Universiteit
Leiden

# Master Computer Science

## Faster Community Detection Without Loss of Quality:

## Parallelizing the Leiden Algorithm

Name: Geerten Verweij
Student ID: s1420062

Date: 25/08/2019

Specialisation: Advanced Data Analytics

1st supervisor: Frank Takes (LIACS)
2nd supervisor: Vincent Traag (CWTS)

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

# Abstract

Community detection algorithms are used to make an abstraction of a large complex network in order to make the structure of the network more comprehensible. These take more compute time for larger networks and networks can get as large as millions of nodes and billions of edges. This is why there is a need for fast community detection algorithms. CPU core count continues to increase and community detection algorithms could benefit from this if they would be implemented using parallelization techniques. The sequential Leiden community detection algorithm is an improvement over the widely adopted Louvain community detection algorithm. In this thesis we will try to parallelize and speed up the Leiden algorithm while trying not to reduce the quality of the solution. The result is a lock free parallel adaptation of the Leiden algorithm which is faster than the original sequential implementation.

# Acknowledgements

Thanks go out to LIACS, CWTS and LUMC for providing the tools and knowledge to do this research.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

This research is focused on parallelizing part of a community detection algorithm with the goal to improve execution speed while maintaining the quality of the result. The subject of network science is introduced in Section 1.1 and community detection is explained in Section 1.2. Section 1.3 explains why we parallelize an existing community detection algorithm. Section 1.4 gives an overview of the content of this thesis.

## 1.1 Network Science

Network science is a field of science that focuses on analyzing and understanding networks. A network, also often referred to as a graph, is a collection of entities with connections between each other. Examples are social networks, where the entities are human beings and the connections are friendships. Other examples are financial networks, biological networks or scientific collaboration networks. Formally, such a network can be described by $G = (V, E)$ where the network $G$ contains a set of nodes $V$ and a set of edges $E \subseteq V \times V$. The number of nodes in a network is $n = |V|$ and the number of edges in a network is $m = |E|$. The neighbourhood of a node $i$ is $N_i = \{j : (i, j) \in E\}$ where $j$ is another node in the network and $(i, j)$ is an undirected edge connecting node $i$ to node $j$ and vise-versa. The degree of a node $i$ is $k_i = |N_i|$ which is the number of edges connected to node $i$. Additional information about a node or

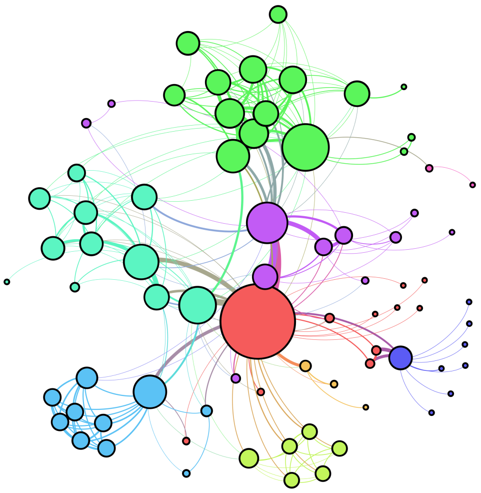

Figure 1.1: Example of a network (graph).

edge can be captured in metadata. This metadata can be quite diverse and very specific to a network, thus it is not often considered in general purpose network analysis methods. One exception is the weight of an edge which is very commonly used in network science. The weight of an edge indicates the intensity of the connection represented by the edge. For example, in a financial network the amount of money that flows between two entities can be represented by the weight of the edge between the two nodes. Figure 1.1 shows a simple example network based on mock data. The size of a node in this example is based on the degree of the node. This type of visualisation is a very common way of visualizing a network. However, for very large networks this representation is no longer usable because the image would become cluttered. In network science we try to extract knowledge from real world networks. Some examples are finding influential persons in a social network such as Twitter, discovering the largest conduits in a financial network, discovering what friends to recommend on Facebook, finding related products in a webshop or discovering crucial proteins in a protein interaction network. A major challenge in network science is the scale of the network and the speed of execution. Many algorithms used to analyze networks scale quadratically or worse when the network increases in size making analysis of large networks very time consuming. This research focuses on the scale and speed aspect of a particular network analysis algorithm.

## 1.2 Community Detection

A community in a network is a group of entities that has relatively more connections within the group than with entities outside of the group. An example would be a group of friends who all know each other, creating a strongly connected community. Sometimes interesting information about communities is captured in the metadata which can then serve as a ground truth. In other cases such information is unavailable or even a true group structure does not exist. Knowing the communities in a network can be useful to understand the structure of large networks. The community structure provides a higher level overview of how the network is constructed. The lower level structure of a network is too complex to understand when there are millions of entities and connections involved. The higher level community structure provides a better overview and helps to summarize many entities into one larger entity (community). We can apply community detection to the network in Figure 1.1 and merge all nodes that are in the same community into a single node. Figure 1.2 shows the network resulting from that operation. In both figures the

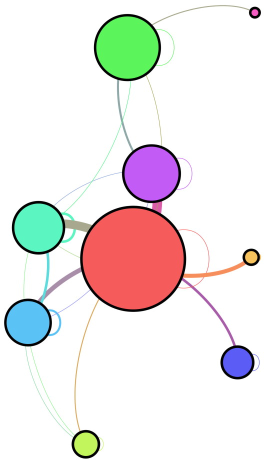

Figure 1.2: Example of communities in Figure 1.1 merged into single nodes.

community is also represented by the node color.

There are different methods to find these communities. One is optimizing modularity which is what the widely adopted Louvain algorithm tries to do heuristically. The algorithm moves each node to the community that gives the highest modularity gain until no further gain in modularity can be made. We will now explain what modularity is.

Let $C$ be a community partition for network $G = (V, E)$ where $C$ is a set of communities. A community $c \in C$ contains a set of nodes from $V$, for example $c = \{i, j, k, ...\}$, where $i, j, k \in V$. The actual number of edges in a community $c$ is given by $e_c = |\{(i, j) \in E : i, j \in c\}|$ where $(i, j)$ is an edge between node $i$ and node $j$ and both these nodes are in community $c$. Each node can only be in one community so $c_x \cap c_y = \emptyset$ for all $c_x, c_y \in C$. The sum of degrees of all nodes $v$ in a community $c$ is $K_c = \sum_{v \in c} k_v$ where $k_v$ is the degree of node $v$. The number of edges connected to any of the nodes in a community can be approximated by $\frac{K_c}{2}$ since summing over the degrees causes edges that are completely within $c$ to be counted twice. The fraction of edges from the entire network connected to any of the nodes in the community would then be $\frac{K_c}{2m}$ where $m$ is the total number of edges in the network. This also approximates the chance that an edge is involved in the community if the edges were connected at random. The expected number of of edges in a community $c$ is $\frac{K_c^2}{2m}$ if the edges were connected at random. This is derived by first breaking all edges leaving only the endpoints of the nodes. For a node $i$ there are then $k_i$ endpoints. For each of those endpoints we chose a random endpoint to connect to and the probability that an endpoint is chosen from the community $c$ is $\frac{K_c}{2m}$. When this is done for all $K_c$ endpoints we obtain $\frac{K_c^2}{2m}$. The modularity is calculated by comparing the expected number of edges in a community with the actual number of edges in a community and summing the difference over all the communities,

$$\mathcal{H} = \frac{1}{2m} \sum_c \left( e_c - \gamma \frac{K_c^2}{2m} \right). \tag{1.1}$$

The resolution parameter is $\gamma$: a higher resolution will result in more communities since it expects more edges to be inside the community by increasing the effect of $\frac{K_c^2}{2m}$. To optimize modularity many possible community partition have to be considered and the number of possible partition grows exponentially with the size of the network. Heuristic algorithms like the Louvain algorithm do not test every possible partition since this takes exponential time, instead the algorithm searches for a maximum modularity by moving nodes to the community with the highest modularity gain until no further gains can be found.

## 1.3 Parallelization

Community detection is more important for larger networks but also computationally harder. In fact some networks are so large that it would take many minutes to finish one analysis, which is a problem in real-

time analysis contexts. The larger a network the more valuable community detection becomes, as mentioned before. Therefore, this research is focused on speeding up an existing community detection algorithm.
The Leiden algorithm as described in [TWvE19] by Traag *et al* provides a speed and quality improvement compared to the existing Louvain algorithm. We will not try to change the algorithm itself but rather how it is executed. The Leiden algorithm is a sequential algorithm, meaning it does all operations step by step, in order. Many modern computers have multiple cores allowing for operations to be distributed over multiple threads which are then executed in parallel. However, implementing a parallel version of an algorithm is not always trivial. The different threads might share some data, in which case locking is needed. Locking data in parallel computing means that the data will be used for reading and/or writing by only one thread at one time in order to preserve the correctness of this data. This locking of data might result in a bottleneck in the computation because threads will wait for other threads when the data is needed simultaneously. Not using any locking might result in incorrect results, while locking data too often might result in a large overhead and slow execution speed. When parallelizing a community detection algorithm this might prove to be a challenge since all the threads will be working on the same community partition data. This raises the question: **Could the Leiden algorithm be executed with greater speed using a parallel implementation without sacrificing the quality of the community detection?**

## 1.4 Thesis Outline

This thesis will cover other work related to community detection and parallelization in Chapter 2. Chapter 3 is about our approach of parallelizing community detection and covers the Louvain and Leiden algorithm it also discusses different approaches and challenges in parallelizing the Leiden algorithm, including our proposed approach. Chapter 4 explains the experimental setup and shows the results of experiments and evaluates these results. Finally, conclusions are drawn in chapter 5 and future work is discussed.

# Chapter 2

# Related Work

Section 2.1 will explain several community detection algorithms. Section 2.2 will discuss related works on parallel community detection algorithms, concluding in Section 2.3 with what we learned from that.

## 2.1 Community Detection Algorithms

A community detection algorithms takes as input a network and produces a community partition of the nodes from the input. The community partition is an assignment of each node to a specific community.

### 2.1.1 Louvain

The Louvain algorithm is a widely adopted heuristic community detection algorithm. It was introduced in the work of D Blondel et al. [BGLL08]. The Louvain algorithm will be explained in more detail in Chapter 3.

### 2.1.2 Leiden

In the work of Traag et al. an adaptation of the Louvain algorithm is presented [TWvE19] called the Leiden algorithm. This adaptation shows improved execution speed and solution quality over the Louvain algorithm. This will be discussed in more detail in Chapter 3.

### 2.1.3 Infomap

In [RB08] by Rosvall et al. the Infomap algorithm is introduced. The Infomap algorithm is comparable to the Louvain method in how it tries to optimize the communities assigned to the nodes. However it does not optimize modularity but it optimizes the map equation. The algorithm considers the limits of encoding random walks over the network as a series of bit codes assigned to all the nodes. The prefixes of the bit codes describe the community a node is in. The map equation will give a higher score when these walks can be expressed in a more compact way. This happens when bit codes can be reused by nodes in different communities because they have a different prefix.

## 2.2 Parallelized Community Detection

One way of making community detection algorithms faster is to execute the computational work in parallel over multiple compute units. We will now discuss different approaches of parallelization applied to different community detection algorithms.

### 2.2.1 Message Passing Interface

The Message Passing Interface (MPI) is an interface designed to allow the writing of code that can be executed by distributed systems containing many machines that work together in a highly scalable parallel fashion. In [ZY18] by Jianping Zeng et al. an MPI implementation of the Infomap community detection algorithm is presented. They managed to scale the Infomap algorithm to massively distributed systems allowing thousands of CPU threads to work on the same community optimization in parallel. This is useful for doing community detection for large networks on clusters of computers. Using the MPI implementation of Infomap they show a small speed-up for small data-sets and a larger speed-up for larger data-sets. This is typical since the parallelization overhead created by the communication between compute-units becomes less significant when each compute-unit has to do more work. On their largest data-set of 105.9 million nodes and 3.78 billion edges they achieve a 6 times speed-up.

### 2.2.2 GPU Acceleration

Graphical Processing Units (GPU's) contain many (thousands) of compute units on a single processor chip to do simple tasks in a massive parallel fashion. GPU's are commonly applied to speed-up 3D rendering in games and animation. However, nowadays GPU's are also very popular in scientific computations such as neural networks and physics simulations. Naim et al. have worked on speeding up the existing Louvain algorithm [NMHT17]. In their work they show a GPU implementation of the Louvain algorithm that is also adaptive. The algorithm is adaptive in the quality threshold, meaning that the algorithm will stop early once the quality does not seem to improve much anymore. These two improvements showed a speedup of 2.7 up to 312 compared to the sequential algorithm. When adding the adaptive part to the sequential algorithm they showed that the GPU algorithm gives a speedup of 1 to 27 compared to the sequential adaptive algorithm. The modularity decrease was less than 0.13%. A challenge they overcame was the large difference in degree between nodes. A node with a high degree requires more computation time and this results in an unbalanced run time for each node visited. This was solved by making the number of threads assigned to each computation depend on the degree of a node. This means that nodes with a high degree get more threads assigned to them, making the computation time for all nodes more balanced. This is specific for GPU computations since the idea behind GPU computation is to have many simple compute units doing the same simple task on different parts of the data in a synchronized streaming way.

### 2.2.3 Heterogeneous Computing

In [HVPPLP15], Heldens et al. presents a heterogeneous approach to community detection called Het-SCD. Their basis is the Scalable Community Detection (SCD) which is introduced in [PPDSLP14] by Prat-Pérez et al. SCD uses the Weighted Community Clustering (WCC) which was introduced in [PPDSBLP12] also by Prat-Pérez et al. WCC scores a community partition based on the relative amount of closed triangles in a community. They show that GPUs give a performance increase to the SCD algorithm. They also acknowledge that the size of a GPU's memory is not sufficient for very large graphs. They present a heterogeneous method where the computation is split over the larger CPU memory and the smaller GPU memory. This resulted in a speedup over a parallel CPU-only implementation for a network that was too large for GPU memory. They did not implement the entire algorithm with the hybrid approach but only what they call the refinement phase which was 41.5 times faster compared to the sequential version for the Live Journal data-set. For end-to-end execution this resulted in a reduction in execution time between 25% and 50% depending on which GPU was used.

### 2.2.4 Locking

As discussed in Section 1.3, parallel programming often comes with the challenge of managing the shared data between threads. A non-trivial problem are read/write collisions which happens when one thread is altering data while another thread is reading that data. When this happens and one wants to make sure each thread gets the correct data there is need for locking of the data. This means that only one of the multiple threads can access the data at any given time. This could potentially result in threads being idle while they wait on data becoming available which reduces efficiency. RelaxMap is a method proposed by Seung-Hee Bae et al., in [BHW$^{+}$17]. It is a variation of the Infomap algorithm with parallelization and prioritization. The parallelization achieves high efficiency thanks to their relaxed lock-free approach. The lock-free approach is needed because the Infomap algorithm is sequential by nature and would remain sequential if locking would be applied since threads would be waiting on each other resulting in threads taking turns. Among other results they show a 17.6x speed-up without loss of quality using 60-way parallelism. They empirically find that the lock-free approach does not cause loss of output quality due to the sparse nature of the networks. Their expectation is that this concept is general enough so that it could also apply to other community detection algorithms. Another observation they make is a correlation between the average degree of a network and the effectiveness of the parallelization 'where denser (higher average degree) graphs tend to achieve better performance than sparser (lower average degree) graphs' [BHW$^{+}$17]. The meaning of the word graph in this quote is the same as the meaning of the word network in this thesis. The reason for this difference could be that for denser networks each threads spends relatively more time on optimizing the community assignment for each node end less time on communicating the results to other threads.

## 2.3 Inspiration from Related Works

These related works show that parallel implementations of community detection algorithms can be very effective in providing faster execution without significant loss of quality in the results. Many of these approaches depend on specific hardware, such as clusters of multiple computers or GPU's. In this work we focus on a solution that is meant for CPU's on a single machine. We hope this makes the solution more applicable for community detection on consumer level desktops. This makes the algorithm more suitable as a plugin for graph analysis tools such as Gephi [BHJ09]. However, to explore the full capability we will test our implementation of the Leiden algorithm on many core machines with high RAM capacity. We will also put the lock-free approach from [BHW$^{+}$17] to the test and try to see how this impacts the quality of the algorithm's results.

# Chapter 3

# Approach

In this chapter we will explain our approach for improving the execution speed of community detection without loss of quality. Section 3.1 will explain in more detail how the Louvain and Leiden community detection algorithms work. In Section 3.2 we will discuss approaches for parallelizing the Leiden algorithm that have been considered while developing our proposed parallelization approach. Section 3.3 will explain our parallel implementation of the Leiden community detection algorithm that was further analysed in our experiments in Chapter 4.

## 3.1 Existing Sequential Approaches

### 3.1.1 Louvain

The Louvain algorithm, introduced in [BGLL08] by Blondel et al., is a widely adopted heuristic community detection algorithm that uses modularity. In Section 1.2 we defined modularity with the help of Equation 1.1. The Louvain algorithm starts with a community partition where each node is in its own community. Modularity is then optimized by moving nodes to communities that give the highest modularity gain. This is done by calculating the potential modularity gain for each potential community when the node would be moved to that community. Once no more moves are found that increase modularity, all nodes that are in the same community are aggregated into new nodes. Modularity is optimized again for this aggregated network resulting in a new community partition. This moving and aggregating is repeated until each community in the aggregated network contains one node. The aggregated network is then de-aggregated or flattened back into the original network. The resulting community partition can be the input for a new iteration of the algorithm to improve the quality of the partition or the community partition can be returned and the algorithm terminates. The number of iterations of the algorithm is configurable.

Algorithm 1 shows the outer loop of the Louvain algorithm and the MoveNodes function. Other parts of the algorithm are left out of this pseudo code because the remainder of this research focuses on the MoveNodes function. This pseudo code comes from [TWvE19]. Community partition $P$, which is a of the set of communities just like $C$ mentioned in Section 1.2, starts initially as a singleton partition where each node is assigned to a community containing just itself. The algorithm terminates when the condition on line 4 is met where there are as many communities as nodes. This happens after all nodes in a community are aggregated into a single node and then put into a singleton partition on line 6, 7 and when subsequently no nodes are moved during the MoveNodes function on line 3. On line 10 the Flat function is called which unfolds the node aggregation creating as many nodes as there were at the start but each node is assigned to the community that was found while the nodes were aggregated. This is further explained in [TWvE19]. The MoveNodes function is a loop over all nodes and for each node all neighboring communities are considered, as well as a new empty community. The community that provides the highest modularity gain is chosen on line 16. Only if there is a strictly positive gain in modularity the node is actually moved, which happens on line 18. Line 21 describes the condition for termination of the MoveNodes function: if there was no modularity gain in one loop over all nodes, the MoveNodes function terminates. This means that if even one node is moved, the loop will in line 13 will repeat over all nodes. We will see later that this is a big difference compared to the Leiden algorithm.

**Algorithm 1** parts of the Louvain algorithm.

```
 1: function Louvain(Graph $G$, Partition $P$)
 2:     do
 3:         $P \leftarrow$ MoveNodes$(G, P)$
 4:         done $\leftarrow |P| = |V(G)|$
 5:         if not done then
 6:             $G \leftarrow$ AggregateGraph$(G, P)$
 7:             $P \leftarrow$ SingletonPartition$(G)$
 8:         end if
 9:     while not done
10:     return Flat*$(P)$
11: end function

12: function MoveNodes(Graph $G$, Partition $P$)
13:     do
14:         $\mathcal{H}_{old} = \mathcal{H}(P)$
15:         for $v \in V(G)$ do
16:             $C' \leftarrow argmax_{C \in P \cup \emptyset} \Delta\mathcal{H}_P(v \mapsto C)$
17:             if $\Delta\mathcal{H}_P(v \mapsto C') > 0$ then
18:                 $v \mapsto C'$
19:             end if
20:         end for
21:     while $\mathcal{H}(P) > \mathcal{H}_{old}$
22:     return $P$
23: end function
```

### 3.1.2 Leiden

In [TWvE19] by Traag et al., an adaptation of the aforementioned Louvain algorithm is presented. The presented adaptation of the Louvain algorithm, called the Leiden algorithm, has two improvements over the Louvain algorithm. First, it is faster because it uses a fast local moving algorithm that only reconsiders optimizing modularity for nodes that are in the neighbourhood of a previously moved node but are no longer in the same community. This is different from the Louvain algorithm, which continues to optimize modularity for each node until no further modularity gain is found. Second, the Leiden algorithm solves the issue that Louvain has where it creates badly connected (or even disconnected) communities. A badly connected community is one that would give a higher modularity if the community was split up in its well-connected parts. The Louvain algorithm only moves one node at a time so these badly connected communities can remain intact, leaving the community partition in a bad local optimum. These badly connected communities arise when a node is moved out of a community and the community left behind becomes badly connected due to that move during the MoveNodes function. The Leiden algorithm solves this by refining each community after one iteration of the MoveNodesFast function. This might split a community into multiple communities when they have become badly connected after a node move which increases the connectedness of the remaining communities. The Leiden algorithm guarantees well connected communities and does so with a higher speed and a higher quality of the result than the Louvain algorithm. This is why the work in this research is focused on further improving the Leiden algorithm.

Algorithm 2 shows parts of the Leiden algorithm as written in [TWvE19]. The outer loop at line 2 contains the MoveNodesFast function on line 3, the stopping condition on line 4, the community partition refinement on line 6 and aggregation of nodes based on this refinement on line 7. However, the community partition that was not refined is kept as the resulting partition which is done on line 8. On this line $C$ is a community present in the non-refined partition $P$ which was the result of the MoveNodesFast function. Each node $v$ that is in both the aggregated graph $G$ and in a community $C$ is included in the new version of partition $P$ that is used for the remainder of the algorithm. This approach maintains a high modularity while also maintaining well-connected communities. This is described in more detail in [TWvE19]. Our research focuses on the fast local moving algorithm. The fast local moving algorithm fills a queue $Q$ with nodes from the graph on line 14, which is done randomly. The fast local moving algorithm then loops over the queue which means that for the first $n$ (number of nodes) iterations the fast local moving algorithm from the Leiden algorithm is similar to the local moving algorithm from the Louvain algorithm. On line 16 a node is removed from the queue and selected for modularity optimization. If a node is selected, the algorithm assigns the node to the community that gives the highest modularity gain just like the Louvain algorithm on line 17, 18 and 19. However, it also adds all neighbouring nodes that are not in the new community to the queue on line 20 and 21. This is where the fast local moving algorithm is different from the Louvain algorithm. The loop continues over the queue

**Algorithm 2** parts of the Leiden algorithm.

```
1:  function Leiden(Graph $G$, Partition $P$)
2:      do
3:          $P \leftarrow$ MoveNodesFast$(G, P)$
4:          done $\leftarrow |P| = |V(G)|$
5:          if not done then
6:              $P_{refined} \leftarrow$ RefinePartition$(G, P)$
7:              $G \leftarrow$ AggregateGraph$(G, P_{refined})$
8:              $P \leftarrow \{\{v|v \subseteq C, v \in V(G)\}|C \in P\}$
9:          end if
10:     while not done
11:     return Flat*$(P)$
12: end function

13: function MoveNodesFast(Graph $G$, Partition $P$)
14:     $Q \leftarrow$ Queue$(V(G))$
15:     do
16:         $v \leftarrow Q$.remove$()$
17:         $C' \leftarrow argmax_{C \in P \cup \emptyset} \Delta\mathcal{H}_P(v \mapsto C)$
18:         if $\Delta\mathcal{H}_P(v \mapsto C') > 0$ then
19:             $v \mapsto C'$
20:             $N \leftarrow \{u|(u,v) \in E(G), u \notin C'\}$
21:             $Q$.add$(N - Q)$
22:         end if
23:     while $Q \neq \emptyset$
24:     return $P$
25: end function
```

until it is empty which is shown on line 23. This can save a lot of extra work since only the necessary nodes are considered after a node is moved, instead of the complete set of nodes.

Our focus is on parallelizing the Leiden community detection algorithm, in particular the fast local moving step. The first reason to do so is to limit the scope since this can be seen as an isolated step in the Leiden algorithm. The second reason is because this part of the Leiden algorithm seems to take up about half of the processing time after testing it with some preliminary benchmarks using the Live Journal dataset (see Chapter 4). For this research a JAVA implementation of the Leiden algorithm was adjusted to partially run in parallel.

## 3.2 Parallel Approaches

In this section we discuss different approaches for parallelizing the fast local moving step from the Leiden community detection algorithm. First we discuss the requirements needed for implementing any of these approaches.

### 3.2.1 Requirements

To parallelize the fast local moving step the so-called worker tasks and datamanagement tasks need to be separated. The worker tasks contain the modularity optimization operations and the datamanagement tasks are bookkeeping operations that maintain information about the community partition. The idea is that multiple threads exist that execute the worker tasks and optimize the community assignment for different nodes in parallel and that only one instance of a datamanagement class exists which does all the bookkeeping of which nodes belong to which community which separates the writing operations done in the datamanagement tasks from the reading operations done in the worker tasks. The possible approaches we present do not use any locking on the community partition data. This was because this data is shared over all threads and putting locking on it would likely result in an effectively sequential algorithm. Since this is a heuristic algorithm a loss in solution quality might be worth the speedup. Besides that, the number of threads typically found in a computer is much lower than the number of nodes in a network. If the network is sparse enough there should not be many conflicts in the data. This is also mention in [BHW$^+$17].

### 3.2.2 Control Mechanisms

To determine which thread works on optimizing modularity for which node, a control mechanism is needed that distributes the nodes in the queue over the threads. Section 3.2.2, Section 3.2.2 and Section 3.2.2 discuss three different approaches to implement such a control mechanism and we will compare their benefits and downsides in Section 3.2.2. These approaches were created during the development of our parallel implementation of the Leiden algorithm and we used preliminary benchmarking and software profiling to analyze their effectiveness.

#### Threadpool Executor Approach

The JAVA threadpool executor approach uses the existing threadpool executor class from the JAVA concurrent utilities library which gets a set of task objects as input and then executes them over multiple threads. Since a created task object cannot be altered a task object needs to be created for each node on the queue in order for a task to work on optimizing that node's contribution to the modularity value. This approach required a way to deal with constantly creating new tasks for each new node on the queue. Since each task object requires large arrays to store all the required network and community data this method can create a lot of overhead due to array initialization. An object called the 'array shop' can solve this issue by initializing enough arrays for all the threads at the start of the fast local moving algorithm and then giving arrays to each new task initialized by the threadpool executor. Once a task is done the array is free to be used by a new task. The array shop object keeps track on which arrays are in use and which are free. We did preliminary experiments with this approach and it does execute correctly. However, it is slightly slower than the sequential Leiden algorithm and gives worse modularity results.

#### Chopped Queue Approach

Interaction with the queue requires locking of the queue since threads should not be optimizing the modularity contribution of the same node. Considering the frequency of queue interaction this results in a large overhead. This approach therefore intends to minimize queue interaction by dividing (or chopping) the main queue, as seen in Algorithm 2 line 14, into a number of smaller queues equal to the set number of threads. Each worker thread processes only one of those smaller queues. This approach of delivering work in bulk reduces interaction between a worker thread and the main queue. Each thread then finds the community with the highest modularity for each node in their assigned queue. Once all threads are done, a new main queue is created based on which nodes were moved. Only nodes whose neighbour moved to a different community are added back to the queue. This queue is then chopped into pieces again and new threads are initialized to repeat the process again. This approach shows a very small speedup over the original algorithm but only for a specific dataset. However collecting a new queue, chopping it up and re-initializing the threads repeatedly for each iteration of the fast local moving algorithm has a big overhead.

#### Self Fetching Approach

In this approach is very similar to Algorithm 2 except that the loop starting on line 15 is executed by multiple threads. The queue interaction on line 16 and 21 are locking operations so threads will wait for each other to finish removing or adding nodes. This causes a large overhead because of threads idly waiting on the queue to be unlocked. This approach is slightly slower than the sequential Leiden algorithm but has the same modularity results.

#### Overview of Potential Approaches

Table 3.1 shows an overview of the intended benefits and the downsides of the aforementioned approaches.

## 3.3 Proposed Parallel Approach

The concept of using the queue in bulk from the chopped queue approach is combined with the simplicity of the self fetching approach in our proposed parallel approach. The original queue from the Leiden algorithm is

| Approach | Intended benefit | Downsides |
|---|---|---|
| Threadpool Executor | Out of the box thread management optimization | Large overhead in initializing many tasks and managing their resources |
| Chopped Queue | Reducing locking on the queue by delivering work in bulk | Large overhead in collecting new work and re-initializing threads |
| Self Fetching | Simple and straight forward approach without repeated re-initialization of threads | Overhead from very frequent queue interaction |

Table 3.1: Overview of attempted control mechanisms.

implemented as a linked integer list instead of as an array. This allows for different lists to be added together by only adjusting the pointers at the head and the tail and adjusting the meta-data such as list-length, list-start and list-end. This linked list approach makes it possible to quickly split a list without initializing new arrays or having to copy data to new arrays. Using this linked list implementation worker threads can fetch a certain amount of nodes from the queue to work on with only one locking operation. This saves many locking operations compared to fetching one-by-one. The same advantage applies to adding nodes to the queue. The community data manager collects all nodes that need to be added to the queue in a linked list and returns it to the worker thread. The worker thread collects all lists for each of the nodes it processes and then adds that entire list to the queue at once in one locking operation.

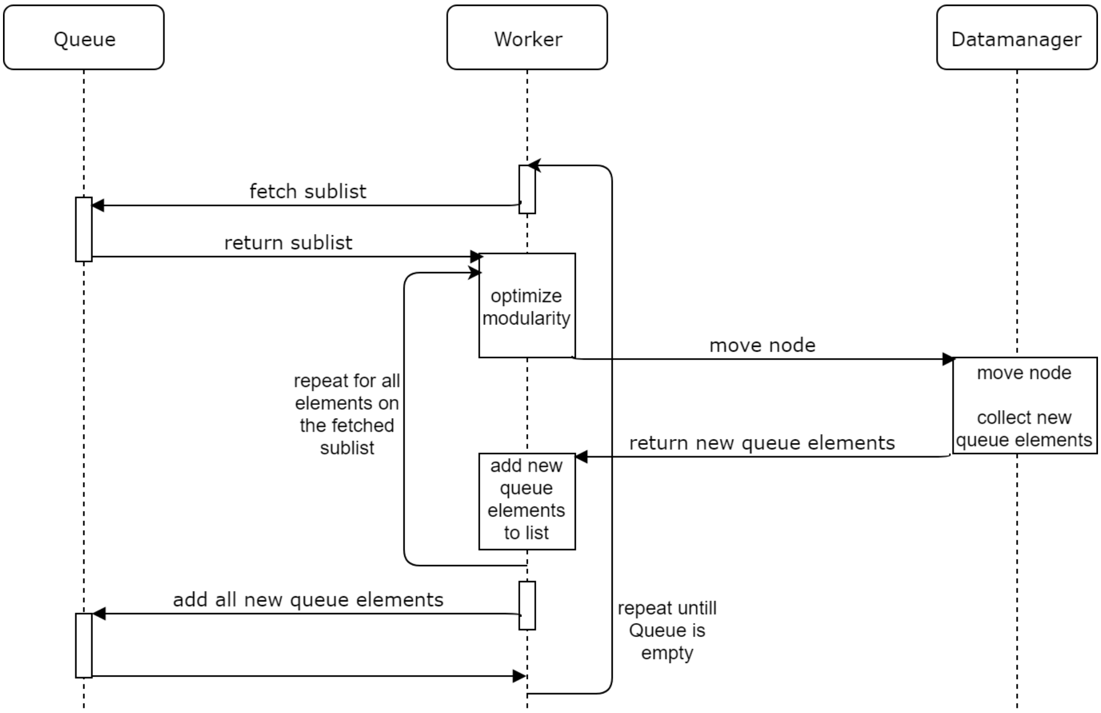
Queue
Worker
Datamanager
fetch sublist
return sublist
optimize modularity
move node
repeat for all elements on the fetched sublist
move node
collect new queue elements
return new queue elements
add new queue elements to list
add all new queue elements
repeat untill Queue is empty

Figure 3.1: Interaction between worker threads, the queue and the community data manager.

Figure 3.1 shows the interaction between the worker thread(s), the linked list queue and the community data manager. The worker thread(s) start by fetching a sublist of nodes that still need to be moved to a modularity optimal community. For each element on the sublist this is done in the inner loop in the middle of the image. During this inner loop the moving of the nodes and determining the new queue elements is through the

**Algorithm 3** Parallel Fast Local Moving Algorithm.

```
1:  function ParallelMoveNodesFast(Graph $G$, Partition $P$, Integer $numberOfThreads$)
2:      $Q \leftarrow \text{LinkedQueue}(V(G))$                  ▷ Create linked list of nodes as the main queue
3:      $threads = NodeMoverThread[numberOfThreads]$             ▷ Create threads array
4:      for $nodeMoverThread \in threads$ do
5:          $nodeMoverThread.\text{start}()$                     ▷ Start each thread
6:      end for
7:      WaitFor(threads)                                         ▷ Wait until all threads are done
8:      return $P$
9:  end function

10: function NodeMoverThread(Integer $numberOfThreads$)
11:     do
12:         $fetchSize = \frac{Q.length + numberOfThreads - 1}{numberOfThreads}$
13:         $Q_{sub} \leftarrow Q.\text{fetchSubQueue}(fetchSize)$   ▷ Locking call that fetches a sub-queue
14:         do
15:             $v \leftarrow Q_{sub}.\text{remove}()$           ▷ Remove item from sub-queue
16:             $C' \leftarrow argmax_{C \in P \cup \emptyset} \Delta\mathcal{H}_P(v \mapsto C)$   ▷ Find community with optimal modularity gain
17:             if $\Delta\mathcal{H}_P(v \mapsto C') > 0$ then
18:                 $v \mapsto C'$                           ▷ Move node with a non-locking call on the partition
19:                 $N \leftarrow \{u | (u, v) \in E(G), u \notin C'\}$
20:                 $Q_{new}.\text{add}(N)$                  ▷ Collect new queue elements
21:             end if
22:         while $Q_{sub} \neq \emptyset$                   ▷ Continue while queue part is not empty
23:         $Q.\text{add}(Q_{new} - Q)$                      ▷ Locking call that adds all the new queue elements
24:     while $Q \neq \emptyset$                             ▷ Continue while the main queue is not empty
25: end function
```

datamanager. These new queue elements are not yet directly added to the main queue. Once a worker thread is done with processing it's sublist all the new queue elements are added in bulk to the main queue. After this the worker thread requests a new sublist from the main queue. Only the interaction between the worker thread(s) and the queue is a locking interaction which means that a worker thread can work uninterruptedly for each whole sub-list it processes.

Algorithm 3 is the pseudo-code that replaces the fast local moving algorithm of the Leiden algorithm in our parallel implementation of the Leiden algorithm. On line 2 all the nodes in the network $G$ are added to the linked list queue $Q$ in random order. A set of threads is created on line 3 where $numberOfThreads$ is a value given to the algorithm by the user through a commandline parameter. This is done so that users can configure the thread count based on the hardware that they are using. All the threads are started on line 5 and the main thread then waits for all worker threads to be done on line 7. Each of the threads that were initialized by the main thread execute as described in the NodeMoverThread function. The $fetchSize$ is calculated on line 12 based on the $numberOfThreads$. This way the nodes are evenly distributed among threads but at the same time there is always a piece of queue left for the next fetch so threads are not starved for work. In the actual implementation in code a thread will fetch the entire queue once the $fetchSize$ is lower than 10 in order to prevent threads from doing a full loop over only a hand full of nodes. On line 13 a part of the main queue $Q$ is fetched and put in the thread's local queue called $Q_{sub}$ based on $fetchSize$.

On line 14 a loop starts that goes over all the elements in $Q_{sub}$ by removing them one by one on line 15 and continuing until $Q_{sub}$ is empty on line 22. The optimization of the modularity is the same as the original Leiden algorithm on the lines 16-19. The new queue elements are appended to a temporary local queue called $Q_{new}$ on line 20 which is later added to the main queue on line 23. The difference with the self fetching approach from Section 3.2.2 is that multiple threads access the same queue by fetching multiple elements at once and queue elements are added back to the queue by bulk as well. $Q_{sub}$, $Q_{new}$, $v$ and $N$ are local variables within each node mover thread. Graph $G$, main queue $Q$ and community partition $P$ are global variables and that data is shared among threads. $G$ consists of the edges $E$ and all the nodes $V$. $P$ contains all communities such as $C$ and $C'$. All parts of graph $G$ do not need locking since that data is not manipulated but only read. The parts of partition $P$ could use locking to preserve data correctness but we chose not to as discussed earlier. Only the interaction with the main queue $Q$ is done in a locking fashion to avoid errors in execution and node $\mapsto$ community assignment. This is because by locking the main queue $Q$ every node is present in only one $Q_{sub}$ of a thread at most. Therefore the node $\mapsto$ community assignment is done by only one thread at a time for a specific node ensuring singular assignments of nodes to communities. The node $\mapsto$ community assignment itself therefore does not need locking to ensure that. The only likely error that this causes is that the community assignment of a node is read incorrectly by a thread that is reading the community of a neighbouring node of the node that thread is currently working on. However, this is only an error in the heuristic calculation of the modularity gain and we chose to ignore that possible error and discover empirically what the impact on the quality becomes.

# Chapter 4

# Experiments

In this chapter we will cover what datasets were used for these experiments in Section 4.1. In Section 4.2 the hardware used for these experiments is described. Section 4.3 explains how the experiments were setup. The results of these experiments are presented in Section 4.4 and these results are evaluated in Section 4.5. Finally an extra experiment on the density of a network in relation to the speedup of parallelization is covered in Section 4.6.

## 4.1 Datasets

Six empirical real-world networks were used to analyse the performance of the parallelization of the Leiden algorithm. These are the same data-sets that were used in [TWvE19].

Table 4.1 shows the number over nodes, number of edges and the average degree of the nodes in the networks used in our experiments. The average degree can be calculated by $\frac{m}{2n}$ because these networks are all undirected networks.

| | Nodes ($n$) | Edges ($m$) | Average Degree |
|---|---:|---:|---:|
| DBLP | 317080 | 1049866 | 6.62 |
| Amazon | 334863 | 925872 | 5.53 |
| IMDB | 374511 | 15014839 | 80.18 |
| Live Journal | 3997962 | 34681189 | 17.35 |
| Web of Science | 9811130 | 104436131 | 21.29 |
| Web UK | 39252879 | 781439892 | 39.82 |

Table 4.1: Empirical real-world data-sets that were used in the experiments.

## 4.2 Hardware

The experiments were done on a machine containing two AMD EPYC 7601 32-Core processors totalling 64 cores and 128 threads with 1TB of RAM. The experiments themselves were not run simultaneously, so only one algorithm was running at any one time on this machine. Due to technical difficulties the Web UK experiments were run on a different machine which has two Intel Xeon E5-2697 v2 processors totalling 24 cores. For that reason the maximum number of threads configured was 32.

## 4.3 Experimental Setup

Eight versions of the algorithm were tested: the original Leiden algorithm and seven versions with the parallelized fast local moving algorithm. These seven version are identical except for the configured number of worker threads. Experiments were done using 1, 2, 4, 8, 16, 32 and 64 threads. Testing all six datasets with all eight algorithm results in 48 experiments. These were all repeated 10 times giving a total of 480 runs. The results were averaged back into 48 results. Each of the 480 runs consists of 10 iterations of the Leiden algorithm just as the experiments in [TWvE19]. Measurements were done separately for the run-time of the fast local moving part of the the algorithm and complete the algorithm to see how well the parallelization works by itself and what the consequences are for the Leiden algorithm as a whole. The first iteration of the complete algorithm was also measured separately to give insight in how effective parallelization is since that first iteration gives the largest modularity gain according to [TWvE19]. The final modularity is also recorded to see if there are any significant losses or gains in the quality of the solution. The code for the used implementation can be found on github [1].

## 4.4 Results

Figures 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6 summarize the results of the experiments for each dataset in a graph. Each graph shows the used algorithm on the horizontal axis and the total runtime on the verical axis. The runtime is split up into the runtime of the fast local moving part and the runtime of the algorithm that is not the fast local moving part. The top error bars show the standard deviation of the total runtime of the algorithm. The error bars in the middle show the standard deviation of the runtime of the fast local moving part of the algorithm.

[1] https://github.com/Geertex/networkanalysis/tree/LeidenParallelV1.0

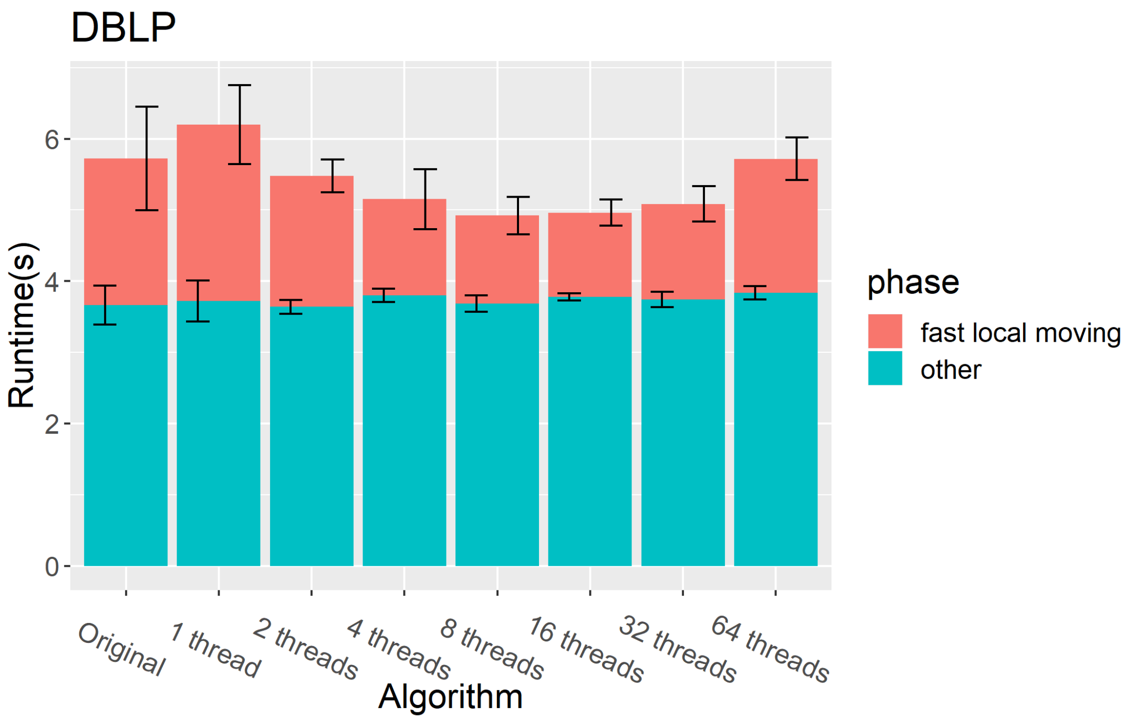

Figure 4.1: DBLP runtimes of 10 iterations with the fast local moving algorithm run-time highlighted.

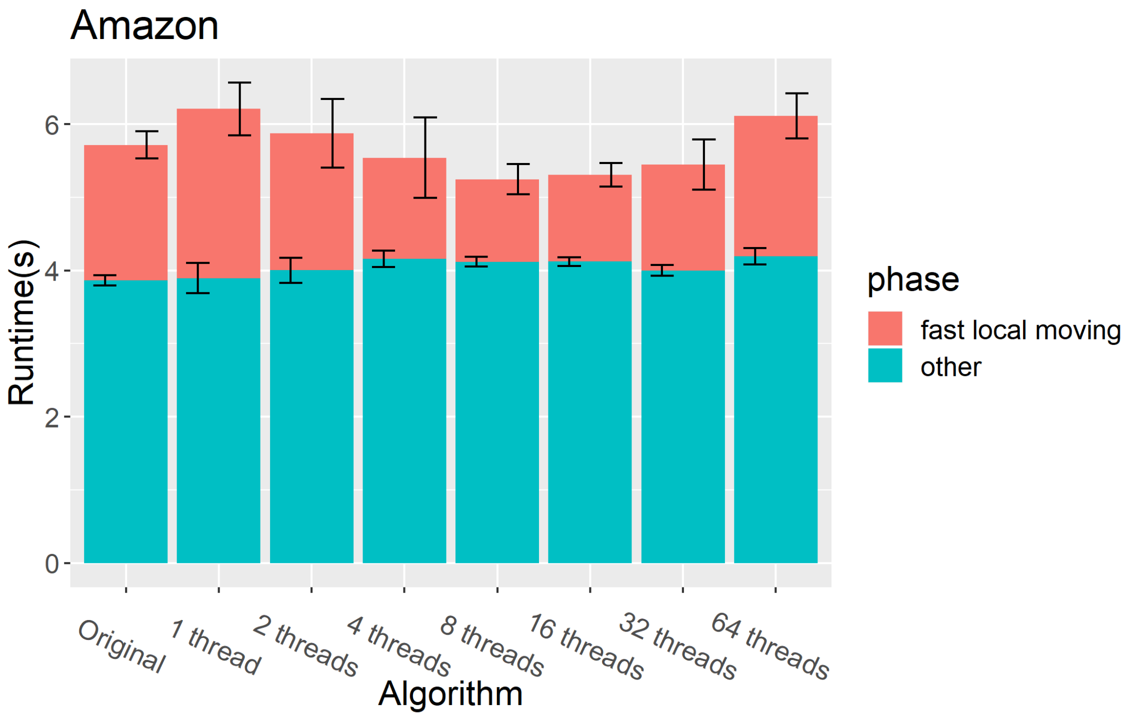

Figure 4.2: Amazon runtimes of 10 iterations with the fast local moving algorithm run-time highlighted.

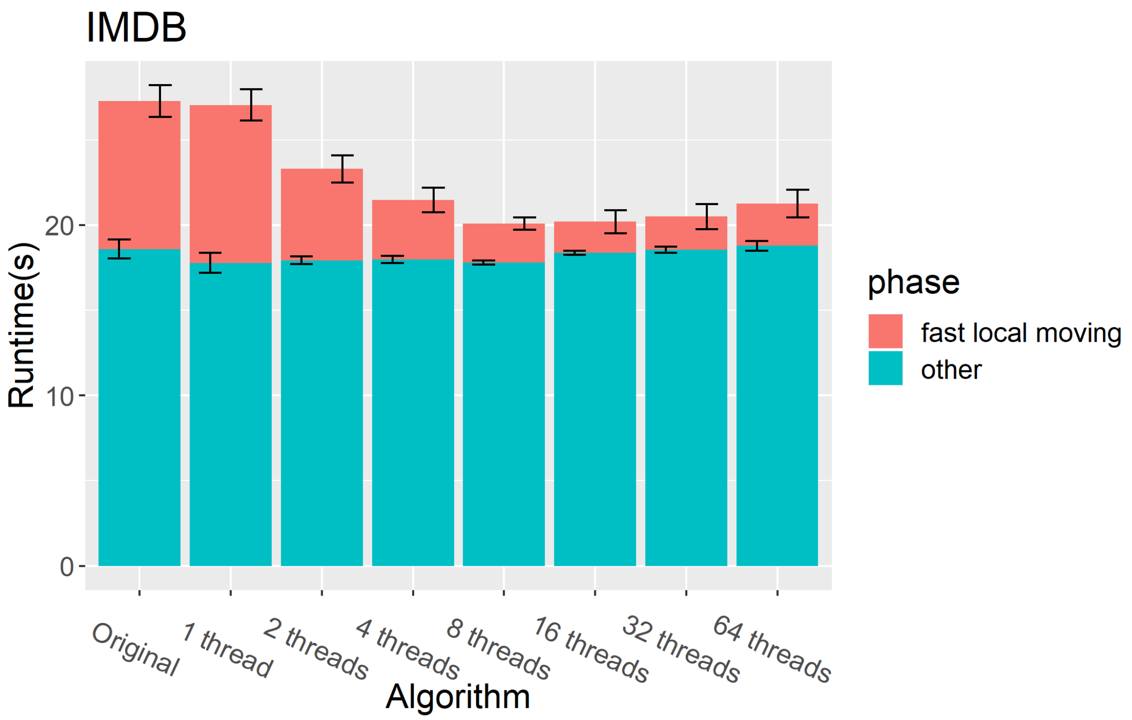

Figure 4.3: IMDB runtimes of 10 iterations with the fast local moving algorithm run-time highlighted.

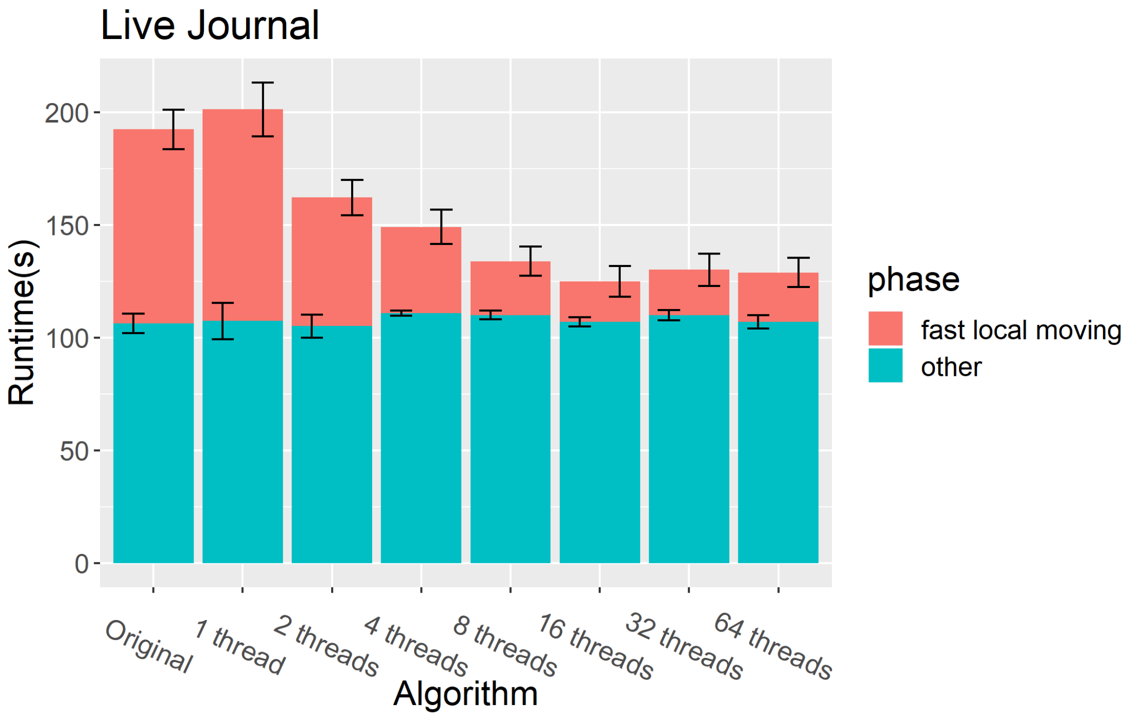

Figure 4.4: Live Journal runtimes of 10 iterations with the fast local moving algorithm run-time highlighted.

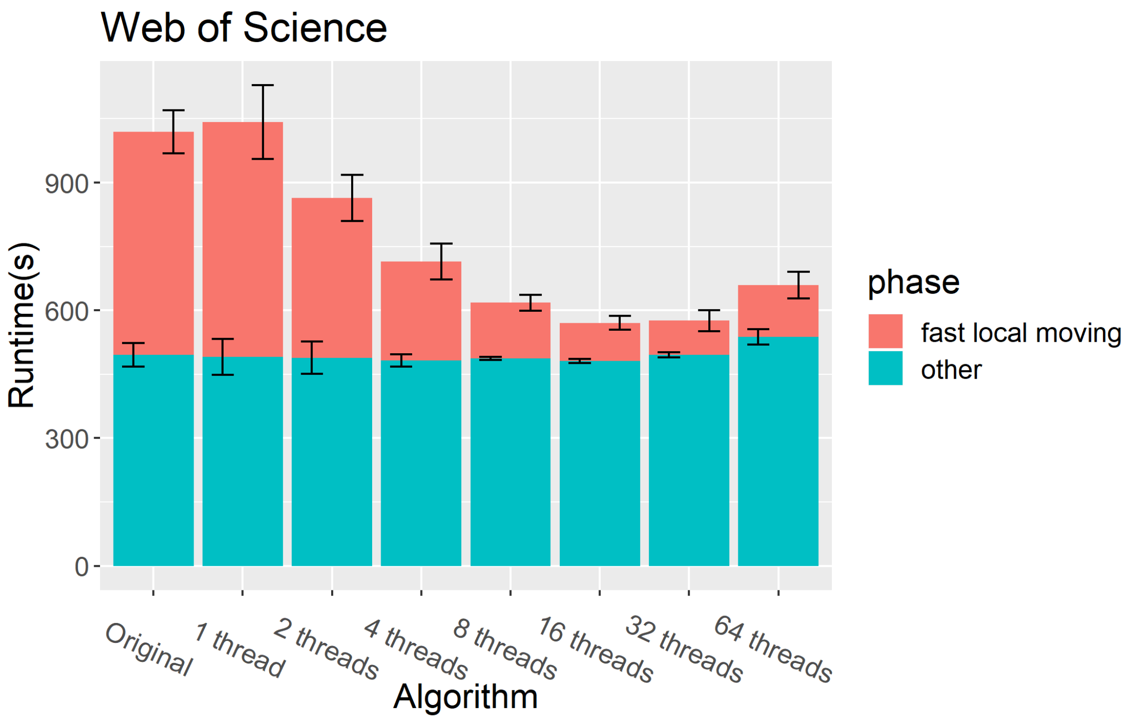

Figure 4.5: Web of Science runtimes of 10 iterations with the fast local moving algorithm run-time highlighted.

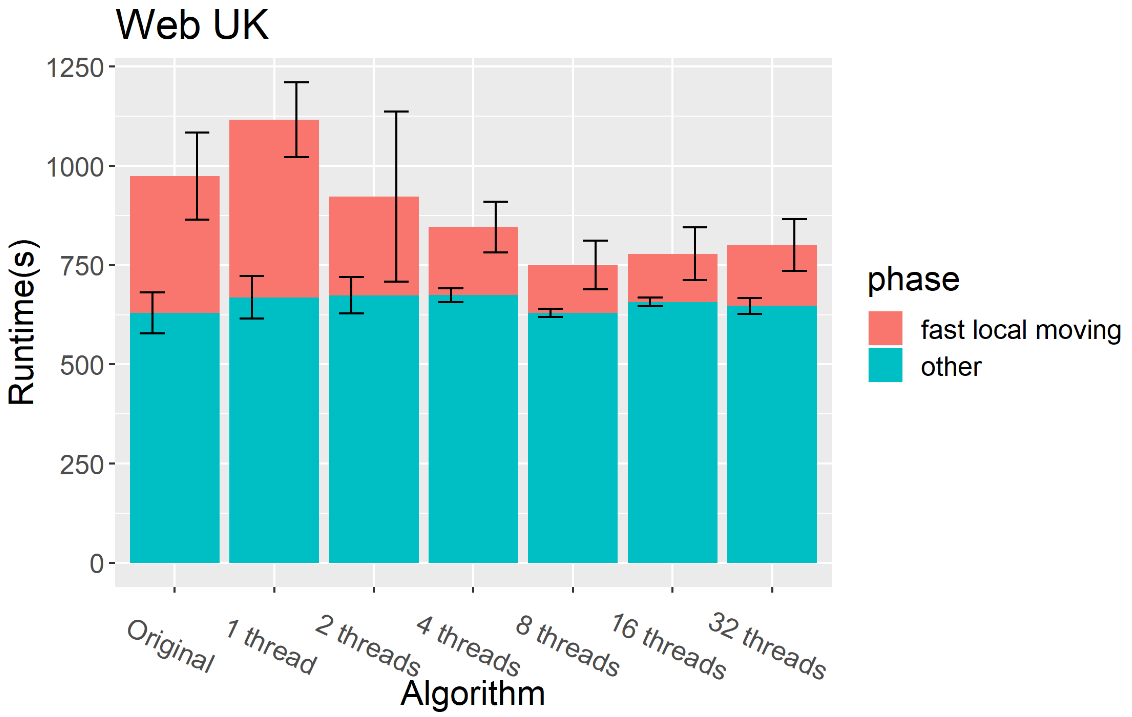

Figure 4.6: Web UK runtimes of 10 iterations with the fast local moving algorithm run-time highlighted.

In Table 4.2 an overview of the results is presented as the relative speed of each of the parallel versions compared to the original Leiden algorithm. This is shown separately for the fast local moving step, the complete algorithm and the first iteration of the complete algorithm. Table 4.2 also shows the increase in modularity value of the solution as a percentage gained over the original Leiden algorithm or percentage lost in the case of negative values.

| | no. of threads → | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|---|
| DBLP | local move | 0.83 | 1.12 | 1.52 | 1.66 | 1.74 | 1.53 | 1.09 |
| | full alg | 0.92 | 1.04 | 1.11 | 1.16 | 1.15 | 1.13 | 1.00 |
| | 1st iteration | 1.01 | 1.05 | 1.26 | 1.33 | 1.28 | 1.26 | 1.14 |
| | modularity diff | 0.05% | 0.07% | 0.07% | 0.04% | 0.11% | 0.10% | 0.08% |
| Amazon | local move | 0.80 | 0.99 | 1.34 | 1.64 | 1.57 | 1.28 | 0.97 |
| | full alg | 0.92 | 0.97 | 1.03 | 1.09 | 1.08 | 1.05 | 0.93 |
| | 1st iteration | 0.93 | 0.92 | 1.08 | 1.13 | 1.14 | 1.07 | 0.93 |
| | modularity diff | 0.00% | -0.02% | -0.01% | -0.01% | -0.01% | -0.01% | -0.01% |
| IMDB | local move | 0.94 | 1.62 | 2.50 | 3.79 | 4.74 | 4.45 | 3.51 |
| | full alg | 1.01 | 1.17 | 1.27 | 1.36 | 1.35 | 1.33 | 1.28 |
| | 1st iteration | 0.97 | 1.33 | 1.61 | 1.90 | 2.06 | 2.16 | 1.81 |
| | modularity diff | -0.07% | 0.00% | 0.04% | 0.00% | -0.06% | -0.11% | 0.01% |
| Live Journal | local move | 0.92 | 1.51 | 2.24 | 3.60 | 4.77 | 4.28 | 3.95 |
| | full alg | 0.96 | 1.19 | 1.29 | 1.44 | 1.54 | 1.48 | 1.49 |
| | 1st iteration | 1.01 | 1.36 | 1.62 | 1.75 | 2.23 | 2.27 | 2.48 |
| | modularity diff | 0.01% | -0.10% | 0.00% | -0.13% | -0.03% | -0.11% | -0.16% |
| Web of Science | local move | 0.95 | 1.40 | 2.25 | 3.99 | 5.84 | 6.50 | 4.28 |
| | full alg | 0.98 | 1.18 | 1.43 | 1.65 | 1.79 | 1.77 | 1.55 |
| | 1st iteration | 0.98 | 1.30 | 1.72 | 2.40 | 3.15 | 3.51 | 3.53 |
| | modularity diff | 0.17% | 0.12% | 0.04% | 0.02% | 0.01% | 0.12% | 0.02% |
| Web UK | local move | 0.77 | 1.38 | 2.01 | 2.85 | 2.83 | 2.25 | |
| | full alg | 0.87 | 1.06 | 1.15 | 1.30 | 1.25 | 1.22 | |
| | 1st iteration | 0.85 | 1.01 | 1.31 | 1.53 | 1.48 | 1.38 | |
| | modularity diff | 0.00% | 0.00% | 0.00% | 0.00% | 0.00% | 0.00% | |

Table 4.2: Average speedups for 10 iterations of the local moving algorithm, 10 iterations of the full algorithm, the first iteration of the full algorithm and the modularity increase compared to the original Leiden algorithm for the different number of threads used in the experiments.

## 4.5 Evaluation

### 4.5.1 Speed

Figures 4.1 and 4.2 show that there is some speedup in the fast local moving part when using parallelization for these two smallest datasets in our experiment, DBLP and Amazon. However, using more that 8 threads on these datasets does not seem to be very effective since there is hardly an increase in speed and even decreases in speed when using 16 threads or more. This is different compared to the results shown in Figures 4.3 and 4.4 where using 16 threads for the larger datasets, IMDB and Live Journal, is the fastest and using 32 and 64 threads is not significantly slower. Figure 4.5, with the results for the even larger Web of Science dataset, shows the largest speedup for the fast local moving part of the algorithm when using 32 threads. The results for the Web UK dataset, as shown in Figure 4.6, are less successful and show a smaller speedup than the Web of Science experiment and also larger error bars. Table 4.2 shows that the speedup of the algorithm as a whole is the smallest for the Amazon dataset with a maximum of 1.09 when using 8 threads. The DBLP speedup is higher with a maximum of 1.16 when using 8 threads. the IMDB and the Live Journal datasets show higher maximum speedups of 1.36 and 1.54 for 8 and 16 threads respectively. The highest speedup is achieved with the Web of Science dataset which is 1.79 when using 16 threads. The Web UK dataset saw a maximum speedup of 1.30 when using 8 threads which is lower than the speedup of the IDMB dataset which is about 50 times smaller in the number of edges. It would seem that parallelization is more effective on larger datasets but this trend does not hold for the largest Web UK dataset. Considering the technical difficulties in running the algorithm on that dataset and the large error margin this might be a misrepresentation of the effectiveness of parallelization.

When looking at the speedups of the first iteration of the algorithm in Table 4.2 it can be seen that the speedup of the first iteration is generally greater than the speedup of the full algorithm. This could be attributed to the fact the during the first iteration there is more work to be done. By this we mean that the community partition is still completely atomic with each community containing one node and therefore the algorithm still needs to move many nodes around to create a better community partition. This is in line with the hypothesis that the more work there needs to be done the more effective the parallelization is which we also see when we increase the size of the network.

Overall the 8 thread implementation was most often the fastest when considering the runtime of the complete algorithm. Table 4.3 summarizes these speedups and based on this the user can expect a 9% to 65% speedup compared to the original sequential Leiden algorithm when using the parallel Leiden algorithm with 8 threads.

| Data-set | speedup |
|---|---:|
| DBLP | 1.16 |
| Amazon | 1.09 |
| IMDB | 1.36 |
| Live Journal | 1.44 |
| Web of Science | 1.65 |
| Web UK | 1.30 |

Table 4.3: Speed-up when using 8 threads for each data-set.

### 4.5.2 Modularity

Table 4.2 also shows us that the maximum modularity loss is 0.16% and the maximum modularity gain is 0.17%. Moreover, all the average modularities fell within one standard deviations of each other. Therefore we think that there is no significant impact on the modularity of the solution when using this parallel implementation of the Leiden algorithm.

## 4.6 Density Experiment

To test if the effectiveness of the parallelization is dependent on the density of a network, meaning the number of edges relative to the number of nodes, an extra experiment was conducted. In this experiment the Live_Journal dataset was subsampled and the previous experiment was repeated on those subsampled datasets. Table 4.4 shows the properties of these new subsampled networks. The subsampling was done by deleting a percentage of the edges randomly. The remaining percentage of edges is shown in Table 4.4. After the edges were removed the largest connected component was selected and the network was saved to be repeatedly used in the experiments. Some nodes are lost in this process since they can become disconnected from the largest connected component when edges are removed. The hardware, experimental setup and measurements were identical to the previous experiment.

Figure 4.7 shows the speedup of the fast local moving step when using different numbers of threads compared to the original sequential algorithm for the different sub sampled Live_Journal networks. For the sample sizes of 100%, 90%, 80% and 70% there are some differences but they are not significant nor show a clear trend. The trend that would be expected is that a smaller sample size, which in turn means a smaller density,

| Sample percentage | Nodes | Edges | Average Degree |
|---|---:|---:|---:|
| 100 | 3997962 | 34681189 | 17.35 |
| 90 | 3904679 | 31203751 | 15.98 |
| 80 | 3798859 | 27719966 | 14.59 |
| 70 | 3676989 | 24240270 | 13.19 |
| 60 | 3533664 | 20761008 | 11.75 |
| 50 | 3364273 | 17284657 | 10.28 |

Table 4.4: Properties of the largest component of the reduced datasets.

would result in smaller speedup from the parallelization. This was discussed in Section 2.2.4 based on the hypothesis from [BHW$^+$17]. One clear trend that we can see is the greatly reduced effectiveness of using 32 and 64 threads for the 50% and 60% sub sampled networks.

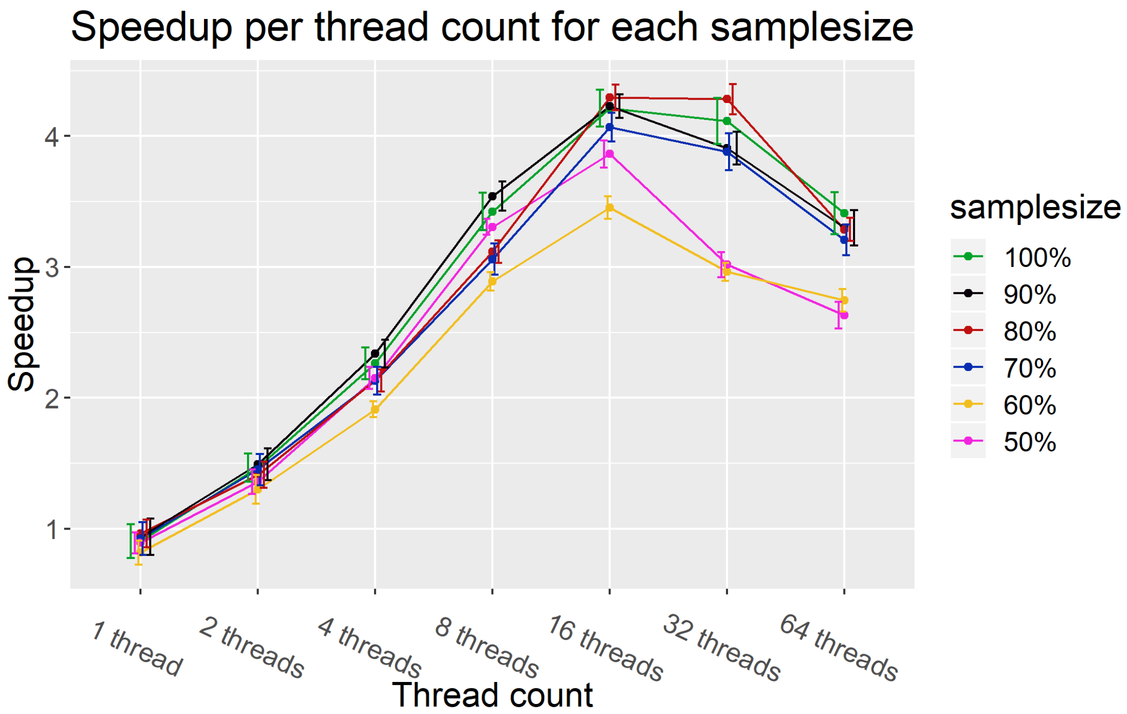

Figure 4.7: Live Journal speedups of the parallel fast local moving algorithm for different sub sampled version of the Live Journal dataset.

# Chapter 5

# Conclusion and Future Work

## 5.1 Conclusion

In this thesis we study the effectiveness of parallelizing the Leiden community detection algorithm. Our question is whether or not we can speedup the Leiden algorithm with a lock free parallel approach while having no loss in the quality of the solution as measured through modularity. We present an implementation of the Leiden algorithm with a parallel fast local moving step. We have shown that this adaptation makes the Leiden algorithm faster without significant loss in modularity. We found that our implementation does not scale well beyond 8 or 16 threads in most cases. A general trend we see is that the more work each thread needs to do the more effective the parallelization is. This is based on the observation that larger datasets benefit more from parallelization and that the early stages of the algorithm, when there is more work to be done, also benefit more from paralellization. The experiment on the Web UK dataset are the exception to this trend. Overall the 8 thread configuration was most often the best performing. Table 4.3 summarizes the speedup of the 8 thread configuration of the parallel fast local moving algorithm when measuring the runtime of the complete Leiden algorithm with 10 iterations. This tells us the actual speedup a user would experience. Based on these experiments a user can expect a 10%-65% speedup over the sequential Leiden algorithm without quality loss when using an 8-core machine. This means that our parallel implementation can benefit users who need faster community detection and have a multi core machine without reducing the quality of the community detection. Because we used a lock free approach our findings on the lack of impact on the quality of the solution is in line with the hypothesis in the work of Seung-Hee Bae where it is suggested that due to the sparseness of most networks parallelization should not effect the quality of heuristic community detection algorithms significantly [BHW+17].

## 5.2 Future Work

It is not uncommon for algorithms not to scale well beyond 4 or 8 threads. However in this research we have not analyzed our algorithm thoroughly enough to know why the parallelization does not scale well beyond 8 threads. This would be especially interesting for the very large datasets such as Web UK. For these large datasets performance improvement is also the most impact full due to their long runtimes. In the density experiment we have seen that the density of a network does impact the effectiveness of parallelization but we have not been able to show a clear trend through all our measurements.

The main performance gain with this research came when the queue was implemented as a linked list and the worker threads were made to interact with it in bulk. However reading from and writing to the queue was implemented at once. This means we do not know whether the queue reading or writing optimization had the largest impact. An experiment where bulk reading and bulk writing are implemented separately could give an answer to this.

An obvious future step is to parallelize the remaining refinement and aggregation steps of the Leiden algorithm as well to give future users more benefit of running the Leiden algorithm on multi core CPUs.

# Bibliography

[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, oct 2008.

[BHJ09] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. 2009.

[BHW$^+$17] Seung-Hee Bae, Daniel Halperin, Jevin D. West, Martin Rosvall, and Bill Howe. Scalable and efficient flow-based community detection for large-scale graph analysis. *ACM Transactions on Knowledge Discovery from Data*, 11(3):32:1–32:30, mar 2017.

[HVPPLP15] Stijn Heldens, Ana Lucia Varbanescu, Arnau Prat-Pérez, and Josep-Lluis Larriba-Pey. Towards community detection on heterogeneous platforms. In Sascha Hunold, Alexandru Costan, Domingo Giménez, Alexandru Iosup, Laura Ricci, María Engracia Gómez Requena, Vittorio Scarano, Ana Lucia Varbanescu, Stephen L. Scott, Stefan Lankes, Josef Weidendorfer, and Michael Alexander, editors, *Euro-Par 2015: Parallel Processing Workshops*, pages 209–220, Cham, 2015. Springer International Publishing.

[NMHT17] M. Naim, F. Manne, M. Halappanavar, and A. Tumeo. Community Detection on the GPU. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 625–634, may 2017.

[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*, CIKM '12, pages 1677–1681, New York, NY, USA, 2012. ACM.

[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*, WWW '14, pages 225–236, New York, NY, USA, 2014. ACM.

[RB08] Martin Rosvall and Carl T. Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4):1118–1123, 2008.

[TWvE19] Vincent A. Traag, Ludo Waltman, and Nees Jan van Eck. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific reports*, 9:5233, 2019.

[ZY18] Jianping Zeng and Hongfeng Yu. A distributed infomap algorithm for scalable and high-quality community detection. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 4:1–4:11, New York, NY, USA, 2018. ACM.