UPPSALA
UNIVERSITET

Optimization Of A Community Detection Algorithm

Louvain Algorithm On Multilayer Networks

Raghid Abdeljawad, Sina Mohammadi

Project in Computational Science

January 2020

Project Report

Abstract

The amount of data available for analysis is far greater today than just a decade ago. With the rise of social media and the internet, there has been an increasing demand for data analytics and data mining. One of the important analysis that can be done on networks is to detect communities (clustering of data), due to the fact that communities usually have different properties with respect to each other. Finding these communities are hence of grave importance when trying to gain useful insight about a particular network. However the tools available for this type of analysis are not optimal in regards to performance due to the vast amount of data that can be used, the detection of communities takes to much time. There is a range of different algorithms that are used for community detection. One that has gained in popularity during recent years is the Generalized Louvain algorithm. Due to the performance issues with the Generalized Louvain algorithm, this project has been conducted in order to find optimizations on a particular implementation of the algorithm. Since, the data available for analysis is always increasing, high performance tools are required. This project has resulted in the characterization of bottlenecks within the algorithm and obtained a speedup of two times with the use of concurrency.

Contents

1 Introduction **3**

2 The Objective **3**

3 Prerequisite knowledge **3**
- 3.1 Multinet . . . 3
- 3.2 Networks . . . 3
- 3.3 Generating Networks . . . 5
- 3.4 Community detection Algorithm . . . 5
  - 3.4.1 Modularity . . . 5
  - 3.4.2 Generalized Louvain Algorithm . . . 6
- 3.5 Concurrency in OpenMP . . . 7
- 3.6 Implementation of OpenMP . . . 8
  - 3.6.1 Modularity Matrix . . . 8
  - 3.6.2 Update step function . . . 8

4 Workflow **9**

5 Results **11**
- 5.1 Black Box testing . . . 11
  - 5.1.1 Results for one layer, changing k_{min} and η . . . 11
  - 5.1.2 Results for three layer, changing k_{min} and η . . . 13
  - 5.1.3 Results for five layer, changing k_{min} and η . . . 14
  - 5.1.4 Summarized computational times . . . 16
  - 5.1.5 Results for 10k nodes changing μ and p . . . 17
- 5.2 White Box testing . . . 18
  - 5.2.1 The while loop computational time, update steps of nodes in the network . . . 18
  - 5.2.2 The computational time of the modularity calculations . . . 20
- 5.3 Concurrency results . . . 21
- 5.4 Analysis of the memory usage . . . 22

6 Discussion **25**

7 Conclusion **26**

1 Introduction

The amount of data available for analysis is far greater today than just a decade ago. With the rise of social media and the internet, there has been an increasing demand for data analytics and data mining. A lot of data can be visualized as multidimensional networks, see Figure 1. Usually there is a lot of important information that can be extracted from such data, by using advanced algorithms or methods. Each network contains communities or a group of nodes, finding these communities are of importance because they usually have very different properties with respect to each other. Therefore it is important to use algorithms to detect these communities, since they contain more specific information about the subsets of the network. Several types of algorithms exist for revealing the community structure in networks, such as the Generalized Louvain algorithm which is the one optimized during this project. The performance of this algorithm, however, is not as good as expected. The problem with this algorithm is the time complexity which is O($n^2 logn$). In order to improve the time complexity, one can use concurrency to achieve a better speedup.

In this project we will aim at improving the performance of the Generalized Louvain algorithm by using concurrency. Since the generalized Louvain algorithm is written in C++, we use OpenMP, also in part due to the simplicity of programming concurrent operations in the parallelizing framework.

2 The Objective

The main objective of this project is to use concurrency in order to optimize the community detection algorithm Generalized Louvain. The path of doing that is planed as follows:

- Generating different multidimensional networks with different parameters. These are refereed to as benchmarks and have different characteristics with respect to each other.
- Test the algorithm (Black box testing) on the generated networks and to find the computational time. Every test, if possible is to be ran more that 10 times, since the average value is taken later.
- Find computational time of the different parts of the algorithm to get an idea where the algorithm spends most time. There are specially two big parts that has been considered, where almost 90 % of the time is spent within.
- Use concurrency to increase the performance of the algorithm. Several parallelization paradigms can be used, in this project OpenMP is used.

3 Prerequisite knowledge

3.1 Multinet

Multinet is a package in the CRAN repository. This package is written by three Authors at Uppsala University Matteo Magnini, Davide Vega and Mikael Dubik. The objective of this package is to analyze multilayer networks (explained below). It also contains functions that preprocess, analyse and mine networks.

One of the features that exists in this package is the Generalized Louvain. This algorithm is useful, since it has the ability to find and extract communities from different multilayer networks, see [4].

3.2 Networks

A simple way to describe a network is to represent it as nodes/points that are connected to each other. An example of a network could be how different individuals interact with each other on different social media platforms. A network can have different dimensions. By dimensions we mean the "planes" were the nodes are located. In the rest of the report we refer to these dimensions as *layers*. In some other scientific sources these are named *slices*. Every layer in the network contains a number of nodes. These nodes have connections (refereed to as *edges*) between each other, these connections could be friendships, two products from the same company etc. The significance of the edge is decided upon by an analyst so essentially they may represent anything. These connections are called *intralayer edges* or *intralayer dependencies*. There are other connections between nodes

in different layers and these are called *interlayer connections*. The interlayer edges are usually between the same entities in different layers. For instance, the LinkedIn profile and Facebook profile of one and the same person have a interlayer connection between them.

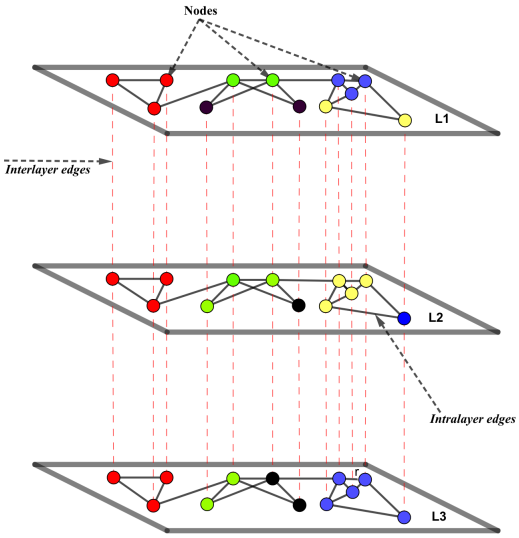

Figure 1: The multiplex network 1

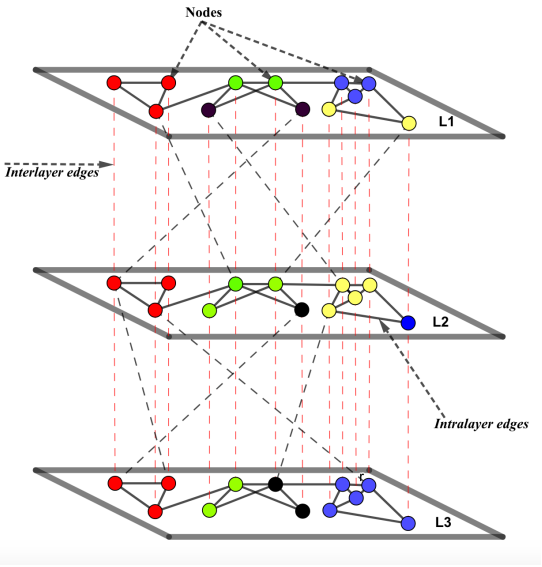

Figure 2: The multiplex network 2

There are networks that have interlayer diagonal connections (connections between a node X in layer 1 and node Y layer 2, see Figure 2) and networks that only have interlayer connection between the same nodes from different layers, see Figure 1. The type of networks that are considered in this project is Figure 1. A simple example of such a network could be a social media network with users as nodes and their common properties as the edges between them. Figure 1 and Figure 2 should be considered as a illustrative example. Networks with more than three layers are considered in this paper.

An important concept regarding networks is communities. A community structure refers to the occurrence of a group of nodes that are more densely connected to each other than the rest of the network. In Figure 3, each color refers to a community in the network. Communities are often related to a partition of the set of nodes, which means that every node only belongs to a unique community.

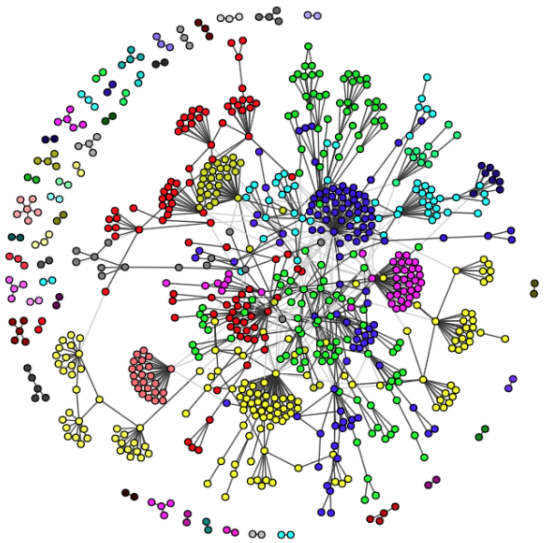

Figure 3: A network with communities where each color corresponds to a single community [9]

There are various methods used to detect these communities, specially in large networks. Problems, such as community detection in social networks are often considered as an optimization problem, where different appropriate algorithms are employed to optimize a function. This function must have a property of computing values (in our case the modularity) that are related to communities. Furthermore, each network contain information that can be extracted using different methods. You can extract the average properties by doing simple computations on the network, but it is not enough. Since the individual communities have quite different properties than the average properties of the network. The existence of communities normally affects different processes like a spreading of deceases or a group of people in the same course etc. Using a community detection algorithm helps to make an overall understanding of such processes [6], [5].

3.3 Generating Networks

In this project, two scientific papers have been part of the pilot study for the project. The first paper is about the generation tool that the authors have implemented. This framework [7] has been used to generate the different benchmarks used during the project, see [6]. The second one describes the algorithm and the implementation of a new feature that has been implemented in order to avoid local maxima, see [5]. The generation tool has different parameters that control different characteristics of the network. The parameters that are considered in this project are k_{min}, k_{max} and η parameters (η is also referred to as exp parameter). These parameters control the power law distribution in Equation 1. From this distribution we sample the amount of edges a node has. As we see later in the report, the density of nodes/edges is one of the important characteristics that are considered for the generated networks. Power law distribution is a distribution function that controls edge density in the framework used to generate benchmarks.

$$p(x) = \begin{cases} Cx^{-n}, & x_{min} \leq x \leq x_{max} \\ 0, & \text{Otherwise} \end{cases} \tag{1}$$

where,

$$C = \frac{\eta - 1}{x_{min}^{-(\eta-1)} - x_{max}^{-(\eta-1)}}.$$

In Equation 1, x_{min} is the minimum number of edges of a single node, while x_{max} is the maximum number of edges for each node. The number of edges can not increase over the limit ($\#\, nodes - 1$), that would be a node that is connected to every other node in the layer. By choosing different values for these parameters in this distribution, we create multiplex network with different edge densities.

We generate networks with different parameter combinations. Mainly, we change the parameters x_{min}, and η, since they are found to have the greatest impact on the density of the edges in the network. Other parameters have also been subject to exploratory testing, however no impact has been detected for those.

3.4 Community detection Algorithm

3.4.1 Modularity

Modularity can be explained as a measure of a network structure. It is designed to estimate the strength of a group of nodes (community/ cluster of nodes) in a network. What is meant by the strength here is how dense a cluster of nodes is compared to the rest of the network, i.e. the density of edges in a network is higher with higher value of the modularity and vice versa. The formula for computing the modularity is given in Equation 2,

$$Q_{\text{single layer}} = \frac{1}{2m} \sum_{i,j} (a_{i,j} - \frac{k_i k_j}{2m}) \delta(\gamma_i, \gamma_j), \tag{2}$$

where **m** is the number of nodes in the network, **i**,**j** are the node identifiers, $a_{i,j}$ is the connection constant, which is 1 if **i** and **j** are connected and 0 elsewhere, k_a is the degree of node **a** i.e. the number of edges node **a** has, $\delta(a, b)$ is the δ-function which is 1 if nodes a and b are in the same community and 0 if they are not.

We note that Equation 2, describes the modularity of a single layer network. The modularity of a multilayer

network needs to modify Equation 2. The difference in the formula for multilayer modularity is the interlayer connections between the layers. Equation 3 describes the modularity of a multilayer network,

$$Q_{\text{multilayer}} = \frac{1}{2\mu} \sum_{ijsr} \left(\left(A_{ijs} - \gamma_s \frac{k_{is} k_{js}}{2m_s} \right) \delta_{sr} + \delta_{ij} C_{jsr} \right) \delta(g_{is}, g_{jr}). \tag{3}$$

Observe, that **i** and **j** are the indices for two different nodes and **s** and **r** are the indices for two different layers. Each network layer is represented by a adjacency matrix A_{ijs} with the connections of node **i** and **j** in layer **s**. The resolution parameter γ_s for layer **s**. The matrix C_{jsr} contain the interlayer connections that connects node **j** in layer **s** to itself in layer **r**. The community identifier $\delta(g_{is}, g_{jr})$ is a **Dirac Delta**, which is 1 if node **i** at layer **s** is connected to node **j** in layer **r** and 0 if they are not connected. In this project this function will be equal to 1 just when **i = j**, see Figure 1. The δ_{sr} is 1 if the two layers **s** and **r** are connected and 0 otherwise and similar to δ_{ij} where it will take the value 1 if node **i** and **j** are connected and 0 otherwise. Finally, the parameter k_{is} describes the number of edges node **i** has at layer **s**.

The modularity is usually used in optimization methods for network structures community detection. Modularity can be seen as the fraction of the edges that exist in a unique community and the expected fraction if edges were randomly distributed, see [5].

3.4.2 Generalized Louvain Algorithm

A general method, used to extract communities from large networks is the Louvain algorithm. This method is a method for modularity optimization, explained in Section 3.4.1. Optimizing the modularity will theoretically give the best possible grouping of the nodes of a given network. The Louvain algorithm finds small communities by optimizing the modularity locally and later each of these small communities are aggregated into one group. The groups are then considered as a node. The first step is then repeated, until all nodes become associated with a group.

Every node in the network is assigned to its own community or group. For each node **i**, the change in modularity is computed for removing **i** from its community and placing it in the community of each neighbor **j** of **i**. This value is easily calculated in two steps: (1) removing **i** from its original community, and (2) inserting **i** to the community of **j**,

$$\Delta Q = \left[\frac{\sum_{in} + 2k_{i,in}}{2m} - \left(\frac{\sum_{tot} + k_i}{2m} \right)^2 \right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m} \right)^2 - \left(\frac{k_i}{2m} \right)^2 \right]. \tag{4}$$

Equation 4 describes the change of the modulaity for a node, where $\sum_{in}$ is all the weights of the connections between nodes inside the target community, $\sum_{tot}$, k_i is the weighted degree of **i**, $k_{i,in}$ denotes all the weights of the connections that connect **i** and all other nodes in the target community, m is the sum of all the weights of all connections in the network. When ΔQ is computed for each community **i** is connected to, **i** is replaced into the community with the greatest modularity. If there is no increase in the modularity, **i** stays in its original group. This process is repeated to all nodes until there is no modularity increase anymore. Once the local maximum of the modularity is reached, that phase is completed. The Generalized Louvain algorithm is the modified Louvain algorithm that is used for more than one layer, see [1].

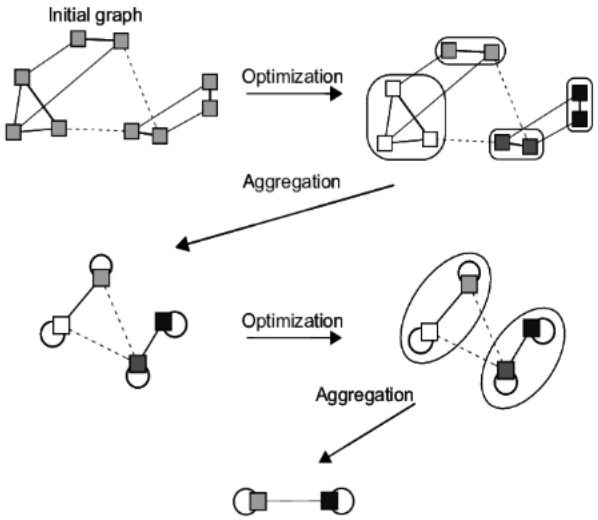

Figure 4: Sketch of the algorithm

3.5 Concurrency in OpenMP

OpenMP is a programming paradigm used for parallel programming. This model supports shared memory multi-processing programming in several programming languages as C and C++. OpenMP is based on two parallelism concepts: threads and the fork/join model. Suppose that your computer executes a sequence of instructions and think of a thread as a process that can be executed. The fork/join model begins when a thread (Master thread) splits into a number of threads (called forking). The different threads are then used as workers and joined together when the process is finished. The part between the fork and join is the region where the parallelization is happening, often refereed to as the **"Parallel region"**.

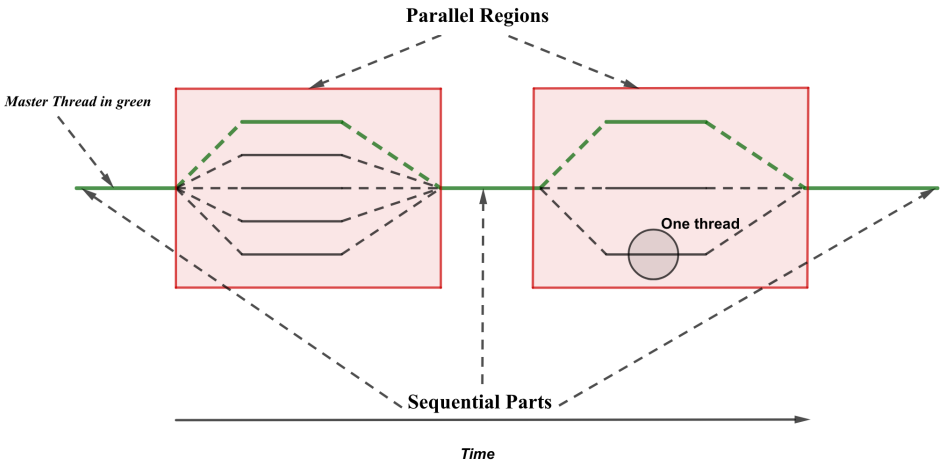

Figure 5: A visualization of OpenMP

We include some more details in the sequel, the threads that are forked are all copies from the master thread. As mentioned above hence, they have access to all the computations until the concurrency step (Shared data). However, the threads are not identical in the sense that they work on different data. Each thread must also have private data and the threads can be identified by their unique thread number. Another important concept is the division of work, when using OpenMP to parallelize a for-loop, OpenMP will instruct each thread to work on a

partition of the loop. Leading to balanced work among the different threads.

Another equally important concept is the choice of the number of threads. To get a good speedup one should typically set the number of threads to be equal to the number of cores of the computer used. However, creating more than that works as well but it will not necessarily give you more speedup.

3.6 Implementation of OpenMP

3.6.1 Modularity Matrix

OpenMP was implemented in the part of the code where the computation of the modularity matrix takes place, since it does a lot of independent tasks. The computation of the modularity iterates through all the nodes as well as through all the layers. Clearly, this part of the code is expected to consume larger time for larger networks.

Algorithm 1 The pseudo code of modularity calculation

```
num_threads= the number of cores that will be used in the parallel region
omp_set_num_threads(num_threads)
#pragma_omp_parallel{
#pragma_omp_critical{
for i = 0 to number of layers do
    for j = 0 to number of nodes do
        for k = 0 to number of nodes do
            Add elements to a triplet list. tlist.push_back()
        end
    end
end
}
}
```

Algorithm 2 The pseudo code of modularity calculation

```
num_threads= the number of cores that will be used in the parallel region
omp_set_num_threads(num_threads)
#pragma_omp_parallel{
#pragma_omp_critical{
for i = 0 to number of layers do
    for j = 0 to number of nodes do
        for k = 0 to number of nodes do
            Add elements to triplet list. tlist.push_back()
            Add elements to triplet list2. tlist21 .push_back()
        end
    end
end
}
}
```

#pragma is one of OpenMP's commands. The task of this command is to enable the program to run in parallel using several threads. The number of threads can be chosen by changing the **num_threads** in **omp_set_num_threads(num_threads)**, see Algorithms 1 and 2.

3.6.2 Update step function

The update step function and the parallel section illustrated in psuedo code in 3.

Algorithm 3 The pseudo code of the update step function

```
num_threads= the number of cores that will be used in the parallel region
omp_set_num_threads(num_threads)
#pragma omp parallel shared(vars){
while condition do
    #pragma_omp_critical{
    for each column in modularity matrix do
        find_best_moves
    end
end
}
}
```

4 Workflow

Step 1: There are two scientific papers (see [6] and [5]) on multilayer networks and the modularity of a network that has been part of the pilot study. These papers has been read carefully by us in order to understand the contents.

Step 2: Paper [6] provides a MATLAB program that generates multilayer networks. The code is used in our project to generate different networks (benchmarks). The network is produced in that form of a tensor. Depending on the parameters we choose, the network has different characteristics. In Table 1 an example of a generated network in the format is visualized (example in Figure 1, in Section 3.2, where the network has three layers). From the tensor in Table 1 we can conclude that the network contains three layers (each represented by one table in, Table 1) and 100 nodes on each layer. The Boolean edge column indicates that there is an edge between Node i and Node j. In addition to the information we have from the tensor, we know that all nodes have a connection to them selves in different layers, also called interlayer connections (see Figure 1 in Section 3.2) . This tensor must be converted to a different format that the multinet package can read. The format is a ASCII file format (.mpx) file with the structure displayed in Table 2. Looking at the first row in Table 2, (n1 , n45 , 1), means that node 1 and nodes 45 are connected and both lay in layer 1.

Layer 1		
Node i	Node j	Bool Edge
1	99	1
2	4	1
2	21	1
2	46	1
3	32	1
4	64	1
4	3	1
⋮	⋮	⋮
99	22	1
100	1	1

Layer 2		
Node i	Node j	Bool Edge
1	100	1
2	53	1
2	22	1
3	41	1
3	11	1
4	98	1
4	2	1
⋮	⋮	⋮
99	22	1
100	1	1

Layer 3		
Node i	Node j	Bool Edge
1	79	1
2	6	1
3	11	1
3	46	1
3	52	1
4	14	1
5	3	1
⋮	⋮	⋮
100	72	1
100	13	1

Table 1: Format of the generated network.

Node 1	Node 2	Layer
n1	n45	1
n2	n11	1
n3	n23	1
n4	n55	1
	⋮	⋮
n99	n100	1
n100	n25	1
n1	n52	2
n2	n56	2
n3	n21	2
	⋮	⋮
n100	n1	2
n1	n57	3
n2	n96	3
n3	n78	3
n4	n22	3
⋮	⋮	⋮
n98	n43	3
n99	n15	3
n100	n12	3

Table 2: The format the of the network (.mpx).

Step 3: There are eight different parameters that one can change to generate different networks. Networks with different number of layers, different number of nodes, different number of edges etc. All the combination between these parameters is tested for 1000 nodes (number of nodes chosen due to time constraints).

Step 4: The algorithm Generalized Louvain is tested on each one of the benchmarks. During this step the algorithm is considered as a black box and the performance of the algorithm is evaluated. After the exploratory testing, we noticed a trend for increasing amount of edges. Hence, the parameters in the power law distribution are to be considered and tested in further detail.

Step 5: Benchmarks with the following parameters has been generated for the testing:

- **UpdateSteps=200**, The update steps of the computation of the modularity.
- **n_layers=i**, The number of layer in the network.
- **n_nodes=1000**, the number of nodes in the network.
- **p = 0.5**, vector of probabilities where p(i) is the probability for a state node to copy its community assignment from corresponding state nodes in the ith aspect.
- **mu = 0.2**, fraction of random edges.
- **theta = 1**, concentration parameter for Dirichlet null distribution.
- **comm = 1**, The number of communities in the network.
- **q = 1**, probability for a community to be active in a given layer.
- η **= j**, power law exponent for expected degree distribution.
- k_{min} **= k**, minimum expected degree.
- k_{max}**= 200**, maximum expected degree.
- **maxr** = 100, maximum number of rejections before bailing out and issuing a warning (the resulting network has less than the desired number of edges).

See pseudo code for the generation of benchmarks in Algorithm 4. The index i is for the number of layers , j and k are the parameters in the Power Law Distribution that controls the number of edges in the network, see Equation 1.

Algorithm 4 How the benchmarks was generated

```
for i = 1:3 do
    for j = -5:1:5 do
        for k = 10:10:100 do
            Generate network with specific characteristic
        end
    end
end
```

Step 6: For this test, the computer has to generate 10*11*3 = 330 benchmarks. The benchmarks are then tested and the computational time is saved to a text file. We used the results from the text file to plot the results in a 2D color map.

Step 7: Analyse the most time consuming parts of the code. This part began by analysing the algorithms bottlenecks in the previously done tests. The algorithm contains a lot of loops that can be parallelized using OpenMP. However, we need to find time consuming ones. Hence we whitebox test the algorithm.

Step 8: Implementing OpenMP where it could be implemented in the code. There are different ways to parallelize loops with OpenMP. Some features of the loop need to be considered before, such as shared variables, etc.

Step 9: When the implementation of OpenMP is done, all the 330 tests are ran again in order to see whether the performance has improved or not.

5 Results

The results are presented in two major groups. Section 5.1 contains the result from the algorithm before using concurrency and explains how the computational time of the algorithm depend on the type of the network. Note that the parameters used in the subsections of Section 5.1 are displayed in the beginning of each subsection. The computational time of the algorithm has been divided into two parts, the read time of the networks and the time of the algorithm itself. The second part of the results Section 5.3 shows the effect of parallelizing the code using OpenMP.

In Section 5.1 you will see the results from different networks, changing the k_{min} parameter and η parameter, since they control the density of edges in the networks (see Equation 1).

5.1 Black Box testing

5.1.1 Results for one layer, changing k_{min} and η

Figure 6 illustrates how the computational time increases while incrementing the number of edges. This gives an idea of how the code performs on different edge densities by changing the parameters in the power law distribution. We still can not see which part of the code takes most of the time, i.e. what part of the code is most influenced by the variation of the edge density. The important information is the asymptotic behavior of the computational time relative to the number of edges, which is one of the bottlenecks of the algorithm.

$$\Big\{ \boxed{N = 1000}, \boxed{\textbf{L = 1}}, \boxed{\text{Update steps} = 200}, \boxed{\text{Theta} = 1}, \boxed{\text{Community} = 1}, \boxed{q = 1}, \boxed{\text{K}_{max} = 200}, \boxed{\mu = 0.1}, \boxed{\text{max-rej} = 100}, \boxed{p = 0.5} \Big\}$$

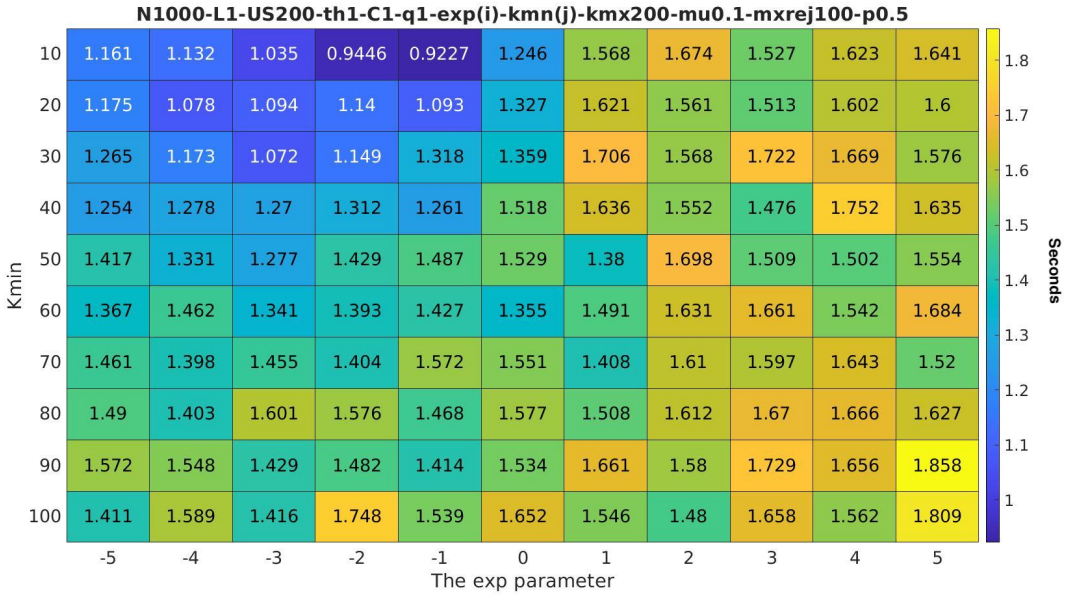

Figure 6: Computational time of the algorithm when changing exp and k-min parameters from the powerlaw distribution

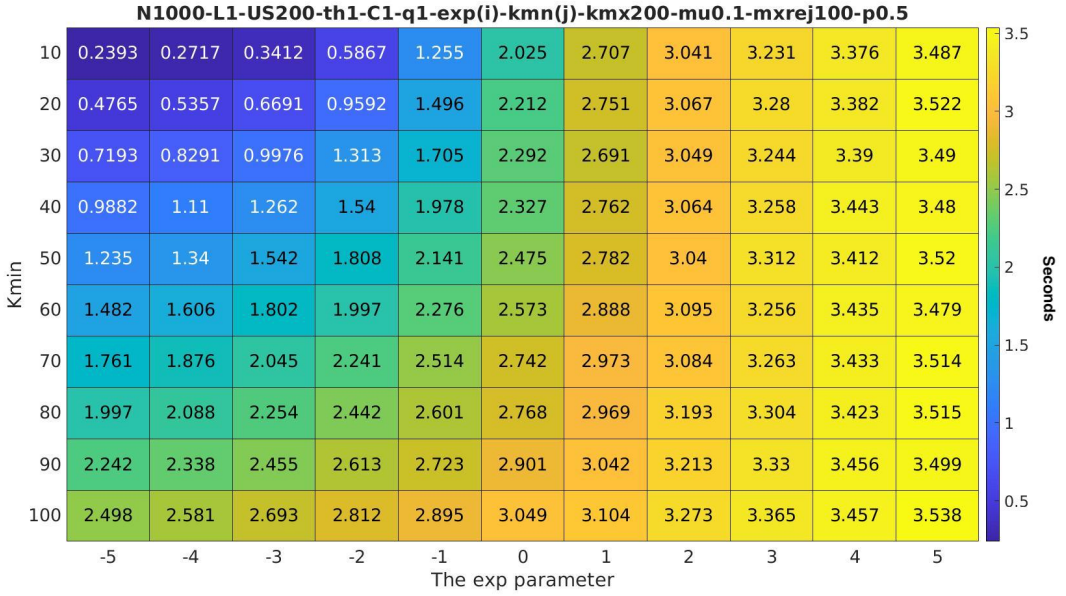

Figure 7: Read time when changing exp and k-min parameters from the power law distribution.

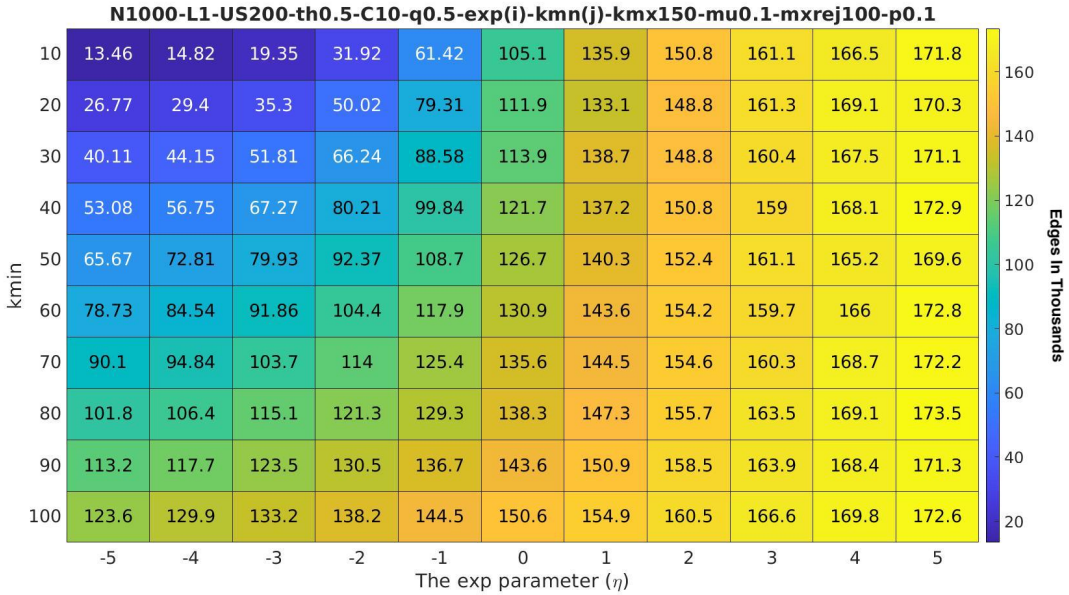

Figure 8: Number of edges in the network when changing the parameters k-min and exp parameters from thepower law distribution.

5.1.2 Results for three layer, changing k_{min} and η

We see the same behaviour in Figure 9 as in Section 5.1.1, however the computational time here is a bit larger than the computational time in Figure 6, since this is a three layer network.

$$\Big\{ \boxed{N = 1000}, \boxed{\textbf{L = 3}}, \boxed{\text{Update steps} = 200}, \boxed{\text{Theta} = 1}, \boxed{\text{Community} = 1}, \boxed{q = 1}, \boxed{\text{K}_{max} = 200}, \boxed{\mu = 0.1}, \boxed{\text{max-rej} = 100}, \boxed{p = 0.5} \Big\}$$

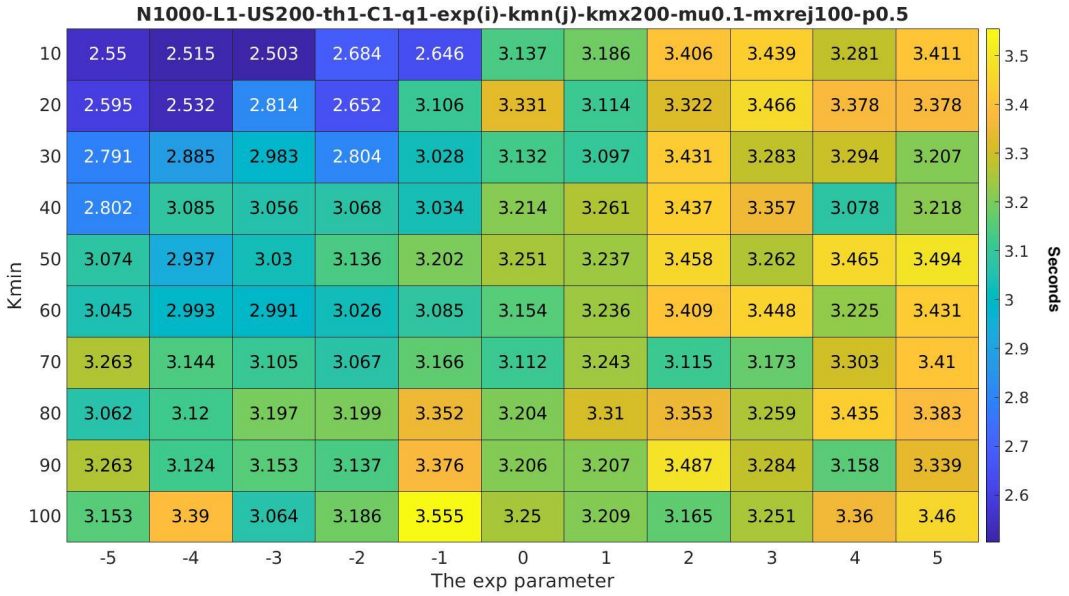

Figure 9: Computational time of the algorithm when changing exp and k-min parameters from the powerlaw distribution.

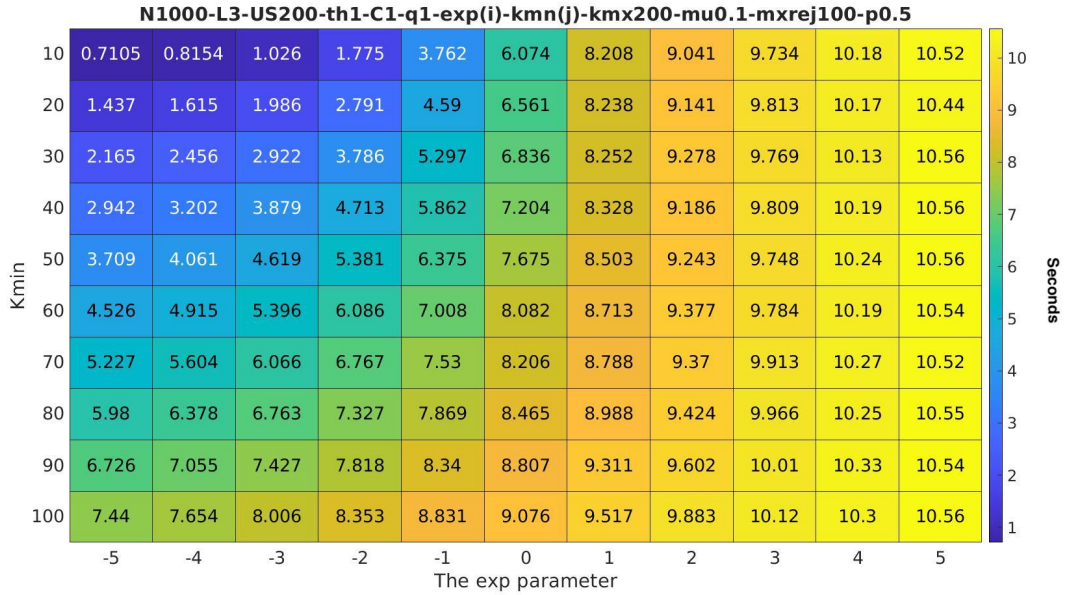

Figure 10: Read time of the algorithm when changing exp and k-min parameters from the powerlaw distribution.

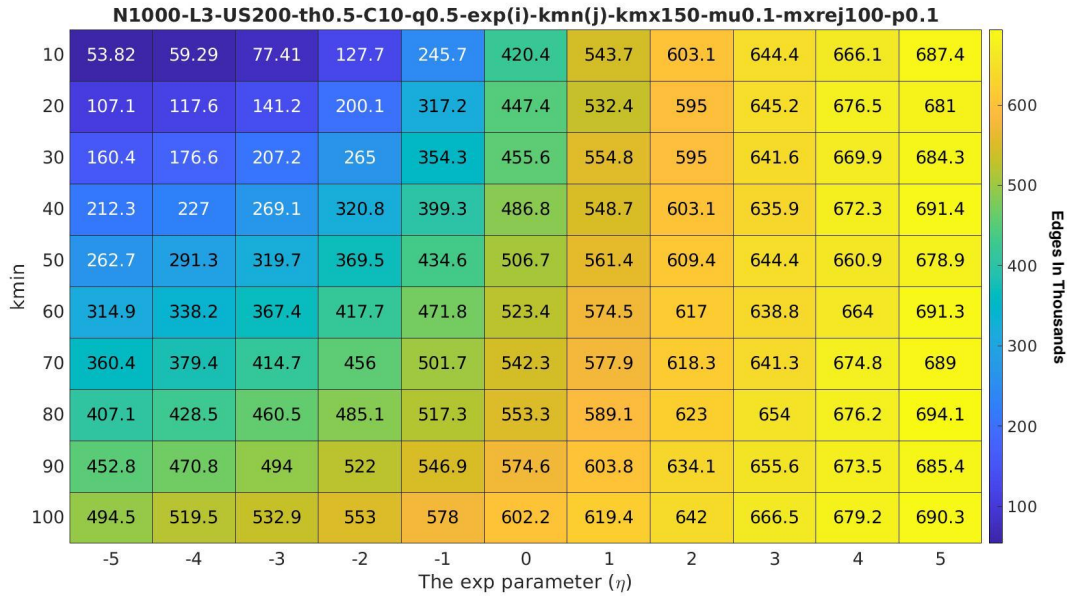

Figure 11: Number of edges in the network when changing the parameters k-min and exp parameters from the power law distribution.

5.1.3 Results for five layer, changing k_{min} and η

In Figures 6, 9 and 12, we see a certain asymptotic behaviour of the computational time with respect to the number of edges. This is a clear bottleneck that is analysed further.

$$\Big\{ \boxed{N = 1000}, \boxed{\textbf{L = 5}}, \boxed{\text{Update steps} = 200}, \boxed{\text{Theta} = 1}, \boxed{\text{Community} = 1}, \boxed{q = 1}, \boxed{\text{K}_{max} = 200}, \boxed{\mu = 0.1}, \boxed{\text{max-rej} = 100}, \boxed{p = 0.5} \Big\}$$

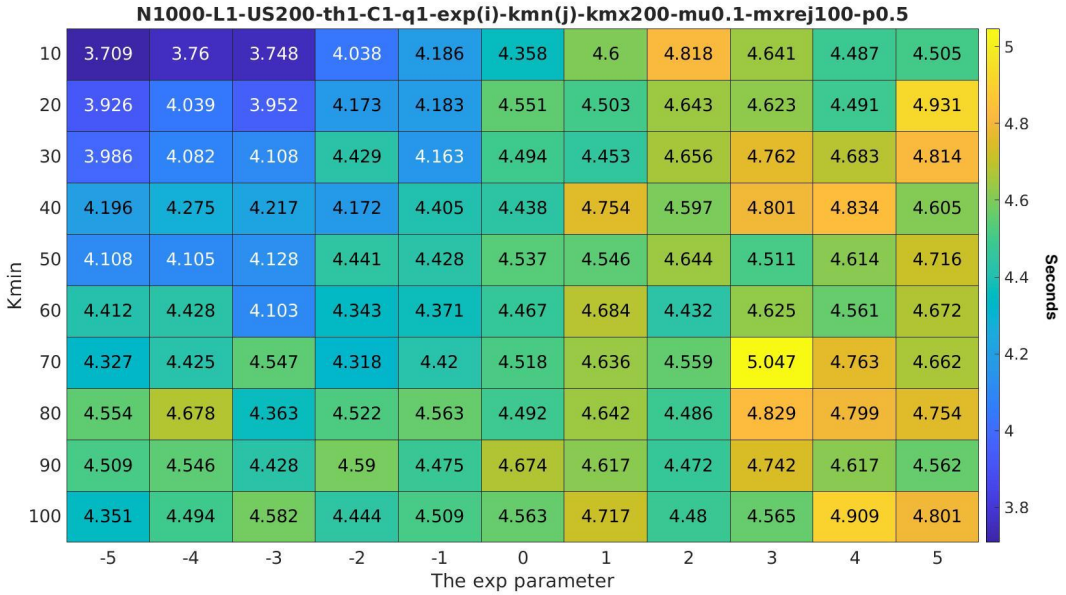

Figure 12: Computational time of the algorithm when changing exp and k-min parameters from the power law distribution.

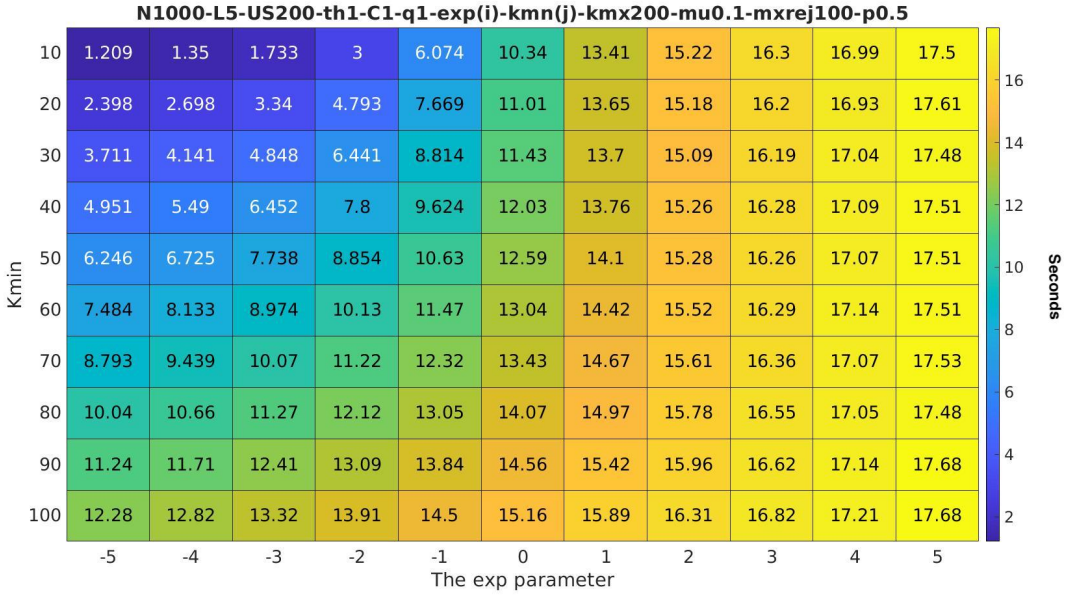

Figure 13: Read time of the algorithm when changing exp and k-min parameters from the power law distribution.

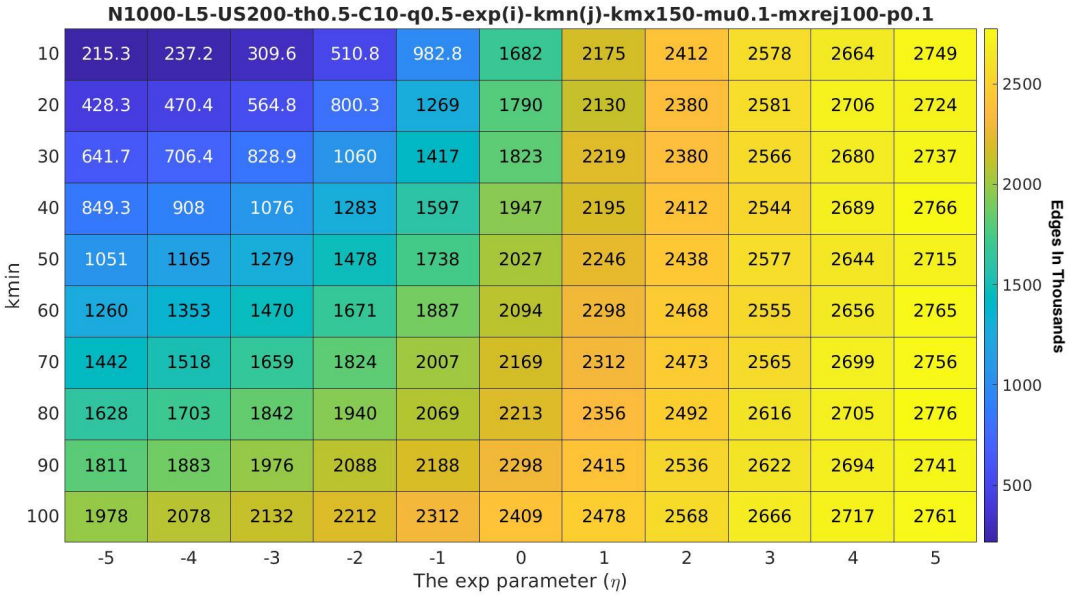

Figure 14: Number of edges in the network when changing the parameters k-min and exp parameters from the power law distribution.

5.1.4 Summarized computational times

The results shown in Figures 6, 9 and 12 are summarized in one figure in order to see more clearly how the computational time changes with respect the number of edges. Figure 15 contains three figures for the three data sets (1, 3, 5) layers. Observe that this result is just for the algorithm itself without the read time of the networks.

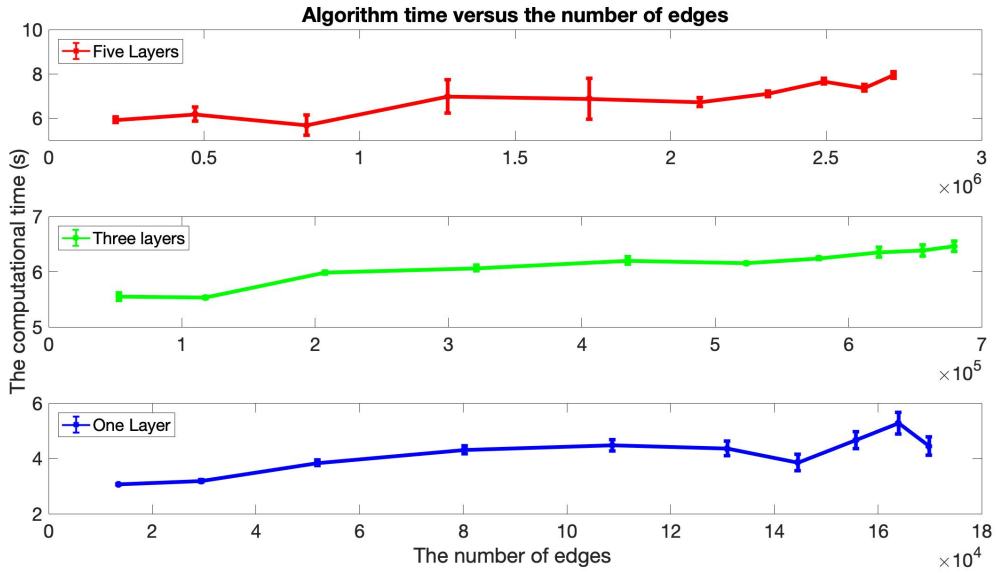

Figure 15: Computational time of the algorithm for a 1000 nodes network for (L1 ,L3 ,L5). Each test is ran 12 times and the average value has been noted. The standard deviation of each test is visualised as well.

Further, the read time of the networks in the three data sets (L1, L3, L5) are also visualized in same way, see Figure 16. Clearly the read time is almost proportional to the number of edges, since the size of the input matrix increases, explained in Section 4.

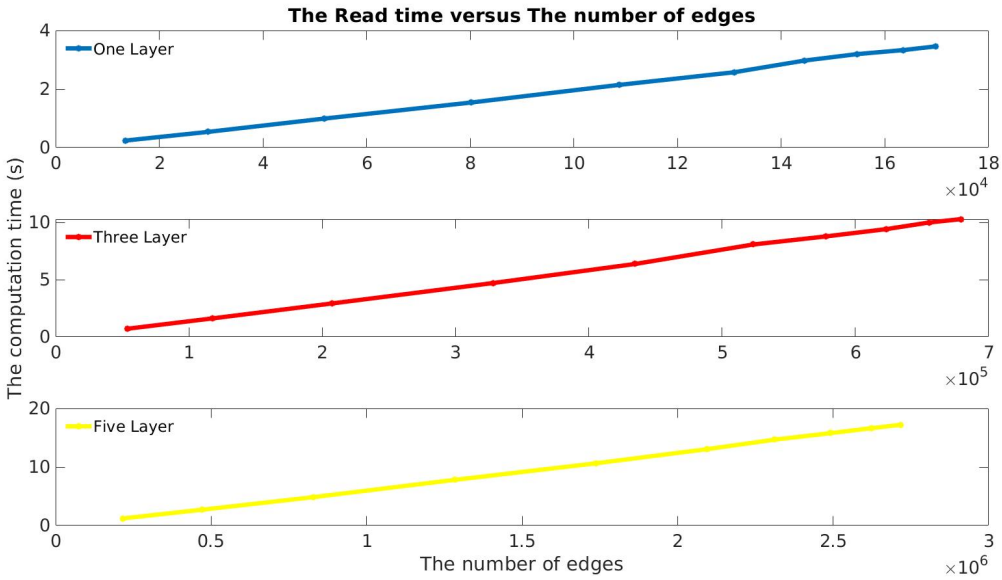

Figure 16: Read time of the algorithm for a 1000 nodes network with (1, 3, 5) layers. Each test has been ran 12 times and the average value has been noted. The standard deviation of each test is visualised as well.

Figures 15 and 16 clearly show that the computational time of the algorithm is affected by the number of edges. It is obvious, from Table 2, that the size of the input matrix increases with the number of edges. This is due to the calculation of the modularity in the Generalized Louvain, since the algorithm goes through all the nodes and communities (See section 3.4.1). However, the time in Figure 16 increases linearly, unlike for the algorithm time in Figure 15. The reason is that the algorithm basically takes more time to read matrices of larger sizes. Having more edges implies larger matrix sizes and that implies larger read time. Observe that a part of the time growth in both Figures 15, 16 is strongly affected by the number of layers as well, since more layers implies more nodes and more edges.

5.1.5 Results for 10k nodes changing μ and p

These results clarify that the tests are indeed very time-consuming. These tests are ran for 10000 nodes with three layers. However, since the number of iterations on this test is too small, we should not draw any conclusions based on the graph. Figure 17 is the **only** one that shows both **read time and clustering time** aggregated in one graph.

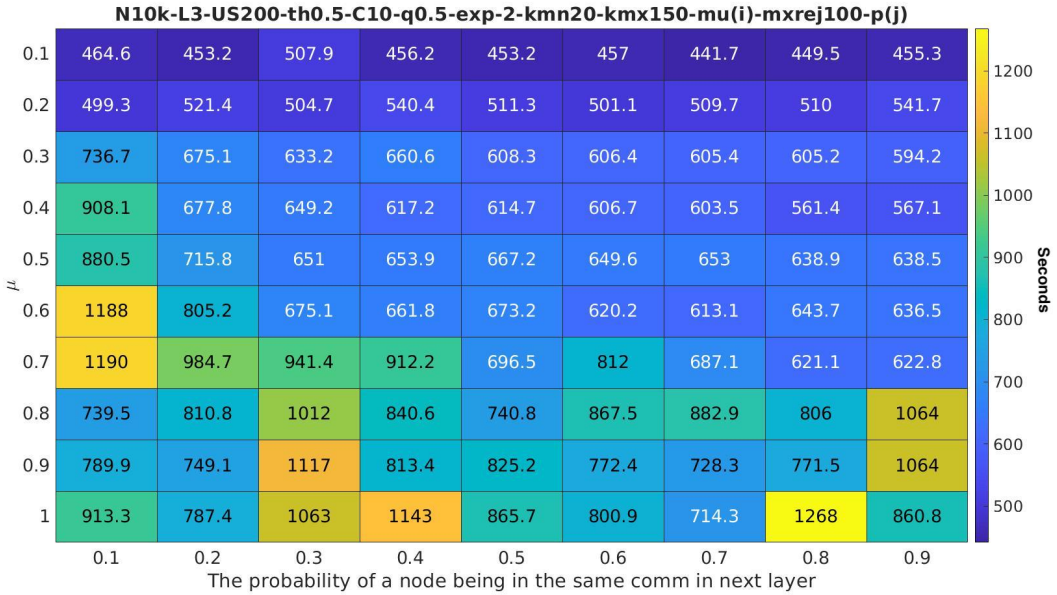

Figure 17: Computational time of the algorithm ran for 10k, changing the parameters μ and p.

Looking at the first "Box" $\mu = 0.1$ and p = 0.1 in Figure 17, the computational time for just that test is 464.6 seconds which is approximately 8 minutes. If we sum up all the times, we get approximately $= 64253s = 1071\text{min} \approx 18h$. In order to get better result, these tests had to be ran about 10 times each and the total time for doing that is $10 * 18 = 180h \approx 8days$. Recall that this test is for three layers. However, running for five layers implies larger networks and therefore consumes much more time.

5.2 White Box testing

As described in Section 3.4.2 the Generalized Louvain algorithm is divided into two main parts which are the most time consuming parts. The first one is the computation of the modularity matrix and the second one is the update steps.

5.2.1 The while loop computational time, update steps of nodes in the network

A detailed analysis of the code has been made, in order to find the parts of the code which are most time consuming. Figures 18, 19 and 20 visualize the computational time of the update step function (modularity optimization) depending on the number of edges for layers equals to 1, 3 and 5.

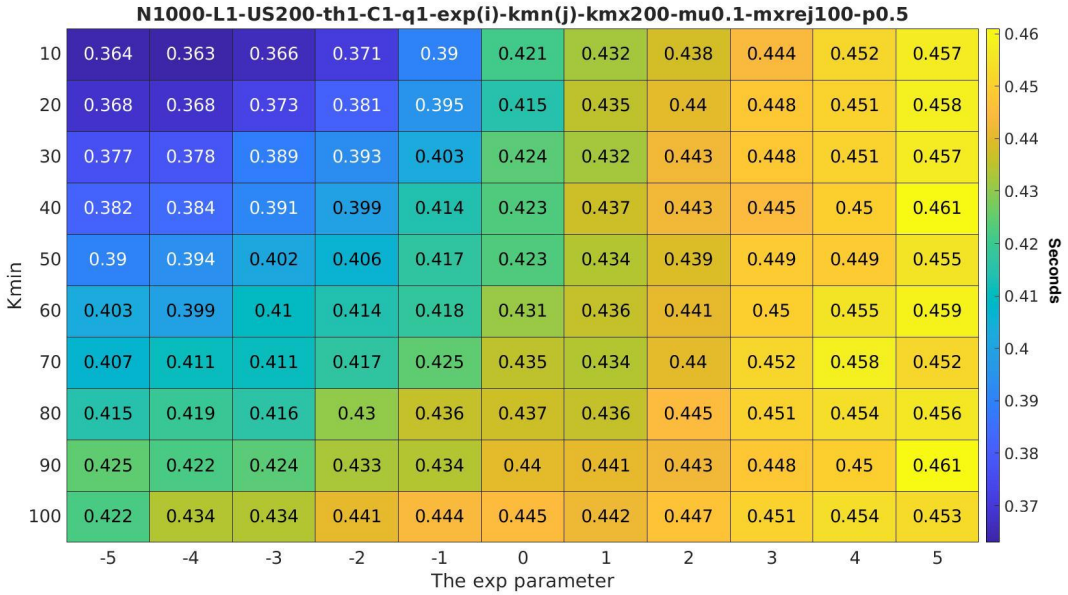

Figure 18: Computational time of the while loop when changing the parameters k-min and exp, from the power law distribution for one layer

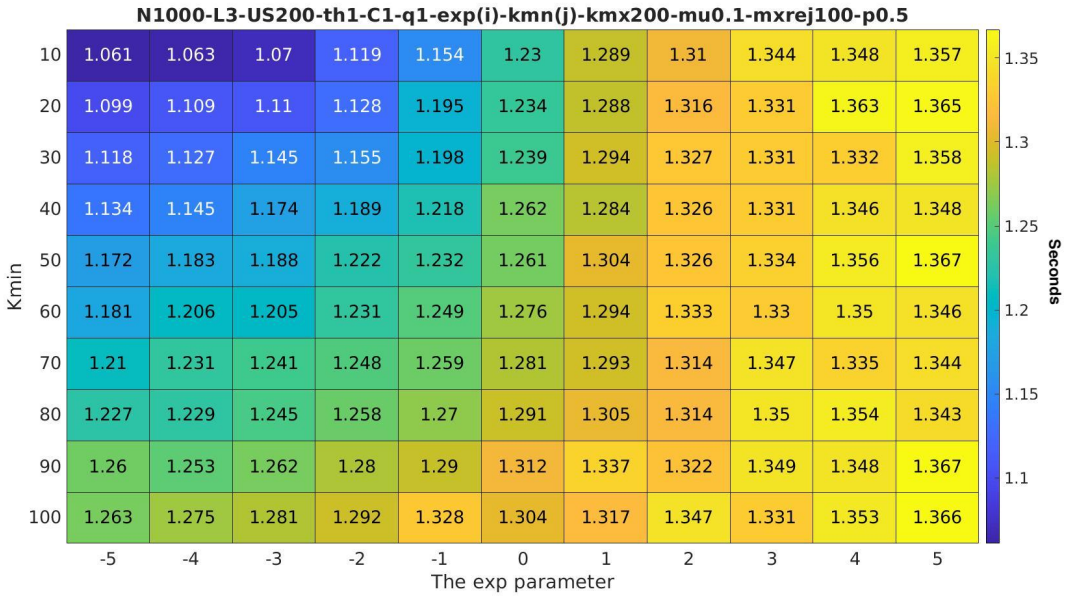

Figure 19: Computational time of the while loop when changing the parameters k-min and exp, from the power law distribution for three layers

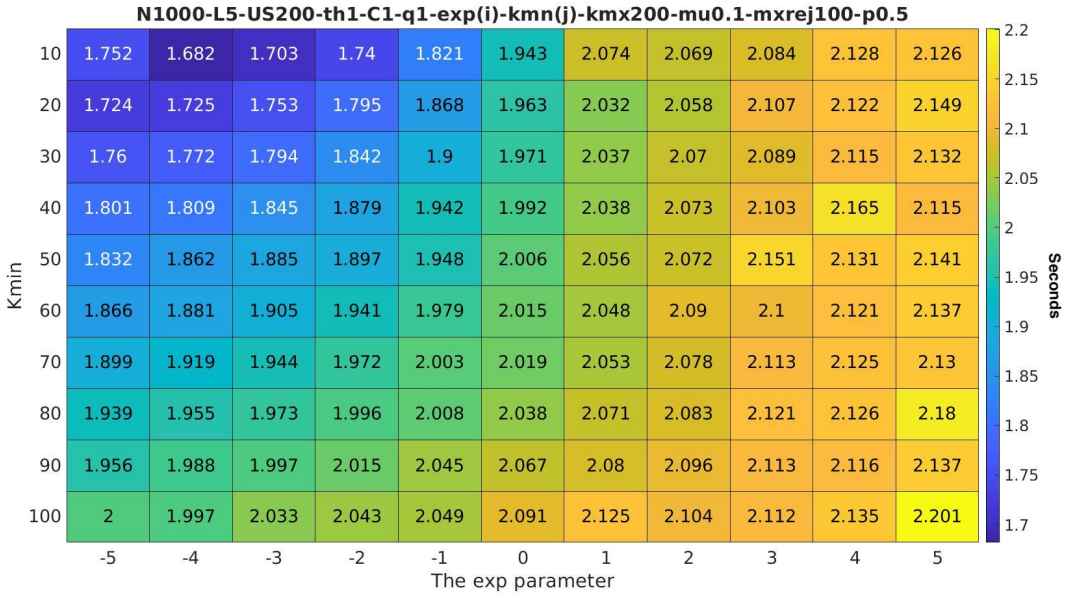

Figure 20: Computational time of the while loop when changing the parameters k-min and exp, from the power law distribution for five layers

5.2.2 The computational time of the modularity calculations

Another part of the code that also consume a lot of time. However, it does not have an asymptotic behaviour as in Figures 18-20. This part of the code computes the original modularity matrix when each node is considered its own group.

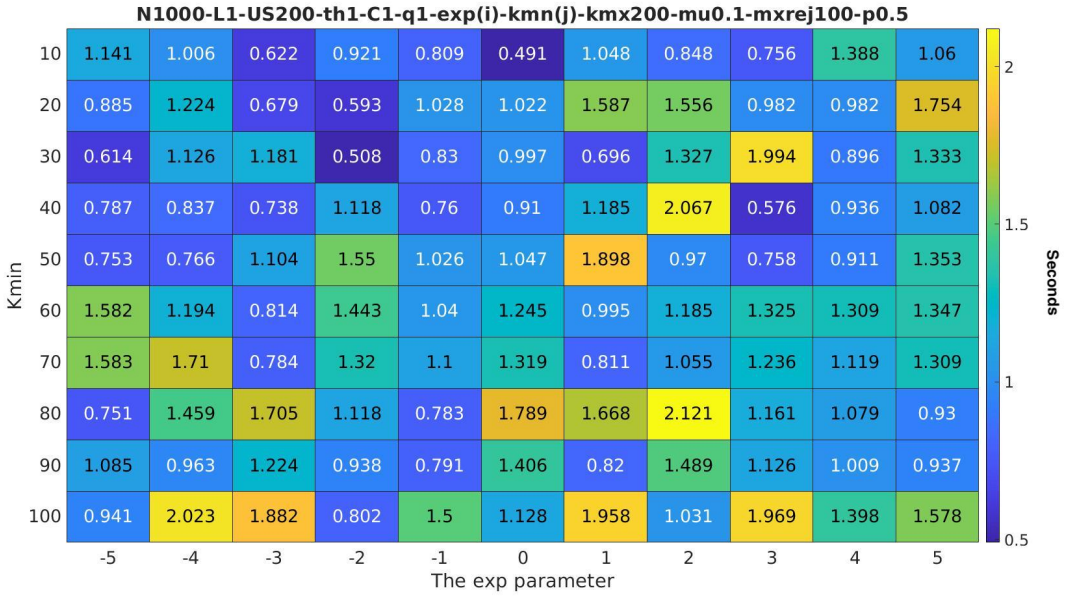

Figure 21: Computational time of the modularity calculation when changing the parameters k-min and exp, from the power law distribution for one layer

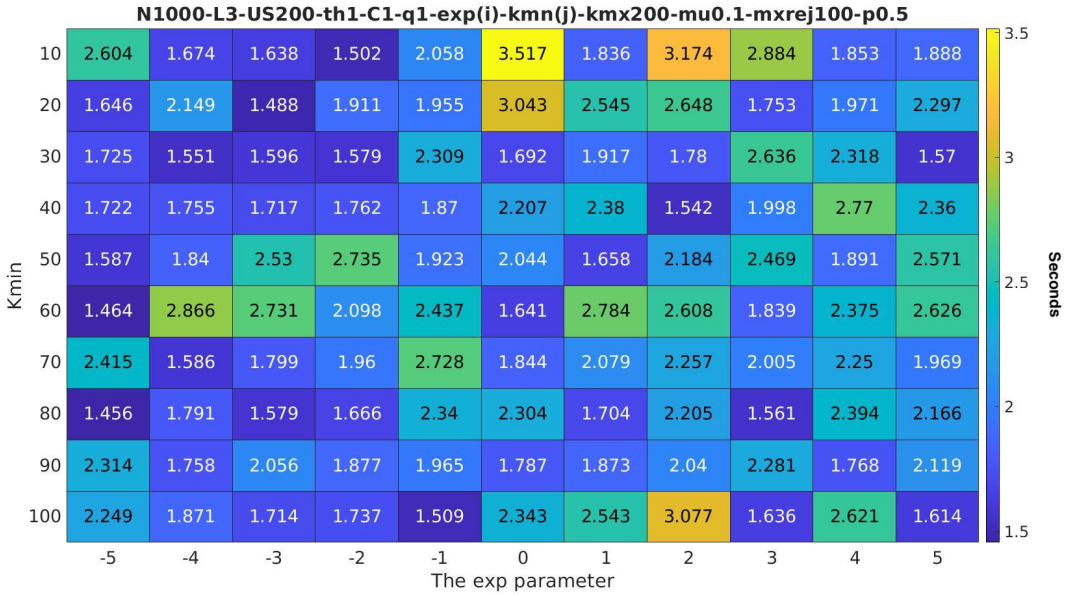

Figure 22: Computational time of the modularity calculation when changing the parameters k-min and exp, from the power law distribution for three layers

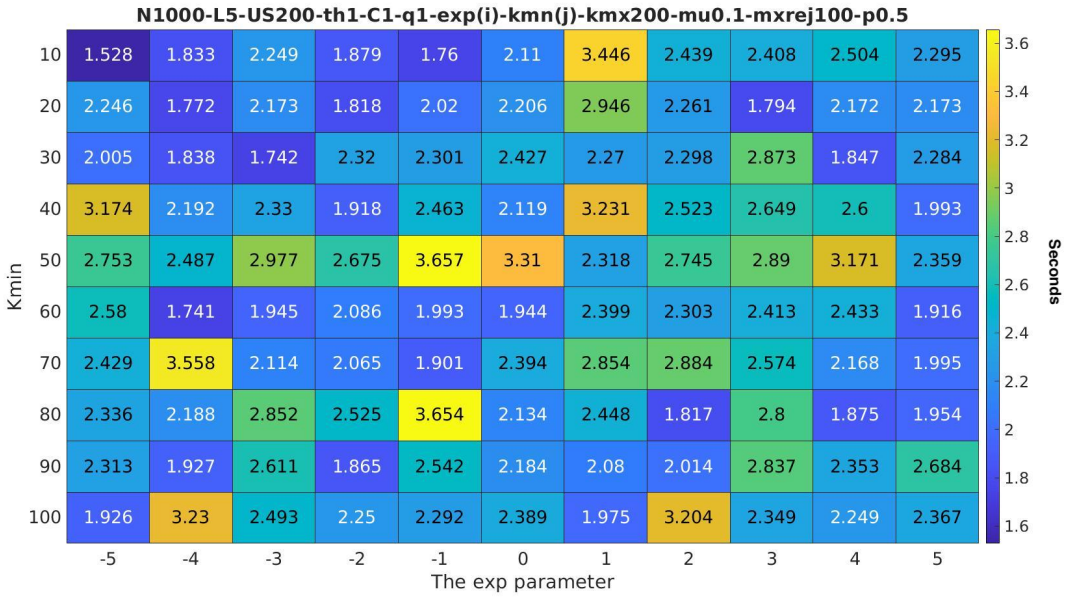

Figure 23: Computational time of the modularity calculation when changing the parameters k-min and exp, from the power law distribution for five layers

5.3 Concurrency results

The tests from Section 5.2.1 show that with increasing number of edges the computational time for the modularity (see Figures 21-23) is not influenced as much as for the loop of the update step, see Figures 18-20. Since we are looking for the asymptotic behaviour, concurrency is not to be used within the read time part of the code. In order to achieve best results, we implemented concurrency on both parts, the update step and the initial modularity calculation of the algorithm. The OpenMP implementation was useful and the speedup reached 2 i.e. the code now runs twice as fast, see Figure 24.

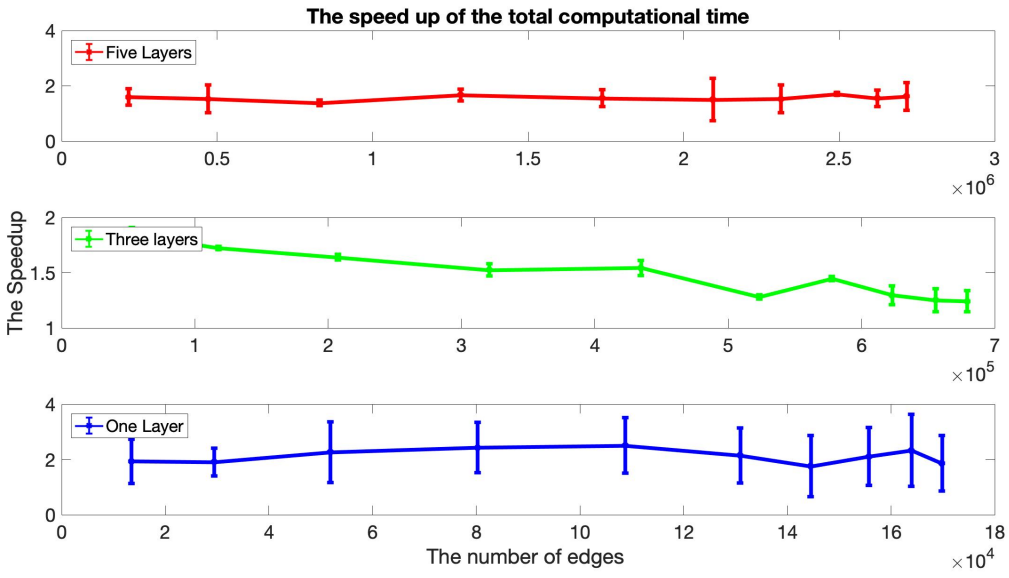

Figure 24: The speedup of the algorithm after using OpenMP for (1, 3, 5) layers

5.4 Analysis of the memory usage

During the project we generated a lot of benchmarks with different characteristics. We noticed some issues for some of these networks with large number of nodes and number of layers. The algorithm either has a large time consumption or it crashed before completing the run with the error bad_allocation. To find out what is causing the issues, we have done some analysis on the memory usage of the algorithm.

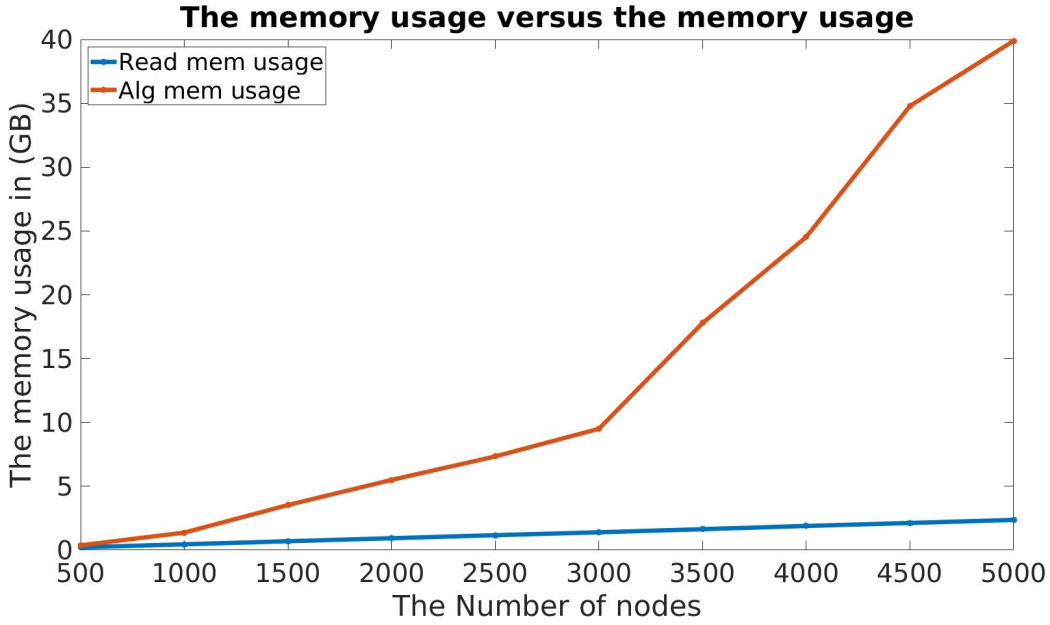

Figure 25: Total memory allocations of the Generalized Louvain Algorithm and the read function separately for three layers

The memory complexity of the read function is linear, i.e. proportional to the number of nodes, as expected. However, the memory usage of the algorithm itself is much larger, hence, the analysis will be done on the algorithms memory usage.

Read (%)	36.91	25.01	16.51	14.50	13.76	12.81	8.45	7.17	5.76	5.61
Algorithm (%)	63.09	74.99	83.49	85.50	86.24	87.19	91.55	92.83	94.24	94.39

Table 3: Memory usage percentage of the total memory usage for both the read function and the algorithm

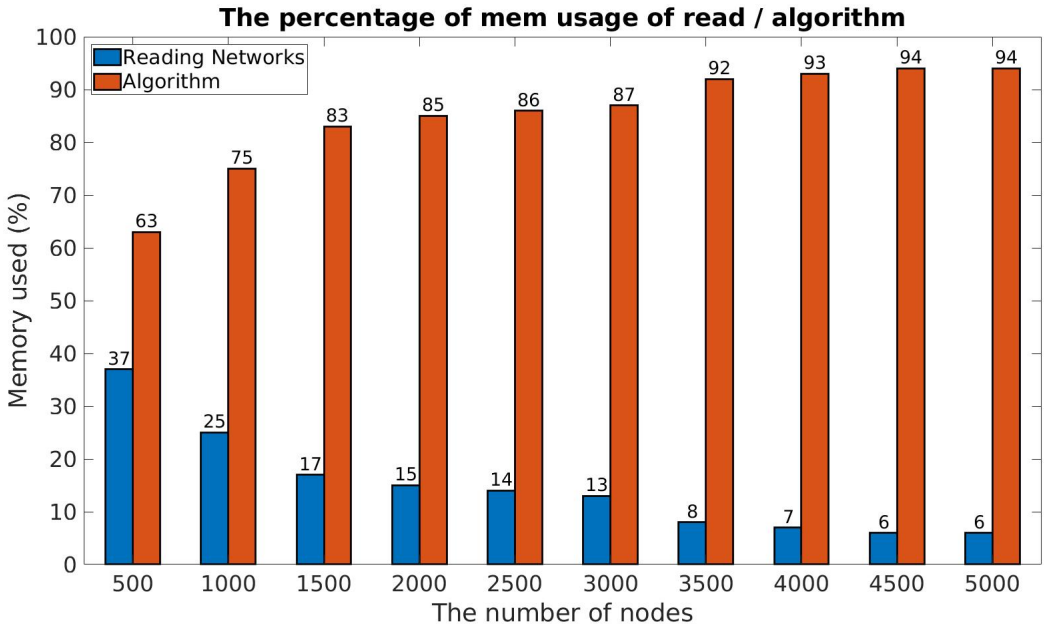

Figure 26: Amount of memory used in reading the networks and running the Generalized louvain algorithm

The algorithm is not that complex, hence the behaviour of the memory usage in Figure 25 needs to be investigated. The reason could be the way the algorithm was implemented or it could be due to the architecture of the algorithm. In the algorithm we work with large sparse matrices which take a lot of memory (in our case), this could be the reason. Some memory analysis has been done using **Valgrind –tool=massif ./program**. This creates a massif file that contain information about the memory allocations during the run. This file has been visualized by the **massif-visualizer** tool .

Due to the scarcity of tools available for memory analysis the **massif-visualizer** has been used during this project. There are tools with more functionality though they are usually commercial. Some functionality in the massif-visualizer is missing such as renaming functions and adjusting legends. Its mainly a tool useful for analysis and should not be considered a graphing tool. However, most of our useful insights into the memory usage has been obtained via the graphs in Figures 27, 28 and 29, therefore we will include them and explain what is going on.

In Figure 27 we see the memory usage and the different functions that are allocating memory. The functions that are using the most memory are displayed in red, orange and yellow in this graph. The red and yellow ones are storage functions in the Eigen library (that is used a lot in order to store matrices in the implementation). These functions are used whenever we are allocating for a matrix with the Eigen library. The orange one is a allocation of a triplet (a storage scheme for sparse matrices), essentially also Eigen storage. In the beginning of the program we see that a lot of allocations that are not using a lot of space, these are the allocations for reading the input file. However, after some time the memory increases quickly due to the computation of the modularity matrix. The modularity matrix uses a lot of memory both during computing it as well as storing itself takes a lot of memory. We see that right before the peak, the memory is skyrocketing and then being quite stable with a slight increase until it reduces again step by step. The huge spike is when the algorithm is being called, in the beginning of the algorithm one of the first things computed is the modularity matrix.

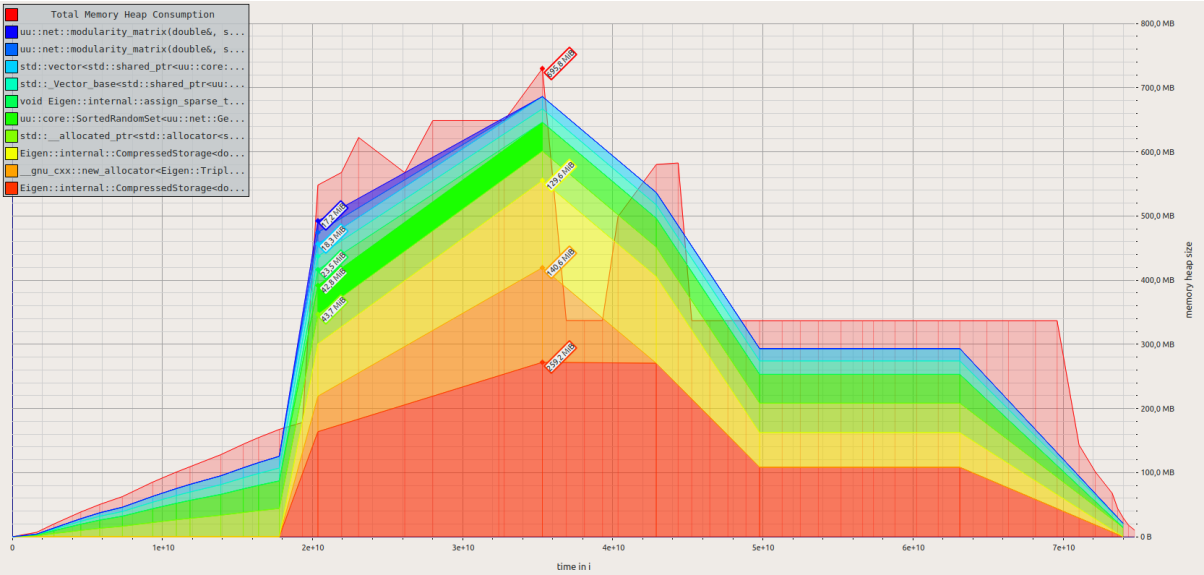

Figure 27: Different memory allocations during the whole program (reading and clustering), the example shown is for N=1500 and L=3

As one would expect the memory increases with larger data sets. In Figure 28 we see that the spike observed in Figure 27 has increased and has even more significance on the memory usage of the algorithm. In this example we see that the modularity matrix function is allocating a lot of memory, this is again memory for computing the matrix itself.

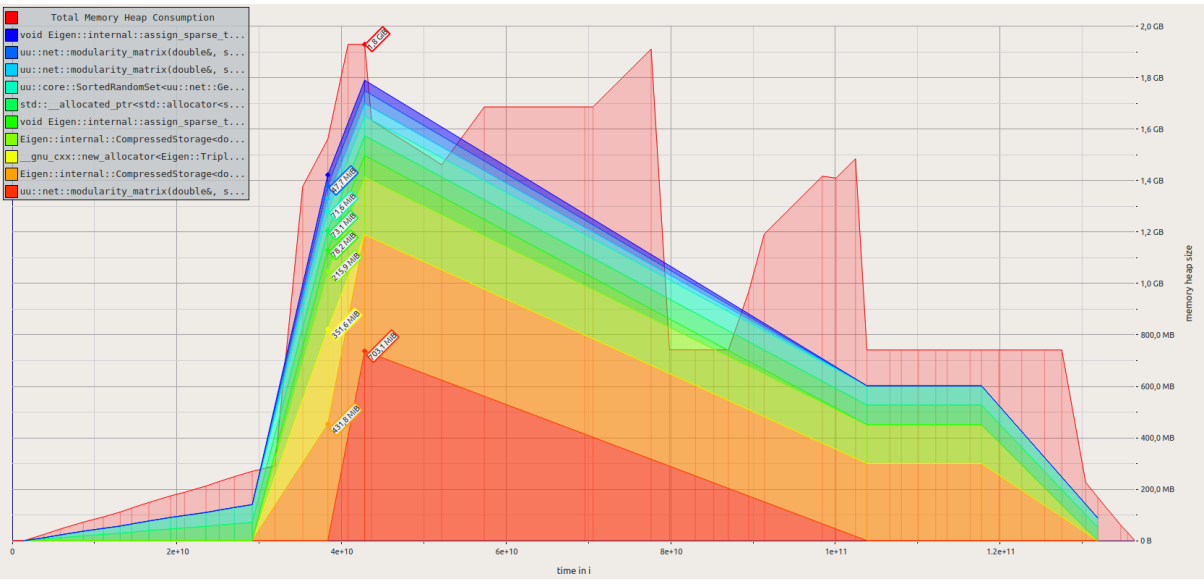

Figure 28: Different memory allocations during the whole program (reading and clustering), the example shown is for N=2500 and L=3

Finally, to show the behaviour as the number of nodes increases we have Figure 29. Here we have three main components that are making up almost all the heap. It is storage for Eigen::Triplet in blue, and Eigen::CompressedStorage in orange and red. These are both responsible for the spikes as well as most of the memory used. These are all functions called during computation of the modularity matrix.

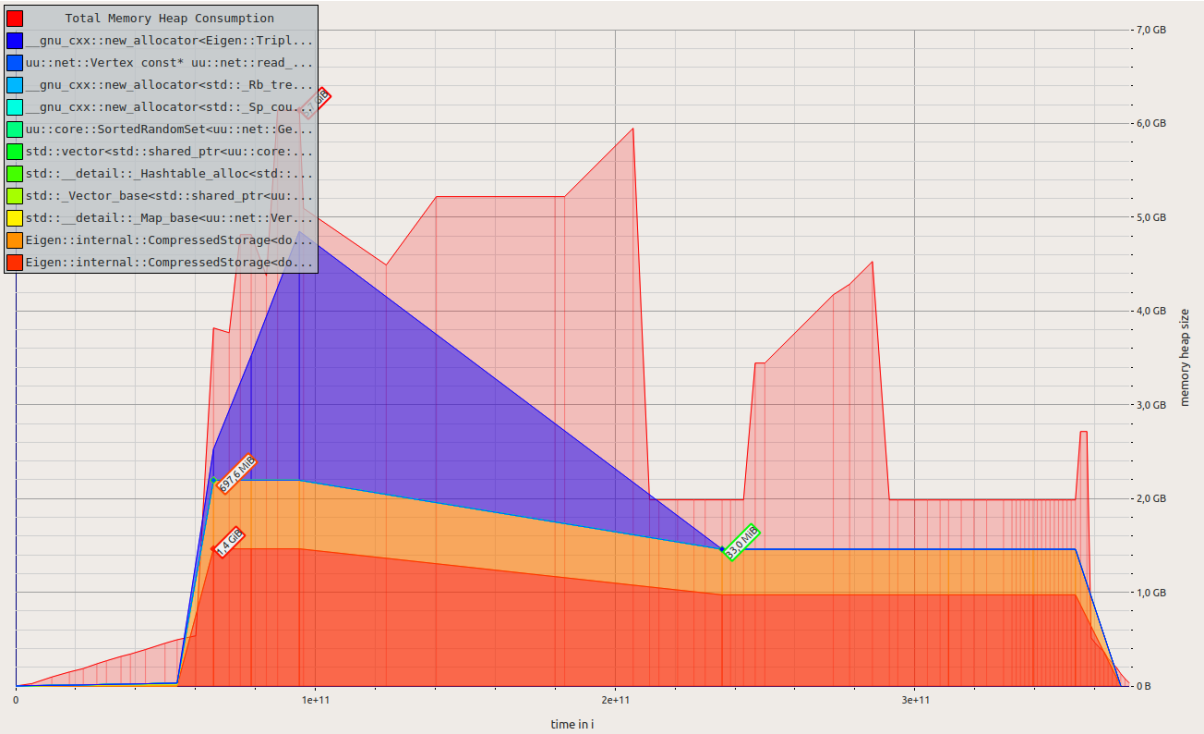

Figure 29: Different memory allocations during the whole program (reading and clustering), the example shown is for N=4500 and L=3

6 Discussion

The algorithm performs when the functionality needed is being used. On our machines there has been issues with compiling the source code and running it as intended. The programmer who has implemented the algorithm has had the foresight that memory consumption will be a problem. Due to this insight he has implemented a partitioning scheme that will partition the modularity matrix into smaller pieces and finding communities in a way that doesn't require the whole modularity matrix in one large chunk of memory. Since our objective has been to find ways to optimize the algorithm and due to time constraints we did not investigate why this functionality does not work on our computers. In part also due to the rigorous testing that CRAN already does before releasing a new version of the package. Instead we focused our efforts on the problem of optimizing the algorithm.

Through the use of OpenMP we where able to obtain a speedup of around two times the speed of the sequential run code. During the final parts of the project (which has been the memory analysis) we found paper [8] that archives a speedup of 16 for the same algorithm. Though this would have been useful, unfortunately we did not find this earlier. The project of optimizing this algorithm for the package will however continue, hence the paper will be useful for this work. We can therefore say with high certainty, that there are more optimizations that can be implemented.

The problems with the memory consumption has been quite hard to track down. We always suspected the modularity matrix, though we did not have the tools to verify it. This is mainly due to the fact that there are not a lot of great tools available for this type of analysis. The massif visualizer although not having the functionality that one is used to with the advancement of software, did help us to reach the conclusions necessary. We can say that the memory consumption is large due to the whole matrix being allocated. However, an evaluation must be made in order to know if the partitioning scheme is sufficient to solve the problem or not. What we can say is that a different implementation for this part of the algorithm will give a performance increase, since a lot of the tests ran in addition to RAM also use swap memory.

7 Conclusion

The algorithm has been optimized during this project, however some work remains to be done. As mentioned in Section 6, one needs to consider whether a implementation with memory optimization will give a performance increase or if the current solution is sufficient.

Even though the algorithm has been optimized, further optimization is possible by using concurrency. Paper [8] states that they have achieved 16 times speedup by using 32 threads.

References

[1] Wikipedia, https://en.wikipedia.org/wiki/Louvain_modularity, *"Louvain modularity"* , 11 January 2020

[3] Eijkhout, Victor, http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-basics.html, *"Parallel Programming for Science Engineering"* ,2016

[4] Matteo Magnani, Davide Vega, Mikael Dubik, https://cran.r-project.org/web/packages/multinet/multinet.pdf, *"Analysis and Mining of Multilayer Social Networks"*, 1 January 2020

[5] Peter J. Mucha , Thomas Richardson, Kevin Macon, Mason A. Porter, and Jukka-Pekka Onnela, https://arxiv.org/pdf/0911.1824.pdf, *"Community Structure in Time-Dependent, Multiscale, and Multiplex Networks"*, 12 Jul 2010

[6] Marya Bazzi, Lucas G. S. Jeub, Alex Arenas, Sam D. Howison, and Mason A. Porter, https://arxiv.org/pdf/1608.06196.pdf, *"A framework for the construction of generative models for mesoscale structure in multilayer networks"*, 11 Dec 2019

[7] Lucas G. S. Jeub and Marya Bazzi, https://github.com/MultilayerGM/MultilayerGM-MATLAB, *"A generative model for mesoscale structure in multilayer networks implemented in MATLAB"*, 2016-2019

[8] Hao Lu, Mahantesh Halappanavar, Ananth Kalyanaraman, https://arxiv.org/pdf/1410.1237.pdf, *Parallel Heuristics for Scalable Community Detection*, 7 oktober 2014

[9] https://www.kernix.com/article/community-detection-in-social-networks/, *Community detection in social networks*, 7 November 2016