \tag{5.2}$$

The merging benefit function can be defined as follows:

$$f_{merge}(X,Y) = \frac{1}{2m}\sum_{u\in X, v\in Y}\left(A_{uv} - \frac{d(u)d(v)}{2m}\right) + \frac{1}{2m}\sum_{u\in Y, v\in X}\left(A_{uv} - \frac{d(u)d(v)}{2m}\right) \tag{5.3}$$

$$= \frac{1}{m}\sum_{u\in X, v\in Y}\left(A_{uv} - \frac{d(u)d(v)}{2m}\right) \tag{5.4}$$

$$= \frac{1}{m}\sum_{u\in X, v\in Y} A_{uv} - \frac{1}{2m^2}\sum_{u\in X, v\in Y} d(u)d(v) \tag{5.5}$$

$$= \frac{m_{XY}}{m} - \frac{d(X)d(Y)}{2m^2} \tag{5.6}$$

Eq. 5.3 is valid since only the edges between X and Y are affected. The first rewrite step is allowed because the graph is undirected, i.e. $A_{uv} = A_{vu}$. In Eq. 5.6, $d(X)$ is defined as the total degree of community X and m_{XY} is the number of edges between X and Y. We see that merging X and Y is only beneficial if the number of edges between X and Y (m_{XY}) is greater than the expected number of edges ($\frac{d(X)d(Y)}{2m}$).

5.3.2 Constant Potts Model (CPM)

Recall from Section 3.2.6 that CPM is defined as in Eq. 5.7.

$$f(\mathcal{P}) = \frac{1}{2m}\sum_{u,v\in V}(A_{uv} - \gamma)\delta(C_{i_u}, C_{i_v}) \tag{5.7}$$

The merging benefit function can be defined as follows:

$$f_{merge}(X,Y) = \frac{1}{2m}\sum_{u\in X, v\in Y}(A_{uv} - \gamma) + \frac{1}{2m}\sum_{u\in Y, v\in X}(A_{uv} - \gamma) \tag{5.8}$$

$$= \frac{1}{m}\sum_{u\in X, v\in Y}(A_{uv} - \gamma) \tag{5.9}$$

$$= \frac{1}{m}\sum_{u\in X, v\in Y} A_{uv} - \frac{|X||Y|\gamma}{m} \tag{5.10}$$

$$= \frac{m_{XY} - \gamma|X||Y|}{m} \tag{5.11}$$

The rewrite steps are similar to those used for modularity. In the final equation, we can see that merging is only beneficial if the edge density between X and Y is greater than γ, i.e., $\frac{m_{XY}}{|X||Y|} > \gamma$.

5.3.3 Weighted Community Clustering (WCC)

Recall from Section 3.2.5 that the WCC defines the cohesion of vertex v to community C as in Eq. 5.12.

$$WCC(v,C) = \begin{cases} \frac{t(v,C)}{t(v,V)} \cdot \frac{vt(v,V)}{|C\backslash\{v\}|+vt(v,V\backslash C)} & \text{if } t(v,V) \neq 0 \\ 0 & \text{if } t(v,V) = 0 \end{cases} \tag{5.12}$$

The quality of an entire partition is defined as the average cohesion over all vertices to their communities. The merging benefit function for WCC is as follows:

$$f_{merge}(X,Y) = \sum_{v\in X}(WCC(v,X\cup Y) - WCC(v,X)) + \sum_{v\in Y}(WCC(v,X\cup Y) - WCC(v,Y)) \tag{5.13}$$

$$= \frac{1}{|V|}\sum_{v\in X}\frac{t(v,X\cup Y)}{t(v,V)} \cdot \frac{vt(v,V)}{|(X\cup Y)\backslash\{v\}| + vt(v,V\backslash(X\cup Y))} \tag{5.14}$$

$$- \frac{1}{|V|}\sum_{v\in X}\frac{t(v,X)}{t(v,V)} \cdot \frac{vt(v,V)}{|X\backslash\{v\}| + vt(v,V\backslash X)} \tag{5.15}$$

$$= \frac{1}{|V|}\sum_{v\in Y}\frac{t(v,X\cup Y)}{t(v,V)} \cdot \frac{vt(v,V)}{|(X\cup Y)\backslash\{v\}| + vt(v,V\backslash(X\cup Y))} \tag{5.16}$$

$$- \frac{1}{|V|}\sum_{v\in Y}\frac{t(v,Y)}{t(v,V)} \cdot \frac{vt(v,V)}{|Y\backslash\{v\}| + vt(v,V\backslash Y)} \tag{5.17}$$

Similar to the insertion benefit function for WCC (Section 4.3), this equation is computationally expensive since it requires one to count the number of triangles each vertex from X and Y closes with X, Y, $X\cup Y$ and V. In collaboration with Prat et al., we have extended the constant-time approximation model they designed for SCD to support merging of communities (see Appendix B). This approximation will be used for the evaluation.

5.4 Evaluation

We have evaluated parallel merging for the two merging strategies (Section 5.2.2) and the three quality metrics (Section 5.3) on the same datasets as used during the evaluation of label propagation (Table 4.1). All results in this section are the average of 5 runs since some randomness is part of the algorithm. Error bars have been omitted since the errors were found to be negligible.

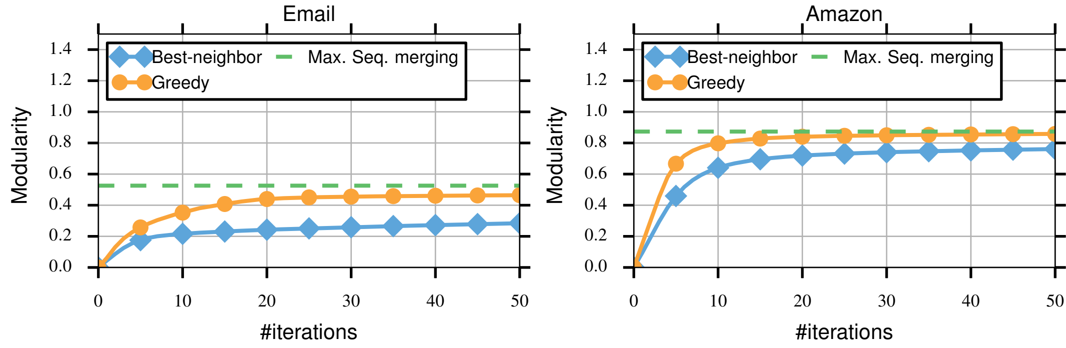


Figure 5.4: Modularity versus number of iterations of parallel merging for Email and Amazon. Higher is better

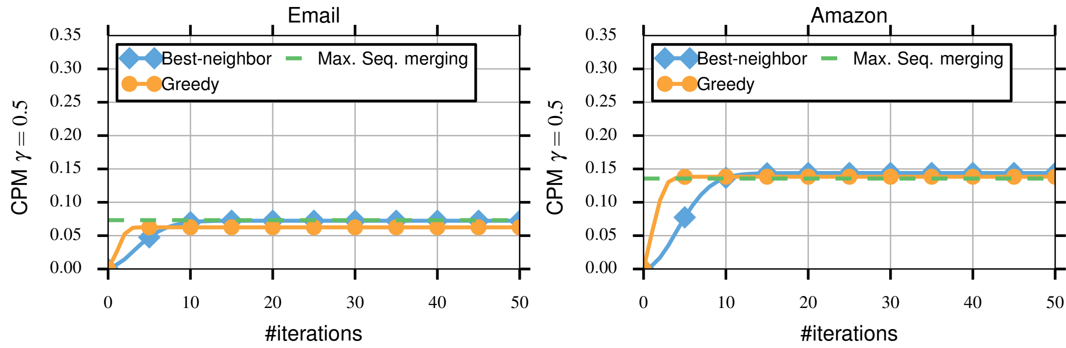


Figure 5.5: CPM for $\gamma = 0.5$ versus number of iterations of parallel merging for Email and Amazon. Higher is better

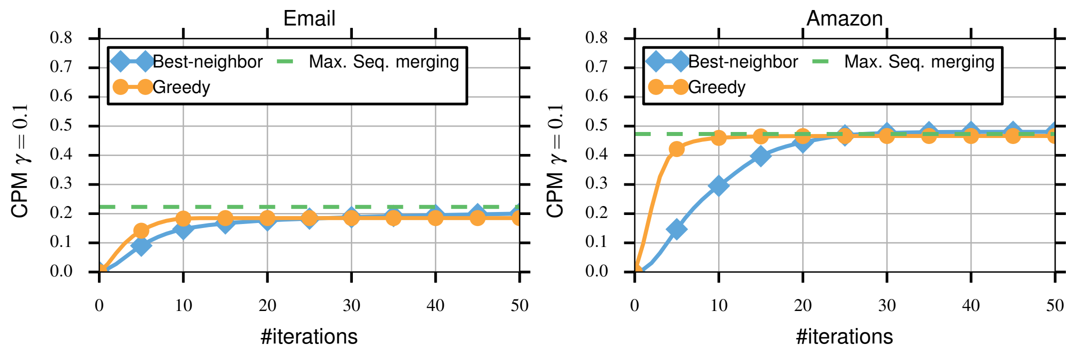


Figure 5.6: CPM for $\gamma = 0.1$ versus number of iterations of parallel merging for Email and Amazon. Higher is better

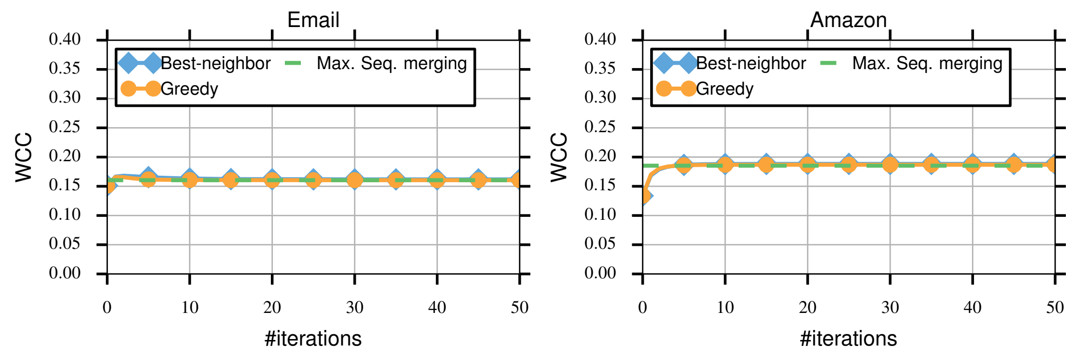


Figure 5.7: WCC versus number of iterations of parallel merging for Email and Amazon. Higher is better

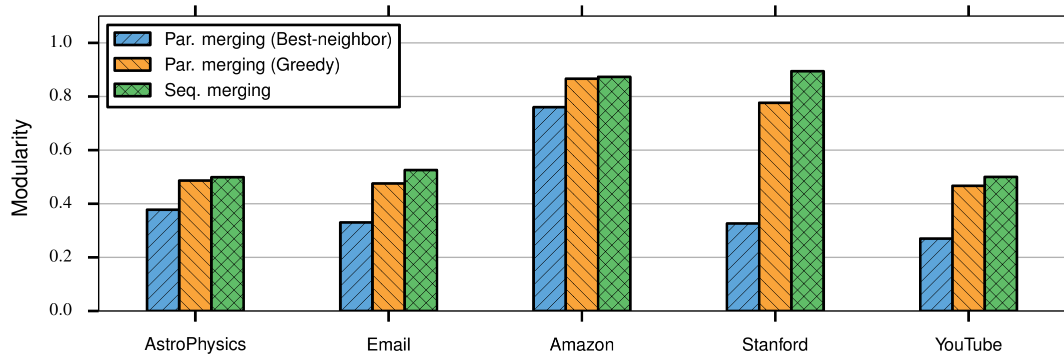


Figure 5.8: Modularity reached for different graphs. Higher is better

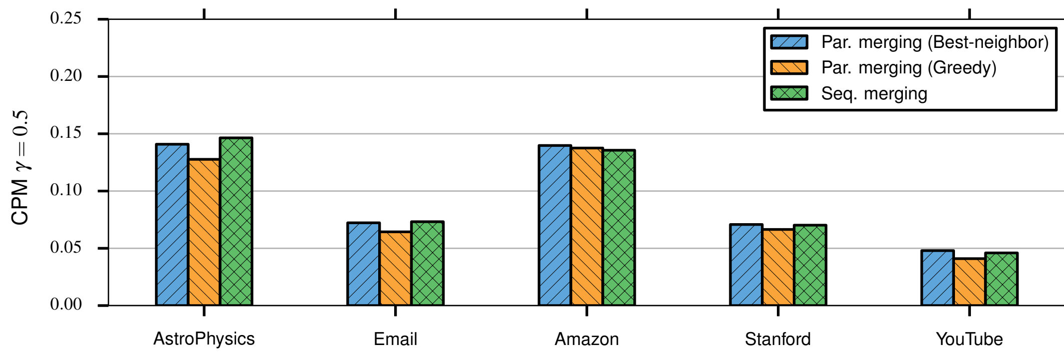


Figure 5.9: CPM for $\gamma = 0.5$ reached for different graphs. Higher is better

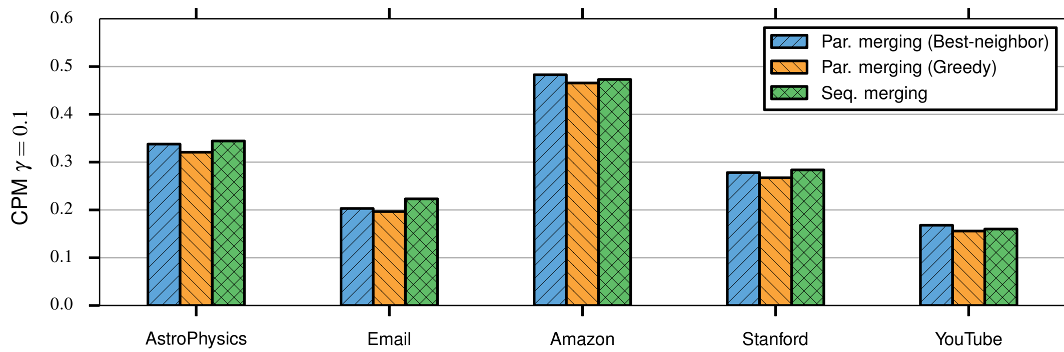


Figure 5.10: CPM for $\gamma = 0.1$ reached for different graphs. Higher is better

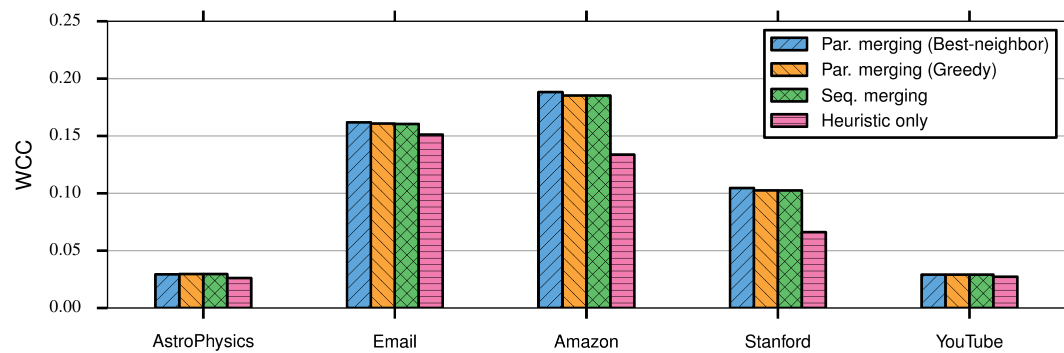


Figure 5.11: WCC reached for different graphs. Higher is better

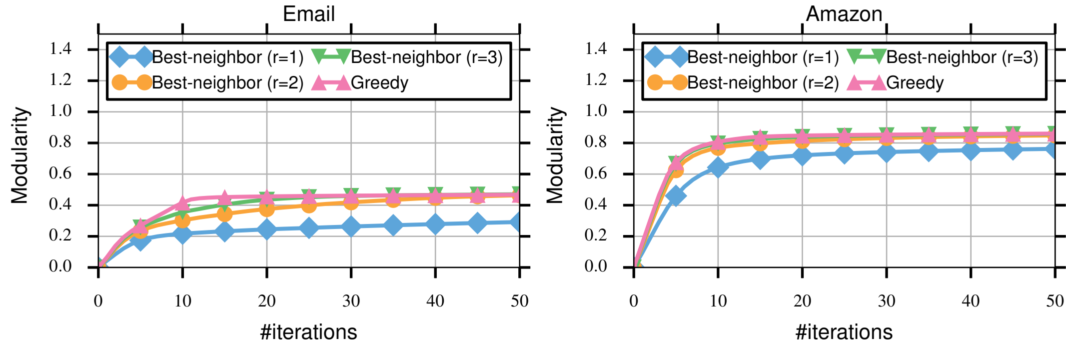


Figure 5.12: Modularity reached for different graphs when using best-neighbor parallel merging with multiple rounds. Higher is better

Fig. 5.4, Fig. 5.5, Fig. 5.6 and Fig. 5.7 shows scores obtained versus the number of iterations performed for modularity, CPM for $\gamma = 0.5$, CPM for $\gamma = 0.1$ and WCC. Only the results for Email and Amazon are included. These graphs show that 50 iterations are sufficient to reach convergence. Fig. 5.8, Fig. 5.9, Fig. 5.10 and Fig. 5.11 show the score obtained for all graphs after 50 iterations. In all figures, the maximal score obtained using sequential merging has been included for comparison.

Fig. 5.4 and Fig. 5.8 show that, for modularity, greedy parallel merging performs almost as good as sequential merging. The best-neighbor strategy converges much slower than the greedy strategy and obtains a lower score after 50 iterations. For example, after 10 iterations, the modularity is only 0.21 for the best-neighbor strategy and 0.34 for greedy parallel merging. After 50 iterations, the modularity is 0.33 and 0.48, respectively. It is clear that greedy parallel merging should be preferred when optimizing for modularity.

Fig. 5.9 and Fig. 5.10 show that for CPM, best-neighbor parallel merging and sequential merging obtain roughly the same scores. The greedy strategy obtains lower scores, but the difference is always less than 5%, which is not significant. On the other hand, Fig. 5.5 and Fig. 5.6 show that the greedy strategy converges twice as fast compared to the best-neighbor strategy. We conclude that the greedy strategy should be preferred since it converges much quicker than the best-neighbor strategy and obtains scores similar to sequential merging.

For WCC, the heuristic from SCD (Section 4.2) was used to generate an initial partition of network before applying the merging algorithm, since it is not possible to form triangles by merging singleton communities. However, Fig. 5.7 and Fig. 5.11 show that merging is not effective for WCC. The improvement by merging is less than 25% compared to omitting the merging stage and only using the initial heuristic. This could be the result of the approximation model for WCC, which is not accurate.

We have seen that the greedy strategy performs better than the best-neighbor

strategy when optimizing modularity or CPM. However, one should also consider the computational cost of both strategies. For the greedy strategy, all pairs of communities need to be sorted and sequentially enumerated. Both operations are expensive and difficult to parallelize. For the best-neighbor strategy, only the best adjacent community for each community needs to be selected. This operation is cheaper and trivial to parallelize using a community-centric approach.

We can combine both strategies to get the best of both worlds. To be able to achieve this, we have extended the best-neighbor strategy by performing multiple *rounds* in each iteration. During each round, each community selects as merging partner the adjacent community with the highest benefit and which has not yet been added to the matching. If two communities select each other as merging partner in the same round, they are added to the matching. The procedure ends when r rounds are performed or when every community is added to the matching. Note that, with an infinite number of rounds, this strategy is equivalent to the greedy strategy.

Fig. 5.12 shows the modularity obtained using this round-based strategy for two graphs. This figure shows that the round-based strategy is very effective. Even performing only a few rounds in every iteration causes the parallel merging algorithm to converge much quicker. For three rounds, the results are almost identical to the results of the greedy strategy.

5.5 Summary

In this chapter, we discussed community detection using merging. We generalized Newman's greedy algorithm and proposed a parallel version of the algorithm which merges more than one pair of communities in every iteration. We considered two strategies for deciding which pairs of communities to merge in one iteration: **best-neighbor** and **greedy**. Results show that the greedy strategy finds communities with a higher quality than the best-neighbor strategy. However, the greedy strategy is also computationally more expensive than the best-neighbor strategy. By extending best-neighbor with multiple **rounds** we get the performance of the best-neighbors strategy and the quality of the greedy strategy.

CHAPTER 6

Design of Par-CD

In this chapter, we present a generic framework for parallel community detection. We discuss the motivation for this framework, its requirements, and its design.

6.1 Motivation

In Chapter 3, we have seen a number of metrics for measuring the quality of a partitioning of a network into communities. Many metrics have been proposed and it is likely that more metrics will be proposed in the future. Intuitively, a high-quality community should have many internal edges and few boundary edges. Quality metrics are designed to capture this intuition into a formal mathematical definition. Additionally, these metrics turn the problem of community detection into an optimization problem where the goal is to find the partitioning of a network into communities having the best score for a given metric.

In Chapter 4 and Chapter 5 we discussed two optimization techniques for community detection: label propagation and community merging. We have seen that both these techniques can be used for three different metrics: Modularity, CPM and WCC. Benchmarks showed that both techniques work well and find high-quality communities. Combining these two techniques might even improve the results. It is possible that other optimization techniques exist that increase the quality of the results. For example, we have not considered techniques based on a divisive method which attempt to split communities.

In this chapter, we present a flexible, generic and user-friendly framework for community which integrates all our findings so far. This framework allows the user to simply select a metric and the optimization techniques to be attempted and automatically generates a partitioning of any given network into communities by optimizing for the chosen metric using the chosen techniques.

6.2 Requirements

For Par-CD, we defined a set of requirements enumerated in this section.

The input of the framework is an *undirected*, *unweighted*, *simple* graph G consisting of a set of vertices V and edges E. Undirected means that edges have no directionality, unweighted means edges have no weights assigned to them and simple means there are no *self-edges*[1] or *multi-edges*[2]. Note that support for such networks could be added with minor modifications, but we restrict ourselves to the most basic type of network to simplify the design of the framework.

The output of the framework is a partitioning $\mathcal{P}$ of the network into k communities $C_1, \ldots, C_k$. Because this is a partitioning of the network, each vertex is assigned to one and only one community, i.e., $C_1 \cup \cdots \cup C_k = V$ and $C_i \cap C_j = \emptyset$ if $i \neq j$. Overlapping communities are not allowed.

Besides the graph to be processed, the user should also provide a quality metric $f : \mathcal{P}(V) \rightarrow \mathbb{R}$ which maps a partitioning of G to a quality score. Without loss of generality, we will assume that higher values of f indicate higher quality results and thus f needs to be maximized. Minimizing a metric is possible by defining a negative metric $f'(\mathcal{P}) = -f(\mathcal{P})$.

There are a number of additional requirements. First of all, the framework should be *fast*. Over the last few decades, large-scale networks consisting of millions or even billions of edges have become more common. Our framework is intended for these types of network and should process them within a reasonable amount of time.

Second, our framework should be *highly parallel* and expose a lot of *fine-grained parallelism*. This is mainly a consequence of our performance requirement. Modern architectures rely on parallelism to obtain high performance. By designing our framework for parallel processing, it is possible to create implementations for parallel architectures such as GPUs, multi-core systems, SIMD CPUs and distributed systems.

Finally, the framework should be *flexible*. The framework should offer enough parameters to allow the user to experiment with different settings. The user should be in control of the execution and be able to make trade-offs between quality and performance.

[1]A self-edge is an edge which has both endpoints attached to the same vertex.
[2]A multi-edge is an edge which consists of multiple sub-edges.

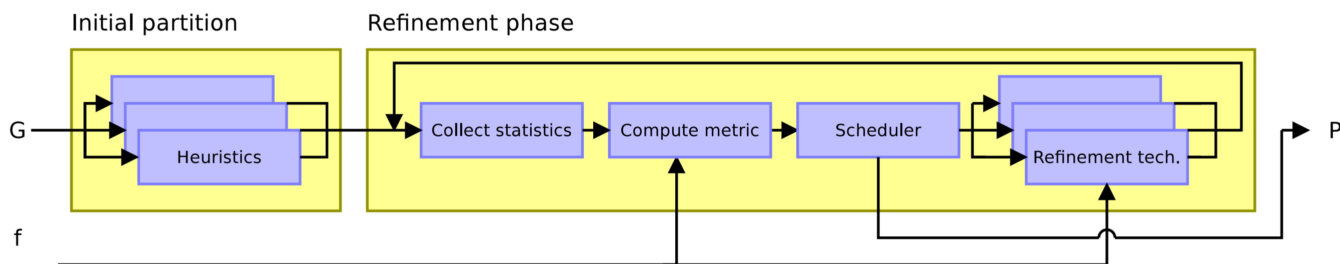


Figure 6.1: Diagram showing components of framework

6.3 Design

Fig. 6.1 shows the design of our framework. This design draws inspiration from the label propagation algorithm by Raghavan et al. [RAK07], the SCD algorithm by Prat et al. [PPDSLP14] and Newman's greedy algorithm [New04]. The framework consists of two phases: the *initial partitioning* phase and the *refinement* phase.

6.3.1 Initial Partitioning Phase

The initial partitioning phase receives as input the graph G and generates an initial partitioning $\mathcal{P}$ of the graph into communities. This initial partitioning serves as starting point for the refinement phase in which it will be iteratively improved. Since the initial partitioning will be improved, heuristics can be used to generate this partitioning. Examples of possible heuristics are the heuristic from SCD (Section 4.1.3) or simply assigning each vertex to a unique label such as done by LPA (Section 4.1.1), the Louvain method (Section 4.1.2) and Newman's greedy algorithm (Section 5.1). The heuristic should be cheap, such that so the time spent in this phase is negligible.

6.3.2 Refinement Phase

The refinement phase takes the initial partitioning generated during the first phase and improves it using an iterative process. In iteration t, a new partitioning $\mathcal{P}_{t+1}$ is generated based on the partitioning from the previous iteration $\mathcal{P}_t$. This new partitioning will have higher quality than the previous partitioning in terms of f, i.e., $f(\mathcal{P}_{t+1}) > f(\mathcal{P}_t)$.

Each iteration of the refinement phase consists of four steps: collect statistics of the communities, calculate the value of $f(\mathcal{P}_t)$, decide which refinement technique to apply, and, finally, apply the chosen refinement technique.

Collect Statistics

Most quality metrics requires the statistics of communities, such as their sizes or volumes, as input. The first step of each refinement iteration is collecting statistics in order to compute the chosen metric. The following three statistics, chosen based on the metrics from Chapter 3, are collected for each community C:

- The number of vertices: $n_C = |C|$
- The total internal degree: $d_{in}(C) = \sum_{x \in C} d_{in}(x, C)$
- The total external degree: $d_{out}(c) \sum_{x \in C}(d(x) - d_{in}(x, C))$

Compute Metric

The next step of each refinement iteration is calculating the value of the chosen metric f for current partitioning $\mathcal{P}$. Computing the metric is straightforward using its definition and the statistics from the previous step.

Scheduler

In each iteration, the scheduler decides which refinement technique will be applied. Only one technique can be applied in every iteration. Different schedulers are possible. For example, all refinement techniques could be applied in a round-robin fashion. Another option is to repeatedly apply the least computationally expensive technique until it is no longer efficient, then the second least expensive technique, etc. The optimal scheduler will likely depend on many factors such as the metric used, the topology of the graph, and the initial partitioning used. Additionally, the scheduler also decides when to terminate the refinement phase.

Refinement techniques

In the last step, the current partitioning is refined using the refinement technique chosen by the scheduler. A refinement technique takes the current partitioning and the community statistics as its and input and it generates a new partitioning based on the old partitioning. For the initial prototype of our framework, we propose the following refinement techniques:

- Semi-parallel label propagation with factor p (Chapter 4).
- Parallel merging with r rounds (Chapter 5).

An important question is how the refinement techniques should take the chosen metric into account. We want to avoid having separate refinement techniques for each possible metric since this will reduce the flexibility of our framework. To solve this problem, we use a concept introduced by Traag et al. [TVDN11] known as *adhesion*. The adhesion between communities X and Y is defined as the improvement for some quality metric f that results from merging these communities into a single community.

$$f_{adhesion}(X,Y) = f(\mathcal{P}) - f((\mathcal{P} \backslash \{X,Y\}) \cup \{X \cup Y\}) \tag{6.1}$$

Intuitively, the adhesion determines whether two communities “attract” or “repel” each other. If the adhesion between two communities is positive, merging them is beneficial and would increase the value of f. If the adhesion is negative, merging them is disadvantageous and would decrease the value of f.

Note that, the adhesion function is equivalent to the merging benefit function (Section 5.2), so the adhesion function has already been defined for modularity, CPM and WCC in Section 5.3.

$$f_{merge}(X,Y) = f_{adhesion}(X,Y) \tag{6.2}$$

The insertion benefit function (Section 4.3) for f can be defined as follows.

$$f_{insert}(v,C) = f_{adhesion}(\{v\},C) \tag{6.3}$$

6.4 Summary

In this chapter, we have introduced a generic framework for community detection which finds communities by optimizing a given community quality metric. The proposed framework is designed to be *fast*, *parallel* and *flexible*. The framework works by generating an initial partitioning of the network using a chosen heuristic and iteratively refining this partition by applying refinement techniques.

CHAPTER 7

Implementation of Par-CD for Multi-Core Architectures

In the previous chapter, we presented the design of Par-CD, without considering the implementation details for any specific architecture. In this chapter, we present an implementation of Par-CD for multi-core CPUs. We describe a sequential implementation of Par-CD and discuss how this implementation has been parallelized for multi-core processors. Finally, we evaluate the performance of our solution.

7.1 Background

A symmetric multiprocessor (SMP) system is a system consisting of a single shared memory unit and multiple processors which are all connected using a system bus. In the case of multi-core CPUs, the processors refer to the *cores* of the CPU. These cores all work in parallel, i.e., each individual core can process data independently from the other cores. Programs can use multiple cores in parallel by launching multiple *threads*. CPUs usually have several levels of *caches* between the cores and main memory to reduce the cost of memory operations.

Fig. 7.1 shows the architecture of the multi-core system which will be used during the evaluation. This system contains of two modern Intel Xeon E5620 CPUs, which are based on the Westmere x86-64 micro-architecture. Each CPU has four cores and a single L3 cache (12MB). Each core has its own L2 cache (256kB) and L1 (64kB) cache. The L1 caches are split evenly into instruction caches (L1i) and data caches (L1d).

The Intel Xeon E5620 CPU supports Intel's *Hyper-Threading Technology* (HTT) [MPS02]. HTT allows each core to execute two threads in parallel. Each *physical* core is composed of two *logical* cores. The physical cores contain a mixture of duplicated and

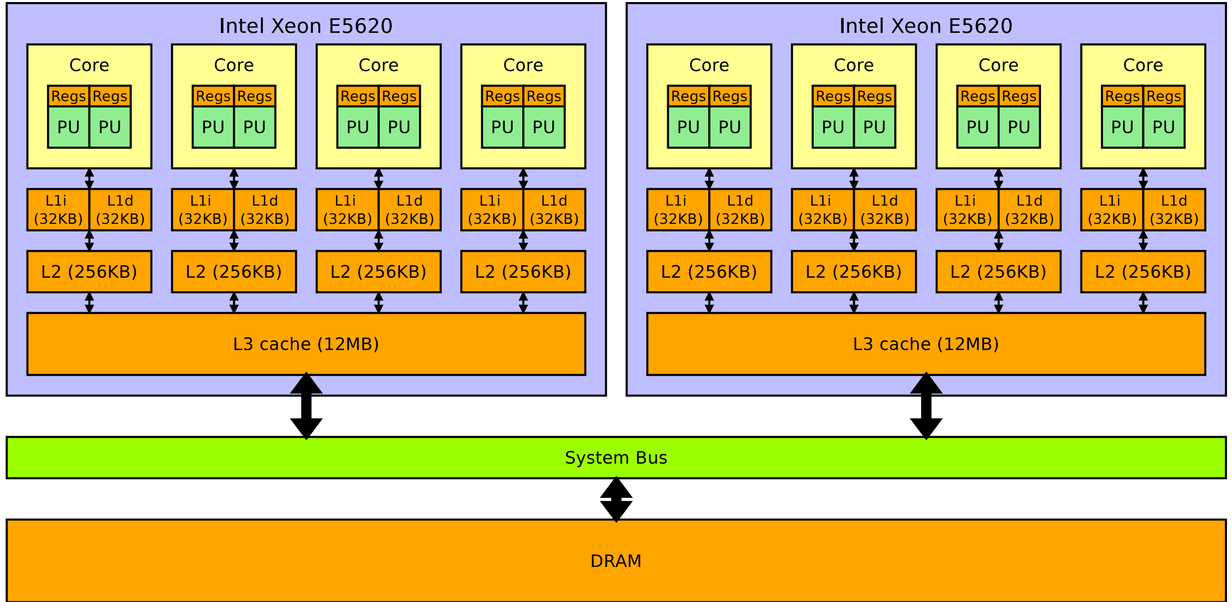


Figure 7.1: Example of modern multi-core architecture consisting of two Intel Xeon E5620 CPUs.

shared resources. Some resources, such as the TLB, are duplicated for each logical core. Other resources, such as the L1 cache, are shared among the two logical cores.

Overall, this system consists of two CPUs, each CPU consists of four cores and each core supports two threads. Therefore, this system can execute 16 threads in parallel.

7.2 Implementation

In this section, we describe our implementation of Par-CD for multi-core architectures. We discuss the data structures, a serial implementation and parallelization of this implementation.

7.2.1 Data Structures

Graph Representation

An important implementation decision is how the input network is represented in memory. Let $G = (V, E)$ be an undirected graph where $V = \{1, \ldots, n\}$ and $m = |E|$. In practice, three memory representations are commonly used (Fig. 7.2) to store such a graph.

Figure 7.2: Three different data structures for the representation of the same graph: adjacency matrix, edge list and compressed sparse row (CSR) format

An **edge list** is simply an unordered list of pairs of vertices which represent the edges of the graph. An **adjacency matrix** A is a two-dimensional $n \times n$ matrrix where the $A_{ij} = 1$ if edge (i, j) exists and $A_{ij} = 0$ otherwise. This matrix is symmetrical since the graph is undirected. Usually, most entries of an adjacency matrix are zero since real-world networks are often sparse. The **compressed sparse row** (CSR) format attempts to reduce the memory overhead of a sparse adjacency matrix by only storing the locations of the non-zero entries. CSR uses two arrays: *neighbors*, which stores the column indices of the non-zero entries by reading them from left-to-right and top-to-bottom, and *indices*, which stores the offsets into *neighbors* that start each row. The neighbor of vertex v can thus be found at $neighbors[indices[v]]$ to $neighbors[indices[v + 1]]$.

There are two important considerations when choosing the right data structure: **memory overhead** and **complexity of operations**. The memory overhead is n^2 elements for the adjacency matrix, $2m$ elements for the edge list and $n+1+2m$ elements for CSR. In Chapter 4 and Chapter 5 we have seen that iterating over the neighbors of a vertex is the most common operation. The complexity of this operation is $\mathcal{O}(n)$ for the adjacency matrix, $\mathcal{O}(m)$ for the edge list and $\mathcal{O}(d)$ for CSR where d is the degree of the vertex.

We argue that CSR is the best memory representation for our CPU implementation. The complexity of iterating over the neighbors of an arbitrary vertex is significantly lower for CSR than the other two representation. The memory overhead of CSR is slightly more than that of an edge list, but less than that of an adjacency matrix.

Histogram

Each vertex of the network is given a label which indicates the community to which the vertex belongs. A common operation is determining the frequency of each label from a set of vertices. For label propagation, it is necessary to determine the number of edges towards each neighboring community. For parallel merging, it is necessary to determine the number of edges between a community and each neighboring community.

Listing 7.1: Collecting community statistics

```
1  input: graph: G = (V, E), vertex community labels: labels
2  output: community sizes: sizes, community internal/external
3          degree: deg_int/deg_ext
4
5  for each vertex v do
6    label ← labels[v]
7    d_int ← 0, d_ext ← 0
8
9    for each neighbor u of v do
10     if labels[u] = label do
11       d_int ← d_in + 1
12     else
13       d_ext ← d_ext + 1
14
15   sizes[label] ← sizes[label] + 1
16   deg_int[label] ← deg_int[label] + d_int
17   deg_ext[label] ← deg_ext[label] + d_ext
```

The act of determining the frequency of each element from an arbitrary set of elements is known as a *histogram* operation. An obvious implementation of a histogram would be to allocate a fixed-size vector of bins and use the label of each vertex as an offset into this vector. However, in practice, this implementation is inefficient since most vertices/communities only have a few neighboring communities thus only a few bins will be used. This results in wasted memory and inefficient cache usage. In Section 7.4 we will evaluate the performance of a number of different *sparse* histogram implementations.

7.2.2 Sequential Implementation

We have implemented a serial version of Par-CD using C++11. In this section, we will describe the different steps of the refinement phase (Fig. 6.1): collecting statistics, calculating metric and applying the refinement techniques (semi-parallel label propagation or best-neighbor parallel merging). We will not discuss the initial partitioning phase since its implementation is straightforward and significantly less computationally expensive than the refinement phase.

Collecting Statistics

The first step of each refinement iteration is collecting the following statistics for each community: the number of vertices, the total internal degree and the total external degree. A straightforward implementation is shown in Listing 7.1. The complexity of this step is $\mathcal{O}(m)$ since each edge of the graph is visited twice.

Listing 7.2: Computing modularity

```
1 input: graph: G = (V,E), community sizes: sizes,
2         community internal/external degree: deg_int/deg_ext
3 output: modularity: Q
4
5 Q ← 0, m ← |E|
6
7 for each community l do
8   Q ← Q + ( deg_int[l]/2m − (deg_int[l]+deg_ext[l])^2/4m^2 )
9 end
```

Calculate Metric

The second step of each refinement iteration is calculating the value of the chosen metric. Implementing this step depends on the definition of the metric. For example, Listing 7.2 shows pseudo code for calculating modularity. The complexity of this step depends on the chosen metric. For modularity, the complexity is $\mathcal{O}(k)$ where k is the number of communities.

Semi-Parallel Label Propagation

The final step of each refinement iteration is applying an optimization technique. We describe semi-parallel label propagation in this section and best-neighbor parallel merging in the next section.

For semi-parallel label propagation, a fraction p of the vertices needs to be updated by applying one of three actions (Section 4.2): (1) keep the current label, (2) pick one of the neighboring labels or (3) generate a fresh label. The benefit of these actions can be calculated using the adhesion function for the chosen metric (Section 4.3). For example, the adhesion function for modularity is defined as follows:

$$f_{adhesion}(\{v\}, C) = d_{in}(v, C)/m - d(v)d(C)/2m^2 \tag{7.1}$$

The values of m, $d(v)$ and $d(C)$ are available from the community statistics. Only the value of $d_{in}(v, C)$ needs to be determined for each neighboring community. In order to update a vertex v, we need to count the number of edges from v to each neighboring community C.

Listing 7.3 shows our implementation of semi-parallel label propagation. For each vertex, there is a probability p that it gets updated (line 5). If a vertex v gets updated, three steps are applied. First, a histogram is created and the labels of the neighbors of v are added to the histogram (lines 6-9). Next, the improvement

Listing 7.3: Semi-parallel label propagation.

```
1  input: graph: G = (V, E), vertex community labels: labels, fraction: p
2  output: new vertex community labels: new_labels
3
4  for each vertex v do
5    if random number from [0, 1] < p do
6      create histogram counter
7
8      for each neighbor u of v do
9        insert labels[u] into counter
10
11     leave_score ← f_leave(v, labels[v], counter[labels[v]])
12     best_transfer_score ← −∞
13     best_transfer_label ← ⊥
14
15     for (label, freq) in counter do
16       score ← leave_score + f_join(v, label, freq)
17
18       if score > best_transfer_score do
19         best_transfer_score ← score
20         best_transfer_label ← label
21
22     if best_transfer_score ≤ 0 and leave_score ≤ 0 do
23       new_labels[v] ← labels[v]
24     else if best_transfer_score > leave_score do
25       new_labels[v] ← transfer_label
26     else
27       new_labels[v] ← generate fresh label
28   else
29     new_labels[v] ← labels[v]
```

for the chosen metric is calculated for each label and the best label is selected (lines 11-20). Finally, one of the three actions is applied (lines 22-27).

Best-neighbor Parallel Merging

Listing 7.4 shows our implementation of best-neighbor parallel merging.

The implementation of parallel merging consists of a number of steps. First, the members of each community are stored (lines 5-6) and, for each community, the variables *partner* and *active* are initialized to $\perp$ and *true* (lines 8-9), respectively. The first variable stores the best merging candidate for each community and the second variable indicates whether a community is still available to be merged with another community.

Next, a number of rounds is performed. Each round consists of two stages. During the first stage (lines 12-24), for each *active* community, the active neighboring community is determined having maximal adhesion. During the second stage

Listing 7.4: Parallel merging

```
1  input: graph: G = (V,E), vertex community labels: labels,
2         number rounds: rounds
3  output: new vertex community labels: new_labels
4
5  for each vertex v do
6    add v to members[labels[v]]
7
8  for each community l do
9    active[l] ← true, partner[l] ← ⊥
10
11 repeat rounds times
12   for each community l do
13     if active[l] and (partner[l] = ⊥ or active[partner[l]]) do
14       create histogram counter
15       best_partner ← ⊥, best_score ← 0
16
17       for each vertex v in members[l] do
18         for each neighbor u of v do
19           if labels[u] ≠ l and active[labels[u]] do
20             insert labels[u] into counter
21
22       for each (label, freq) in counter do
23         score ← f_merge(l, label, freq)
24
25         if score > best_score do
26           best_partner ← label, best_score ← score
27
28       partner[l] ← best_partner
29
30   for each community l do
31     if partner[l] = ⊥ or partner[partner[l]] = l do
32       active[l] ← false
33
34
35 for each vertex v do
36   l ← labels[v]
37
38   if not active[l] and partner[l] ≠ ⊥ do
39     new_labels[v] ← minimum(l, partner[l])
40   else
41     new_labels[v] ← l
```

(lines 30-32), the communities which have selected each other as merging partner are deactivated. After the last round, communities are merged be relabelling the vertices (lines 38-41). The complexity of this implementation is $\mathcal{O}(mr)$ where r is the number of rounds since, in the worst case scenario, each edge is traversed twice in each round.

Note that, in our implementation, each active community needs to recompute its best neighboring community in every round. Alternatively, it is also possible to

compute the adhesion between each pair of communities and store it into memory. The pairs that will be merged can then be selected using a number of rounds. This approach is less computationally expensive but requires more memory than our approach. However, in practice, we found that most communities are deactivated in the first round so the amount of computation in subsequent rounds is minor. The cost of using more memory does not outweigh the cost of additional computation.

7.3 Parallelization

To parallelize our sequential implementation, we have chosen OpenMP [DM98]. OpenMP is an API that allows C, C++ or Fortan code to be parallelized by using *pragma* directives to mark sections of code that can be run in parallel. OpenMP is based on the *fork-join* model where a single thread execute sequential sections and a pool of worker threads together execute parallel sections.

Par-CD has been designed to expose massive amounts of parallelism and most computation is embarrassingly parallel. Most stages of the framework can be parallelized by applying a vertex/community-centric approach by randomly dividing the vertices/communities evenly over the available threads. Note that the amount of work for each individual vertex/community might not be equivalent since vertices have different degrees and communities have different sizes. However, since the vertices/communities are randomly divided, the *average* amount of work per vertex/community is roughly equivalent. Load imbalance is thus negligible.

Collecting the community statistics (Listing 7.1) can be parallelized using a vertex-centric approach under the condition that the three counters (lines 15-17) are incremented using atomic operations to prevent race conditions. Since atomic operations are more expensive than regular memory operations, it is interesting to see whether the benefit of parallelism outweighs the additional cost of the atomic operations compared to regular memory operations. Semi-parallel label propagation (Listing 7.3) consists of a large loop which can be parallelized using a vertex-centric approach. Best-neighbor merging (Listing 7.4) can be parallelized by applying a community-centric approach to the two larger loops (lines 12-28, 30-32) and a vertex-centric approach to the relabelling loop (lines 35-41).

7.4 Evaluation

We have evaluated the performance and scalability of our implementation. Table 7.1 lists the networks used for the evaluation which were chosen from the SNAP repository [LK14]. The platform used for the evaluation contains two Intel Xeon E5620 CPUs (Fig. 7.1) and 24GB memory. For compilation, `gcc` 4.8.2 was used with the flags `-O3` and `-ffast-math`. Since this section focuses on perfor-

Table 7.1: Networks used for evaluation

Name	Vertices	Edges	Avg. Deg.	Max. Deg.	Type
Amazon	334,863	925,872	3.37	549	Co-purchase network
DBLP	317,080	1,049,866	6.62	343	Collaboration network
YouTube	1,134,890	2,987,624	5.16	28754	Social network
LiveJournal	3,997,962	34,681,189	17.18	14815	Social network
Orkut	3,072,441	117,185,083	76.28	33313	Social network

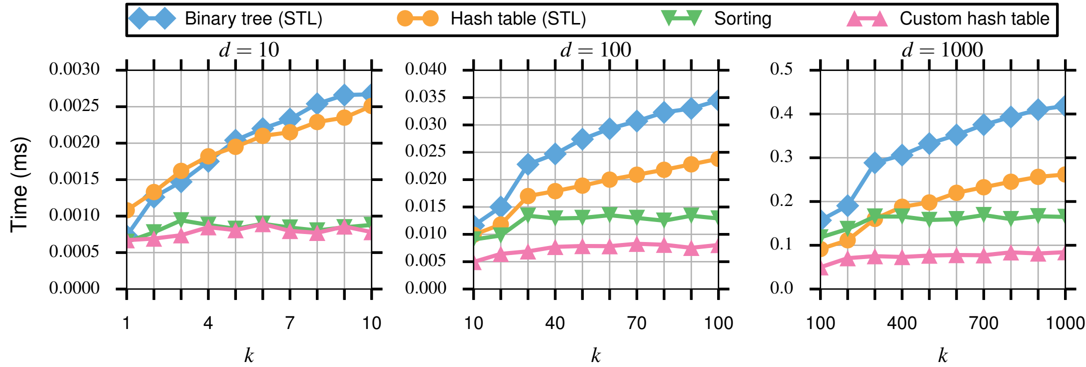


Figure 7.3: Performance of different sparse histogram implementation. The parameter k refers to the number of items inserted and d refers to the number of different items.

mance and not quality of the results, we will only consider modularity as a metric. The unique label heuristic (Section 6.3.1) was used for the initial labels.

7.4.1 Sparse Histogram

A sparse histogram is histogram in which most bins are unused. To store a sparse histogram, it is more efficient to only store the bins that are actually used instead of wasting memory on unused bins. Different implementation of sparse histograms exist. One option is to use an associative array, such as a binary tree or hash table. Each key-value pair (k, v) indicate that elements k occurs v times. Another options is to copy the elements to a buffer and sort this buffer. Sorting the buffer places matching elements adjacent to each other, making counting them possible.

We compare the performance of these two methods, see Fig. 7.3. For this experiment, we first generated a set of k different 32-bit integers. Next, d times one integer was randomly selected from this set and inserted into the histogram. We used the binary tree, hash table and sorting algorithm from the C++ standard library (STL). We also implemented a custom hash table based on *robin hood hashing* [CLM85]. The results show that the custom hash table has superior performance. Sorting the elements is just slightly slower than the custom hash table.

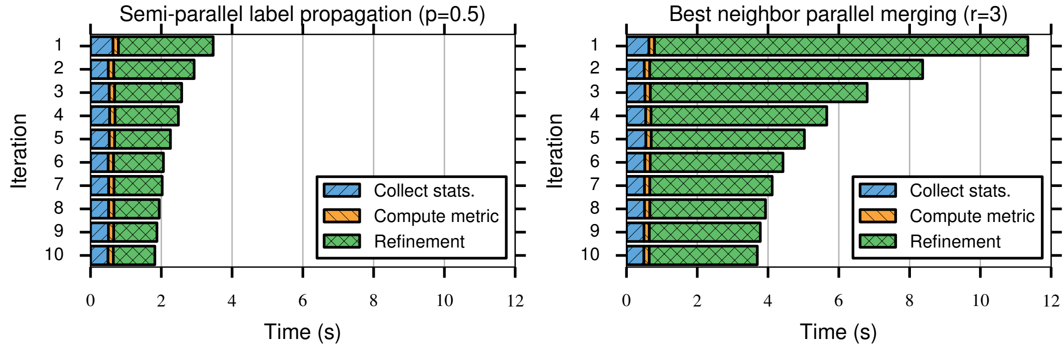


Figure 7.4: Execution time of different refinement techniques for first ten iterations on LiveJournal.

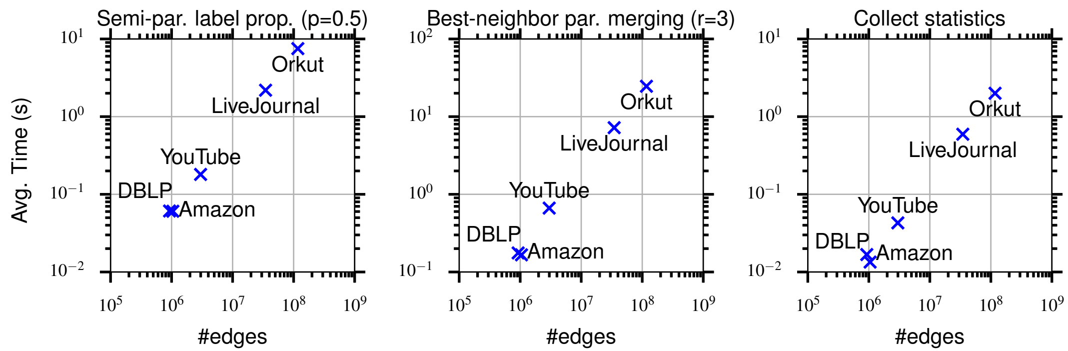


Figure 7.5: Average time of different stages over first ten iteration for different networks. Note that the scales on the horizontal axes are different.

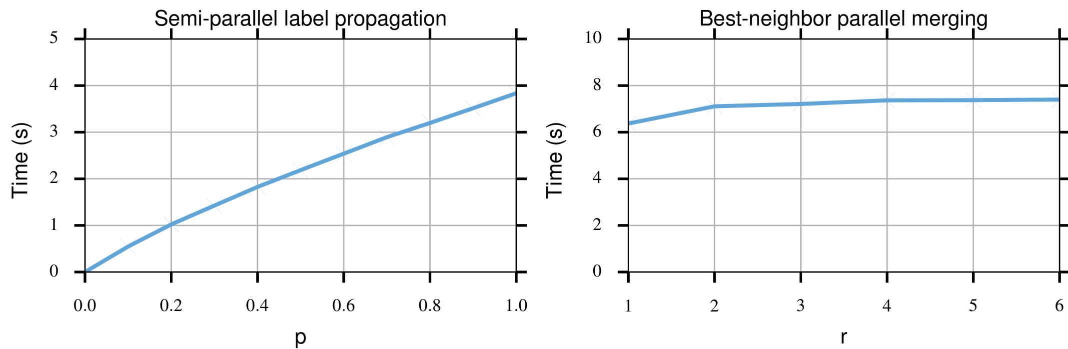


Figure 7.6: Effect of parameters on performance of label propagation and parallel merging for LiveJournal. Times are average over first ten iteration. Note that the scales on the horizontal axes are different.

7.4.2 Sequential Implementation

Next, we focus our attention to the performance of our sequential implementation. Fig. 7.4 shows the performance the different stages of each refinement iterations for the first ten iterations of label propagation (p=0.5) or ten iterations of parallel merging ($r = 3$). This figure shows that the time to perform a single iteration

decreases after each iteration. This happens because the number of communities decreases after each iteration. The number of communities determines the number of times the adhesion function is called and thus the amount of work in each iteration.

The figure also shows that the cost of calculating the metric is negligible. Collecting the statistics is slightly more expensive and takes roughly between 20% and 30% of the time for each iteration of label propagation and between 5% and 10% for parallel merging. Also, parallel merging is more computationally expensive than label propagation. A single iteration of parallel merging requires between 2x and 3.5x more time than label propagation.

Fig. 7.5 shows the average time over the first ten iterations for each stage and for the different networks. Calculating the metric has not been included since its time is negligible. Each stage has complexity $\mathcal{O}(m)$. This figure clearly shows that the execution time indeed scales linearly with the number of edges.

It is also interesting to see how the parameters p for label propagation and r for parallel merging affects the performance. Fig. 7.6 shows the effect of these parameters on the run-time. For label propagation, the execution time scales linearly with the parameter p. This is expected, since p determines the fraction of vertices to update and thus the amount of work. For parallel merging, the execution time increases with the number of rounds up to three rounds. Using more than three rounds has little effect on the performance. This happens because, after a certain number of rounds, each community has been deactivated and increasing the number of rounds has no effect on the amount of work.

7.4.3 Parallel Implementation

Next, we shall focus our attention to the parallel implementation. Fig. 7.7 shows the execution time and speedup of the different phase when varying the number of threads between 1 and 16. We observe that semi-parallel label propagation scales excellent up to 8 threads. Parallel merging also scales excellent up to 8 threads, but slightly worse than label propagation since more threads need to synchronize more often. Collecting statistics scales reasonably up to 8 threads, but worse than label propagation and parallel merging. This is due to the cost of the atomic operations.

In all three cases, we can see that the speedup does not increase significantly when using more than 8 threads. For example, for label propagation, the speedup is 6.2x when using 8 threads and only 8.1x when using 16 threads. The most likely explanation for this behavior is that this is due to Hyper-Threading (Section 7.1). Each E5620 CPU can execute 8 threads in parallel since each of the four *physical* cores exposes two *virtual* cores (vcores) to the operating system. However, since some of the resources of a physical are shared between the two vcores, contention

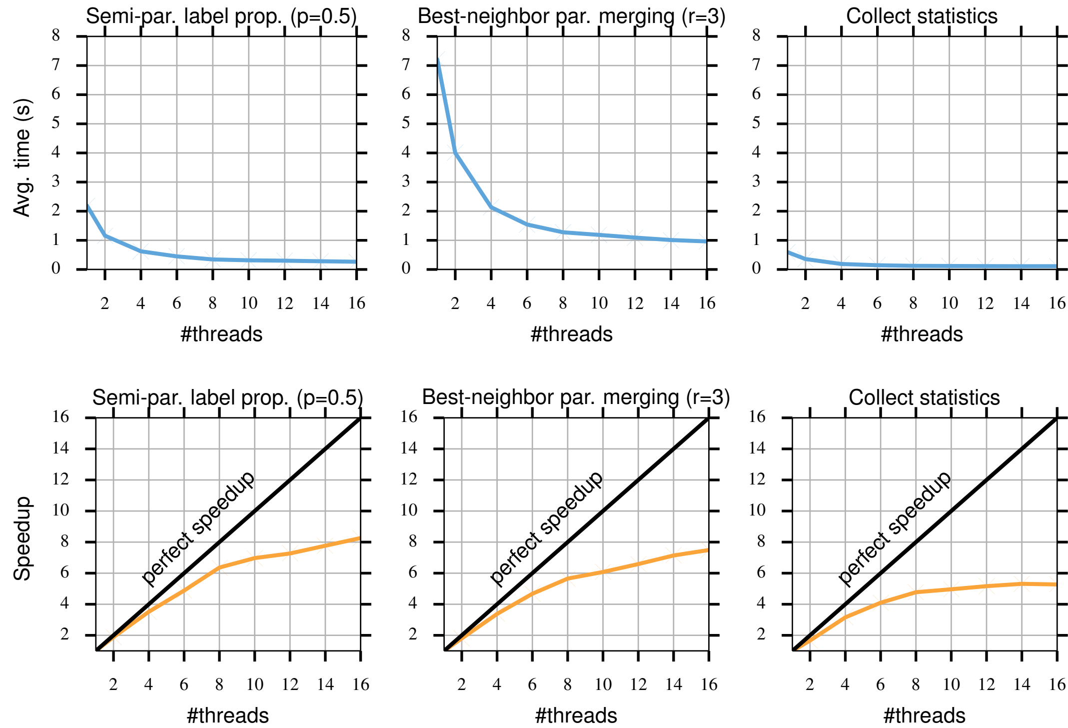


Figure 7.7: Execution time and speedup for different phases on LiveJournal when increasing the number of threads. Times are average over first ten iterations.

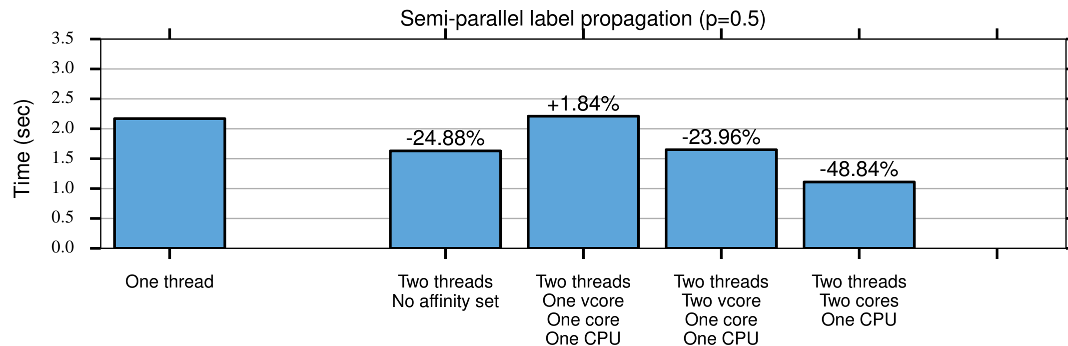


Figure 7.8: Execution time of semi-parallel label propagation on LiveJournal for different execution strategies by setting the affinity of threads. Times are average over first ten iterations.

occurs which limits speedup.

To validate this hypothesis, we have executed our implementation using different execution strategies. By manually setting the affinity of threads, it is possible to bind each thread to a specific vcore. Fig. 7.8 shows the results when using four different execution strategies for two threads. The figure shows that binding two threads to the same vcore has no effect on the performance compared to a single thread (0.98x speedup). Binding two threads to the different vcores of the same

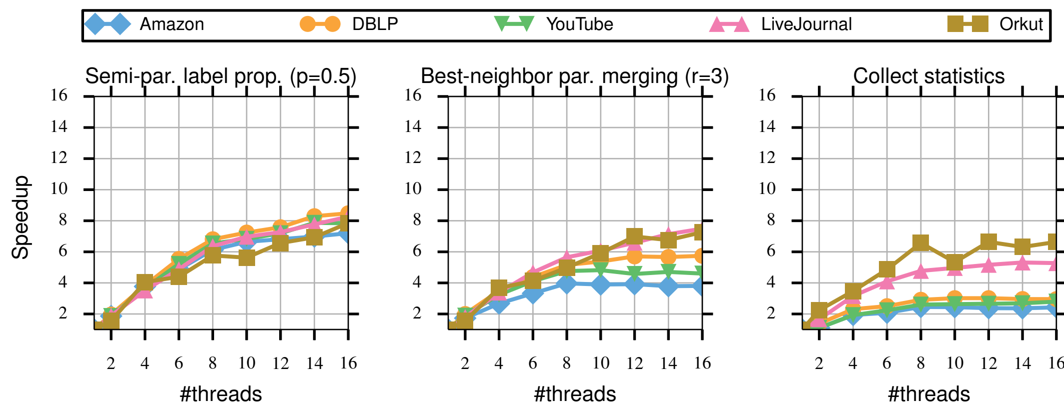


Figure 7.9: Execution time and speedup for different phases on different networks when increasing the number of threads. Times are average over first ten iterations.

core gives 1.2x speedup. Binding two threads to different cores of the same CPU gives 2.0x speedup.

This shows that this application is not suitable for Hyper-Threading, likely because there a lot of contention for resources within each core. Fig. 7.8 also shows that if the affinity is manually set, the speedup obtained is 1.2x. If the affinity is not explicitly set by the user, the operating system will determine the scheduling of the threads. Since this speedup is similar to the speedup obtain when using two vcores of the same core, it is likely that the operating system uses this execution strategy. For good performance when using fewer threads than cores, it is thus important to manually bind the threads to different cores.

Fig. 7.9 shows the speedup for all five networks. The speedup for all networks is similar to the speedup obtained for LiveJournal. The only exception is that collecting statistics does not scale for the three smallest networks (Amazon, DBLP, YouTube). This is most likely since these networks have fewer communities so there is more contention on the atomic operations.

7.5 Summary

In this chapter, we presented a implementation of Par-CD for multi-core CPUs. We have seen a serial implementation of Par-CD which has been parallelized using OpenMP. Results show that the serial implementation is fast: one refinement iteration on a network having 34M edges takes only ∼3 seconds for parallel label propagation and ∼6 seconds for best-neighbor parallel merging. The parallel implementation scales excellent with the number of threads.

CHAPTER 8

Implementation of Par-CD for Many-core Architectures

In the previous chapter, we presented an implementation of Par-CD for multi-core systems. In this chapter, we focus our attention to GPUs, since they are a good example of a modern many-core architecture. The architecture of a GPU differs significantly from that of a CPU and thus requires to different programming approach in order to obtain performance. We describe how GPUs and CPUs differ and discuss why the implementation of Par-CD from the previous chapter is not suitable for GPUs. Next, we present an alternative implementation of Par-CD specifically designed for GPUs. Finally, we evaluate the performance of this implementation and compare the performance of the version for CPUs against the version for GPUs.

8.1 Background

A *Graphics Processing Unit* (GPU) is a specialized processor which offers massive parallelism. While common CPUs usually have between two and twenty cores, GPU usually have several hundreds or even thousands of cores. Originally, GPUs were intended for graphics-intensive tasks, such as video games and 3D applications. However, due to their massively parallel design, GPUs turned out to be also highly efficient for parallel applications. Nowadays, GPUs are often used as an alternative to CPUs for many different types of parallel workloads. Using GPUs for general-purpose tasks is commonly referred to as "General-Purpose GPU computing" (GPGPU) [Har05].

GPUs differ significantly from CPUs, both in their programming model and their hardware design. We discuss both topics in this section.

8.1.1 Programming Models

In practice, two models are commonly used for programming GPUs: OpenCL and CUDA.

OpenCL (Open Compute Language) [SGS10] is a framework for writing programs for different types of parallel devices, including CPUs, GPUs and FPGAs (field-programmable gate arrays). OpenCL is an open standared mainted by the Khronos group. OpenCL requires the programmer to write special OpenCL-functions (*kernels*) in a C-like language which can be executed on a parallel device. These kernels are compiled at run-time for the chosen device by the OpenCL library. The main advantages of OpenCL are that it is an open standard and that code is cross-platform, i.e., the exact same OpenCL code can be executed on different devices without any modifications.

CUDA (Compute Unified Device Architecture) [SGS10] is a framework by NVIDIA for programming their GPUs. Similar to OpenCL, CUDA requires the programmer to define *kernels* which are executed on the GPU. However, an important difference is that CUDA is an extension to the C/C++ language and kernels are defined and called in a way similar to regular functions. This means that C/C++ programs can easily be accelerated using GPUs by only changing a few lines of code. Rewriting the entire source code is not necessary. The main advantages of CUDA are its ease of use and its ubiquity in GPU computing. CUDA is also more mature than OpenCL and offers better tools and libraries. However, contrary to OpenCL, CUDA code can *only* by executed on *GPUs* by *NVIDIA*. CUDA code cannot run on GPUs by other vendors or other parallel devices.

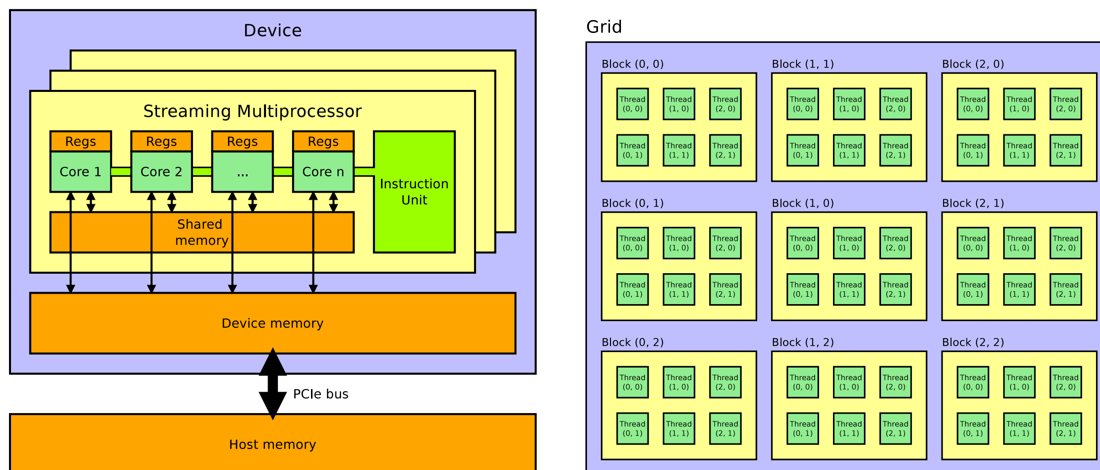
We refer to the overview by Su et al. [SCL+12] for a comprehensive comparison of the differences between CUDA and OpenCL and to the comparison by Fang et al. [FVS11] for a performance comparison.

8.1.2 Architecture

Fig. 8.1a shows a simplified model of the hardware of a GPU [NVI07]. In this section, we use NVIDIA terminology, although other vendors use similar terminology.

A GPU has its own dedicated GDDR (Graphics Double Data Rate) memory unit (**global memory**). The memory of the GPU and the memory of the CPU are not shared. Data needs to be explicitly transferred from the host to the device and from the device back to the host. The amount and type of memory differs per device. For example, the GTX TITAN has 6144MB GDDR5 RAM.

A GPU has a number of processors called *streaming multi-processors* (**SMs**). Each SM contains a fixed number of cores. Each core has its own private set of registers and has access to local memory unit (**shared memory**) which is shared among



(a) Simplified model of the hardware of a NVIDIA GPU.

(b) Execution model of CUDA showing hierarchy of threads.

all cores on the same SM. The number of SMs and the number of cores per SM differs per GPU. For example, the GTX TITAN has 14 SMs and 192 cores per SM, resulting in a total of 2688 cores.

The architecture of a GPU has an effect on its execution model. Executing code on a GPU is done by launching a large number of **threads** which all execute the same kernel. Threads are grouped into n-dimensional groups called **thread blocks**. Thread blocks are grouped into a n-dimensional **grid**. Only one grid can be active on the GPU at a time. Although each thread executes the exact same code, each thread has a unique identifier which allows it to determine which data element it will process. Thread blocks of a grid are scheduled onto the SMs of the devices. Threads within the same thread block can synchronize and communicate through shared memory. Threads within different blocks cannot cooperate. Fig. 8.1b demonstrates this model.

GPUs use a technique known as **latency hiding**. Each thread block is divided into groups of 32 consecutive threads known as **warps**. These warps are scheduled onto the cores of an SM. SMs can rapidly switch between different warps. If the active warp is stalled and waiting for memory transactions, the SM can quickly switch out the stalled warp and replace it with an idle warp. In other words, the latency of memory access is *hidden*. This allows GPUs to achieve performance without using complex techniques which are common on CPUs, such as caching, branch prediction, prefetching and out-of-order-execution.

Threads within the same warp execute in lock-step: they must all execute the same instruction at the same time. This has two major consequences: **thread divergence** and **memory coalescing**.

Thread divergence refers to the following problem. Since threads within the same warp execute in lock-step, they must all follow the same execution path and cannot diverge. If a warp reaches a branch instruction and some threads

want to take the branch and other want to skip the branch, the warp will need to perform both possible execution paths. Threads that want to take the branch are deactivated when the branch is not taken and vice versa. In other words, both paths are not executed in parallel, but serial. Thread divergence significantly reduces performance since the time it takes to execute both branch is the *sum* of the time it takes to execute each path, *not* the maximum time of both paths. This problem can occur for all if-statements and loops in the code.

Memory coalescing means combining multiple memory accesses to global memory into a single memory transaction. Since threads within the same warp run in lock-step, all threads will execute the same memory access instruction simultaneously. If all threads access the same 128 byte word in the same cycle, their accesses will be *coalesced* and only a single memory transaction is required. If memory accesses are not coalesced, multiple transactions are necessary which are serialized. In the worst case, the memory accesses are completely random and 32 memory transactions are necessary, one for each thread in a warp.

Note that OpenCL uses a similar execution model, only the terminology is different: *thread* is called *work-item*, *block* is *work-group*, *SM* is *computing unit*, *shared memory* is *local memory* and *registers* are *private memory*.

8.2 Implementation

For our implementation, we have chosen to use CUDA instead of OpenCL. This decision was made primarily because of CUDA's ease of use and its mature tools and libraries. However, implementing Par-CD in CUDA is not straightforward.

One possible solution would be to take the version for CPUs from the previous chapter and adapt it for GPUs. With the CPU version, the vertices/communities are randomly distributed over the available threads. The number of threads is set to the number of cores of the CPU. Since the number of cores is only a few dozens, each core gets a large number of vertices/communities and the work is equally distributed.

GPUs, on the other hand, have a high number of cores and rely on latency hiding to achieve performance. GPUs thus require massive amounts of threads to keep the cores occupied. The simplest approach to adapt the CPU version is to assign a CUDA thread to every vertex/community. However, this approach is not efficient for a number of reasons.

- Since vertices have different degrees and communities have different sizes, the amount of work per vertex/community differs. This load imbalance leads to severe **thread divergence**. For example, if a thread iterates over the neighbors of its vertex, all threads within the same warp must execute the

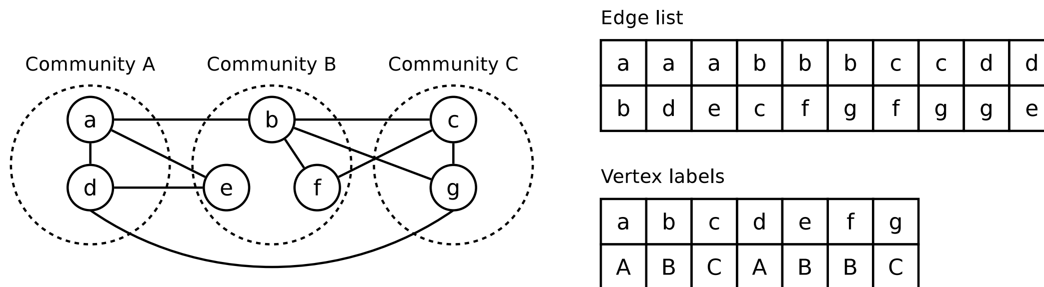


Figure 8.2: Example data structures used to store a network and its communities in memory

same number of iterations. The thread which owns the vertex with the largest degree thus determines the total number of iterations of the entire warp.

- The CSR format is not suitable for GPUs since its leads to little **memory coalescing**. For example, if all threads within the same warp access the i-th neighbor of their vertex, the memory accesses will not be coalesced since these memory locations are most likely not within the same word.
- Both refinement techniques (semi-parallel label propagation and best-neighbor parallel merging) use a sparse histogram to count the number of labels in the local neighborhood of a vertex/community. Each individual thread cannot allocate its own sparse histogram since this requires **dynamic memory allocation** by threads, which is not supported by most GPUs. A solution would be to pre-allocate fixed amounts of memory for each thread, but this requires massive amounts of memory.

Because of these reasons, we designed an alternative implementation of Par-CD specifically intended for GPUs. We abandoned the vertex/community-centric approach used for the CPU and instead use an edge-centric approach. Instead of storing the network in CSR format and frequently iterating over the neighbors of each vertex, we store the network as an undirected edge list and apply sequences of collective primitives (e.g., sorting, reduction, gather) to the edge list. This strategy allows us to utilize the existing knowledge on how these primitives can be efficiently implemented for GPUs. Fig. 8.2 shows an example of how a network and its communities are stored in GPU memory.

In the following sections we will describe the implementation of the different steps of the refinement phase: collecting statistics, calculating the metric and applying the refinement techniques (semi-parallel label propagation or best-neighbor parallel merging). Note that all steps are performed entirely on the GPU: data is only transferred to the the GPU before the refinement phase and results are transferred to the CPU after the refinement phase. We will not discuss the initial partitioning phase since it is not computationally expensive and thus does not need to be performed on the GPU.

Listing 8.1: Collecting community statistics

```
1  input: graph: G = (V, E), vertex community labels: labels
2  output: community sizes: sizes, community internal/external
3          degree: deg_int/deg_ext
4
5  parallel for each vertex v do
6    atomic_add(sizes[labels[v]], 1)
7  end
8
9  parallel for each edge (u, v) do
10   label_u ← labels[u]
11   label_v ← labels[v]
12
13   if label_u = label_v do  # internal edge
14     atomic_add(deg_int[label_u], 2)
15   else                     # external edge
16     atomic_add(deg_ext[label_u], 1)
17     atomic_add(deg_ext[label_v], 1)
18   end
19 end
```

Listing 8.2: Computing modularity

```
1  input: graph: G = (V, E), community sizes: sizes,
2         community internal/external degree: deg_int/deg_ext
3  output: modularity: Q
4
5  m ← |E|
6  scores ← allocate memory
7
8  parallel for each community l do
9    scores[l] ← (deg_int[l]/2m − (deg_int[l]+deg_ext[l])^2/4m^2)
10 end
11
12 Q = reduce(+, scores)
```

8.2.1 Collecting Statistics

Listing 8.1 shows pseudo-code for how community statistics are collected. For each vertex, the size of its community is incremented by one. For each edge, it is checked whether the edge is internal or external. If the edge is internal, the internal degree of the community is incremented by two. If the edge is external, the external degree is incremented by one for the communities at both endpoints of the edge. Note that atomic operations are used to increment the counters.

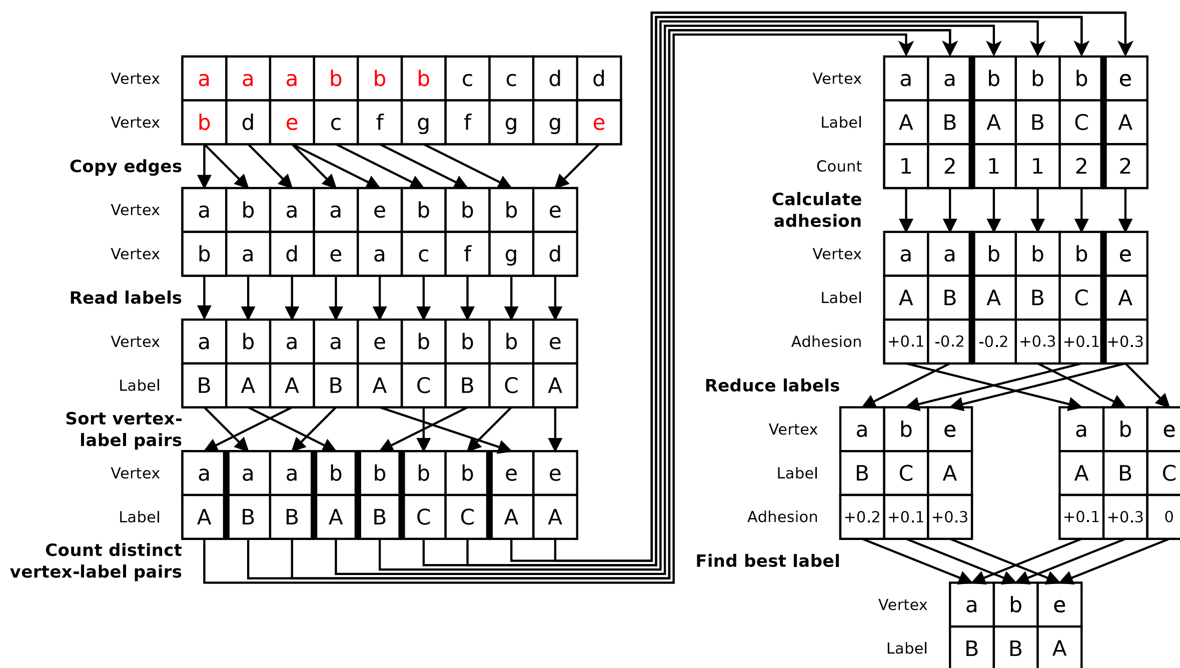


Figure 8.3: Data-flow for semi-parallel label propagation on the network from Fig. 8.2. The vertices a, b and e are updated.

8.2.2 Calculating Metric

Calculating the chosen metric depends on the definition of the metric. Listing 8.2 shows an example of calculating modularity. The scores of all communities are calculated and then added using a reduction operation.

8.2.3 Semi-Parallel Label Propagation

The implementation for collecting community statistics and calculating the metric is straightforward. The implementations for semi-parallel label propagation and best-neighbor parallel merging are more complex. We will use diagrams for these two stages instead of pseudo-code. These diagrams visualize the different collective operations and show how data flows through these operations. Fig. 8.3 visualizes the implementation of semi-parallel label propagation. The implementation involves a number of steps.

First, a random set of vertices that will be updated is selected. In this example, the vertices a, b and e are set to *active* since they will be updated. Next, for each edge (u, v), the tuples (u, v) or (v, u) are copied to a new buffer depending on whether u or v is active. If u is active, the tuple (u, v) is copied. If v is active, the tuple (v, u) is copied. If u and v are both active, (u, v) and (v, u) are both copied. The result is a list of pairs where each tuple (x, y) corresponds to an active vertex x having neighbor y.

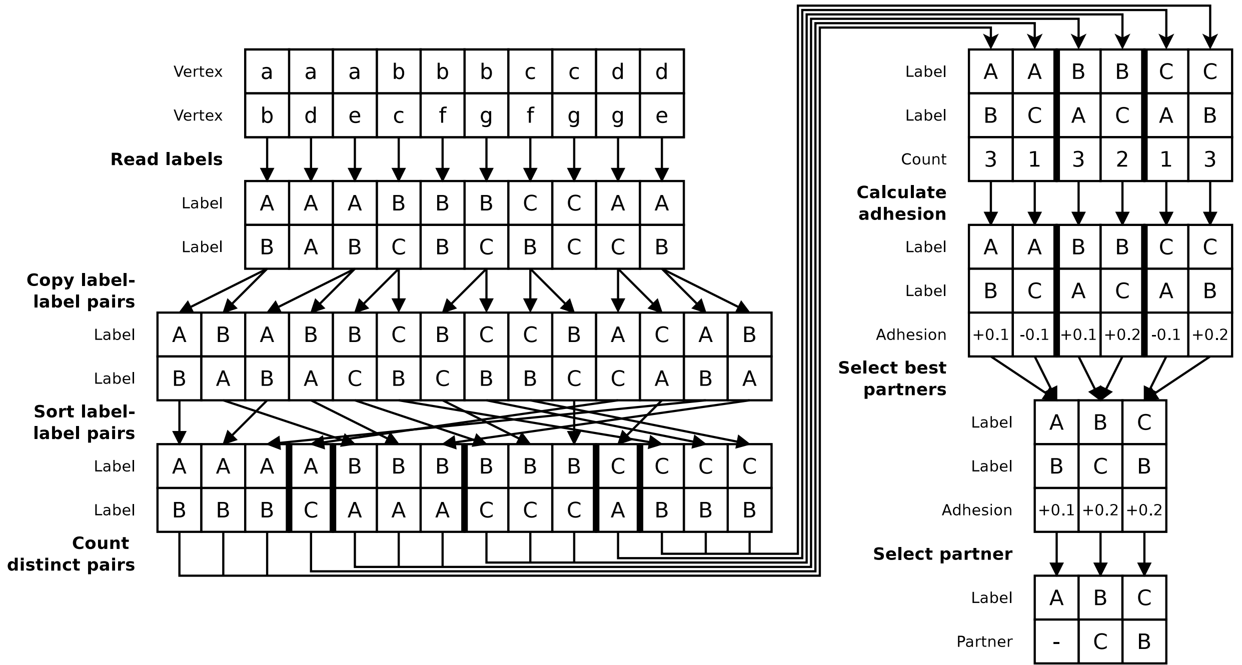


Figure 8.4: Data-flow for best-neighbor parallel merging on the network from Fig. 8.2.

After copying the edges, the labels of the vertices are read. This results in a list of vertex-label pairs where each pair (x, L) indicates that vertex x has an edge towards community L. These pairs are sorted such that all matching (x, L) pairs are adjacent. The frequency of each unique (x, L) pair is determined by a run-length encoding[1]. This gives the number of times each unique pair (x, L) occurs, i.e., the number of edges from vertex x towards community L. This information, together with the community statistics, is used to calculate the adhesion for each vertex-label pair. The result is a list of triples where each triplet (x, C, f) indicates that the adhesion between vertex x and community C is $f = f_{adhesion}(\{x\}, L)$.

Finally, two reduce-by-key operations [2] are performed. The first operation determines, for each vertex, the neighboring community having maximal adhesion. The second operation determines, for each vertex, the adhesion between the vertex and its current community. Using these values, it is determined which of the three possible actions (Section 4.2) is most beneficial. Finally, the action is applied and the new label of each vertex is determined.

Of all operations, sorting will most likely be the most computationally expensive operation since it has complexity $\mathcal{O}(m \log m)$. The other steps all have complexity $\mathcal{O}(m)$ and thus the amount of work scales linearly with the number of edges.

[1] Run-length encoding compresses a sequence of elements by encoding each "run" of consecutive matching elements using a single element and the length of the run.

[2] Reduce-by-key divides a sequence of key-value pairs into segments having consecutive matching keys and reduces the values in each segment to a single value.

Table 8.1: Devices used for evaluation

Name	Generation	Compute Capability	SM count	Core count	Core clock (Mhz)	Memory (MB)	Bandwidth (GB/s)
C2050	Fermi	2.0	14	448	575	2688	144.0
GTX480	Fermi	2.0	15	480	700	1536	177.4
GTX680	Kepler	3.0	8	1536	1006	2048	192.2
Tesla K20m	Kepler	3.5	13	2496	706	5120	208.0
GTX960	Maxwell	5.2	8	1024	1127	2048	112.0
GTX-TITAN	Kepler	3.5	14	2688	837	6144	288.0

8.2.4 Best-Neighbor Parallel Merging

Fig. 8.4 visualizes the implementation of best-neighbor parallel merging.

First, for each edge (u, v), the labels (X, Y) of the two endpoints are read. If the labels match, the edge is internal and it is ignored. If the labels are different, the pairs (X, Y) and (Y, X) are both copied to a buffer. Next, the buffer is sorted such that all matching pairs (X, Y) are adjacent. The frequency of each unique pair is determined using a run-length encoding. The result is a list of triples where each (X, Y, c) triplet indicates that there are c edges between communities X and Y. Note that if a (X, Y, c) triplet exist, there must also exists a reversed (Y, X, c) triplet. For each triplet, the adhesion between the two communities is calculated. The neighboring community having maximal adhesion is determined for each community using a reduce-by-key operation. Finally, the communities which have selected each other as partner are merged by relabeling the vertices.

Note that this implementation only performs a single merging round and does not support multiple rounds (Section 5.4). Adding multiple rounds is possible by, in each round, removing the triples (X, Y, c) where X or Y has already found a partner and applying a reduce-by-key operation to the remaining triples. Similar to label propagation, each operation has complexity $\mathcal{O}(m)$, except for sorting which has complexity $\mathcal{O}(mlogm)$. Sorting is thus the most expensive operation.

8.3 Evaluation

We have evaluated the performance of our solution for six GPUs (Table 8.1) and five networks (Table 7.1). Similar to Chapter 7, only modularity was considered as a metric since our focus is on performance of the implementation and not the quality of the results. For compilation, `nvcc` 5.5 was used with the flags `-O3` and `--use_fast_math` enabled. CUB [DM] was used to perform the collective operations. CUB (CUDA UnBound) is a library of reusable high-performance parallel primitives that operate at different levels of the hardware (device-wide,

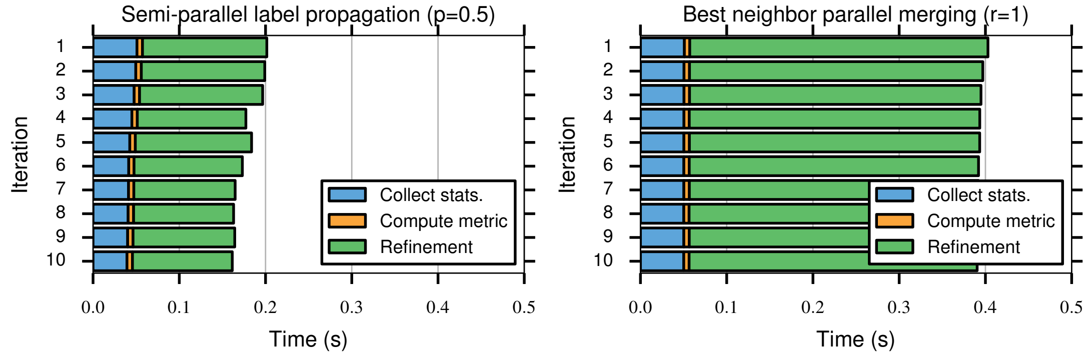


Figure 8.5: Execution time of different refinement techniques for first ten iteration for the K20m on LiveJournal.

thread-block level or warp level).

8.3.1 Performance

We first consider the performance the implementation for a single device and a single network. Fig. 8.5 shows the performance of the different stages of each iteration for the first ten iterations of label propagation($p = 0.5$) or parallel merging ($r = 1$). The figure shows that the cost of calculating the metric is negligible. Collecting the statistics is slightly more expensive and apply the refinement techniques is the most expensive stage.

It is also interesting to see how the time of each iteration is divided over the different operations performed during each iteration. Fig. 8.6 shows these times for both label propagation and parallel merging. We can see that, for both refinement techniques, sorting the pairs is the most expensive operation. This is not surprising since this operation has highest complexity.

For label propagation, the second most expensive operation is collecting internal/external degrees of the communities. Although this operation has complexity $\mathcal{O}(m)$, it is an expensive operation due to the high number of atomic instructions performed. The two most expensive operations, sorting the pairs and collecting degrees of the communities, take over 50% of the total execution time of one iteration.

For parallel merging, the second most expensive operation is calculating the adhesion between communities. This is most likely due to the high number of random memory reads, necessary to fetch the statistics for each community which are used to calculate the adhesion. This operation, together with sorting the pairs, takes over 60% of the total execution time of one iteration. For label propagation, calculating the adhesion is the third most expensive operation.

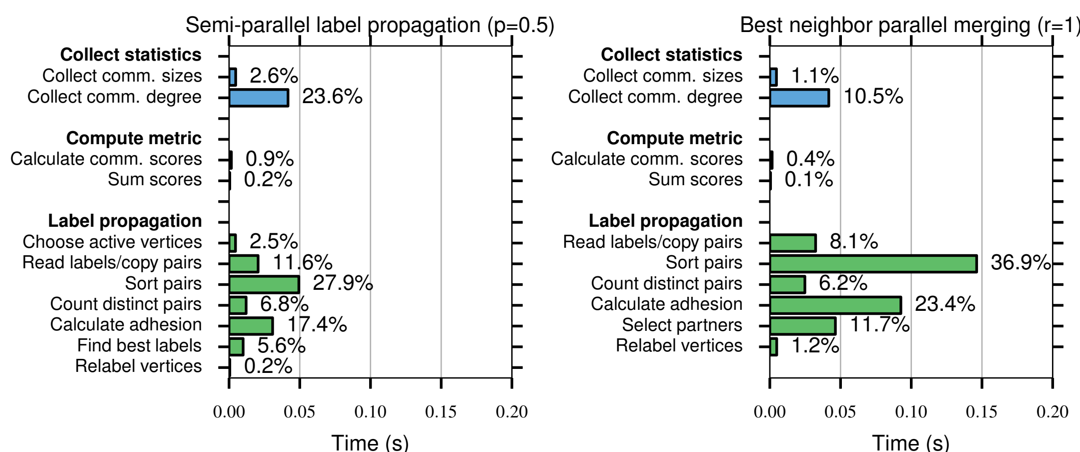


Figure 8.6: Average execution time of each of the steps of each iteration for the K20m on LiveJournal. These times are the average over the first ten iterations.

Note that, the two diagrams in Fig. 8.6 use the term "pairs" to refer to different types of pairs. For label propagation, each pair is a vertex-label tuple which represents and outgoing edge from an active vertex to one of its neighbors. The total number of pairs in each iteration can be estimated as $2m \times p$. For parallel merging, each pair is a label-label tuple which represents a directed edge from one community to another community. The total number of pairs is equal to twice the number of external edges of the network. Since each vertex is initially assigned a unique label in the first iteration, the number of pairs is close to $2m$ during the first few iterations. Since there are more pairs to consider, parallel merging is more expensive than label propagation.

Fig. 8.7 shows the average time per iteration for the six devices and five networks. Note that the times have been normalized by dividing them by the size of each network. The figure shows that the GTX-TITAN is clearly the most powerful GPU and the C2050 is the least powerful GPU. Also, note that the amount of memory of the GPUs is a realistic limitation which cannot be ignored. Only the K20m and GTX-TITAN provide enough memory to perform label propagation on Orkut. None of the GPUs provide enough memory to perform parallel merging on Orkut.

Fig. 8.8 shows the *speedup* of the GPU version over the serial CPU version from the previous chapter. The speedup of the multi-threaded CPU version has been included for comparison. The results show that the GPU is always significantly faster than the CPU. The most powerful GPU, the GTX-TITAN, obtains speedups between 13x and 28x for label propagation and between 17x and 29x for parallel merging, while the multi-threaded CPU version only obtains a speedups of between 5x and 8x. Only for the least powerful GPU, the C2050, the difference in speedup compared to the multi-threaded version is negligible.

The figure also shows that the speedup increases when the size of the network

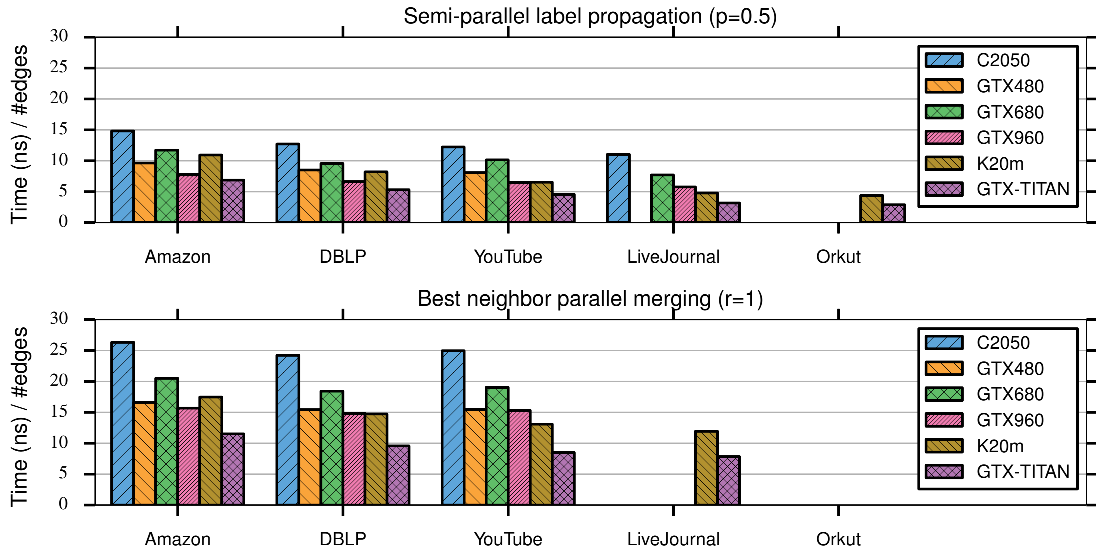


Figure 8.7: Average time per iteration over the first ten iterations for both refinement techniques. Lower is better. Note that the execution times are divided by the number of edges of each network to normalize the measurements. Missing bars indicate failures due to insufficient GPU memory.

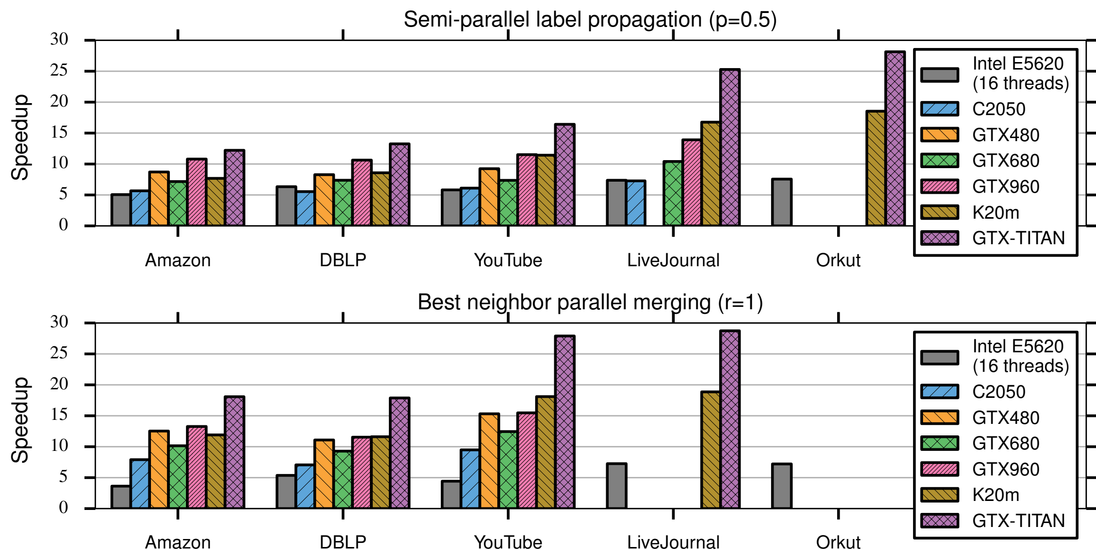
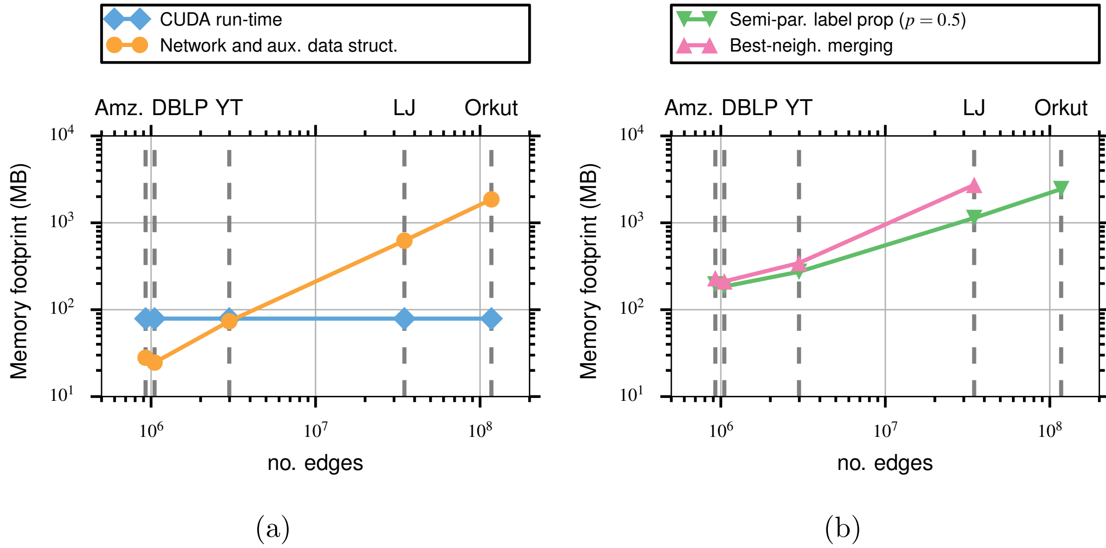


Figure 8.8: Speedup of GPU version over serial CPU version on an Intel Xeon E5620. Performance of the multi-threaded CPU version has been included for comparison.

increases. For example, for the semi-parallel label propagation on the K20m, the speedup is 7x for Amazon (0.9M edges) and 17x for Orkut (117M edges). This is most likely due to the difference in architecture. For the CPU, increasing the size of the network means that size of the data structures increases, leading to more cache misses and thus less efficient cache utilization. GPUs mostly rely



(a) (b)

Figure 8.9: Memory footprint of different networks on the K20m.

on memory latency hiding instead of caching to achieve high performance. For the GPU, increasing the size of networks means better SM occupation and thus improved latency hiding. In other words, when increasing the size of the network, the GPU becomes more efficient while the CPU become less efficient. This leads to an increase in speedup of the GPU over the CPU.

8.3.2 Memory Footprint

Fig. 8.7 shows that not all GPUs have sufficient memory to process all networks. For example, the GTX480 only has 1536MB memory, which is not enough to perform semi-parallel label propagation on LiveJournal. In this case, the total amount of available memory is exceeded by the memory footprint, i.e., the total amount of memory used by the application.

The memory footprint can be divided into three categories. First of all, there is the memory required by the CUDA run-time environment for the application. This amount is fixed and does not depend on the size of the network. Second, there is the memory required for the network and auxiliary data structures, such labels and community statistics. This amount scales with the size of the network, both the number of vertices and number of edges. Finally, there is the memory required for temporary buffers used during each iteration.

Fig. 8.9a shows the memory usage of the first two categories. The amount of memory required for the network and auxiliary data structures scales with both the number of vertices and the number of edges of the network. However, since the number of edges is an order of magnitude larger than the number of vertices, the memory used for the vertices is negligible compared to the memory used for

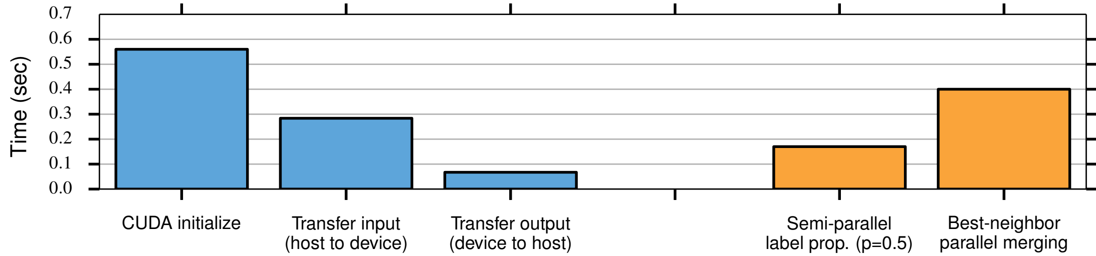


Figure 8.10: Overhead of data transfers for the K20m on LiveJournal. The average times from Fig. 8.5 to perform one iteration of each refinement technique have been included for comparison.

the edges. The memory usage thus scales almost linearly with the number edges.

Fig. 8.9b shows the additional memory usage when perform ten iterations of each refinement technique. We can see that parallel merging requires more memory than label propagation. For both refinement techniques, the memory usage scales with the number of edges.

Note that for semi-parallel label propagation, the memory usage also scales linearly with the parameter p. For example, doubling the parameter p, doubles the number vertices that will be updated and thus roughly doubles the memory usage. This allows us to perform label propagation on GPUs which have little memory by lowering the value of p.

For example, 78.9MB is required for the CUDA run-time and 622.5MB is required to store LiveJournal and the auxillarly data structures in memory (Fig. 8.9a). Loading LiveJournal on the GTX480 only leaves 794.6MB unused. This is not enough to perform semi-parallel label propagation for $p = 0.5$ since this requires 1136.8MB (Fig. 8.9b). However, by lowering the value of p it is possible to process LiveJournal on the GTX480. For example, it can be estimated that for $p \approx 0.3$, only $\sim$700MB will be required for label propagation. This means that the GTX480 provides more than enough memory to process LiveJournal if $p \approx 0.3$. Additional experiments on the GTX480 confirm that this GPU indeed provides enough memory if $p = 0.3$.

8.3.3 End-To-End Performance

Up to this point, we have only considered the time it takes to process the network on the GPU. However, for a complete picture of the performance of the GPU, one should also consider additional overheads, such as initializing the CUDA run-time environment and transferring data between host and device. Note that only two transfers are needed: once to transfer the network and initial labels to the device and once to transfer the final labels back to the host. Fig. 8.10 shows the cost of

these transfers for LiveJournal on the K20m. The average times to perform one iteration of each refinement technique have been included for comparison. This figure shows that the cost of the data transfers are small, considering one usually performs tens or hundreds of iterations, while the data is only transferred once.

8.4 Summary

In this chapter, we focused our attention to GPUs. We discussed how the architectures of CPUs and GPUs differ and why the implementation from Chapter 7 is not suitable for GPUs. We presented a version of Par-CD which has been specifically designed for GPUs. Benchmarks show that this version obtains excellent performance. For example, for one iteration of semi-parallel label propagation on LiveJournal (34M edges), the GTX-TITAN obtains a speedup of 25x over the serial CPU version and a speedup of 3.5x over the multi-threaded CPU version for 16 threads. For best-neighbor parallel merging, the speedups are 29x and 3.6x, repetitively. However, the limited amount of memory of the GPUs restricts the size of the networks that can be processed. For example, our benchmarks showed that only the K20m and GTX-TITAN provide enough memory to perform semi-parallel label propagation on Orkut (117M edges). However, semi-parallel label propagation can still be performed on low-end GPUs which provide little memory by tweaking the parameter p.

CHAPTER 9

Evaluation of Par-CD

In Chapters 4 and 5, we analyzed the two refinement techniques of Par-CD and focused on the quality of the results. In Chapters 7 and 8, we presented implementation for two different platforms and focused on analyzing their performance. In this chapter, we analyze both performance and quality. We show how Par-CD can be used for three different metrics and evaluate the results by comparing them against other community detection algorithms.

9.1 Case-studies

We have evaluated the performance and quality of Par-CD for three different metrics from Chapter 3: modularity, *Weighted Community Clustering* (WCC) and *Constant Potts Model* (CPM). The definition of the adhesion function for these three metrics can be found in Sections 4.3 and 5.3. For each metric, we demonstrate how the two refinement techniques of Par-CD can be combined and compare the results against those of other community detection algorithms.

The networks used for the evaluation have been chosen from the SNAP repository [LK14] are shown in Table 9.1. These networks were chosen because of the wide range of sizes: medium (0.2M edges), large (0.9M edges) and very large (34M edges). We will only consider the performance of the CPU implementation (Chapter 7) and not the GPU implementation (Chapter 8) since we will compare Par-CD against CPU implementations of other algorithms. Few GPUs implementations of community detection algorithms exist. Additionally, we have already demonstrated the increased speedups of Par-CD on GPUs in Section 8.3. The platform used for the evaluation contains two Intel Xeon E5620 CPUs and 24GB memory.

Table 9.1: Networks used for evaluation.

Name	Vertices	Edges	Avg. Deg	Max. Deg.	Type
Email	36,692	183,831	10.0	1,383	Communication Network
Amazon	334,863	925,872	5.5	549	Co-purchasing Network
LiveJournal	3,997,962	34,681,189	17.18	14815	Social network

9.1.1 Modularity

First, we focus our attention to the most popular quality metric: modularity. We investigate what parameters to use for the two refinement techniques and analyze how these techniques can be combined.

Fig. 9.1 shows the modularity over time when applying semi-parallel label propagation for different values of p. This figure shows that the parameter p does not have an effect on the final modularity obtained. However, the parameter p does have an effect on the convergence speed: lowering the value of p increases the convergence speed. For example, after two seconds, the modularity is already 0.45 for $p = 0.25$ and only 0.27 for $p = 1$. As discussed in Section 4.2, lowering the value of p reduces the amount of parallelism but increases the stability of the algorithm. The figure shows that lowering the value of p below 0.5 does not have a significant impact on the convergence speed as there is little difference between the curves for $p = 0.25$ and $p = 0.5$. A value of $p = 0.25$ seems to be a good choice for modularity.

Fig. 9.2 shows the modularity over time when applying best-neighbor parallel merging for different values of r. The figure shows that the number of rounds has a major impact on the final modularity obtained. For example, after 25 seconds, the modularity is 0.5 for $r = 5$ and only 0.22 for $r = 1$. A value of $r = 5$ is a good choice.

It is possible that combining both refinement techniques yields better communities than using each technique individually. Fig. 9.3 shows the modularity over time when alternating the two refinement techniques in every iteration. The results shows combing both techniques gives higher modularity than each technique on its own. This is happens because the two methods complement each other. Label propagation works on a local level: it refines the "border" between communities by relabelling individual vertices. Merging works on a global level: it refines the partitioning by relabelling entire communities.

We can also see that label propagation is more efficient during the first few iterations. For example, after 5 seconds, modularity is 0.61 for label propagation and only 0.20 for parallel merging. Ideally, one would like to perform label propagation during the first iterations and using parallel merging when label propagation is no longer efficient.

Based on this observation, we propose the following strategy. During the first iterations, semi-parallel label propagation is applied until the increase in modularity obtained in one iteration is less than a given threshold λ. Once this occurs, one iteration of best-neighbor parallel merging is applied. If the increase of modularity is greater than λ, semi-parallel label propagation is again repeatedly performed. If the increase is less than λ, the algorithm terminates. Fig. 9.3 shows the results for this threshold-based strategy for $\lambda = 0.1\%$. It is clear that its performance is superior compared to simply alternating the two techniques.

For modularity, we chose the following five algorithms for comparison:

- The Louvain method by Blondel et al. [BGLL08].
- The CNM "fast modularity" algorithm by Clauset, Newman & Moore [CNM04].
- Walktrap by Latapy & Pons [PL05].
- GANXiS (also known as SLPA) by Xie, Szymanski & Liu [XSL11].
- Community-EL by Riedy et al. [RMEB12].

These algorithms have been chosen since they find disjoint communities, have been shown to produce communities with high modularity, and the authors have publicly released their source code. The first three algorithms are well-known classic community detection algorithms. GANXiS is a modern algorithm based on label propagation which can be seen as an improvement of the classic label propagation algorithm by Raghavan et al. [RAK07]. Community-EL is a modern algorithm based on merging. Note that Community-EL is the only parallel algorithm and has support for multi-core systems. The other algorithms are single-threaded.

Fig. 9.4, 9.5 and 9.6 shows the performance of Par-CD compared to the other algorithms. The figures show the performance of Par-CD for only semi-parallel label propagation, only parallel merging, and the threshold-based strategy which combines both techniques. Note that the times shown in the figure do not include the time required to load the network from disk into memory since different implementations use different file formats. The maximum run-time was set to 15 minutes. Note that the scale on the horizontal axes is logarithmic.

The figures show that the quality and performance of Par-CD is competitive with the other algorithms. Par-CD is over an order of magnitude faster than CNM, WalkTrap and GANXiS. Community-EL is faster than Par-CD for `Email` and `Amazon`, but its memory usage does not scale for LiveJournal and the quality for `Email` is low. The modularity found by Par-CD is comparable to that of the Louvain method, which always obtains the highest modularity out of the five algorithms. This is not surprising, since the Louvain method is also based on label propagation and merging. However, Par-CD is significantly faster for the two largest networks: roughly 2 times faster for `Email` and roughly 10 times faster for `Amazon`. This is due to the parallelism of Par-CD .

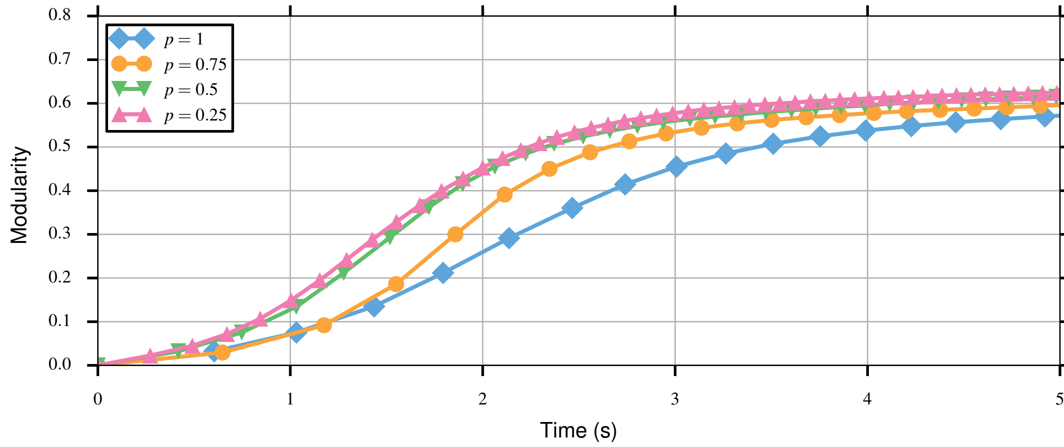


Figure 9.1: Semi-parallel label propagation for modularity on LiveJournal.

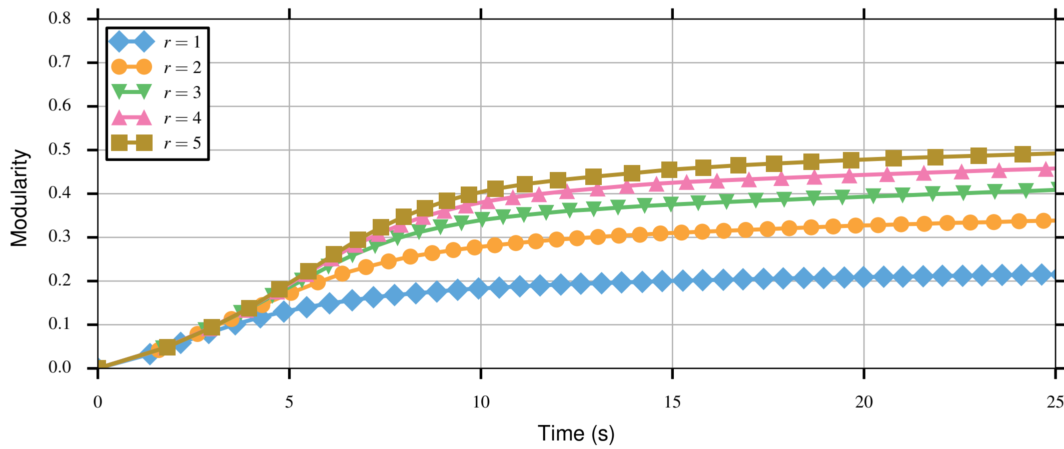


Figure 9.2: Best-neighbor parallel merging for modularity on LiveJournal.

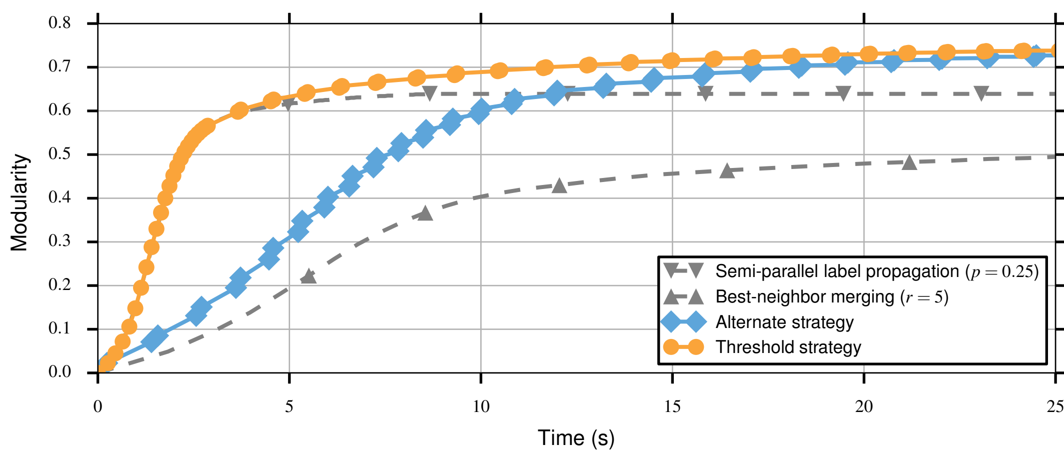


Figure 9.3: Combining both refinement techniques for modularity on LiveJournal.

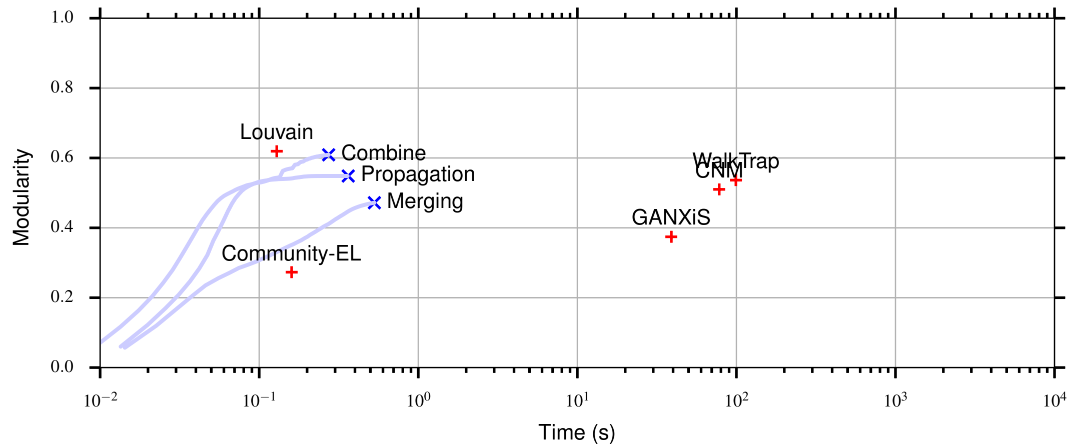


Figure 9.4: Performance of Par-CD for modularity and other algorithms on Email.

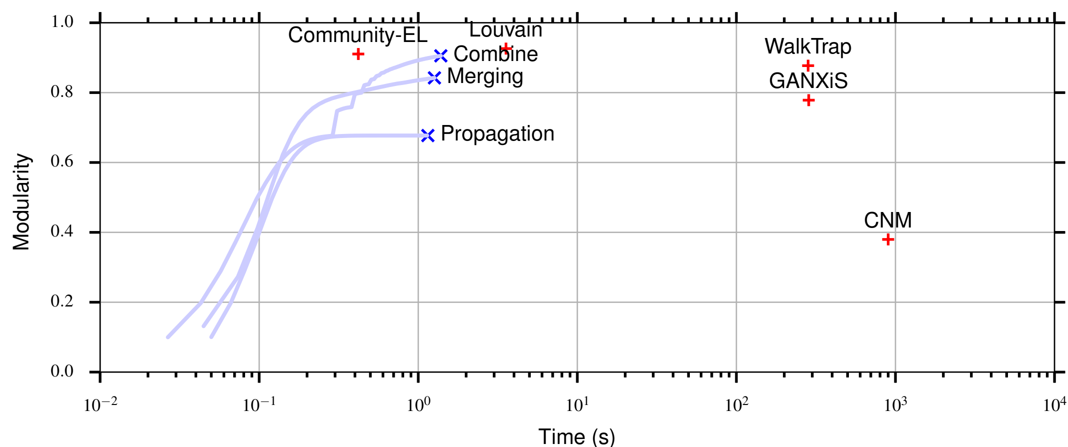


Figure 9.5: Performance of Par-CD for modularity and other algorithms on Amazon.

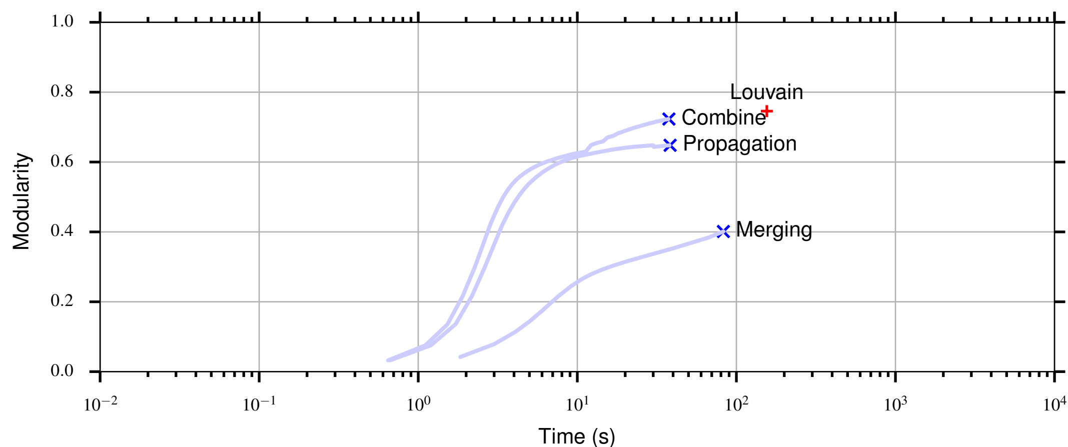


Figure 9.6: Performance of Par-CD for modularity and other algorithms on LiveJournal. Missing points indicate failure due to exceeding the 15 minute deadline or insufficient memory.

9.1.2 WCC

Next, we focus our attention on WCC, a quality metric based on triangle counting. Again, we first investigate what parameters to use for the two refinement techniques and how these techniques can be combined. Note that the heuristic from SCD will be used to generate the initial labels (Section 6.3.1) instead of the unique labelling heuristic.

Fig. 9.7 shows the WCC over time when applying semi-parallel label propagation for different values of p. Note that this figure only shows the run-time of the refinement phase, not the initial partition phase. The parameter p does not have an effect on the final WCC obtained, but it does have an effect on the convergence speed. Contrary to modularity, for WCC we see that increasing the value of p increases the convergence speed. A value of $p = 1$ gives the best performance.

Fig. 9.8 shows WCC over time when applying best-neighbor parallel merging for different values of r. We can see that the value of r does not have a large impact on the WCC obtained. Increase the value of r actually lowers the WCC obtained. A value of $r = 1$ thus gives the best performance.

Fig. 9.9 shows WCC over time when alternating semi-parallel label propagation and best-neighbor merging. The figure shows that combining the two techniques actually lowers the quality of the results compared to only using label propagation. This is most likely due to the approximation used to estimate the adhesion when merging communities (Section 5.3). For WCC, it is best to only use semi-parallel label propagation and omit best-neighbor parallel merging.

At the time of writing, only one algorithm exists which is based on WCC maximization: SCD by Prat et al. [PPDSLP14]. As discussed in Section 4.1.3 is an algorithm based on label propagation. In fact, SCD is almost equivalent to Par-CD when optimizing WCC by applying semi-parallel label propagation for $p = 1$. The main difference is that SCD includes a preprocessing phase which removes edges which do not close any triangles. These edges do not contribute to the WCC, so removing them reduces memory usage and increases performance.

Fig. 9.10, 9.11 & 9.12 show the performance of Par-CD compared SCD. As can be seen, the performance and quality obtained by both algorithms is similar. Par-CD appears to be slightly faster, which is due to a small optimization in the source code. The implementation of SCD counts all triangles in the network in each iteration. Par-CD, on the other hand, counts all triangles once at the start of the refinement phase and only counts the *internal* triangles in every iteration.

However, SCD obtains higher WCC than Par-CD . This is due to the preprocessing phase of SCD. The constant-time approximation for WCC is based on the assumption that each edge closes at least one triangle. This approximation is more accurate if edges which do not close a triangle have been removed.

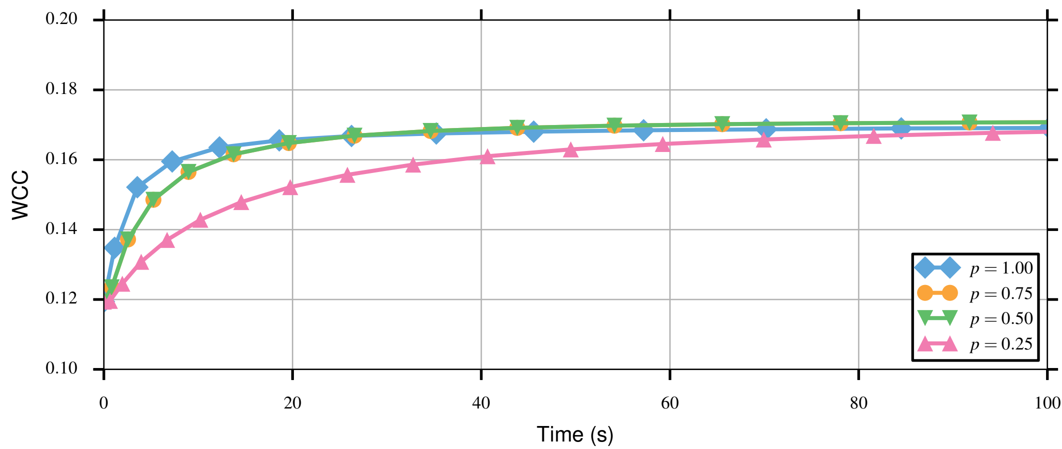


Figure 9.7: Semi-parallel label propagation for WCC on LiveJournal.

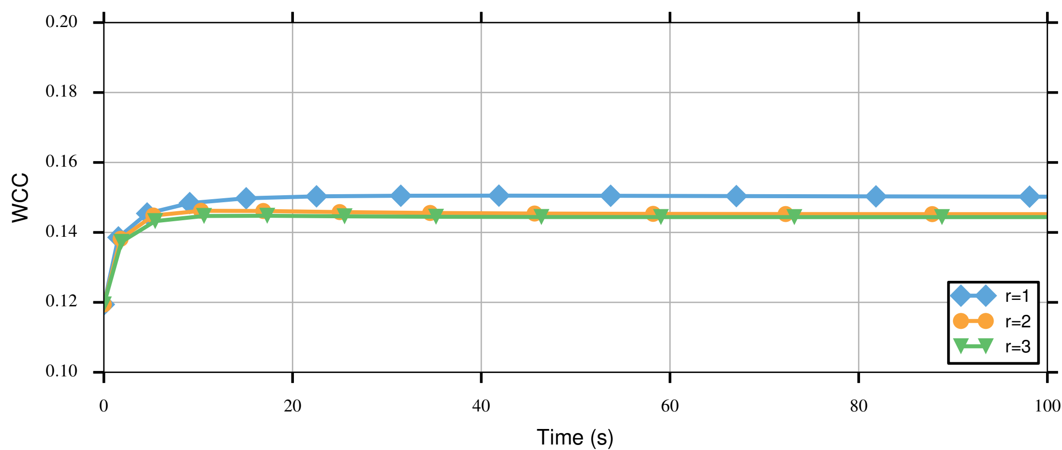


Figure 9.8: Best-neighbor parallel merging for WCC on LiveJournal.

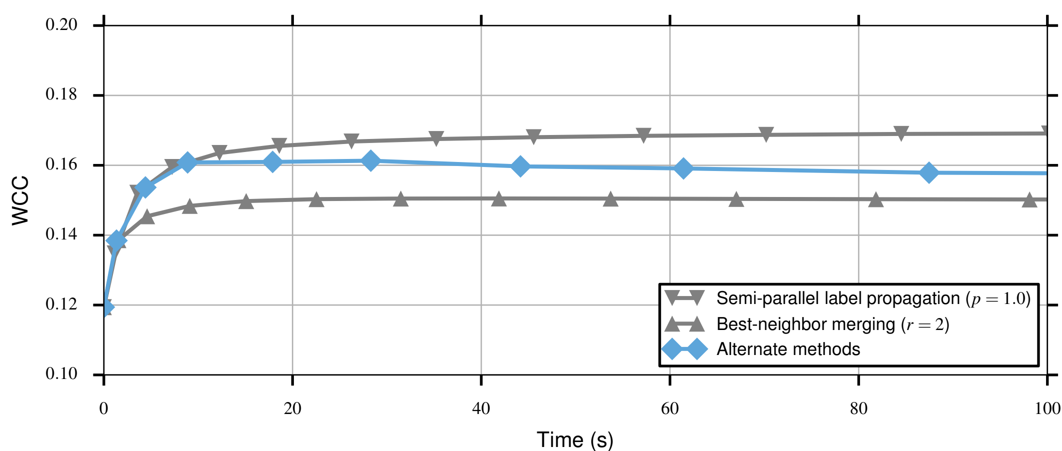


Figure 9.9: Combining both refinement techniques for WCC on LiveJournal.

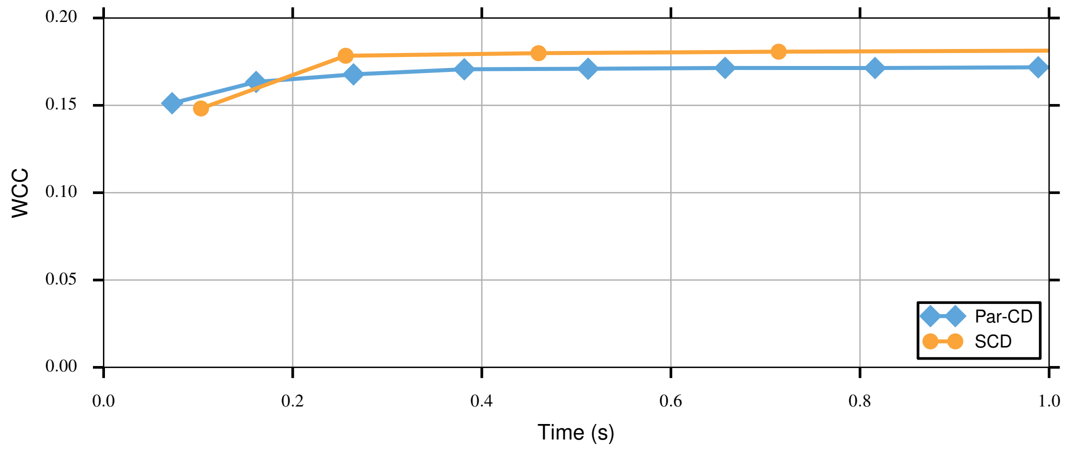


Figure 9.10: Performance of Par-CD for WCC and SCD on Email.

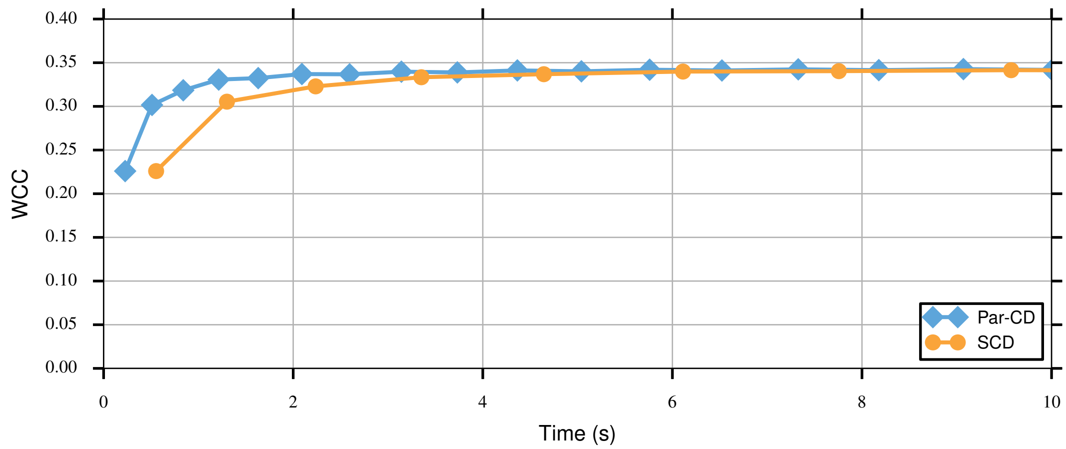


Figure 9.11: Performance of Par-CD for WCC and SCD on Amazon.

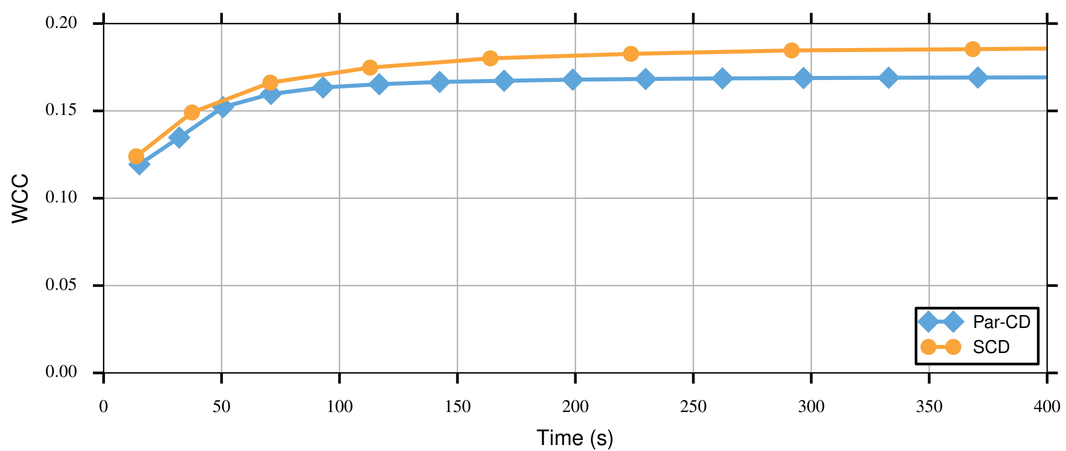


Figure 9.12: Performance of Par-CD for WCC and SCD on LiveJournal.

9.1.3 CPM

Finally, we focus our attention on CPM, a quality metric which includes a resolution parameter γ. The first step is investigating what parameters to use for the two refinement techniques. The next step is analyzing how these techniques can be combined. The unique labelling heuristic will be used to generate the initial labels.

Fig. 9.13 and 9.16 show the CPM over time for $\gamma = 0.1$ and $\gamma = 0.5$ when applying semi-parallel label propagation different values of p. These figures clearly show that a value of $p = 1$ is not suitable when optimizing CPM: the final score obtained is significantly higher for $p < 1$ than for $p = 1$. As discussed in Section 4.4, this is due to oscillation of labels. Lowering the value of p increase the stability of the algorithm, but also reduces the amount of parallelism. The figures show that a value of $p = 0.5$ is a good trade-off for both $\gamma = 0.1$ and $\gamma = 0.5$.

Fig. 9.14 and 9.17 show the CPM over time for $\gamma = 0.1$ and $\gamma = 0.5$ when applying best-neighbor parallel merging for different value of r. These results show that the number of rounds r has little impact on the convergence speed or the final CPM obtained Fig. 9.14 shows that the algorithm convergences slightly faster for $r = 2$ compared to $r = 1$, so the value $r = 2$ seems a good choice for CPM.

Fig. 9.15 and 9.18 shows the CPM over time for $\gamma = 0.1$ and $\gamma = 0.5$ when alternating the two refinement techniques. Both refinement techniques for the chosen parameters have been included in the figures for comparison. These results show that semi-parallel label propagation yields much higher quality than best-neighbor parallel merging. Combing the two techniques does not increase the quality of the results.

Unfortunately, at the time of writing, no algorithm exists yet which is based on CPM maximization. This means it is not possible to asses the performance of Par-CD by comparing it against another algorithm. However, it is possible to asses the *meaningfulness* of the communities found by Par-CD when optimizing for CPM. This gives an indication of whether Par-CD can be used for optimizing CPM. If the communities found are not meaningful, then the results are arbitrary and thus useless.

We used the LFR model (Section 2.4.2) to evaluate the meaningfulness of the results of Par-CD CPM. Fig. 9.19 shows the results of the LFR benchmark for different value of γ and μ. Recall from Section 2.4.2 that the factor μ determines how "entangled" the network is. All communities are isolated from each other for $\mu = 0$ and all communities are completed mixed for $\mu = 1$. The similarity between the ground-truth communities and the communities found by Par-CD is measured by the NMI (Section 2.4.2). The results match exactly if $NMI = 1$ and the results are completely independent if $NMI = 0$. Fig. 9.19 shows that the NMI is close to 1 for small values of μ, which means the results are meaningful. Note that

the parameter γ determines the resolution of the CPM. Increasing the value of γ decreases the size of the communities and vice versa. A value of $\gamma = 0.1$ seems to best match the resolution of the ground-truth communities.

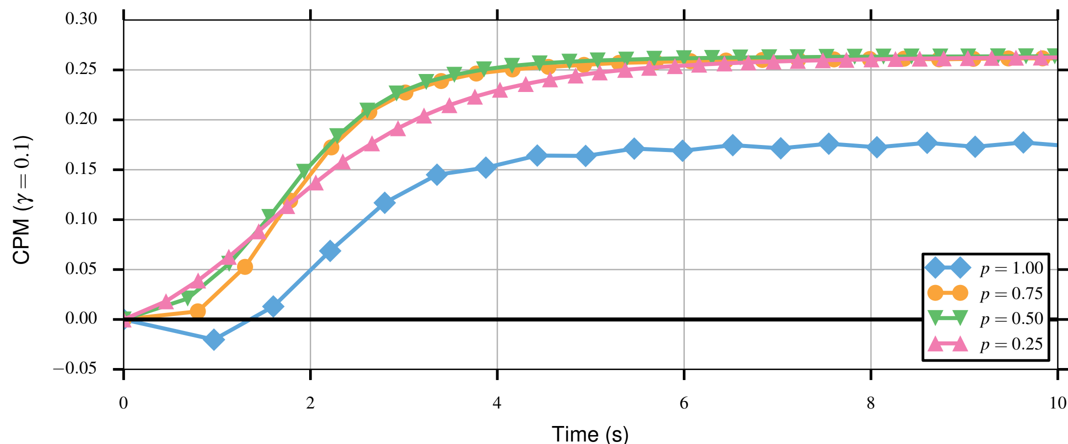


Figure 9.13: Semi-parallel label propagation for CPM ($\gamma = 0.1$) on LiveJournal.

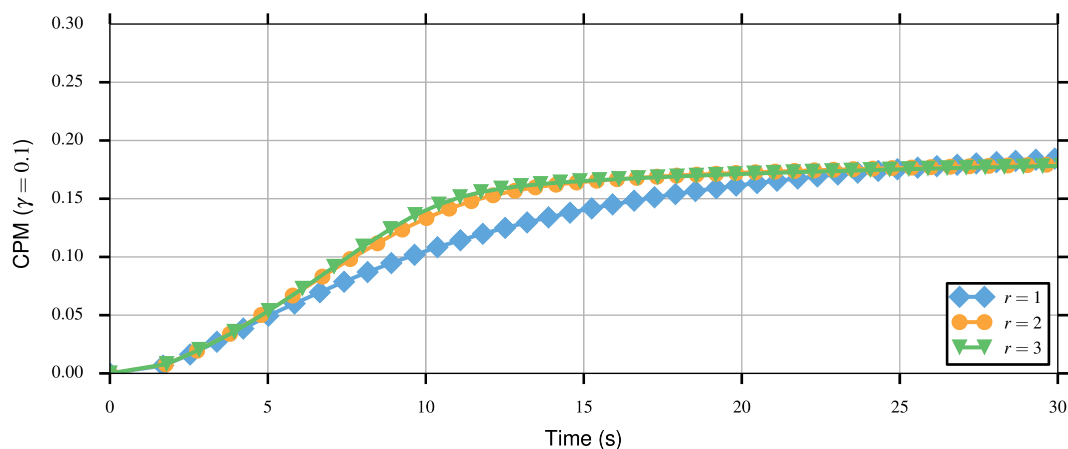


Figure 9.14: Best-neighbor parallel merging for CPM ($\gamma = 0.1$) on LiveJournal.

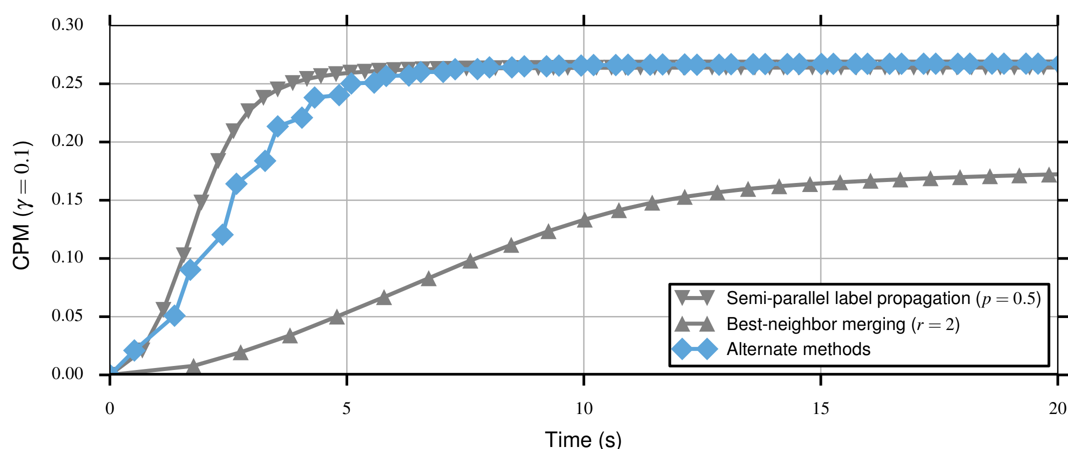


Figure 9.15: Combining both refinement techniques for ($\gamma = 0.1$) on LiveJournal.

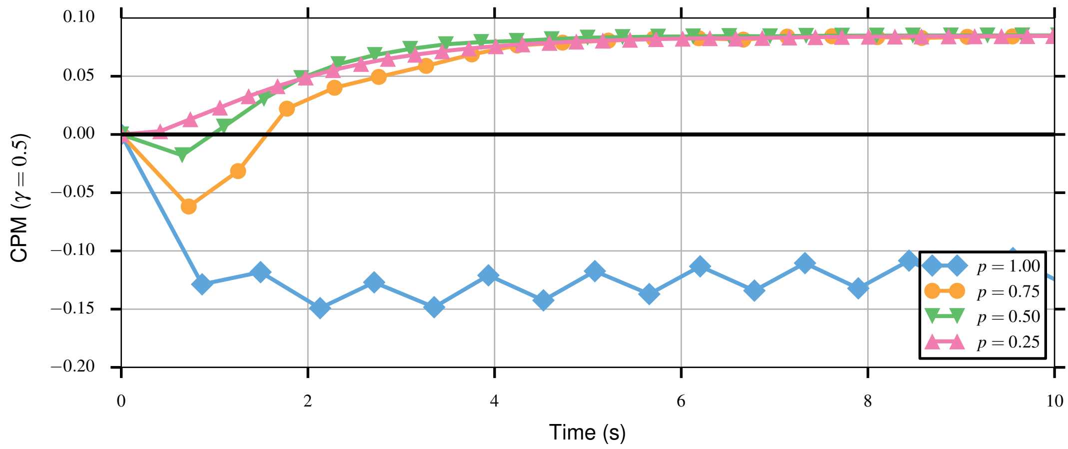


Figure 9.16: Semi-parallel label propagation for CPM ($\gamma = 0.5$) on LiveJournal.

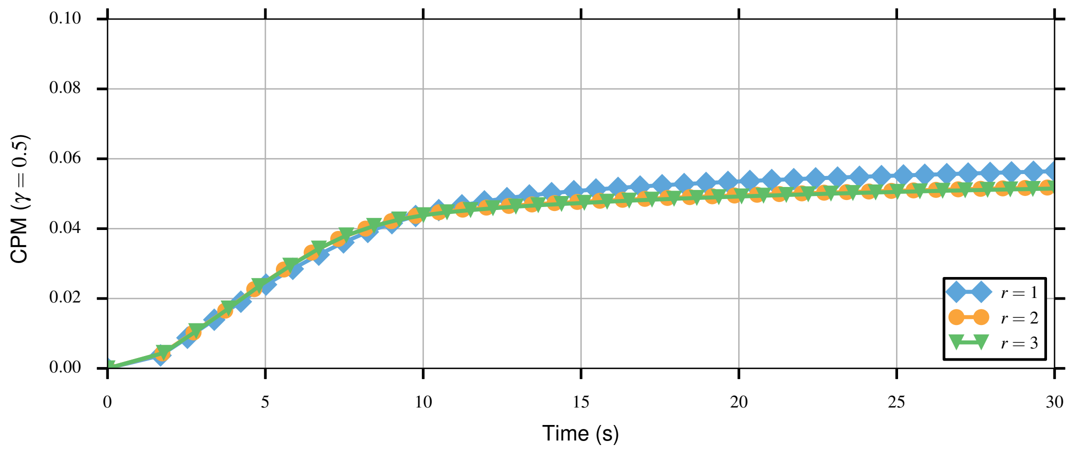


Figure 9.17: Best-neighbor parallel merging for CPM ($\gamma = 0.5$) on LiveJournal.

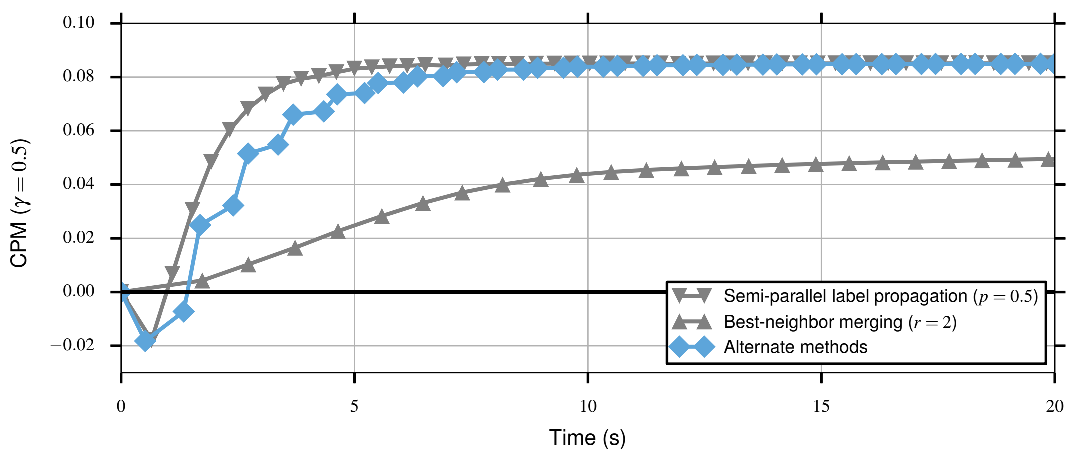


Figure 9.18: Combining both refinement techniques for ($\gamma = 0.5$) on LiveJournal.

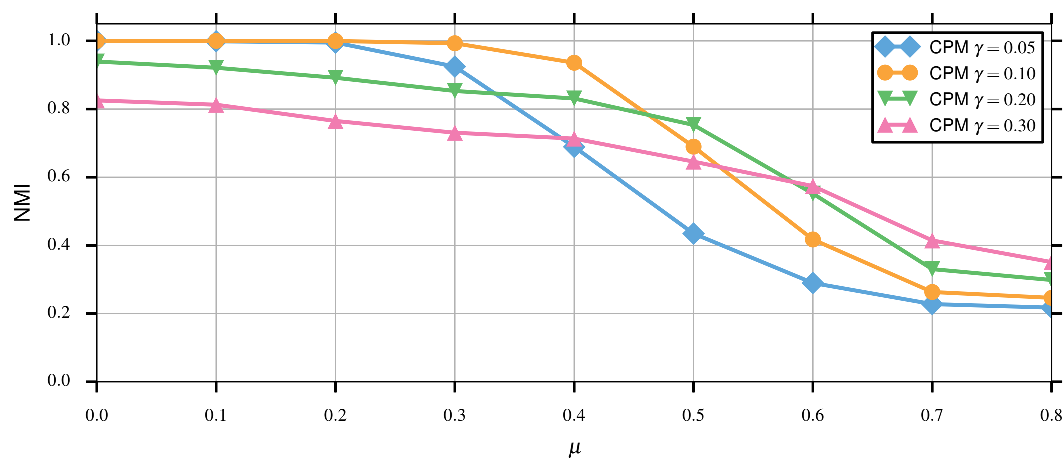


Figure 9.19: Performance of Par-CD for CPM on LFR benchmark ($n = 1000$, $k = 15$).

CHAPTER 10

Conclusions & Future Work

10.1 Summary

Community detection is the problem of partitioning a network into communities, i.e., densely interconnected clusters which are sparsely connected to each other. Finding these communities in an important field of research since they give insight into the structure and the topology of real-world network. However, since the concept of a community is intuitive and informal, community detection is not straightforward. Many different algorithms and quality metrics for community detection have been proposed. No algorithm or metric is universally accepted.

In this thesis, we presented the design, implementation, and evaluation of *Par-CD*: a flexible framework for parallel community detection in large networks. This framework is based on the philosophy that no single unique partitioning of a network into communities exists, but instead, community detection should be seen as a generic optimization problem where the goal is to optimize for a chosen quality metric.

10.2 Contributions

Par-CD represents a clear answer to our research question. We have empirically shown that it is possible to build a framework which is *flexible* (Par-CD currently supports three metrics), *fast* (Par-CD can process a network having 35M edges in less than 38 seconds), and highly parallel (Par-CD has been implemented for multi-core CPUs and GPUs).

In Chapter 3, we analyzed the most well-known community quality metrics. From

these metrics, we have selected the three most interesting ones: **modularity**, since it is the most popular metric, **CPM**, since it includes a resolution parameter, and **WCC**, since it is a novel metric based on triangle counting instead of density.

In Chapters 4 and 5, we analyzed two optimization techniques which are commonly used for community detection: **label propagation** and **merging**. We have seen how both techniques can be adapted to use any quality metric. Additionally, we explored how these methods can be parallelized. Label propagation can be parallelized by updating only a fraction of all vertices in parallel, i.e., **semi-parallel label propagation**. Merging can be parallelized by merging each community with the adjacent community for which merging is most beneficial, i.e., **best-neighbor parallel merging**. Both techniques are integrated into Par-CD.

In Chapters 7 and 8, we presented implementations of Par-CD for two modern parallel platforms: **multi-core CPUs** and **GPUs**. Since Par-CD has been designed to be highly parallel, implementing a parallel version of Par-CD for both platforms is feasible. Par-CD obtains excellent performance on both platforms, although using different approaches.

In Chapter 9, we compared the performance and accuracy of Par-CD against existing community detection algorithms for the three metrics. Benchmarks show that quality of the communities found by Par-CD is competitive to those found by the Louvain method, which performs best in the evaluation. However, Par-CD is significantly faster. On a network with 35M edges, the Louvain method takes 115 seconds and yields modularity of 0.745, while Par-CD takes 38 seconds (∼3 times faster) and yields modularity of 0.723 (∼3% difference). For WCC, Par-CD performs slightly worse than SCD, an algorithm based on WCC maximization. This is due to an additional phase of SCD which preprocesses the input network. For CPM, assessing the performance of Par-CD is difficult since no algorithms based on CPM maximization exist yet.

10.3 Future Research

There are a number of directions in which this work can be extended. First of all, adding support for additional quality metrics to Par-CD is both interesting and challenging. As discussed in Chapter 6, both refinement techniques require the user to define an **adhesion** function for the chosen metric. This function measures the improvement for the metric when merging two communities. It is interesting to see for which metrics such an adhesion function can be defined, or, if it cannot be defined, how much is required to modify Par-CD to support these metrics.

Second, extending Par-CD with additional optimization techniques can further extend its generality. For example, we have not explored optimization techniques

based on splitting communities. We have observed that, under some conditions, communities can get too large and splitting them is beneficial. However, with the current two refinement techniques, these communities cannot be reduced in size.

Third, Par-CD can be extended with support for different types of graphs. In this thesis, we have only focused on graphs which are undirected and unweighted. It can be interesting to see how much work is required to add support for directed or weighted graphs. Additionally, we have only considered *static* networks, i.e., networks which are fixed and known beforehand. Real-world networks, on the other hand, often change during their lifetime and edges/vertices are removed/added over time. It is interesting to see how Par-CD can be used for these *dynamic* networks, without recalculating all communities after each time step.

Finally, in Chapter 8, we found that some networks cannot be processed on low-end GPUs since these devices have insufficient memory. This problem can be solved using a heterogeneous approach by processing some vertices on the CPU and the remaining vertices on the GPU. We have already demonstrated the feasibility of this approach in *Het-SCD* [HVPPLP15], a heterogeneous implementation of SCD which combines the larger computational power of GPUs with the larger memory of CPUs. However, this implementation is specific to SCD and further research is needed to apply a similar approach to Par-CD.

10.4 Final Remarks

The source code of Par-CD has been released [1] under the GNU GPLv3 license. We encourage the reader to explore the source code and give permission to extend, distribute, or modify our work. We are open to suggestions and ideas towards improving Par-CD. We believe community detection is an interesting and challenging field of research and many questions related to community detection remain unexplored and unanswered. Par-CD is a tool that can be used for finding answers to these questions.

[1] `https://github.com/stijnh/Par-CD`

CHAPTER 11

Bibliography

[ACLY00] Rudolf Ahlswede, Ning Cai, S-YR Li, and Raymond W Yeung. Network information flow. *Information Theory, IEEE Transactions on*, 46(4):1204–1216, 2000.

[ADGPV06] Alex Arenas, Albert Díaz-Guilera, and Conrad J Pérez-Vicente. Synchronization reveals topological scales in complex networks. *Physical review letters*, 96(11):114102, 2006.

[AJB99] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Internet: Diameter of the world-wide web. *Nature*, 401(6749):130–131, 1999.

[AK08] Gaurav Agarwal and David Kempe. Modularity-maximizing graph communities via mathematical programming. *The European Physical Journal B-Condensed Matter and Complex Systems*, 66(3):409–418, 2008.

[Alb73] Richard D Alba. A graph-theoretic definition of a sociometric clique. *Journal of Mathematical Sociology*, 3(1):113–126, 1973.

[BA99] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[Bal85] Alexandru T Balaban. Applications of graph theory in chemistry. *Journal of Chemical Information and Computer Sciences*, 25(3):334–343, 1985.

[BBPP99] Immanuel M Bomze, Marco Budinich, Panos M Pardalos, and Marcello Pelillo. The maximum clique problem. In *Handbook of combinatorial optimization*, pages 1–74. Springer, 1999.

[BC10] P. Basuchowdhuri and P. Chen. Detecting communities using social ties. In *Granular Computing (GrC), 2010 IEEE International Conference on*, pages 55–60, Aug 2010.

[BGLL08] Vincent D 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):P10008, 2008.

[BLM+06] Stefano Boccaletti, Vito Latora, Yamir Moreno, Martin Chavez, and D-U Hwang. Complex networks: Structure and dynamics. *Physics reports*, 424(4):175–308, 2006.

[Bol98] Béla Bollobás. *Modern graph theory*, volume 184. Springer Science & Business Media, 1998.

[BPSDGA04] Marián Boguñá, Romualdo Pastor-Satorras, Albert Díaz-Guilera, and Alex Arenas. Models of social networks based on social distance attachment. *Physical Review E*, 70(5):056122, 2004.

[CG10] Gennaro Cordasco and Luisa Gargano. Community detection via semi-synchronous label propagation algorithms. In *Business Applications of Social Network Analysis (BASNA), 2010 IEEE International Workshop on*, pages 1–8. IEEE, 2010.

[CLM85] Pedro Celis, Per-Åke Larson, and Ian J Munro. Robin hood hashing. In *Foundations of Computer Science, 1985., 26th Annual Symposium on*, pages 281–288. IEEE, 1985.

[CNM04] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[DA05] Jordi Duch and Alex Arenas. Community detection in complex networks using extremal optimization. *Physical review E*, 72(2):027104, 2005.

[DDGDA05] Leon Danon, Albert Diaz-Guilera, Jordi Duch, and Alex Arenas. Comparing community structure identification. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(09):P09008, 2005.

[DGDGL07] Emilio Di Giacomo, Walter Didimo, Luca Grilli, and Giuseppe Liotta. Graph visualization techniques for web clustering engines. *Visualization and Computer Graphics, IEEE Transactions on*, 13(2):294–304, 2007.

[DM] NVIDIA Research Duane Merrill. Cub: State-of-the-art, reusable software components for every layer of the cuda programming model. `http://nvlabs.github.io/cub/`. last accessed: July 17, 2015.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[ER61] Paul Erdos and Alfréd Rényi. On the evolution of random graphs. *Bull. Inst. Internat. Statist*, 38(4):343–347, 1961.

[Fac] Facebook. Facebook: Connect with friends and the world around you on Facebook. `https://www.facebook.com/`. Accessed: 2015-06-27.

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

[FCF11] Adrien Friggeri, Guillaume Chelius, and Eric Fleury. Triangles to capture social cohesion. In *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*, pages 258–265. IEEE, 2011.

[FLG00] Gary William Flake, Steve Lawrence, and C Lee Giles. Efficient identification of web communities. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 150–160. ACM, 2000.

[FLM04] Santo Fortunato, Vito Latora, and Massimo Marchiori. Method to find community structures based on information centrality. *Physical review E*, 70(5):056104, 2004.

[FM82] Charles M Fiduccia and Robert M Mattheyses. A linear-time heuristic for improving network partitions. In *Design Automation, 1982. 19th Conference on*, pages 175–181. IEEE, 1982.

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

[FVS11] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[GA05] Roger Guimera and Luis A Nunes Amaral. Functional cartography of complex metabolic networks. *Nature*, 433(7028):895–900, 2005.

[GN02] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[Har05] Mark Harris. Gpgpu: General-purpose computation on gpus. *SIGGRAPH 2005 GPGPU COURSE*, 2005.

[HVPPLP15] S. Heldens, A. L. Varbanescu, A. Prat-Prez, and J. L. Larriba-Pey. Towards community detection on heterogeneous platforms. In *HeteroPar2015: Thirteenth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*. Springer, 2015.

[HW08] Jake M Hofman and Chris H Wiggins. Bayesian approach to network modularity. *Physical review letters*, 100(25):258701, 2008.

[KASM05] Junzo Kamahara, Tomofumi Asakawa, Shinji Shimojo, and Hideo Miyahara. A community-based recommendation system to reveal unexpected interests. In *Multimedia Modelling Conference, 2005. MMM 2005. Proceedings of the 11th International*, pages 433–438. IEEE, 2005.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[KL70] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 49(2):291–307, 1970.

[Kur03] Yoshiki Kuramoto. *Chemical oscillations, waves, and turbulence.* Courier Dover Publications, 2003.

[KW08] Jongkwang Kim and Thomas Wilhelm. What is a complex graph? *Physica A: Statistical Mechanics and its Applications*, 387(11):2637–2652, 2008.

[LFK09] Andrea Lancichinetti, Santo Fortunato, and János Kertész. Detecting the overlapping and hierarchical community structure in complex networks. *New Journal of Physics*, 11(3):033015, 2009.

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

[Lin] LinkedIn. LinkedIn: World's largest professional network. https://linkedin.com/. Accessed: 2015-06-27.

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[LLM10] Jure Leskovec, Kevin J Lang, and Michael Mahoney. Empirical comparison of algorithms for network community detection. In *Proceedings of the 19th international conference on World wide web*, pages 631–640. ACM, 2010.

[LS69] Fabrizio Luccio and Mariagiovanna Sami. On the decomposition of networks in minimally interconnected subnetworks. *Circuit Theory, IEEE Transactions on*, 16(2):184–188, 1969.

[MIH99] Hideo Matsuda, Tatsuya Ishihara, and Akihiro Hashimoto. Classifying molecular sequences using a linkage graph with their pairwise similarities. *Theoretical Computer Science*, 210(2):305–325, 1999.

[Mil67] Stanley Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[Mok79] Robert J Mokken. Cliques, clubs and clans. *Quality & Quantity*, 13(2):161–173, 1979.

[MPS02] William Magro, Paul Petersen, and Sanjiv Shah. Hyper-threading technology: Impact on compute-intensive workloads. *Intel Technology Journal*, 6(1), 2002.

[MV07] Oliver Mason and Mark Verwoerd. Graph theory and networks in biology. *Systems Biology, IET*, 1(2):89–119, 2007.

[New03] Mark EJ Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[New04] Mark EJ Newman. Fast algorithm for detecting community structure in networks. *Physical review E*, 69(6):066133, 2004.

[New06] Mark EJ Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical review E*, 74(3):036104, 2006.

[NG04] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[NL07] Mark EJ Newman and Elizabeth A Leicht. Mixture models and exploratory analysis in networks. *Proceedings of the National Academy of Sciences*, 104(23):9564–9569, 2007.

[NP03] Mark EJ Newman and Juyong Park. Why social networks are different from other types of networks. *Physical Review E*, 68(3):036122, 2003.

[NVI07] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

[PDFV05] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818, 2005.

[PKVS12] Symeon Papadopoulos, Yiannis Kompatsiaris, Athena Vakali, and Ploutarchos Spyridonos. Community detection in social media. *Data Mining and Knowledge Discovery*, 24(3):515–554, 2012.

[PL05] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. In *Computer and Information Sciences-ISCIS 2005*, pages 284–293. Springer, 2005.

[PPDSBLP12] Arnau Prat-Pérez, David Dominguez-Sal, Josep M Brunat, and Josep-Lluis Larriba-Pey. Shaping communities out of triangles. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1677–1681. ACM, 2012.

[PPDSLP11] Arnau Prat-Pérez, David Dominguez-Sal, and Josep L Larriba-Pey. Social based layouts for the increase of locality in graph operations. In *Database Systems for Advanced Applications*, pages 558–569. Springer, 2011.

[PPDSLP14] Arnau Prat-Pérez, David Dominguez-Sal, and Josep-LLuis Larriba-Pey. High quality, scalable and parallel community detection for large real graphs. In *Proceedings of the 23rd international conference on World wide web*, pages 225–236. International World Wide Web Conferences Steering Committee, 2014.

[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.

[RB04] Jörg Reichardt and Stefan Bornholdt. Detecting fuzzy community structures in complex networks with a potts model. *Physical Review Letters*, 93(21):218701, 2004.

[RB06] Jörg Reichardt and Stefan Bornholdt. Statistical mechanics of community detection. *Physical Review E*, 74(1):016110, 2006.

[RCC+04] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences of the United States of America*, 101(9):2658–2663, 2004.

[RMEB12] E Jason Riedy, Henning Meyerhenke, David Ediger, and David A Bader. Parallel community detection for massive graphs. In *Parallel Processing and Applied Mathematics*, pages 286–296. Springer, 2012.

[RN10] Peter Ronhovde and Zohar Nussinov. Local resolution-limit-free potts model for community detection. *Physical Review E*, 81(4):046114, 2010.

[SAS07] Xiaolin Shi, Lada A Adamic, and Martin J Strauss. Networks of strong ties. *Physica A: Statistical Mechanics and its Applications*, 378(1):33–47, 2007.

[SCL+12] Ching-Lung Su, Po-Yu Chen, Chun-Chieh Lan, Long-Sheng Huang, and Kuo-Hsuan Wu. Overview and comparison of opencl and cuda technology for gpgpu. In *Circuits and Systems (APCCAS), 2012 IEEE Asia Pacific Conference on*, pages 448–451. IEEE, 2012.

[Sei83] Stephen B Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.

[SF78] Stephen B Seidman and Brian L Foster. A graph-theoretic generalization of the clique concept*. *Journal of Mathematical sociology*, 6(1):139–154, 1978.

[SGS10] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[SM97] J Shi and J Malik. Normalized cuts and image segmentation. *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition*, 1997.

[Str01] Steven H Strogatz. Exploring complex networks. *Nature*, 410(6825):268–276, 2001.

[TVDN11] Vincent A Traag, Paul Van Dooren, and Yurii Nesterov. Narrow scope for resolution-limit-free community detection. *Physical Review E*, 84(1):016114, 2011.

[VD00] Stijn Marinus Van Dongen. *Graph clustering by flow simulation.* PhD thesis, University of Utrecht, 2000.

[VL06] I Vragović and E Louis. Network community structure and loop coefficient method. *Physical Review E*, 74(1):016105, 2006.

[WC89] Yen-Chuen Wei and Chung-Kuan Cheng. Towards efficient hierarchical designs by ratio cut partitioning. In *Computer-Aided Design, 1989. ICCAD-89. Digest of Technical Papers., 1989 IEEE International Conference on*, pages 298–301. IEEE, 1989.

[WS98] Duncan J Watts and Steven H Strogatz. Collective dynamics of small-worldnetworks. *nature*, 393(6684):440–442, 1998.

[WT07] Ken Wakita and Toshiyuki Tsurumi. Finding community structure in mega-scale social networks. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 1275–1276, New York, NY, USA, 2007. ACM.

[XSL11] Jierui Xie, Boleslaw K Szymanski, and Xiaoming Liu. Slpa: Uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process. In *Data Mining Workshops*

(ICDMW), 2011 IEEE 11th International Conference on, pages 344–349. IEEE, 2011.

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

[Zac77] Wayne W Zachary. An information flow model for conflict and fission in small groups. *Journal of anthropological research*, pages 452–473, 1977.

[ZL04] Haijun Zhou and Reinhard Lipowsky. Network brownian motion: A new method to measure vertex-vertex proximity and to identify communities and subcommunities. In *Computational Science-ICCS 2004*, pages 1062–1069. Springer, 2004.

APPENDIX A

Graph Theory

Formally, a graph is a tuple $G = (V, E)$ consisting of a set V, containing *vertices* or *nodes*, and a set of pairs $E \subseteq V \times V$, called *edges*. If the edges are *ordered* pairs then the graph is said to be *directed*. In other words, edges have an orientation and thus edge (a, b) is different from edge (b, a). If the edges are *unordered* pairs then the graph is said to be *undirected*. For these graphs, edges have no orientation and thus edge (a, b) is equal to (b, a).

The *order* of a graph is the number of vertices and is denoted by $|V|$. The *size* of a graph is the number of edges and is denoted by $|E|$. A graph is called *dense* if the number of edges is very close to the maximum number of edges, otherwise it is called *sparse*. A directed graph of order n can have at most $n(n-1)$ edges, or n^2 edges if one allows *loops* in the graph. A loop is an edge (u, u) which connects a vertex u to itself. An undirected graph of order n can have at most $\frac{1}{2}n(n-1)$ edges, or $\frac{1}{2}n(n+1)$ edges if one allows loops. A graph which has the maximum number of edges is called a *complete graph* or *fully connected graph* since every vertex is directly connected to every other vertex.

Two vertices u and v are said to be *adjacent* to each other if there exists and edge $(u, v) \in E$ which links the two vertices together. Alternatively, we also say that u and v are *neighbors* of each other and that u lies in the *neighborhood* of v. The number of neighbors of a vertex is called the *degree* of the vertex. An *adjacency matrix* is a $n \times n$ matrix where $A_{ij} = 1$ if $(v_i, v_j) \in E$ and $A_{ij} = 0$ otherwise.

A *walk* is a sequence of vertices $v_1, ..., v_k$ such that an edge exists between every two consecutive vertices. The *length* of a walk is the number of edges that it goes over or, equivalently, the number of vertices minus one. A *path* is a walk where all v_i are distinct. A *cycle* or *circuit* is a walk where all v_i are distinct and, additionally, v_1 and v_k are adjacent.

A vertex is said to be *reachable* from another vertex if there exists a valid path

between them. If every vertex is reachable from every other vertex, then a graph is said to be *connected*, otherwise it is *disconnected*. The *geodesic distance*, or simply *distance*, between two vertices is the length of the shortest path between them. If one vertex is not reachable from another vertex then there exists no path between them so their distance is conventionally defined as infinite. The *diameter* of a graph is the greatest geodesic distance over all pairs of vertices. By definition, the diameter of a disconnected graph is infinity.

A graph $G' = (V', E')$ is said to be a *subgraph* of graph G if $V' \subseteq V$ and $E' \subseteq V' \times V' \cap E$. A subgraph is called a *full* subgraph if the set of edges is maximal, so $E' = V' \times V' \cap E$. A disconnected graphs can be partitioned into connected subgraphs called *connected components*. Each connected component of a disconnected graph consists of all vertices which are reachable by all other vertices in the same component. Connected graphs have, by definition, only one connected component.

APPENDIX B

Approximation Model for WCC when Merging Communities

The *Weighted Community Clustering* metric [PPDSBLP12] is a community quality metric based on counting triangles. Let $G = (V, E)$ be an unweighted, undirected graph. We define $t(v, C)$ as the number of triangles vertex v closes with vertices in $C \subseteq V$ and $vt(v, C)$ as the number of vertices in C that close at least one triangle with v and any other vertex from G. If $t(v, V) > 0$ then the *cohesion* of vertex v to community C is defined as follows [PPDSBLP12]:

$$WCC(v, C) = \frac{t(v, C)}{t(v, V)} \cdot \frac{vt(v, V)}{|C \backslash \{v\}| + vt(v, V \backslash C)} \tag{B.1}$$

Consider the following scenario: a vertex v is member of community X and community X will be merged with community Y. From the perspective of vertex v, merging these communities equivalent to saying that v is *transferred* from community X to $X \cup Y$. We define $WCC_{transfer}(v, X, X \cup Y)$ as the difference in WCC for v before and after the merging the two communities.

$$WCC_{transfer}(v, X, X \cup Y) = WCC(v, X \cup Y) - WCC(v, X) \tag{B.2}$$

$$= \frac{t(v, X \cup Y)}{t(v, V)} \cdot \frac{vt(v, V)}{|X| + |Y| - 1 + vt(v, V \backslash \{X \cup Y\})} - \frac{t(v, X)}{t(v, V)} \cdot \frac{vt(v, V)}{|X| - 1 + vt(v, V \backslash X)} \tag{B.3}$$

$$= \frac{vt(v, V)}{t(v, V)} \left(\frac{t(v, (X \cup Y))}{|X| + |Y| - 1 + vt(v, V \backslash \{X \cup Y\})} - \frac{t(v, X)}{|X| - 1 + vt(v, V \backslash X)} \right) \tag{B.4}$$

Unfortunately, the exact computation of $WCC_{transfer}(v, X, X \cup Y)$ is computationally expensive since it requires one to count the number of triangles v closes with X, $X \cup Y$ and V. Prat et al. encounter the same problem when they developed the SCD algorithm [PPDSLP14]. To solve this problem, they proposed a constant-time approximation model to calculate the value of $WCC_{transfer}(v, X, X \cup Y)$ when X or Y is a singleton, i.e. $|X| = 1$ or $|Y| = 1$. In collaboration with Prat et al., we designed a generalization of this model to remove this restriction. This new model is presented here.

The model is based on the following assumptions:

- Every edge closes at least one triangle.
- The edge density inside X, inside Y and between X and Y is homogeneous.
- The clustering coefficient[1] of the graph is homogeneous for all vertices outside of X and Y.

Next, we define the following symbols:

- ω: Average clustering coefficient of the graph.
- $n_X = |X|$, $n_Y = |Y|$: The number of vertices of X and Y.
- n_X^{conn}, n_Y^{conn}: The number of vertices of X/Y which are *connected* to Y/X, i.e. they have at least one edge towards the other community.
- n_X^{disc}, n_Y^{disc}: The number of vertices of X/Y which are *disconnected* from Y/X, i.e. they have no edges towards the other community.
- m_X, m_Y: The number of edges inside X and Y.
- b_X, b_Y: The number of edges at the boundary of X and Y.
- m_{XY}: The number of edges between X and Y.
- $p_X = \frac{m_X}{n_X(n_X-1)/2}$, $p_Y = \frac{m_Y}{n_Y(n_Y-1)/2}$: Edge density of X and Y.
- $p_{XY} = \frac{m_{XY}}{n_X^{conn} n_Y^{conn}}$: Edge density between X and Y.
- $q_X = \frac{b_X - m_{XY}}{n_X}$, $q_Y = \frac{b_Y - m_{XY}}{n_Y}$: Average number of neighbors outside $X \cup Y$ for vertices in X and Y.

Using these definitions, we define *approximations* for $t(v, C)$ and $vt(t, C)$ which can be used to estimate $WCC_{transfer}(v, X, Y)$. We consider two different scenarios.

[1] The clustering coefficient of a vertex is defined as ratio of the number of triangles the vertex closes with its neighbors to the total possible number of triangles is could close. It can be seen as the probability that two neighbors of a vertex are again neighbors of each other.

First, consider the scenario where $v \in X$ is connected to Y.

$$\begin{aligned}
t^{conn}(v, X) =& \binom{n_X - 1}{2} p_x^3 \\
t^{conn}(v, X \cup Y) =& \binom{n_X - 1}{2} p_X^3 + (n_X^{conn} - 1) n_Y^{conn} p_X p_Y p_{XY} + \binom{n_Y^{conn}}{2} p_{XY}^2 p_Y \\
t^{conn}(v, V) =& \binom{n_X - 1}{2} p_X^3 + (n_X^{conn} - 1) n_Y^{conn} p_X p_Y p_{XY} + \binom{n_Y^{conn}}{2} p_{XY}^2 p_Y \\
& + \binom{q_x}{2} \omega + q_X (n_X - 1) p_X \omega + n_Y^{conn} q_X p_{XY} \omega \\
vt^{conn}(v, V \backslash X \cup Y) =& q_X \\
vt^{conn}(v, V \backslash X) =& q_X + p_{XY} n_Y^{conn} \\
vt^{conn}(v, V) =& q_X + p_{XY} n_Y^{conn} + p_X (n_X - 1)
\end{aligned}$$

Second, consider the scenario where $v \in X$ is *not* connected to Y.

$$\begin{aligned}
t^{disc}(v, X) =& \binom{n_X - 1}{2} p_x^3 \\
t^{disc}(v, X \cup Y) =& \binom{n_X - 1}{2} p_X^3 \\
t^{disc}(v, V) =& \binom{n_X - 1}{2} p_X^3 + \binom{q_x}{2} \omega + q_X (n_X - 1) p_X \omega \\
vt^{disc}(v, V \backslash X \cup Y) =& q_X \\
vt^{disc}(v, V \backslash X) =& q_X \\
vt^{disc}(v, V) =& q_X + p_X (n_X - 1)
\end{aligned}$$

By inserting these equation into Eq. B.4, we define $WCC^{conn}_{transfer}(v, X, X \cup Y)$ and $WCC^{disc}_{transfer}(v, X, X \cup Y)$ to approximate $WWC_{transfer}(v, X, X \cup Y)$ for vertices $v \in X$ which are connected and not connected to Y, respective. Since these approximations do not depend on the properties of individual vertices but only rely on global statistics of X and Y, we can also omit the first parameter and simply write $WCC^{conn}_{transfer}(X, X \cup Y)$ and $WCC^{disc}_{transfer}(X, X \cup Y)$.

The exact computation of $WCC_{transfer}(x, X, X \cup Y)$ requires one to count the number of triangles x closes with X, $X \cup Y$ and V. Under the assumption that all vertices of G have degree d, the complexity of this operation is $\mathcal{O}(d^2)$. The approximations $WCC^{conn}_{transfer}(v, X, X \cup Y)$ and $WCC^{disc}_{transfer}(v, X, X \cup Y)$ have complexity $\mathcal{O}(1)$, given that the values of ω, n_X, n_Y, m_X, m_Y, b_X, b_Y and m_{XY} are available.

Furthermore, we assume that the edges between X and Y are connected to different

vertices from X and Y. The values of n_X^{conn}, n_Y^{conn}, n_X^{disc} and n_Y^{disc} can thus be defined as follows:

$$n_X^{conn} = min\{m_{XY}, n_X\} \tag{B.5}$$
$$n_Y^{conn} = min\{m_{XY}, n_Y\} \tag{B.6}$$
$$n_X^{disc} = max\{n_X - m_{XY}, 0\} \tag{B.7}$$
$$n_Y^{disc} = max\{n_Y - m_{XY}, 0\} \tag{B.8}$$
$$\tag{B.9}$$

The overall improvement in WCC when merging communities X and Y can now be approximated as follows.

$$WCC_{merge}(X,Y) \approx \frac{1}{|V|}(n_X^{conn}\ WCC_{transfer}^{conn}(X, X \cup Y) + n_X^{disc}\ WCC_{transfer}^{disc}(X, X \cup Y) + n_Y^{conn}\ WCC_{transfer}^{conn}(Y, X \cup Y) + n_Y^{disc}\ WCC_{transfer}^{disc}(Y, X \cup Y)) \tag{B.10}$$