Karlsruhe Institute of Technology

Leiden-Based Parallel Community Detection

Bachelor Thesis of

Fabian Nguyen

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	PD Dr. Torsten Ueckerdt
	Prof. Dr. Peter Sanders
Advisors:	Dr. Michael Hamann
	Lars Gottesbüren, M.Sc.

Time Period: 1st of June 2021 – 30th of September 2021

Statement of Authorship

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, October 12, 2021

Abstract

Leiden is a community detection algorithm, that seeks to maximize modularity by dividing a graph into densely connected disjoint sets of nodes. It is an improvement of the widely known Louvain algorithm and can be split up into three main phases. The local moving and coarsening phases which are almost the same as in the Louvain algorithm, and the refinement phase which was introduced as an improvement. Considering that graphs can contain billions of edges nowadays using a sequential algorithm is unreasonable. We therefore consider an existing parallelization of the local moving phase and explain how it can be improved by incorporating the active-nodes queue introcuded in the Leiden algorithm, then analyze how the refinement phase can be efficiently implemented and parallelized and which performance gains are to be expected. The coarsening phase wont be covered here as a parallelized function for it is already provided by the used framework. The provided parallel implementation of the Leiden algorithm was written in C++. We show that our implementation achieves speedups up to 15.8 on the largest tested graph with 64 threads. More specifically, our parallel refinement achieves speedups up to 28 with 128 threads whereas local moving only scales up to about 32 threads with a maximum speed up of 12.5.

Deutsche Zusammenfassung

Leiden ist ein sogenannter Community Detection Algorithmus, der durch Optimierung der Modularity einen Graphen in möglichst dicht verbundene Teilmengen von Knoten zu unterteilen versucht. Er stellt eine Verbesserung des bekannten Louvain Algorithmus dar und kann in drei wesentliche Phasen aufgeteilt werden. Die Local Moving- und die Coarsening Phase, im wesentlichen identisch zu Louvain, und die neu eingeführte Refinement Phase die den Louvain Algorithmus weiter verbessert. Da Graphen heutzutage Milliarden an Kanten haben können scheint die Nutzung eines sequentiellen Algorithmus allerdings ungünstig. Wir werfen daher einen Blick auf eine bestehende Parallelisierung der Local Moving Phase und erklären dann wie diese durch kombination mit der active-nodes queue des Leiden Algorithmus verbessert werden kann. Dann analysieren wir, wie die Refinement Phase effizient implementiert und parallelisiert werden kann und welches Ausmaß an Leistungsgewinn gegenüber der sequentiellen Version zu erwarten ist. Die Coarsening Phase wird nicht weiter betrachtet, da dafür vom genutzten Framework bereits eine parallelisierte Funktion bereitgestellt wird. Die hier vorgestellte parallele Implementierung des Leiden Algorithmus wurde in C++ geschrieben. Auf dem größten getesteten Graphen ist unsere parallele Implementierung mit 64 threads bis zu 15.8 mal schneller. Dabei erreicht die Refinement Phase Speedups bis zu 28 mit 128 threads, während die Local Moving Phase lediglich bis ungefähr 32 Threads skaliert, mit einem maximalen Speedup von 12.5.

Contents

1. Introduction

As the usage and consumption of digital services increases so does the amount of associated data that needs to be processed. One important example of this is the analysis of large graphs that are built from real world data like customer shopping behavior, street networks, social media communities or even associations between different parts of the brain.

These graphs can quickly grow to enormous sizes, containing hundreds of millions of nodes and billions of edges and therefore pose significant challenges. How do we analyze specific properties of actual data in its graph representation and how do we do it efficiently? Nowadays efficiency on such a scale usually comes hand in hand with parallelism since performance increases in CPU cores have declined whereas the number of cores on a CPU still increases significantly. Therefore, the question "how do we do it efficiently" can often be rephrased to "how can we do it in parallel".

We take a look at community detection, which is used to to divide a network into densely connected subsets of nodes in order to reveal natural divisions and relationships in the network. A possible use case of this is to find well-connected social communities within large data sets of social media websites/applications like Facebook, i.e. to find communities in which two arbitrary people within the community are very likely to be friends of (or at least know) each other but are unlikely to be friends of anyone outside this community.

More specifically, we discuss the Leiden algorithm presented by Traag et. al. [TWvE19], explain how it can be parallelized and how much of a performance gain one can expect.

The Leiden algorithm is not a completely new algorithm per se but rather an improvement of another widely known algorithm, the Louvain algorithm [BGLL08]. The Louvain algorithm aims to divide a graph into communities (also called clusters) by maximizing modularity [NG04]. While this was the initial proposal by Blondel et al., the algorithm can also be adapted to use other quality measures like the map equation [RAB09].

As Traag et al. show in their paper the Louvain algorithm can find badly connected or even disconnected communities. The Leiden algorithm fixes this flaw by extending the Louvain algorithm with a so called refinement phase, which guarantees (among other things) that communities will never become disconnected. They also show a way to drastically increase the performance of the local moving phase. This phase will attempt to move *all* nodes to a different community to increase modularity and repeats if changes have been made. It can be improved by only re-considering relevant (so called active-) nodes instead of all nodes.

The main contributions of this thesis are two versions of parallelized refinement and two (similar) parallel versions of local moving, followed by an overall evaluation. Local moving was parallelized by combining an existing approach (from PLM) with the active-nodes queue from the Leiden algorithm. In contrast, we have no knowledge of other parallel approaches to the refinement phase. The source code can be found on Github [Ngu].

While a master thesis titled "Faster Community Detection Without Loss of Quality: Parallelizing the Leiden Algorithm" exists [Ver20], this thesis actually only considers parallelizations of the local moving phase and promises only negligible speedups (speedups significantly lower than 2 with 8 threads)

We show that our implementation, which we call Parallel Leiden, consistently achieves significant speedups up to at least 8 threads. The largest tested graph, the web graph uk2007, finishes 15.8 times faster with 64 threads, though using more than 32 threads (which already achieves a speedup of 14.7) is not recommended as speedups diminish at this point. These results may still be drastically improved by further optimizing local moving, which shows significantly worse scaling than refinement in our tests (maximum speedups of 12.5 and 28 respectively).

We start by introducing the basic concepts we will need in chapter 2 and then explain how the Louvain and Leiden algorithm work in chapter 3. Additionally, we briefly describe the main idea of a parallel version of the Louvain algorithm which we will extend for our own work and introduce the framework NetworKit that we use. In chapter 4 we will then explain our approaches to parallelizing both an improved version of the local moving phase and the refinement phase. Lastly, we evaluate our implementation in terms of speedups and impact of aspects like randomization and the amount of intra-community edges.

2. Preliminaries

2.1 Graphs

A graph $G = (V, E)$ consists of a set V of n nodes and a set $E \subseteq \{\{u, v\} \mid u, v \in V\}$ of m edges. Nodes are identified by their node ID (consecutive integers from 0 to $|V| - 1$), edges are identified by the two nodes they connect.

We call an edge e *incident* to a node v if v is one of the two nodes that e connects. Furthermore, we call an edge e a *bridge* of a node-subset (in our case communities) if the removal of e causes the subset to become disconnected, i.e there is at least one pair of nodes that is not connected through a series of edges within the subset.

Every edge can also be attributed a certain weight, in this case the weight function can be denoted as $\mathrm{w} : E \to \mathbb{R}^+$. Unweighted graphs can be interpreted as weighted graphs with weight 1 for all edges.

In the following we assume a graph is undirected (i.e. edges can be used in both directions) and weighted. Loops, i.e. edges from a node to itself, are also allowed.

The degree of a node is equal to the number of neighbors it has:

$$\deg(v) = |\{e \mid e \in E, v \in e\}|$$

The weighted degree instead sums the weight of all incident edges

$$\deg_{\mathrm{w}}(v) = \sum_{e \in E | v \in e} \mathrm{w}(e) + L_v$$

where L_v is the weight of the loop on v if one exists or 0 otherwise.

This means that loops are counted twice like they would be in directed graphs.

The weighted volume of a set of nodes S is the sum of the nodes' weighted degrees:

$$\mathrm{vol}_{\mathrm{w}}(S) = \sum_{v \in S} \deg_{\mathrm{w}}(v)$$

For node subsets $A, B \subset V, A \cap B = \emptyset$ the weighted cut is the summed weight of all edges between A and B:

$$\mathrm{cut_w}(A, B) = \sum_{\{u,v\}|\{u,v\}\in E, u\in A, v\in B} \mathrm{w}(\{u, v\})$$

The cut is symmetric: $\mathrm{cut_w}(A, B) = \mathrm{cut_w}(B, A)$.

We use $\mathrm{cut_w}(A)$ as a shorthand for the cut of a set to the rest of the Graph, i.e $\mathrm{cut_w}(A, V \setminus A)$. Likewise, we write $\mathrm{cut_w}(v, A)$ instead of $\mathrm{cut_w}(\{v\}, A)$

2.2 Partition & Community

Community detection is frequently used in network science to divide a network into disjoint sets of nodes. We call such a division a Partition $\mathcal{P} = \{C_1, ..., C_i\}$ that consists of communities (also called clusters) $C_i \subset V,\ \bigcup C_i = V,\ C_i \cap C_j = \emptyset$.

While communities can be overlapping (which is a realistic assumption for social media communities for example), the Leiden algorithm requires communities to be disjoint, so overlapping communities are not be considered here.

2.3 Modularity

Modularity is a measure of how strongly a graph has been divided into community structures, proposed by Newman and Girvan in 2004 [NG04]. Highly modular graphs are characterized by densely (innerly) connected communities with few connections between them [NG04].

Modularity can range between $-\frac{1}{2}$ (worst) and 1 (best) [BDG+08].

As mentioned by [Ham21] modularity can be written as:

$$\sum_{C\in\mathcal{P}} \frac{\mathrm{vol_w}(C) - \mathrm{cut_w}(C)}{\mathrm{vol_w}(V)} - \frac{\mathrm{vol_w}(C)^2}{\mathrm{vol_w}(V)^2}$$

Modularity can be changed by *moving* nodes between communities. A *move* is the act of removing a node from its current community and adding it to another community (so essentially assigning it a new community ID).

We denote the move of a node v to a community C by $v \to C$.

For the purposes of community detection algorithms we are more interested in a formula that describes the change in modularity for one particular move since we do not want to recalculate the modularity of the entire graph for every move.

Let C be v's current community, D the community we want to move v to and C^-/D^- C and D without v respectively. The modularity difference $\Delta_{v\to D}$ that results from one such move is then given by [Ham21]:

$$\Delta_{v\to D} = 2 * \left(\frac{\mathrm{cut_w}(v, D^-) - \mathrm{cut_w}(v, C^-)}{\mathrm{vol_w}(V)} - \mathrm{deg_w}(V) * \frac{\mathrm{vol_w}(D^-) - \mathrm{vol_w}(C^-)}{\mathrm{vol_w}(V)^2} \right)$$

In essence modularity compares the number of edges within a community to the number of edges one would expect in a random network with the same number of nodes and edges. The more edges within and the less edges between communities, the higher the graph's modularity. However, assumptions made in its underlying model introduce a non-trivial bias to the sizes of communities that can be found by any algorithm using modularity, as

the expected number of edges between two small communities can become smaller than one in large graphs, resulting in them being merged if even a single edge connects them [FB07].

This is known as *resolution limit*, causing algorithms that use modularity to find larger communities as the graph size increases. This can cause small communities to be merged in large graphs, whereas they would not have been merged in a small graph [FB07]. This problem in itself can not be avoided completely when using modularity but can be attenuated by introducing a resolution parameter γ [KSKK07]. It is unclear how this parameter should be chosen prior to analyzing the graph however.

It should also be noted that modularity tends to increase the larger a graph becomes [GdMC10]. This means that modularity values of different graphs can not be simply compared to determine which one has a higher "degree of community structure".

3. Related Work

In this chapter, we discuss relevant algorithms and the framework that was used to implement the parallel version of the Leiden algorithm.

Community detection is often used to reveal relations between nodes of the network in order to get a better understanding of the subject of interest. It can be used in virtually every field that can represent its data in a graph, from social media networks to road maps or even interactions within the brain. Considering the scope of these applications and how important it is to get accurate data for some of them, it is of utmost importance to have community detection algorithms that reliably find communities which truly represent a relation between nodes of these communities. Errors may implicate incorrect associations between unrelated nodes, causing wrong conclusions to be drawn and wasting the researchers' resources.

Over the years many approaches to solve this problem have been proposed, like spectral clustering and hierarchical clustering [For10], the most popular approach is to maximize a quality function. One of these algorithms is the Louvain algorithm proposed by Blondel et al. in 2008 [BGLL08]. The authors decided to use modularity as the quality function to be maximized, which is also the most popular quality function, but other ones can be used in its place as well, e.g. the Constant-Potts-Model (CPM) [TVDN11] or the map equation [RAB09]. In the following, we only consider modularity as the quality function.

Pseudocode for Louvain and Leiden can be found at the end of the respective sections.

3.1 Louvain Algorithm

The Louvain algorithm uses a relatively straight forward approach to maximizing modularity for a given graph, which is basically a multi-level greedy algorithm. It starts by assigning each node its own community. We call such nodes that are alone in their community singletons and a partition in which all nodes are singletons a singleton partition. As we identify each node by its node ID (an integer), we can assign such a community to every node by assigning it its own node ID as community ID. The algorithm then consists of repeating two phases, local moving and coarsening, until each community consists of one node only after a local moving (meaning no changes have been made). A visual example is given in figure 3.1 and pseudocode in algorithm 3.1.

Local Moving:

This phase tries to maximize modularity by greedily choosing moves. For each node $v \in V$ consider all communities v is connected to and move v to the community yielding the

largest modularity *increase*. Additionally, consider moving v to an empty community (i.e. making it a singleton).

If no such move is possible (i.e. all moves would result in a modularity decrease or an equal modularity), ignore the node and continue with the next one.

If at least one node has been moved (hence modularity has increased), consider all nodes once again until no more changes are made.

Coarsening:

Build a new graph in which for every community one node is inserted. For every edge in the initial graph, insert one edge with the same weight between the corresponding nodes of their communities into the coarse graph. Note that edges between nodes of the same community become loops this way, which is why we count loops twice as mentioned before.

In general, this results in multiple edges between some nodes. This can be avoided by adding an edge's weight to an already existing one instead of inserting a new one.

The coarsening phase itself does not change the modularity of the graph since the volume of each community and the sum of the weight of all edges between communities stay the same. However, it effectively grants the local moving phase the ability to move multiple nodes at once since every node in the coarse graph (possibly) represents a multitude of nodes in the initial graph.

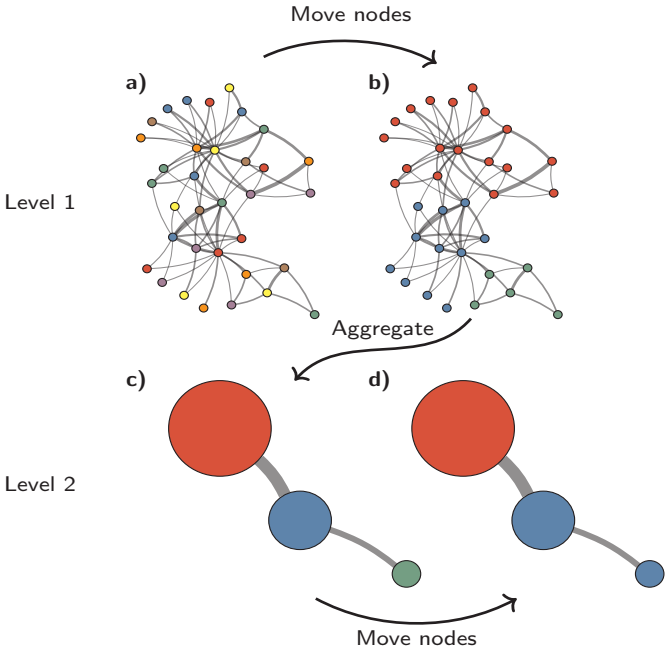

Figure 3.1: Schematic view of the Louvain algorithm taken from [TWvE19]. a) Starting from singleton partition. b) Local moving identified communities. c) Coarsened graph. d) Repeat b and c until no more changes occur.

Pseudocode of Louvain (for the most part as described in [TWvE19], mainly different notation). Note: flatten(P) basically reverses all coarsenings on P, by unfolding every node back to the (possibly multiple) nodes that were aggregated into it (while keeping the associated community). This requires us to save a mapping between coarse nodes and nodes in the initial graph, which has been omitted for the sake of clarity.

Algorithm 3.1: Louvain

Input : Graph G, Partition P

```
1  function Louvain(Graph G, Partition P)
2      do
3          P ← LocalMoving(G, P)
4          done ← |P| == |V(G)|      // Every community consists of one node
5          if not done then
6              G ← Coarsen(G, P)
7              P ← SingletonPartition(G)
8      while not done
9      return flatten(P)
10 function LovalMoving(Graph G, Partition P)
11     changed ← false
12     do
13         for v ∈ V(G) do
14             C_new ←argmax_{C∈P∪∅} Δ_{v→C}
15             if Δ_{v→C_new} > 0 then
16                 v → C_new                          // Move v to Community C_new
17                 changed ← true
18     while changed
19     return P
20 function Coarsen(Graph G, Partition P)
21     V ← P                          // One node in V for every Community in P
22     E ← {{C, D} | {u, v} ∈ E(G), u ∈ C ∈ P, v ∈ D ∈ P}
23     return Graph(V, E)
```

3.2 Parallel Louvain Method

As the name already suggests, the Parallel Louvain Method (PLM) is a parallel version of the Louvain algorithm. We use the same strategy to parallelize the local moving phase in our parallel implementation of the Leiden algorithm as PLM. It is a rather easy approach to parallelism in that it simply turns the main for-loop in the Louvain algorithm into a parallel for-loop. This requires us to make all writes to shared data atomic, though in our implementation there is only one shared data structure, which is a vector saving the volumes of all communities as recalculating them every time would drastically decrease performance. While this can still lead to threads reading outdated data (and subsequently calculating wrong modularity deltas) it will never cause longterm errors as no changes will be lost due to the atomic writes (and no write is based off an incorrect value, all writes are additions or subtractions of a node's degree, which does not change, to the *current* value in the vector). Short term errors can lead to incorrect moves, though it is rather unlikely that threads will cause each other to make a lot of incorrect moves in sufficiently large graphs. In practice, no significant decreases in modularity can be observed (compared to the sequential Louvain algorithm) as shown in [SM16]. The parallel execution of this phase also implicitly randomizes the order in which the nodes are considered, an explicit randomization is not done.

3.3 Leiden Algorithm

As shown by Traag, Waltman and van Eck in 2018, Louvain has a flaw that causes it to find badly connected communities or even disconnected communities [TWvE19]. They fix this flaw by introducing another phase to the algorithm, the so-called refinement phase, which greatly reduces the number of badly connected communities and guarantees that communities never become disconnected.

The refinement phase goes in between local moving and coarsening and aims to improve the partition found by the local moving phase. It may split a community into multiple subcommunities which can then be moved in the following local moving independently, providing more opportunities for beneficial moves. While the coarse graph in the Leiden algorithm is based on this refined partition, the initial partition of the coarse graph is still the one found by the local moving phase. This means that the coarse graph can now contain multiple nodes belonging to the same community instead of every community being represented by exactly one node like in the Louvain algorithm.

The authors also show a minor change to the local moving phase that significantly improves its performance. While the local moving phase of the Louvain algorithm re-considers *all* nodes in the next loop if changes have been made, the Leiden algorithm only considers some of them. Local moving starts by filling a queue with all nodes so that all nodes are considered at least once, like in the Louvain algorithm. When a node v is moved to a community C all neighbors of v that are *not* in C are added to the queue unless they are already in it. This gives neighbors of a node the chance to "follow" that node into the new community and prevents the algorithm from repeatedly considering nodes whose neighborhoods have not changed as they are unlikely to be moved (note that it is not *impossible*, even if a nodes' neighborhood has not changed its optimal community might have [BDG+08]).

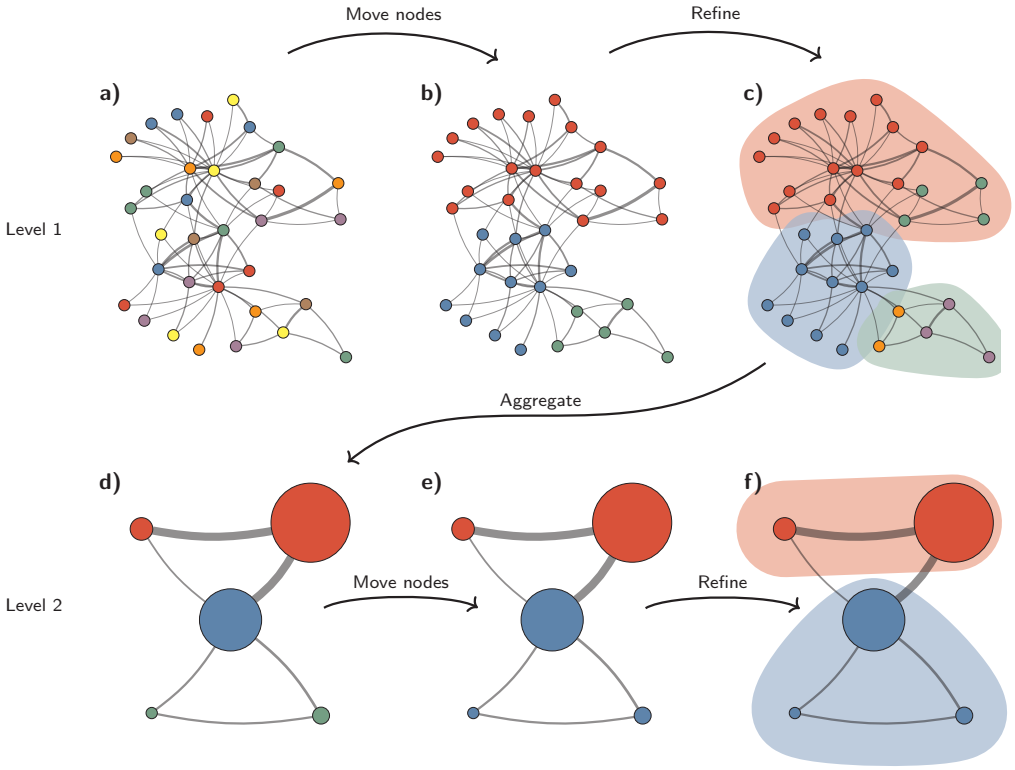

Figure 3.2: Schematic view of the Leiden algorithm, taken from [TWvE19]. The refinment seperates the red and green communities into two subcommunities respectively

In the refined partition every node starts as a singleton and is considered at most once. Nodes can only be moved if they are still a singleton, this guarantees that communities can not be disconnected since edges incident to singleton nodes can never be bridges of a community. Additionally, only nodes that are well connected within their community (the one in the not-refined partition) can be moved. If both of these requirements are met we calculate the modularity deltas for all well-connected neighboring communities in the refined partition.

Instead of taking a greedy approach and moving a node to the community yielding the highest modularity increase the authors instead recommend introducing randomness to the selection of the community with a parameter. However, in their published Leidenalg package [V.T] this approach is not implemented and it did not have any noticable effect on our implementation either so it was not included.

Additionally, while the Louvain algorithm is only run once, the Leiden algorithm can and should be used multiple times with the partition found in the last iteration as the base partition for the next iteration. The authors show that Leiden consistently improves its found partition with every iteration (until no more changes occur) while Louvain may worsen its result by producing more badly connected or even disconnected communities.

3.4 NetworKit

NetworKit describes itself as a "growing open-source toolkit for large-scale network analysis" [SSh16]. While NetworKit is a Python module, its algorithms and data structures are written in C++ for the sake of efficiency and parallelism and then exposed to Python. It provides many parallel implementations of network algorithms and tools that aid in contributing own algorithms. In our implementation, we used many of the classes provided by NetworKit, most notably Graph, Partition and ParallelPartitionCoarsening, as well as Timer and PLM (for testing).

Algorithm 3.2: Leiden

Input : Graph G, Partition P

function Leiden(Graph G, Partition P)
  do
    $P \leftarrow$ FastLocalMoving(G, P)
    $done \leftarrow |P| == |V(G)|$ `// Every community consists of one node`
    if not *done* **then**
      $P_{refined} \leftarrow$RefinePartition(G, P)
      $G \leftarrow$Coarsen$(G, P_{refined})$
      $P \leftarrow$SingletonPartition(G)
  while not *done*
  return flatten*(P)*

function FastLovalMoving(Graph G, Partition P)
  $Q \leftarrow$ Queue$(V(G))$ `// Insert nodes in random order`
  do
    $v \leftarrow Q$.pop()
    $C_{new} \leftarrow$argmax$_{C \in P \cup \emptyset}$ $\Delta_{v \to C}$
    if $\Delta_{v \to C_{new}} > 0$ **then**
      $v \to C_{new}$ `// Move` v `to Community` C_{new}
      $Neighbours \leftarrow \{u \mid \{v, u\} \in E(G), u \notin C_{new}\}$
      Q.push$(Neighbours \setminus Q)$
  while $Q \neq \emptyset$
  return P

function RefinePartition(Graph G, Partition P)
  $P_{refined} \leftarrow$ SingletonPartition(G)
  for $C \in P$ **do**
    $P_{refined} \leftarrow$ MergeNodesSubset$(G, P_{refined}, C)$
  return $P_{refined}$

function MergeNodesSubset*(Graph G, Partition P, Subset S)*
  $R \leftarrow \{v \mid v \in S, \text{cut}_\text{w}(v, S - v) \geq \deg_\text{w}(v)[\text{vol}_\text{w}(S) - \deg_\text{w}(v)]\}$
  for $v \in R$ **do**
    if v is singleton **then**
      $\mathcal{T} \leftarrow \{C \mid C \in P, C \subseteq S, \text{cut}_\text{w}(C, S - C) \geq \text{vol}_\text{w}(C)[\text{vol}_\text{w}(S) - \text{vol}_\text{w}(C)]\}$
      $C_{new} \leftarrow$argmax$_{C \in \mathcal{T}}$ $\Delta_{v \to C}$
      $v \to C_{new}$

4. Parallel Leiden Algorithm

We now discuss how the local moving phase and the refinement phase have been parallelized. While other parallel sections exist, like necessary precalculations and shuffling, these are rather trivial to implement and are therefore not discussed. Lastly, parallel coarsening was already implemented in NetworKit and was used with only a minor change as described in [PR]. Pseudocode can be found at the end of each section (4.1 and 4.2).

4.0.1 Terminology

As our implementation mainly uses C++ `vectors` we refer to them as such, though any form of dynamically resizable data structure with random access could be used instead. Specifically, we refer to the vector that saves cuts from nodes to communities as `cutWeights`, the vector that saves the volumes of all communities is called `communityVolumes`. These are used to avoid recalculating the respective terms in the modularity delta and refinement conditions from scratch every time, since we can continuously update them. Note that `communityVolumes` is a global vector that is shared among threads, while `cutWeights` is a thread-local vector that only exists within one phase at a time. Likewise, we use `activeNodes` for the active-nodes queue introduced in the Leiden algorithm to speed up local moving.

4.1 Parallel Local Moving

Our approach to a parallel Local Moving implementation is to combine the strategy of PLM with the active-nodes queue of the Leiden algorithm.

There are multiple challenges and questions that arise when trying to implement a parallel local moving like this:

- How do we make sure that no duplicates are inserted to the `activeNodes` queue?
- Similarly, how do we make sure that a node is only assigned to one thread?
- Which data updates need to be protected from race conditions?
- How do we achieve load-balancing?
- And lastly, how do we deal with increasing community ID's from local moving, since there is a shared vector?

No duplicates

The first issue is solved by keeping track of the nodes that were already inserted into the queue. We do this by using a `vector<atomic_bool>`, from now on referred to as `inQueue`. Whenever a thread finds a neighbor node and wants to add it to the queue, it will first check whether this node is already in the queue (lines 11 to 18 in 3.1). However, threads may simultaneously read a boolean to be `false`, then set the boolean to `true` and erroneously insert the node multiple times. We therefore use a compare-and-swap function [1] to differentiate between whether a thread set the boolean to `true`, meaning the node is not in the queue already, or whether it was already `true` to begin with, meaning the node should be ignored. If a thread is done with a node, it needs to set the respective boolean to `false` again (line 19).

Shared or private queue?

The second issue depends on the approach that is taken to implement the active-nodes queue. One approach is to have each thread maintain its own queue which it works off until it is empty. This version does not need any synchronization except for the `inQueue` vector to avoid duplicates. A major problem with this is that one thread may find a significantly larger number of nodes to work through, meaning that it will take a much longer time to finish while other threads are waiting (and doing nothing). Load balancing can not be easily done with this approach either since the queues are entirely separate and we would therefore need some kind of explicit work stealing. (An implementation of this approach without work stealing has been included as `Experimental`)

We therefore use a shared queue with locking to prevent race conditions. A lock-free implementation of a thread-safe queue was also tested briefly, namely Cameron Desrocher's `ConcurrentQueue` [Des], but was found to be slower than the implementation with explicit locking and an `std::queue`.

Thread-safe updates

In the case of a shared queue with locking, we can guarantee the integrity of the queue by only pushing or popping the queue while holding the respective lock. To reduce the overhead of locking and unlocking, each thread collects the nodes it wants to insert in a local vector first until a certain number of nodes has been found. If the threshold has been reached, we push the entire vector into the queue (which is an $\mathcal{O}(1)$ operation in C++). Per default this threshold is set to 1000 nodes.

Note that threads will not try to push their remaining nodes to the queue if they finished their current working set of nodes but will instead directly work through the remaining collected nodes. While this reduces the number of locking and push/pop operations needed it also alters the order in which the nodes are considered. Considering that this only affects a small number of nodes at a time (which will in turn typically only cause a small number of new nodes to be added) this should not have a significant impact, especially since we actually *want* the order to be somewhat random. By "somewhat random" we mean small-scale randomization (except for the initial random assignment) since we explicitly do *not* want nodes to starve in the queue. The parallelization in itself also introduces randomness since it can not be guaranteed that a set of nodes taken from the queue will be worked through faster than one taken after it, though in large graphs it is unlikely that nodes from separate sets would have influenced each other anyway.

[1] CAS functions are used to write values while simultaneously checking whether a usly assumed value is still present. This enables us to differentiate between a write that did not change data and one that did. Note that a simple previous read of the value does not solve this problem in a multithreaded environment as simultaneous reads and writes cause race conditions.

Load-balancing

For load-balancing we use a condition variable on which threads can wait if they have finished their nodes and the queue was empty when they checked. A thread that pushes a vector of nodes to the queue will call notify on this condition variable (which will wake all threads waiting on this condition variable), ensuring that threads only sleep when no work is available. Since a thread can only be assigned 1000 nodes at a time, this minimizes the time one thread may take longer than the others. The number of sleeping threads is monitored with a simple atomic integer so the last thread to finish can detect this and wake all threads to conclude the local moving (lines 20 to 25).

Increasing community IDs

Apart from the queue there is one more data structure that is shared by threads, the `communityVolumes` vector. While modifying these values can be done atomically it is unclear how large the vector is supposed to be at the start. This is because nodes may be moved to new singleton communities whose ID's increase and unlike in the sequential version the vector can not thoughtlessly be resized. Reusing ID's of empty communities would require us to keep track of all empty communities and would introduce many edge cases which may still force us to resize the vector after all. Consequently, the volume-vector's initial size and that of the thread-local cut-vectors are set to be 10000 larger than the highest community ID currently assigned. This is sufficient for most graphs as moves to singleton communities are rare and costs a negligible amount of memory (for reference, assuming a `double` is 8 bytes, 128 threads will in total require an extra 10000 * (8+1) * 128 = 11,520,000 bytes, or ~11.5MB.).

Nevertheless, to be able to guarantee correctness we need a way to resize the vector in a thread-safe manner. This is done by blocking all threads except the one that first detected a too large community ID (i.e. it is equal to or greater than the vector's current sizes). The first thread then increases the `communityVolumes` vector's capacity by another 10000 as soon as all threads have been blocked, while each thread individually increases the capacity of its `cutWeights` vector. Considering that this occurs *extremely* rarely and only on very *very* large graphs (in fact, this has not happened even a single time during our tests), this overhead is negligible and should never occur for typical use-cases.

Randomization

Additionally, we explicitly randomize the node order in the beginning. To avoid the overhead of sequentially shuffling all nodes, this is done on a per-thread basis. More specifically, we equally split the nodes into n parts when there are n threads and randomly assign each thread one of the parts. Each thread then shuffles its own nodes.

While this method covers significantly less permutations, it introduces a reasonable degree of randomness combined with the inherent randomness stemming from the parallel execution for minimal overhead.

Avoid unnecessary calculations

Lastly, we also make a small change to how the modularity difference 2.3 is calculated, because we are only interested in the maximum delta, regardless of its actual value. The volume of the entire graph $\mathrm{vol_w}(V)$ does not change since the actual graph is not changed, only the association of nodes to communities is. Hence, we multiply by $\mathrm{vol_w}(V)$ once to get rid of the denominator in the first term. Additionally, we can drop the factor 2, leaving us with:

$$\mathrm{cut_w}(v, D^-) - \mathrm{cut_w}(v, C^-) - \mathrm{deg_w}(v) * \frac{\mathrm{vol_w}(D^-) - \mathrm{vol_w}(C^-)}{\mathrm{vol_w}(V)}$$

Then, we remove both terms containing C^-. These terms are not dependent on the choice of D, we therefore avoid adding these terms multiple times by using them as a threshold instead. Since we are only interested in modularity differences greater than 0 (yielding a modularity *increase*), we are therefore looking for:

$$\arg\max_{D \in \mathcal{P}} \underbrace{\mathrm{cut}_\mathrm{w}(v, D^-) - \deg_\mathrm{w}(w) * \frac{\mathrm{vol}_\mathrm{w}(D^-)}{\mathrm{vol}_\mathrm{w}(V)}}_{modularitydelta\ \delta_{v \to D}} \overset{!}{>} \underbrace{\mathrm{cut}_\mathrm{w}(v, C^-) - \deg_\mathrm{w}(v) * \frac{\mathrm{vol}_\mathrm{w}(C^-)}{\mathrm{vol}_\mathrm{w}(V)}}_{modularitythreshold\ \delta_{Th}}$$

In the following, we call the left hand side of this inequality the *modularity delta* $\delta_{v \to D}$ and the right hand side the *modularity threshold* δ_{Th}.

The general idea of doing this, and a simple interpretation of these terms was mentioned by Blondel et al. [BGLL08] (with different notation): The modularity threshold is the change in modularity that results from removing the node v from its community and the modularity delta is the change in modularity that results from then moving the node v into the new community. Hence, we will only move v when the benefit is larger than the loss.

Algorithm 4.1: Parallel Local Moving

Input : Graph G, Partition P

```
1  Q ← Queue of vector of nodes
2  inQueue ← empty vector
3  – Open parallel region –                                  // OpenMP SPMD style
4  currentNodes ← V(G)/#threads                  // Split nodes among threads
5  while currentNodes ≠ ∅ do
6  |  newNodes ← empty vector
7  |  for v ∈ currentNodes do
8  |  |  C_new ←argmax_{C∈P∪∅} δ_{v→C}
9  |  |  if δ_{v→C_new} ≤ δ_Th then
10 |  |  |  Continue
11 |  |  P[v] ← C_new                              // Move v to Community C_new
12 |  |  for nb ∈ Neighbours(v) do
13 |  |  |  if not inQueue[nb] and P[nb] ≠ C_new then
14 |  |  |  |  inQueue[nb] ← true
15 |  |  |  |  newNodes.push(nb)
16 |  |  |  |  if newNodes.size() = 1000 then
17 |  |  |  |  |  Q.safePush(newNodes)          // lock Q, push, unlock Q
18 |  |  |  |  |  wakeAllThreads()        // wake sleeping/waiting threads
19 |  |  |  |  |  newNodes ← empty vector
20 |  |  inQueue[v] ← false
21 |  if newNodes ≠ ∅ then
22 |  |  currentNodes = newNodes
23 |  |  Continue
24 |  currentNodes ← empty vector
25 |  while not Q.trySafePop(currentNodes) and not all threads finished do
26 |  |  sleep()
27 – Close parallel region –
28 return P
```

4.2 Parallel Refinement

Before diving into the specifics of how we approach the parallelization of the refinement phase it is important to understand which guarantees and properties we need to uphold.

Remember that the refinement phase was explicitly introduced to guarantee the absence of disconnected communities and to increase connectivity within communities overall. This was achieved by restricting the considered nodes to those that are still singletons since these can never act as bridges in a community, so this is the main guarantee to watch out for.

Additionally, there are two more conditions that need to be met (imposed by the $\mathcal{T}$ and R sets as shown in algorithm 3.2), only nodes that are well connected within their initial community and the one created by the refinement phase will be moved. Note that we do *not* explicitly construct these sets and instead verify that the conditions are met for each node individually, we use this representation for ease of notation and clarity only. Both of these conditions contain cuts and volumes of which all except one may change during the refinement ($\text{cut}_\text{w}(v, S - v)$ can not change since S is static, just as v is).

4.2.1 Cluster-wise parallelization

Looking at the way the pseudocode provided in algorithm 3.2 is written, there is an "easy" way to parallelize the refinement phase that should spring to mind. Notice that no conditions or calculations ever depend on nodes outside of the community of the node we are currently trying to move. This means that all communities can be worked in parallel without the need for any synchronization at all, which means we should be able to simply replace the for loop with a parallel-for loop. However, there are several problems with this.

Firstly, the for loop assumes we have saved the affiliation of nodes to communities to begin with. And while this is true, after all we know the community of each node, we only know this for individual nodes, i.e. we have a node $\rightarrow$ community mapping. The loop assumes a community $\rightarrow$ node mapping however. We solve this by sorting all nodes by their community IDs and then saving the intervals in which nodes of the same community lie. We then sort the intervals by their size in descending order to minimize load imbalances instead of randomizing. A shared counter is used to indicate which interval should be used next. (Of course we could also save the mapping in the reverse way to begin with as well, but that would introduce a massive overhead to the local moving as nodes would have to be inserted and removed numerous times.)

The efficiency of this approach depends on the sizes of the individual communities, the larger the smallest communities are the higher load imbalances may become. In the worst case, all threads finish their current communities at the same time with one community left over. This results in all threads except one being inactive for as long as the last community takes to be worked through. If intervals are not sorted by size, this may end up being the largest community.

Additionally, if the thread count is higher than the number of communities this results in sleeping threads right from the beginning, though this should not occur for typical graphs that are large enough to warrant the use of a parallel algorithm.

We therefore give up the restriction that only one thread can work on a community at a time and turn to a "truly" parallel version (though once again, this version has also been included as `EXPERIMENTAL`).

4.2.2 Full parallelization

Unlike in the local moving phase, we now need 4 shared data structures. One of them is a simple `vector<bool>`[2], used to mark whether a node is still a singleton. Two are `vector<double>` and save the volumes of the communities in the (now to be built) refined partition, and the cut of a refined partition C to the initial subset S (the respective term is $E(C, S - C)$ in the $\mathcal{T}$ set as described in algorithm 3.2). Lastly, we need a vector to hold locks, one for each community.

Initializations

While the singleton and lock vectors can simply be initialized (singleton to `true`) for all nodes, the two `double` vectors have to be precalculated. These can be filled during a single parallel loop over all nodes and do not require any synchronization or atomic updates as all nodes are singletons in the beginning, which means that every community will only have their value updated once.

Finding the best community

To determine the optimal community for a node v, we need to calculate several things. The degree of v, all communities C_i to which v is connected (or worded differently, all its neighbor communities) and the cut from v to its initial community S. We can calculate all of these things in a single loop over all of v's neighbors. Once again, this can be done without synchronization as none of this data is shared. Additionally, we save the ID of all neighbors of v that are still singletons. We will need these later to ensure that the community has not changed in a way that causes us to write wrong data to the shared vectors.

Restrictions

There are three conditions that need to be met in order to move a node v:

- v needs to be well-connected to its initial community S
- The target (refined) community C needs to be well-connected.
- v has to be a singleton at the time of the move.

The main issue here is that it is possible for all of them to be met while we determine the best community, but not anymore when we want to move v. We solve this by using locks for every community. Before making a move, we lock the community of v and the target community C we want to move v to, locking the lower community ID first to prevent deadlocks (line 13). When both locks are held, we can verify that v is still a singleton by reading from the singleton vector (line 14). The fact that v is well-connected to S can not change during refinement as both v and S are static, but it may be that the the target community C is not well-connected anymore. There are two reasons as to why this could happen.

Firstly, the target community might have been a singleton community and the node in it was moved. We can verify this has not happened by making sure the node with the same ID as the community is still in that community (or equivalently, that the target community is not empty, as shown in line 15). Since we are holding the lock for this community, it can not change anymore. If the community is in fact now empty, we ignore it and choose

[2]Note: We actually use a `vector<uint8_t>` instead because `vector<bool>` is a specialized container in C++ that does not guarantee thread-safe modification of different elements. This prevents us from atomically updating a single boolean efficiently.

a new one (lines 16 to 23). Note that we may need to unlock v in case the new chosen community (if one even exists, otherwise we ignore the node) has a lower ID than v to prevent deadlocks. This means that we need to confirm that v is still a singleton after reacquiring the locks.

Secondly, communities might have had nodes moved into them in the meantime, which also means that a different community may now be preferable over the one we have determined to be the best community before, even if it is still well-connected. Since locking all possible target communities is infeasible we accept this risk. In our tests, this had no noticeable impact on the quality of the partition (in terms of modularity). We assume that erroneous moves happen rarely and generally still choose a community among the best ones (remember that the Leiden paper even *recommended* not choosing the best one deterministically). Additionally, they might be corrected in subsequent local moving and refinement phases or the next iteration of the Leiden algorithm.

Nevertheless, we still need to guarantee the correctness of updates to shared data. This is why we remembered singleton neighbors of v earlier. The only shared data that is updated upon a move is the cut of the refined community to its initial subset S and the volume of the community (combined into line 25). The volume can be updated regardless of what happened to the community in the mean time, since we only need to atomically add the degree of v to it. The cuts from communities to their initial subsets are influenced by moves however, as this example demonstrates:

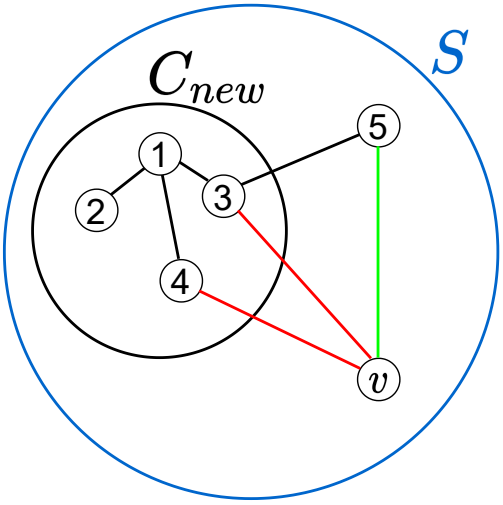

Figure 4.1: Updating cuts from communities to their initial subset

When we move a node v to a new community C_{new}, we need to subtract all edges from v to nodes of C_{new} (edges in red) and add edges from v to nodes outside of C_{new} but still in S (edge in green). The red edges make up the cut from v to C_{new} which we have already calculated, and the green edge is the cut from v to the community of node 5 which was also calculated already. In case node 5 is moved into C_{new} after these calculations however, the cut from v to C_{new} will no longer be correct. We therefore need to update the cut from v to C_{new} (line 24) before moving v and subsequently updating the cut from C_{new} to $S - C_{new}$ (line 25). Additionally, we update the cut to all changed neighbor communities before selecting a new community (see lines 17 to 19). While this does not guarantee correct values (as neighborhoods may change during this update) it increases the chance that the best community is chosen.

Randomization

As in the local moving phase we shuffle nodes per-part in the beginning. Unlike in the local moving phase however we do not split the entire vector of nodes. The fact that this may introduce load imbalances was offset by the existence of the active-nodes queue which provided work for threads that finished their initial share of nodes early. As no nodes are re-inserted in the refinement phase each thread is continuously given a certain amount of nodes (also 1000 per default) until no more remain.

Algorithm 4.2: Parallel Refinement

```
   Input : Graph G, Partition P
 1 P_refined ← SingletonPartition(G)
 2 // Initializations                                          // in parallel
 3 parallel for v ∈ V(G)
 4 |  S ← P[v]
 5 |  if v not singleton or cut_w(v, S − v) < deg_w(v)[vol_w(S) − deg_w(v)] then
 6 |  |  continue
 7 |  criticalNodes ← empty vector
 8 |  for nb ∈ Neighbors(v) do
 9 |  |  if nb ≠ v and P[nb] = S and nb is singleton then
10 |  |  |  criticalNodes.add(nb)
11 |  T ← {C ∈ P_refined | C ⊆ S, cut_w(C, S − C) ≥ vol_w(C)[vol_w(S) − vol_w(C)]}
12 |  C_new ←argmax_{C∈T} δ_{v→C}
13 |  if δ_{v→C_new} ≤ δ_Th then
14 |  |  Continue
15 |  lockLowerFirst(v, C_new)
16 |  if v is singleton then
17 |  |  while C_new = ∅ do
18 |  |  |  unlock(v, C_new)
19 |  |  |  updateCut(criticalNodes)              // due to moved neighbors
20 |  |  |  update T                  // cut values may be outdated by now
21 |  |  |  C_new ←argmax_{C∈T} δ_{v→C}
22 |  |  |  if v is singleton and δ_{v→C_new} > δ_Th then
23 |  |  |  |  lockLowerFirst(v, C_new)
24 |  |  |  else
25 |  |  |  |  Break
26 |  |  updateCut(criticalNodes)                          // C_new is locked
27 |  |  v → C_new              // cut values are guaranteed to be correct
28 |  unlock(v, C_new)
29 return P_refined
```

5. Experimental Evaluation

The following graphs from the DIMACS [DIM] and SNAP [SNA] websites have been used:

Graph	Nodes	Edges	Avg. Degree	Source
as365	7,013,978	11,368,076	1,62	[CLA12]
uk2002	18,520,486	261,787,258	14,14	[BRSV11]
uk2007	105,896,555	3,301,876,564	31,18	[BCSV04]
PGP	10,680	24,316	2,28	[BnPSDGA04]
italy	6,686,493	7,013,978	1,05	[GEO]
lux	114,599	119,666	1,04	[GEO]
europe	50,912,018	54,054,660	1,06	[GEO]
belgium	1,441,295	1,549,970	1,08	[GEO]
amazon	334,863	925,872	2,76	[YL12]
livejournal	3,997,962	34,681,189	8,67	[YL12]
orkut	3,072,441	6,288,363	2,05	[YL12]
google	875,713	5,105,039	5,83	[LLDM08]
LFR	varying	varying	-	Generated

The tested algorithms/implementations are as follows:

- The leidenalg package by V.Traag [V.T] (version 0.8.7). It should be noted that this implementation focuses on flexibility over performance. It is written in C++ but only accessible through python and depends on Igraph.
- The Leiden and Louvain algorithms implemented in Igraph [CN06] (not to be confused with jgraph which used to also go by the name Igraph). While these are implemented in C we use the provided python interface (version 0.9.6).
- Our parallel and sequential implementations of the Leiden algorithm in C++ [Ngu] which depend on NetworKit [SSh16] [1] The parallel implementation is denoted by PL (Parallel Leiden), the sequential version by PL SQ.
- The Parallel Louvain Method (PLM) as implemented in NetworKit.

[1] Note: The results presented here will not be reproducible unless NetworKit is built from source as we made a change to the coarsening function which is not included in any release of NetworKit as of the writing of this thesis. We refer to the pull request on Github for more information [PR]. Additionally, the environment variable `OMP_PROC_BIND` should be set to `true` if a multi-cpu system is used.

Unless otherwise noted, all Leiden-tests were performed with a maximum iteration count of three (Louvain is only run once, as described in 3.3), repeated five times and then averaged to reduce deviations caused by inherent randomness. All measured values were rounded to have at most four decimal digits. All tests were performed on a dual-cpu system with 2x64 cores and 1024GB (16x64GB) DDR4-3200MHz(ECC) memory.

5.1 Sequential Implementations

We start by comparing all sequential algorithms. The parallel version run with only one thread has been included as well, to rule out significant inherent performance differences.

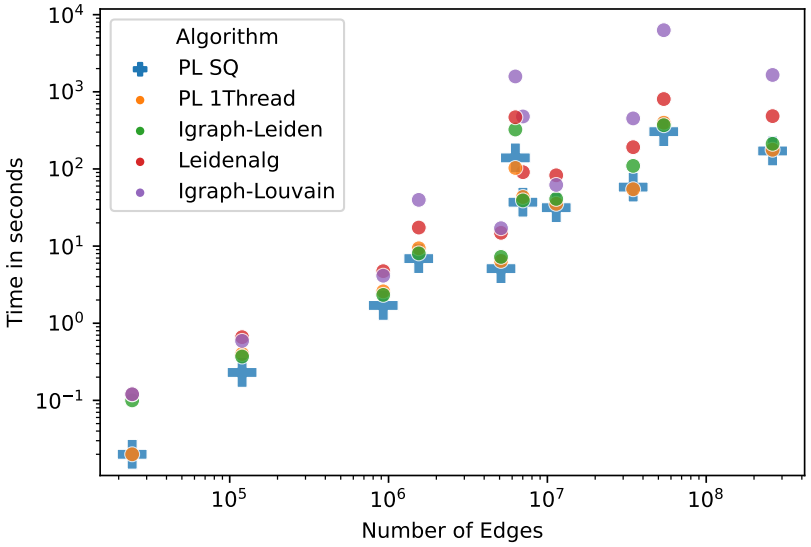

Figure 5.1: Comparison of single threaded performance. PL SQ has been given a different shape to avoid exact overlaps.

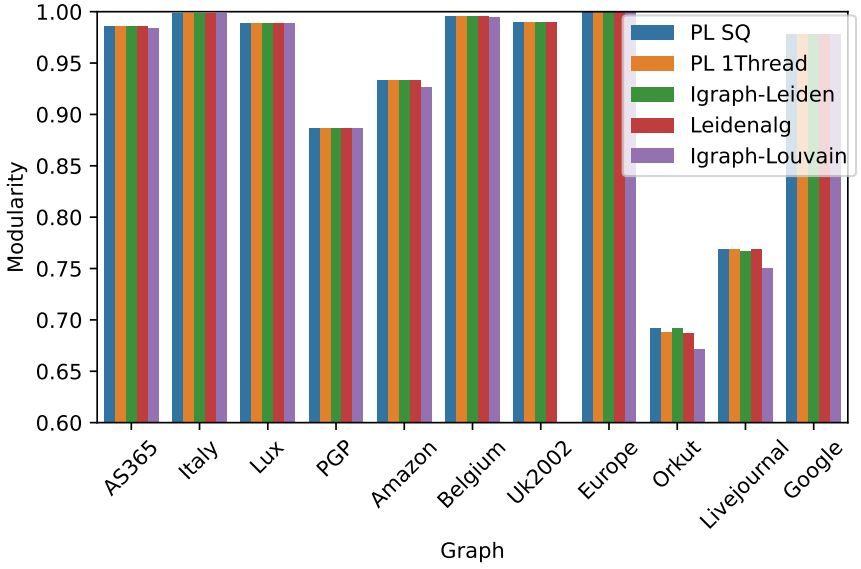

Figure 5.2: Achieved modularity of sequential algorithms grouped by graph

In figure 5.1 we can see that Louvain consistently takes the longest time to finish by far,being up to 20 times slower than PL. While Leidenalg is already considerably faster than Louvain it still falls behind Igraph by a significant amount. Our parallel implementation and Igraph-Leiden run at roughly the same speed in most cases, the parallel version being marginally slower than the sequential one for the most part. The sequential implementation is the fastest on every graph except one, where the parallel version beats it slightly. The margin between Igraph-Leiden and our implementations varies a lot by graph with Igraph being slower by up to 200% than our sequential Leiden, though usually much less.

The achieved modularity is virtually the same for all Leiden-implementations as can be seen in figure 5.2. The largest differences occur on the orkut and livejournal graphs, though for both of them the general deviation is also the highest. For example, for the orkut graph we measured a minimum modularity of 0.6813 and a maximum of 0.6944 with similar deviations for all implementations. Louvain is slightly worse on some graphs with a deficit of at most 3%.

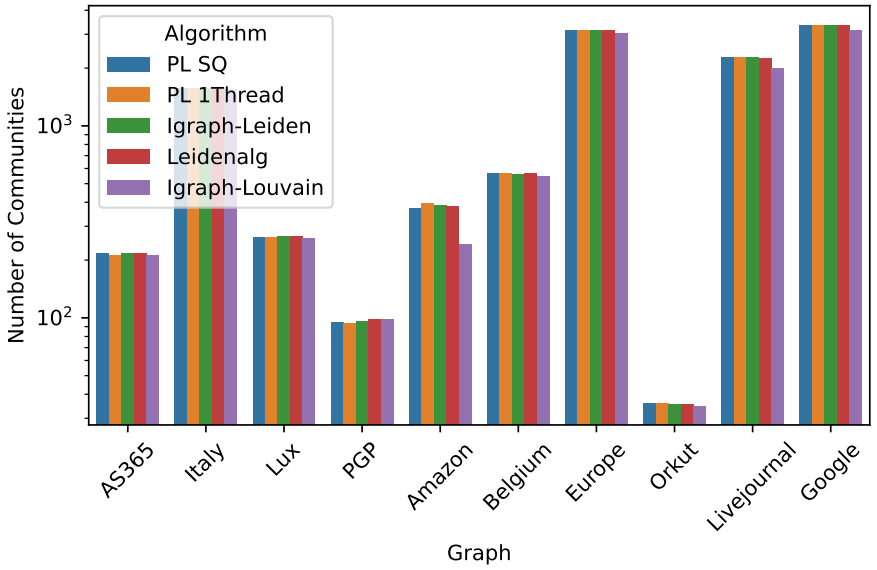

Figure 5.3: Number of communities found by the sequential algorithms grouped by graph

The number of communities found is also similar among the leiden implementations, with average differences of about 2-3%. Louvain is the exception here again and tends to find less communities, e.g. about 37% (150) fewer communities on the amazon graph and roughly 12% fewer on the journal graph.

Note that no data for the Igraph-based implementations is given for the uk graphs. This is because the community count is likely to be incorrect due to issues with integer overflows on large graphs. As the issue has not been completely resolved yet we are unable to circumvent this issue as of the writing of this thesis, see [Igra].

It is unclear whether the modularity and time measurements are reliable, but considering that they fit well compared to our own implementation we assume that they are. Note that Igraph seems to have different limits for different data structures, so it might very well just be the community count that is incorrect, see [Igrb]. The PLM implementation in NetworKit has also been tested on the graph to verify our results and gave similar numbers (roughly 6000, while Igraph-based implementations gave 40000+).

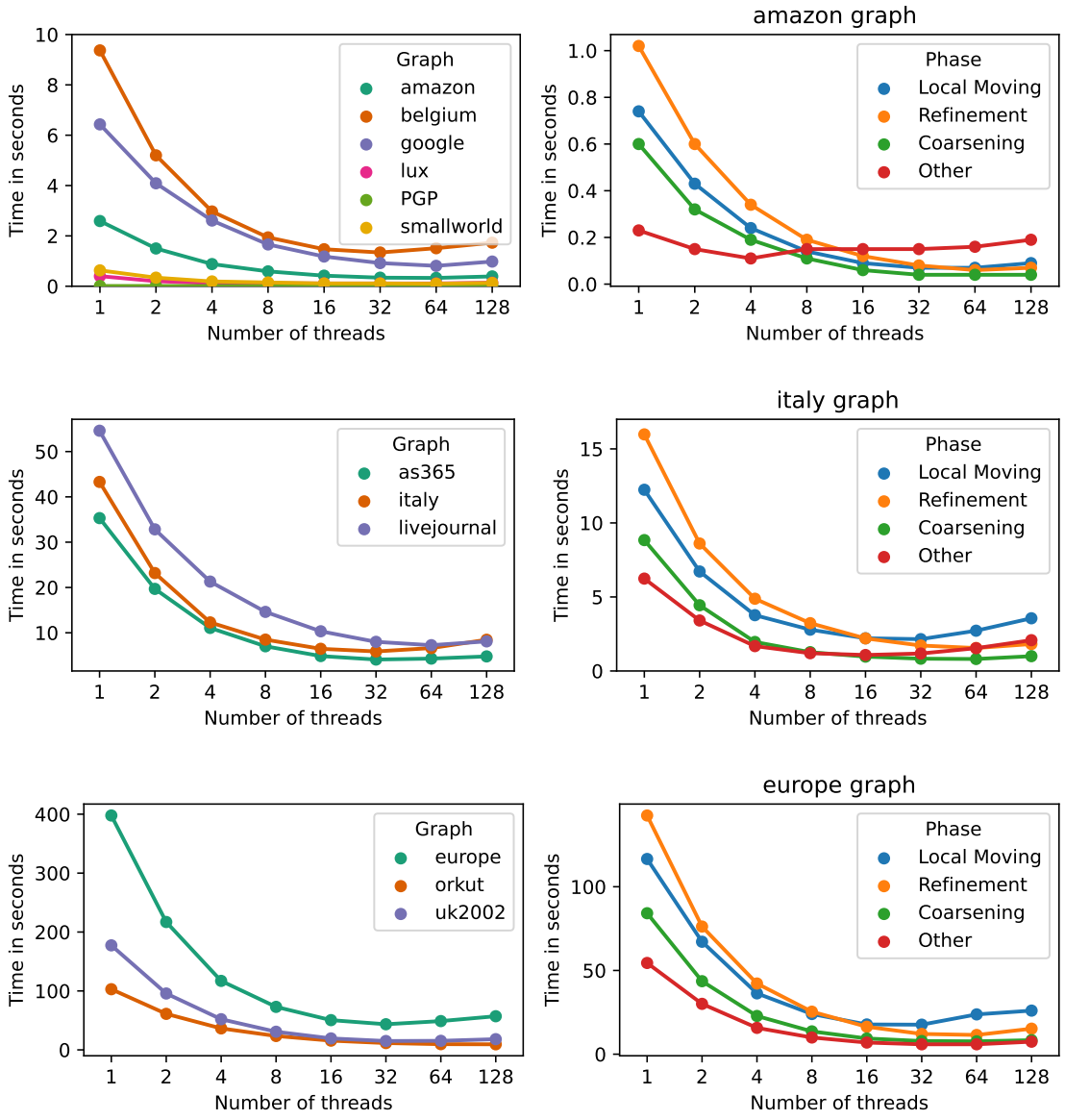

Figure 5.4: Scaling and share of different Phases in the Parallel Leiden algorithm

5.2 Parallel

Continuing with the parallel implementation, the total running time and shares of the different phases (averaged since there are multiple graphs) are given in figure 5.4. *Other* is the total time minus the summed times of the three main phases. For better visibility, graphs with similar running times have been grouped. The uk2007 and big LFR graphs are shown separately for the same reason. Modularity values remained virtually the same in all cases (no differences larger than the typical deviations as in the sequential tests due to randomness) so they will not be addressed separately.

As one would expect, for small graphs the parallelization does not influence the total time significantly. High thread counts (64/128) do not carry heavy impacts as the speedup only decreases slightly or not at all.

For medium to large sized graphs results are similar with 2 threads achieving a speedup of about 1.7 to 1.8 and 4 threads achieving speedups between 2.5 and 3.2. The highest

speedup, up to 6.5, is typically achieved with 64 threads, though the gains past 16 threads are very insignificant.

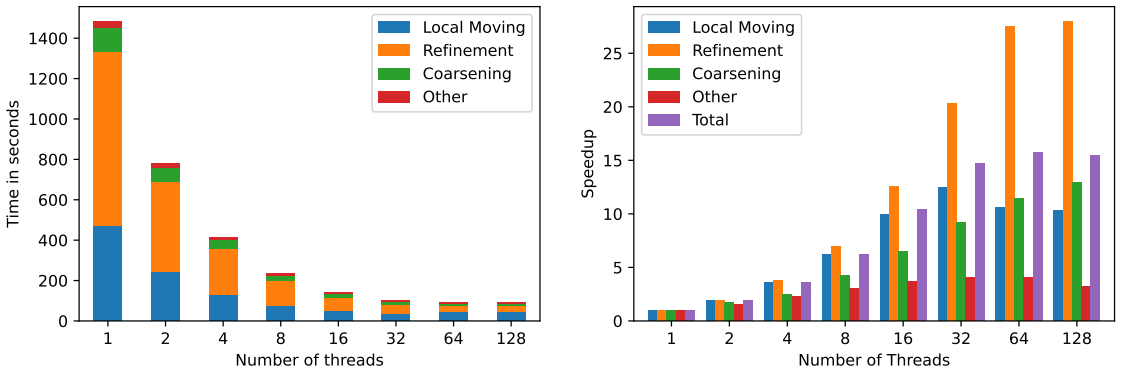

Figure 5.5: Scaling and shares on the largest graph uk2007

The graphs with the highest total time show a significantly better scaling with europe showing a speedup close to 1.8, 3.5 and 5.8 for 2/4 and 8 threads respectively and a maximum speedup of 10.

While the refinement phase initially takes the most time in virtually all graphs it also has the best scaling by far, often beating the local moving phase (in terms of total time) past 16 threads. Figure 5.5 shows this for the largest tested graph uk2007 specifically. The refinement phase shows a maximum speedup of 28 with 128 threads, and efficient speedups up to 16 threads. In contrast, local moving only scales well up to 8 threads and only moderately up to 32 threads, giving a maximum speedup of 12.5. Past 32 threads, speedup even *decreases* down to 10.3 with 128 threads.

Coarsening shows the worst scaling by far though it also takes up barely any time in comparison to the other phases. "Other" contains a lot of sequential code and also takes up very little time, even when compared to coarsening, no significant speedups should be expected here.

The exact speedups are as follows:

Threads	1	2	4	8	16	32	64	128
Total	1	1.90	3.55	6.24	10.37	14.69	15.76	15.49
Local Moving	1	1.92	3.63	6.19	9.94	12.50	10.58	10.28
Refinement	1	1.95	3.8	6.98	12.54	20.29	27.55	27.96
Coarsening	1	1.72	2.51	4.29	6.45	9.16	11.45	12.9

Table 5.1: Speedups uk2007 graph

The last large graph that was tested is a generated LFR graph with 100 million nodes and 980 million edges. The parameter `mu` was chosen as 0.5 which means that roughly half of all edges are inter-community edges. More details on the LFR algorithm are specified in the next paragraph.

In this example, local moving and refinement have similar speedups until 16 threads. From there on up to 64 threads refinement takes a slight lead before falling behind local moving again with 128 threads. While local moving again only scales up to 32 threads at most (and then slows down) refinement speeds up to 64 threads before then also slowing down.

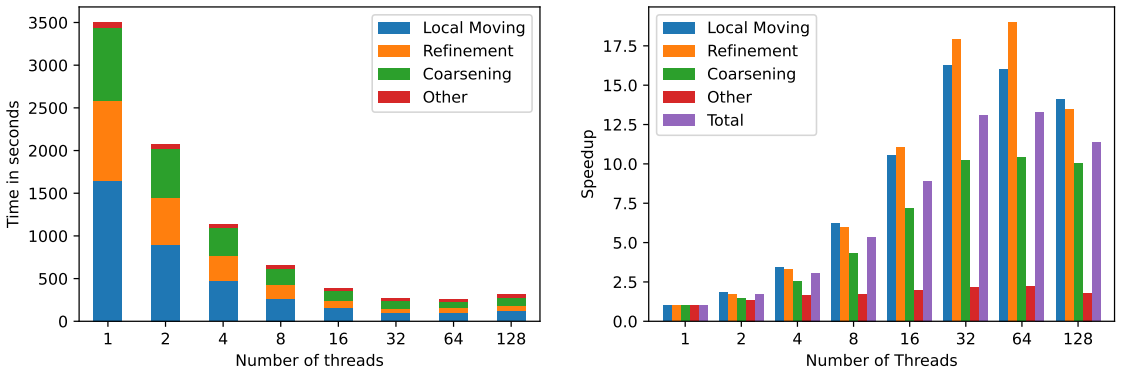

Figure 5.6: Scaling on big LFR graph.

Compared to the uk2007 graph differences vary. While local moving sees an increase in speedup, the refinement phase sees a decrease. This is also true for the absolute time needed however. This is the only tested graph in which local moving takes longer than the refinement phase, which is likely because of the high number of inter-community edges. Coarsening stays roughly the same, though on very high thread counts the speedup also increases slightly.

PLM

We also briefly tested PLM on the uk2007 graph. PLM shows a slightly better overall scaling with a maximum total speedup of 18.7 (PL had a maximum speedup of 15.8). However, unlike our implementation PLM has a maximum count of iterations for the local moving phase (standard Louvain and Leiden repeat until no more changes are made).

Additionally, while the first local moving in our implementation makes up roughly 60% of the total time *spent on local moving* (with 1 thread, roughly 50% else), PLM spends more than 95% of its *total time* on the first local moving phase. Note also that PLM was only run once, unlike all other Leiden implementations which were run 3 times.

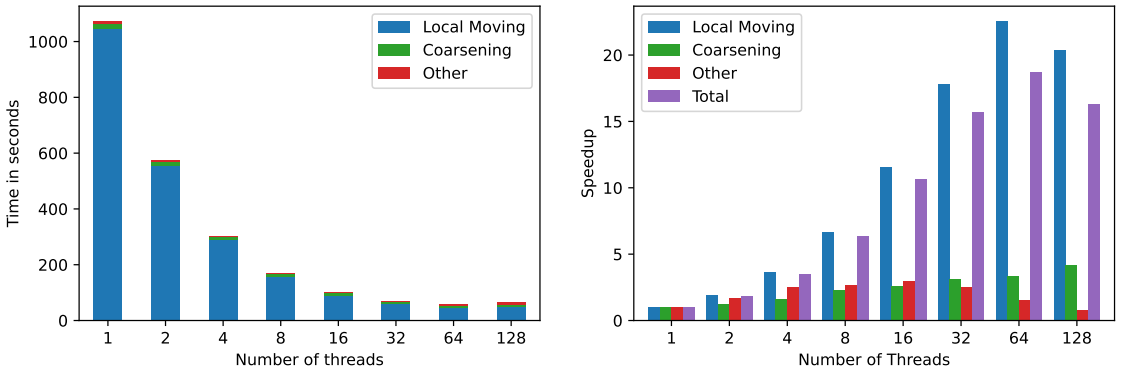

Figure 5.7: PLM scaling on uk2007 graph

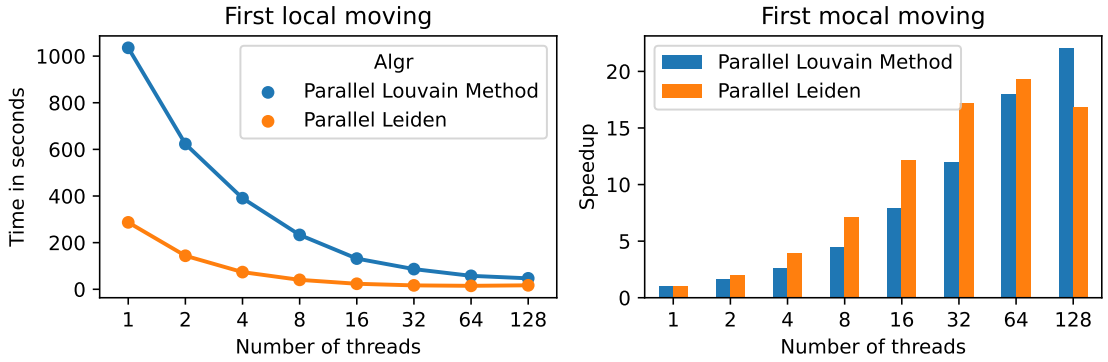

Figure 5.8: Scaling of first local moving phase on uk2007 graph

Randomization

In general (i.e in the sequential Louvain and Leiden algorithms), randomization can help lower the total running time in graphs that have community structure [Tra15]. The reverse effect is also possible however, especially when graphs are saved with a certain node order already present. In most of the graphs used here locality in the order in which nodes are saved usually also implies a high probability that these nodes stem from the same community. This poses a contrast to the fact that we would prefer an order of nodes that are *not* in the same community in a parallel environment, since this allows us to move nodes more freely (that is, without waiting for locks to be free). It is unclear which of these effects dominates or whether this can even be said in general.

Regarding modularity our results confirm observations made in [BGLL08] by showing virtually no changes, whereas our parallel Leiden implementation takes about 5-20% longer to finish when run on the pre-randomized (i.e node ID's have been shuffled before running the algorithm) europe graph depending on the number of threads. While the speedup also increases slightly for the pre-randomized graph (as can be seen in figure 5.9) this is probably due to the longer running time.

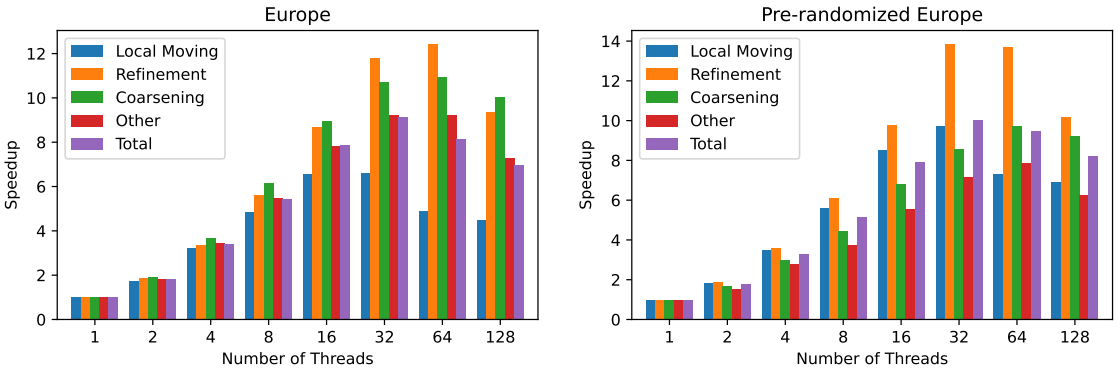

Figure 5.9: Scaling of unrandomized and pre-randomized europe graph in comparison

However, we could not determine any performance gains through randomization in code in any case, regardless of whether the graph was randomized beforehand or not. On the contrary, performance often significantly decreased. For example, the europe graph took 53 seconds to finish when randomized (again, in code) compared to 36 seconds when it was not. Note that we always shuffle nodes in parts only, which influences the number of permutations that can result from that shuffle. However, this saves a significant amount of

time compared to sequentially shuffling as this took up to 10 seconds in total for the europe graph in our tests (and consequently resulted in being slower than a per-parts shuffle).

It is unclear whether this is specific to our implementation or whether the effect of randomization generally does not show in a parallel context like this. Further study of this should definitely be considered if one wishes to use this algorithm.

Number of edges between communities

Both modularity and the time needed scale with the number of inter-community edges in the graph. For modularity a higher number of such edges naturally means a lower score, since the graph can not be partitioned as easily. Likewise, since the communities are less clearly separated it takes more time for the algorithm to identify and separate communities. To show this effect, we compare graphs that were generated by the LFR algorithm [LFR08] implemented in NetworKit with 1 million nodes and roughly 10 million edges. The LFR algorithm is used to generate benchmark graphs for community detection. It generates a graph in which node degrees and community sizes are chosen according to a power law. The mixing parameter `mu` specifies the share of edges that are between different communities. For this test, we chose a minimum of 20, maximum of 50 and an exponent of -1 for the community size power law and 20,50 and -2 respectively for the degree power law.

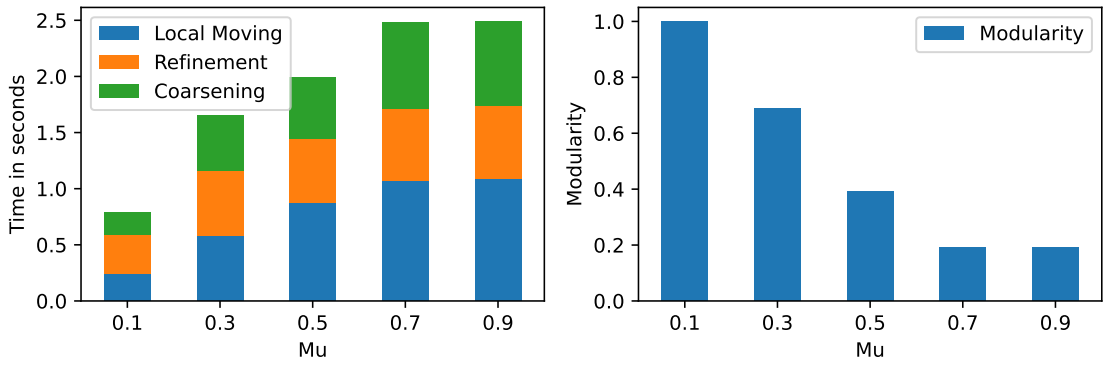

Figure 5.10: Effect of number of edges between communities (32 cores)

While refinement shows a slight increase in time needed when mu is low (which results mainly from more refinement phases happening, as local moving takes more iterations to find a stable partition), no changes can be observed past a mu of 0.3. Considering that the refinement phase only ever considers a node once, this is to be expected. On the other hand, the local moving phase sees a drastic increase up to a mu of 0.7.

For modularity, the achieved values decrease up to a mu of 0.7 as well.

Alternative approaches to parallelization

Both local moving and refinement phase have been implemented in two different ways. The ones we call experimental are local moving with thread-local queues and refinement with parallelization on a per-cluster basis only.

We tested these on two graphs to show a brief impression of the differences.

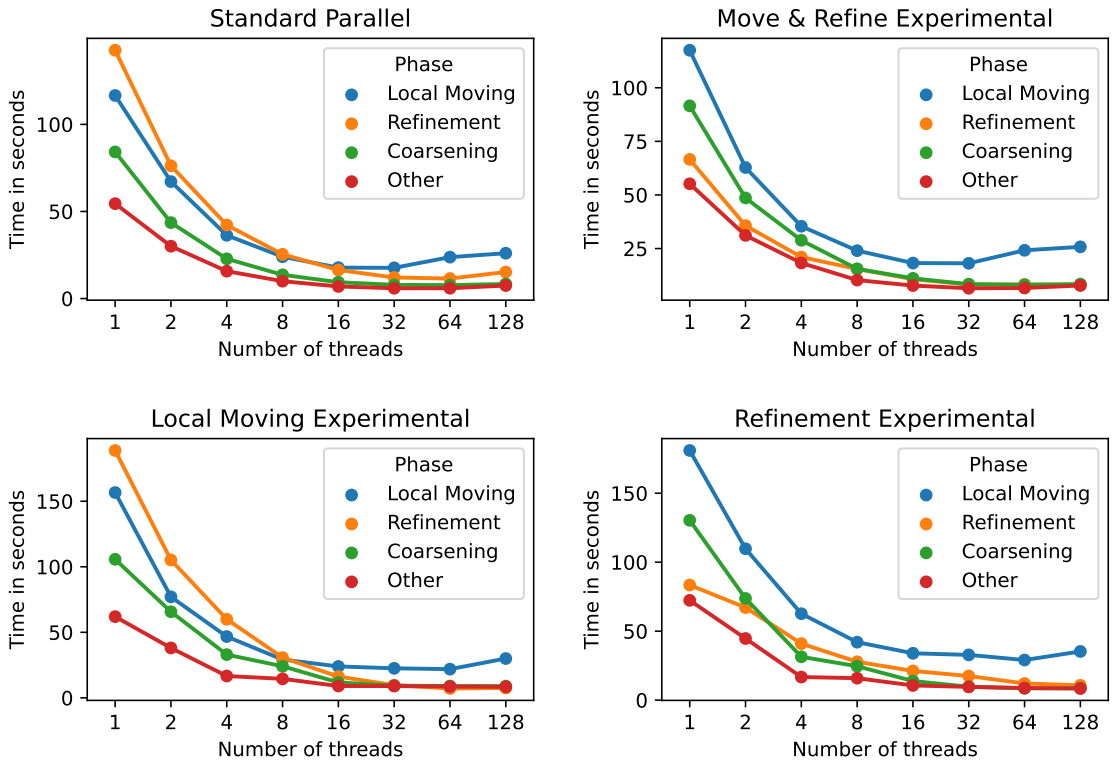

Figure 5.11: Scaling and shares of phases with Experimental methods (europe graph)

For the europe graph, using the experimental local moving results in a slightly higher total time for both local moving and refinement with low thread counts, whereas on high thread counts (8 or more in this case) refinement is slightly faster. Using the experimental refinement results in a significantly faster refinement phase but also slows down all other phases. This is likely a result of the difference in randomization (remember that the per-cluster refinement contains no randomization). Notably, using both experimental versions together results in a *lower* total time. Since the two local moving versions also have different randomization behaviour (or rather, the order in which nodes are considered) the two experimental versions may be complementing each other in regards to the order in which nodes are considered.

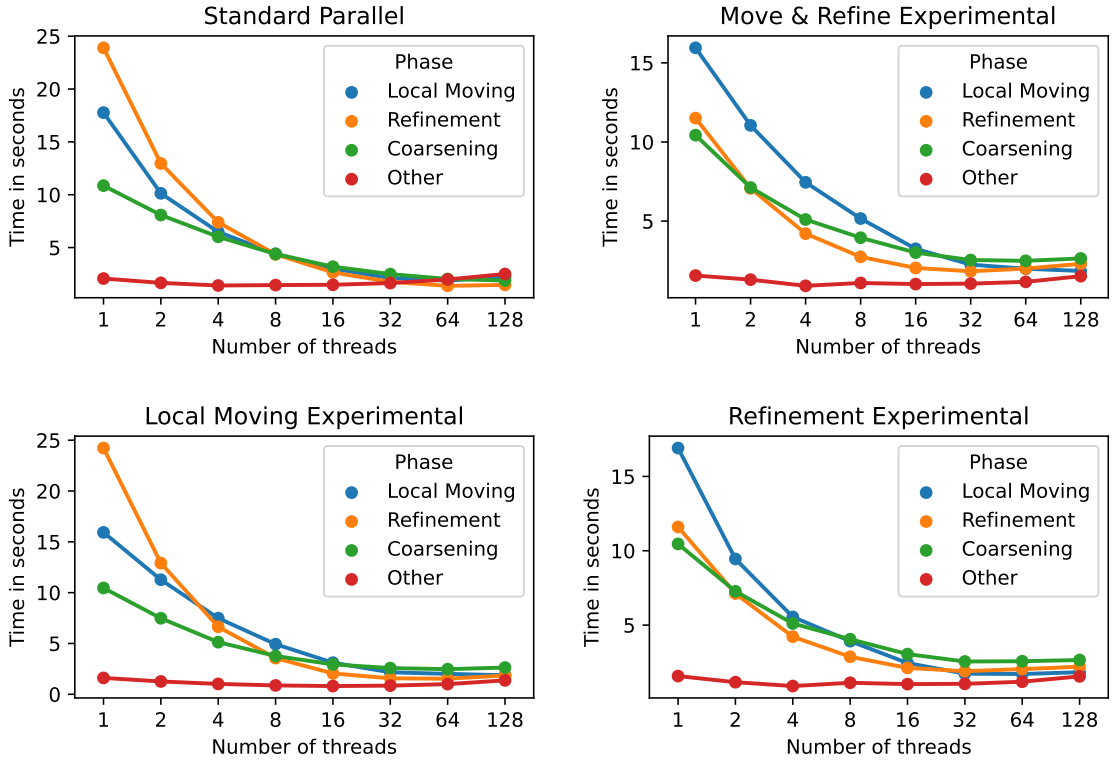

Figure 5.12: Scaling and shares of phases with Experimental methods (livejournal graph)

The second graph we tested, the livejournal graph, shows a different behaviour. For this graph, the experimental local moving version results in virtually no difference, while experimental refinement significantly reduces the total time needed, even if used on its own. Using both of them together is practically identical to using standard local moving with experimental refinement.

These results indicate that it may be preferable to use the experimental versions when only few threads are used, though the differences between the main and experimental versions diminish as the number of threads increases.

It is also unclear to what extent these differences result from randomization (or lack thereof).

6. Conclusion

In this thesis, we introduced a parallelized version of the Leiden algorithm which we call Parallel Leiden and evaluated an implementation in C++.

We implemented local moving by combining the active-nodes queue introduced by the Leiden algorithm with the parallelization strategy of PLM. The main ideas here were to use compare-and-swap to determine whether nodes have already been inserted to the queue or not and to use a lock for the queue to avoid race conditions when inserting or removing nodes from the queue.

As for refinement, we initially proceed like in the sequential version and use locks for communities to prevent race conditions while moving the node. This enables us to uphold the main guarantee of the refinement phase, to never produce disconnected communities. While it can not be guaranteed that the move will be optimal, it will generally be among the best ones.

Coarsening has been used as implemented in NetworKit.

We evaluated our implementation by testing twelve graphs, the smallest one having 24000 edges and the biggest one 3.3 billion. Among other application backgrounds, these include real life street networks, web graphs as well as social media networks and generated benchmark graphs. Our implementation achieves modularity scores virtually identical to those of the sequential leiden algorithm. The refinement phase achieves speedups up to 28 with 128 threads while local moving achieves speedups up to 12.5 with 32 threads, with a maximum total speedup of about 15.8 which is achieved with 64 threads.

More than 32 threads eventually lead to a decrease in performance for local moving, it is unclear whether this stems from an inefficient implementation or from a general limitation of the approach. Considering that PLM (as implemented in NetworKit) shows similar speedups, this may also be a limitation of NetworKit or parallel community detection in general. Further analysis of this issue may significantly improve local moving performance, as its speedups fall far behind those of the refinement phase.

The algorithm may also be improved by finding less restrictive solutions for race conditions as we need to fully lock communities during moves in the refinement phase and check multiple times that the considered node is still a singleton. A possible approach to this is to use colorings which restrict which nodes of the graph may be moved simultaneously as mentioned in [LHK15].

Bibliography

[BCSV04] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[BDG+08] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):172–188, 2008. doi:10.1109/TKDE.2007.190689.

[BGLL08] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large its. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, Oct 2008. doi:10.1088/1742-5468/2008/10/p10008.

[BnPSDGA04] Marián Boguñá, Romualdo Pastor-Satorras, Albert Díaz-Guilera, and Alex Arenas. Models of social networks based on social distance attachment. *Phys. Rev. E*, 70:056122, Nov 2004. doi:10.1103/PhysRevE.70.056122.

[BRSV11] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[CLA12] Siew Yin Chan, Teck Chaw Ling, and Eric Aubanel. The impact of heterogeneous multi-core clusters on graph partitioning: An empirical study. *Cluster Computing*, 15(3), 2012. doi:10.1007/s10586-012-0229-4.

[CN06] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006. URL: https://igraph.org.

[Des] Cameron Desrocher. A fast general purpose lock-free queue for c++. URL: https://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++.htm.

[DIM] 10th dimacs implementation challenge - graph partitioning and graph clustering. URL: https://www.cc.gatech.edu/dimacs10/downloads.shtml.

[FB07] Santo Fortunato and Marc Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007. doi:10.1073/pnas.0605965104.

[For10] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, Feb 2010. doi:10.1016/j.physrep.2009.11.002.

[GdMC10] Benjamin H. Good, Yves-Alexandre de Montjoye, and Aaron Clauset. Performance of modularity maximization in practical contexts. *Physical Review E*, 81(4), Apr 2010. doi:10.1103/physreve.81.046106.

[GEO] Geofabrik.de europe street networks. URL: http://download.geofabrik.de/europe.html.

[Ham21] Michael Alexander Hamann. *Scalable Community Detection*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2021. doi:10.5445/IR/1000133317.

[Igra] Igraph fix integer type issue. URL: https://github.com/igraph/igraph/pull/1626#issuecomment-856873159.

[Igrb] Igraph put integer types into order issue. URL: https://github.com/igraph/igraph/issues/1450#issuecomment-667711046.

[KSKK07] J. M. Kumpula, J. Saramäki, K. Kaski, and J. Kertész. Limited resolution in complex network community detection with potts model approach. *The European Physical Journal B*, 56(1):41–45, March 2007. doi:10.1140/epjb/e2007-00088-4.

[LFR08] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E*, 78(4), Oct 2008. doi:10.1103/physreve.78.046110.

[LHK15] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37, August 2015. doi:10.1016/j.parco.2015.03.003.

[LLDM08] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters, 2008. arXiv:0810.1355.

[NG04] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69:026113, Feb 2004. doi:10.1103/PhysRevE.69.026113.

[Ngu] Fabian Nguyen. Parallel leiden github repo. URL: https://github.com/CxVercility/ParallelLeiden.

[PR] Networkit pull request. URL: https://github.com/networkit/networkit/pull/802.

[RAB09] M. Rosvall, D. Axelsson, and C. T. Bergstrom. The map equation. *The European Physical Journal Special Topics*, 178(1):13–23, Nov 2009. doi:10.1140/epjst/e2010-01179-1.

[SM16] Christian L. Staudt and Henning Meyerhenke. Engineering parallel algorithms for community detection in massive networks. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):171–184, 2016. doi:10.1109/TPDS.2015.2390633.

[SNA] Stanford large network dataset collection. URL: https://snap.stanford.edu/data/index.html#communities.

[SSh16] Christian L. Staudt, Aleksejs Sazonovs, and Henning henke. Networkit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, December 2016. doi:10.1017/nws.2016.20.

[Tra15] V. A. Traag. Faster unfolding of communities: Speeding up the louvain algorithm. *Physical Review E*, 92(3), Sep 2015. doi:10.1103/physreve.92.032801.

[TVDN11] V. A. Traag, P. Van Dooren, and Y. Nesterov. Narrow scope for resolution-limit-free community detection. *Physical Review E*, 84(1), Jul 2011. doi:10.1103/physreve.84.016114.

[TWvE19] V. A. Traag, L. Waltman, and N. J. van Eck. From louvain to leiden: guaranteeing well-connected communities. *Scientific Reports*, 9(1), March 2019. doi:10.1038/s41598-019-41695-z.

[Ver20] G. Verweij. Faster community detection without loss of quality: Parallelizing the leiden algorithm. 2020.

[V.T] V.Traag. Leidenalg. URL: https://github.com/vtraag/leidenalg.

[YL12] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth, 2012. arXiv:1205.6233.