A PARALLEL IMPLEMENTATION OF THE SYNCHRONISED LOUVAIN METHOD

Benjamin Chiêm¹ & Andine Havelange¹ & Paul Van Dooren²

February 16, 2017

ABSTRACT

Community detection in networks is a very actual and important field of research with applications in many areas. But, given that the amount of processed data increases more and more, existing algorithms need to be adapted for very large graphs.

The objective of this project was to parallelise the Synchronised Louvain Method, a community detection algorithm developed by A. Browet, in order to improve its performances in terms of computation time and thus be able to faster detect communities in very large graphs.

To reach this goal, we used the API OpenMP to parallelise the algorithm and then carried out performance tests.

We studied the computation time and speedup of the parallelised algorithm and were able to bring out some qualitative trends. We obtained a great speedup, compared with the theoretical prediction of Amdahl’s law.

To conclude, using the parallel implementation of Browet’s algorithm on large graphs seems to give good results, both in terms of computation time and speedup. Further tests should be carried out in order to obtain more quantitative results.

Acknowledgement

We would like to sincerely thank Etienne Huens for his technical help and his pieces of advice about all coding aspects, and Arnaud Browet for his explanations about his innovative algorithm.

¹ Ecole Polytechnique de Louvain, Université Catholique de Louvain, Louvain-la-Neuve, Belgium
² (Supervisor) Department of Mathematical Engineering, Ecole Polytechnique de Louvain, Université Catholique de Louvain, Louvain-la-Neuve, Belgium
CONTENTS

1 Introduction 3
2 General notions 3
  2.1 Definition of communities ........................................... 3
  2.2 Energy of a partition .............................................. 4
  2.3 Null model ................................................... 5
  2.4 Modularity as fitness measure ................................... 5
3 The algorithm 6
  3.1 ASSIGN (best neighbour) .......................................... 7
  3.2 COMPONENTS (connected components) .............................. 7
  3.3 POSITIVE (split) ............................................... 8
  3.4 MAXIMAL (merge) ........................................... 8
  3.5 AGGREGATE (aggregation of communities ) ....................... 9
4 Parallel implementation 9
  4.1 What is parallel computing? ................................... 9
  4.2 Parallel implementation of Browet’s algorithm .................. 10
  4.3 OpenMP .................................................. 11
5 Results 13
  5.1 Benchmark graphs ............................................ 13
  5.2 Computation time ............................................ 14
  5.3 Speedup ................................................ 15
6 Conclusion 18
A Appendix 19
  A.1 Pseudocodes of main steps of Browet’s algorithm ............... 19

LIST OF FIGURES

Figure 1 Communication communities in Belgium .......................... 3
Figure 2 Configuration Null Model ....................................... 6
Figure 3 Assignment and extraction of connected components steps. ........ 7
Figure 4 Positive correction in a branch. ................................ 8
Figure 5 Positive correction in the SCC. .................................. 8
Figure 6 Maximal correction in a branch. ................................ 9
Figure 7 Maximal correction in the SCC. .................................. 9
Figure 8 Aggregation step. ........................................... 9
Figure 9 Sequential computing. ....................................... 10
Figure 10 Parallel computing. ....................................... 10
Figure 11 Hierarchical organisation of the code to be parallelised (call-graph) .................. 11
Figure 12 Shared memory. ........................................... 12
Figure 13 Fork-join model. .......................................... 12
Figure 14 Influence of mixing parameters. ............................... 13
Figure 15 Computation time versus number of nodes .................... 14
Figure 16 Amdahl’s law. ............................................ 15
Figure 17 Theoretical (for P = 95%) and empirical speedup. .......... 16
Figure 18 Speedup versus number of nodes ................................ 17

LIST OF TABLES

Table 1 Used parameters for the generation of benchmark graphs. ........ 13
INTRODUCTION

Since the last decade, the study of large networks has known a big boost of interest. Graph theory provides interesting new insights for many studies. This is why we need new algorithms to analyse the structure of these networks.

In particular, an interesting question to be raised is: does a certain network contain some kind of communities? This is precisely the question that the Synchronised Louvain method aims to answer. Developed by Arnaud Browet, this algorithm is based on the well-known Louvain method. The goal of our project is to implement a parallel version of Browet’s algorithm in order to improve its performances in terms of computation time and therefore, to be able to quickly detect communities in very large networks.

Accelerating community detection algorithms could be of great interest in the domain of telecommunications (Figure 1) for example, but also in others. In fact, many new studies are trying to apply such tools provided by graph theory to various domains such as neuroscience, social or even gene regulatory networks. The amount of data to process in these cases is quite huge; it is thus well justified to search for the fastest algorithm.

In this document, we will briefly explain some general notions about communities and the different steps of the Synchronised Louvain method, then we will present what is parallel computing. After that, we will detail the way we chose to implement our parallel version and what tools we used. Finally, we will develop our results and give some explanations about them.

![Figure 1: Communication communities in Belgium. (Source: Guigourès, R., Boullé, M., Rossi, F. (s.d.). Segmentation géographique par étude d’un journal d’appels téléphoniques. http://apiacoa.org/publications/2011/guigouresbouleletal2011segmentation-geographique.pdf)]](http://apiacoa.org/publications/2011/guigouresbouleletal2011segmentation-geographique.pdf)

2 GENERAL NOTIONS

2.1 Definition of communities

There is no unique definition of communities. They can informally be defined as sets of nodes with a high internal density (which is defined as the ratio between the actual number of edges in this set and the maximal number of possible edges) and a low external density with the rest of the nodes not being in the community. In other words, a community is a set of nodes who are strongly interconnected and weakly connected with the rest of the network. A typical example is the study of the belgian communication network (see Figure 1)
in which communities can be detected, some of them corresponding to provinces of Belgium.

This definition is unfortunately too informal to allow us to compute what would be an optimal partition of nodes into communities in a given graph. Most of the time, people use functions that measure the quality of a partition of the nodes into communities, called fitness measures or quality functions. The goal is then to find a community structure maximising the quality function.

There exist various quality functions and some of them are based on the concept of energy of a partition. It has been developed by Reichardt and Bornholdt and will be presented in the following subsection.

2.2 Energy of a partition

Reichardt and Bornholdt interpret the community detection problem as finding the ground state of a spin glass, that is a disordered magnet. A spin state $\sigma_i$ is assigned to each node of the graph. This spin state represents the community assignment of the node. So, if the nodes are partitioned into $c$ communities, $\sigma_i \in \{1, 2, ..., c\}$.

In an optimal partition of nodes into communities, edges should ideally connect vertices in the same spin state, i.e. belonging to the same community. And there should be as little as possible edges linking nodes in different spin states. This observation leads to the definition of an energy function rewarding existing edges between vertices in the same spin state and penalising other existing edges.

If the adjacency matrix of the graph is denoted by $A_i$, for any existing edge, $A(i, j) \neq 0$, the partition is

- rewarded by $a_{ij} > 0$ if the nodes $i$ and $j$ are in the same spin state, i.e. $\sigma_i = \sigma_j$
- penalised by $c_{ij} > 0$ if the nodes are in different spin states $\sigma_i \neq \sigma_j$

Non-existing edges must also be taken into account in a similar way. More precisely, for any non-existing edge in the graph, $A(i, j) = 0$, the partition is

- penalised by $b_{ij} > 0$ if the nodes $i$ and $j$ are in the same spin state, i.e. $\sigma_i = \sigma_j$
- rewarded by $d_{ij} > 0$ if the nodes are in different spin states $\sigma_i \neq \sigma_j$

Denoting the Kronecker delta by $\delta(\sigma_i, \sigma_j)$, i.e.

\[
\delta(\sigma_i, \sigma_j) = \begin{cases} 
1 & \text{if } \sigma_i = \sigma_j \\
0 & \text{if } \sigma_i \neq \sigma_j
\end{cases}
\]

the energy of a partition $\sigma$ can be written as (denoting the set of edges of the graph by $V$)

\[
H(\sigma) = -\sum_{i,j \in V} \left[ a_{ij} A(i,j) - b_{ij}(1 - A(i,j)) \right] \delta(\sigma_i, \sigma_j) - \left[ c_{ij} A(i,j) - d_{ij}(1 - A(i,j)) \right] (1 - \delta(\sigma_i, \sigma_j))
\]  

(1)

where the minus sign preceding the summation symbol is a convention such that the optimal partition is characterised by a spin state of minimal energy.

The terms of (1) can be rearranged such that

\[
H(\sigma) = -H_0 - \sum_{i,j \in V} \left[ a_{ij} A(i,j) - \beta_{ij} \right] \delta(\sigma_i, \sigma_j)
\]  

(2)

where $H_0 = \sum_{i,j \in V} \left[ -(c_{ij} + d_{ij}) A(i,j) + d_{ij} \right]$ is independent of the partition and can thus be removed from the fitness measure of the partition.

The values of the parameters

\[
\alpha_{ij} = a_{ij} + b_{ij} + c_{ij} + d_{ij} \\
\beta_{ij} = b_{ij} + d_{ij}
\]

depend on the null model the graph is compared to in order to compute the energy of the partition. The concept of null model is explained in what follows.
2.3 Null model

A crucial question to be raised when dealing with community detection is how to measure the quality of the partition obtained by some algorithm. Indeed, we do not know the \textit{a priori} community structure of most real networks and thus, we can not verify our results. This issue lead to the concept of null model.

A null model is a graph that shares some features (to be chosen) with the original graph, but whose other features are essentially randomly determined. That randomness should avoid any community structure; by comparing the original graph with the null model, we can then prove the community structure of the first.

A major concern about null models is the selection of the features to be preserved. For instance, Reichardt and Bornholdt chose an Erdös-Rényi graph as null model. In this null model, the number of nodes \( N \) and the number of edges \( m \) remain the same as in the original graph, but each edge is randomly (with constant probability) chosen to link two nodes. In other words, this null model is randomly taken in the set of all graphs which have \( N \) nodes and \( m \) edges. More formally, the probability that two nodes \( i \) and \( j \) are rewired, \( p_{ij} \), is chosen in the Erdös-Rényi null model as

\[
p_{ij} = p
\]

with \( p \) constant.

Many other null models were designed either for generic or specific applications since the choice of a suitable null model can highly influence the performance measurements of an algorithm. In fact, two major drawbacks of the Erdös-Rényi model are known. On the one hand, it is hard to be generalised in the case of weighted graphs. On the other hand Erdös-Rényi graphs are known to exhibit a Poissonian degree distribution whereas real networks often have a degree distribution that follows a fat-tail power law.

Following this observation, Newman and Girvan introduced the so-called \textit{configuration null model} which is the basis of the modularity measure. The shared feature in the configuration null model is the degree distribution. To build this model from the original graph (in the directed case, for the sake of generality), the following steps must be followed:

- Build the edge list from the original graph. Let us define the vectors \( i^{\text{out}} \) whose entries are the labels of the nodes having an outgoing edge (one entry for each edge), and \( j^{\text{in}} \) whose entries are the labels of the corresponding nodes, i.e. the ones that have an incoming edge.
- Proceed to a random shuffle on both \( i^{\text{out}} \) and \( j^{\text{in}} \) vectors.
- The nodes of the random graph are rewired according to the shuffled edge list.

These steps are depicted in Figure (2). Intuitively, we can notice that this procedure does not change the degree distribution of the graph, as we wanted.

Based on this model, Newman and Girvan have proposed one of the most famous fitness measure for community detection: modularity.

2.4 Modularity as fitness measure

As said just above, the modularity is a fitness measure based on the configuration model producing a random graph with the same degree distribution as the original network for which an optimal partition of the nodes into communities has to be found.

Given a partition \( \sigma \), the modularity cost function compares the actual edge density within each community to the expected edge density in a random network using the configuration null model. It is given by

\[
Q(\sigma) = \sum_{i,j \in V} \left[ \frac{A(i,j)}{m} - \frac{k_i^{\text{out}}}{m} \frac{k_j^{\text{in}}}{m} \right] \delta(\sigma_i, \sigma_j)
\]

where \( k_i^{\text{out}} \) (resp. \( k_j^{\text{in}} \)) is the number of outgoing (resp. incoming) edges of node \( i \) (resp. \( j \)).

Up to scaling \( \frac{1}{m} \), that does not modify the optimal partition, (4) can be expressed in the framework of the
energy function, given by \( (2) \), with \( \alpha_{ij} = 1 \) and \( \beta_{ij} = p_{ij} \), i.e. the expected number of edge from \( i \) to \( j \).

The modularity can easily be extended to the case of weighted networks:

\[
Q_w(\sigma) = \frac{1}{m_w} \sum_{i,j \in V} \left[ W(i,j) - s^{(\text{out})}_i s^{(\text{in})}_j \right] \delta(\sigma_i, \sigma_j)
\]

where \( W \) is the weighted adjacency matrix of the input graph and \( s^{\text{out}}_i \) and \( s^{\text{in}}_i \) are respectively the outgoing and incoming strengths of nodes \( i \). They are equal to the sum of the weights of the outgoing (respectively incoming) edges incidents to node \( i \). (These notions are similar to those of outgoing and incoming degrees in unweighted graphs). The total weight of the input graph, \( m_w \), is

\[
m_w = \sum_{i,j \in V} W(i,j) = \sum_{i \in V} s^{(\text{out})}_i = \sum_{j \in V} s^{(\text{in})}_j
\]

The modularity score of a given partition belongs to the interval \([-1, 1]\). It is equal to 0 when the partition contains only one community and is in general negative for a partition with \( n \) communities, that is if every node defines a community. So, a partition with a high modularity score is supposed to be an accurate representation of the optimal partition of nodes into communities.

It can indeed be shown that finding a community structure maximising the modularity function (and, more generally quality functions based on the energy of partitions) is a NP-hard problem. Hence, most of the community detection algorithms are greedy algorithms providing a good approximation of the optimal solution.

Modularity cost function is one of the most popular fitness measures and one of its biggest advantages is not to require to know beforehand the number of communities to extract. That is the quality function that is used in this report to determine a good partition of nodes into communities. But other fitness measures could be chosen too, in order to compensate some drawbacks related to the use of the modularity, which will not be discussed here.

## 3 The Algorithm

Browet’s community detection algorithm, also known as the Synchronised Louvain Method, is summarised below by its description using Algorithm 1.

In this report, we focus on the most important parts of the algorithm for the project, i.e. for the parallel implementation of Browet’s algorithm. But it is described in more details in Arnaud Browet’s thesis [1].

We will explain in details the important parts of the algorithm in the following sections. The pseudocodes of these main steps are shown in the appendix (see A.1).
Algorithm 1 Synchronised Louvain Method

\textbf{Input}: a graph \( G(V, E) \)
\textbf{Output}: a community partition matrix \( C \in \mathbb{R}^{k \times n} \)

Initialize \( C = I_n \), \( C_t = 0 \), \( G_t = G \)
\( \triangleright \) \( C \) final partition, \( C_t \) partition at level \( t \) for \( G_t \)

\begin{algorithmic}
\WHILE {\( C_t \neq I \)}
\STATE \( C_t \leftarrow \text{ASSIGN}(G_t) \)
\STATE \( C_t \leftarrow \text{POSITIVE}(C_t, G_t) \)
\WHILE {\( \exists i \in V_t, c \in C_t \text{ with } \Delta H(c_i \rightarrow i \rightarrow c) > 0 \)}
\STATE \( C_t \leftarrow \text{MAXIMAL}(C_t, G_t) \)
\STATE \( C_T \leftarrow \text{POSITIVE}(C_t, G_t) \)
\ENDWHILE
\STATE \( G_t \leftarrow \text{AGGREGATE}(G_t) \)
\STATE \( C = C_t \)
\ENDWHILE
\end{algorithmic}

3.1 ASSIGN (best neighbour)

The first step of the algorithm consists in initialising the community structure by assigning each node of the graph to its best neighbour.

In order to do so, the algorithm computes for each of the neighbours of the current node the gain in modularity obtained by assigning the current node \( i \) to the community formed by this neighbour \( j \) (i.e. \( \Delta H(i \rightarrow \{j\}) \)). The best neighbour of the node is the one for which the modularity gain is the highest. (If all the neighbours of a node are such that this gain is negative, the node is self-assigned, in which case the gain is equal to 0).

At the end of the assignment step, we obtain a set of directed subgraphs spanning the input graph, which is called the assignment graph, with one directed edge going out from each node to its best neighbour, as illustrated in Figure 3.

![Figure 3: Assignation and extraction of connected components steps.](image)

The complexity of the assignment step is \( \mathcal{O}(m) \), where \( m \) is the number of edges in the input graph.

3.2 COMPONENTS (connected components)

The next step is to extract the connected components of the assignment graph, which are defined as the communities. For example, the assignment graph in Figure 3 consists of 3 communities. As illustrated on the same figure, each community contains one and only one directed cycle, the strongly connected component (SCC).

Given that the assignment of each node to its best neighbour is independent from the assignation of the other nodes, the assignation step might result in communities in which the presence of some nodes decreases...
the value of the cost function.
In the following steps, two types of corrections will thus be applied to the assignment graph in order to improve the community partition, by updating some node assignments.

3.3 **POSITIVE (split)**

The positive correction ensures that each node has a positive contribution to its community in terms of the cost function.
The local gain of a node is defined as the modularity gain to assign the node to its current community. For each community containing at least one node with a negative local gain, the algorithm searches for the optimal (i.e. providing the greatest gain of modularity) bisection (split) of these communities.

A positive correction is made by updating at most two node assignments. Indeed, two kinds of bisections can be considered. Either we remove the assignment of a node in a branch of the community (as shown on Figure 4) or two assignments within the SCC (as illustrated in Figure 5). The nodes whose assignments are removed are self-assigned.

![Figure 4: Positive correction in a branch. (Source: Browet, A. (2014) Algorithms for community and role detection in networks. Ph.D. thesis in Engineering Sciences, Ecole Polytechnique de Louvain, Université Catholique de Louvain, Louvain-la-Neuve.)](image)

![Figure 5: Positive correction in the SCC. (Source: Browet, A. (2014) Algorithms for community and role detection in networks. Ph.D. thesis in Engineering Sciences, Ecole Polytechnique de Louvain, Université Catholique de Louvain, Louvain-la-Neuve.)](image)

Positive corrections are applied till there is no node with a negative local gain anymore.

The complexity of the Positive Correction step is dominated by a linear complexity in the number of nodes within the branches of a community.

3.4 **MAXIMAL (merge)**

Once all possible positive corrections have been made, maximal corrections are applied to the assignment graph by allowing some nodes to switch from one community to another in order to further optimise the cost function and thus the community partition.

While positive corrections induce an increase in the number of communities by splitting them, the maximal corrections tend to decrease this number by merging some communities (we will come back on this later on).
One should notice that, when a node is switching from a community to another one, all the nodes in its tail (the set of nodes such that there exists a directed path leading from them to the node in the assignment graph) are also switching to its new community, as illustrated in Figure 6.
In order to decide to which community a node will be assigned, the algorithm computes the gains (given the current partition) obtained by assigning the node and the nodes of its tail to each of the communities in which the node has a neighbour in the input graph (thus not in the assignment graph). Then, the node and its tail are assigned to the community providing the largest non negative gain (it can be its current community). To ensure the convergence of the algorithm, each maximal correction is accepted with a certain probability (see [1] for more details).

A maximal correction results in a merge of two communities if the node that is switched to another community is part of the SCC (as shown on Figure 7) because its tail is the entire community. But this is not the case if the node belongs to a branch of the community (see Figure 6).
The complexity of the Maximal Corrections step is $O(m)$.

Positive and maximal corrections are alternatively applied till convergence is reached.

3.5 AGGREGATE (aggregation of communities)

When no correction providing a strictly positive gain can be done, the community graph is collapsed. All the nodes belonging to the same community are aggregated into one “super node”. Then the same procedure (Assign, Components, Positive, Maximal, Positive, ..., Aggregate) is applied to the aggregated graph, which provides another hierarchical level of clustering. This is illustrated in Figure 8. The algorithm stops when it can not extract any community structure in the last aggregated graph, i.e. when the assignment of any "super node" to any other "super node" results in a negative gain.

The total complexity of the algorithm can not be computed because it depends on the number of positive and maximal corrections, which can not be evaluated.

4 PARALLEL IMPLEMENTATION

4.1 What is parallel computing?

As explained in [2], most of the programs we use today are written to be sequentially executed. This means that the program is broken into a discrete series of instructions who are executed one after another, on a single processor, as illustrated in Figure 9.

Parallel computing takes advantage of the increasing number of processor cores in computers. The main idea of parallel computing is to distribute the work to be done by a program between several cores. It follows in a significant computing time saving.

The problem is broken into discrete pieces of work that can be solved independently, each part being handled by one core. A part consists in a sequence of instructions that are solved one after another by a same core. An overall control mechanism is employed to divide the work and to ensure the coordination and synchronisation between the cores, as shown in Figure 10.
The results of the program when executed in parallel must be the same that when sequentially executed.

Not every algorithm can be parallelised: indeed, it has to be written such that the instructions can be executed independently. Let us consider algorithms 2 and 3. Both are computing the same thing: the index of the elements in the vector \( \text{count} \). But they cannot be both parallelised. Indeed, the for-loop in algorithm 2 can be parallelised, i.e. that, if we have \( n \) cores, \( n \) iterations of the loop can be executed simultaneously, one by each core, because each iteration is independent from the other ones. But this is not the case of the loop in algorithm 3 because each iteration depends on the previous one: you cannot compute \( \text{count}(3) \) if the value of \( \text{count}(2) \) has not been computed yet given that there is a temporal dependence between the iterations.

### Algorithm 2 Parallelisable Program

```plaintext
count = zeros(1,100)
for i = 1..100 do
    count[i] = i
end for
```

### Algorithm 3 Non Parallelisable Program

```plaintext
count = zeros(1,100)
count(1) = 1
for i = 2..100 do
    count[i] = count(i-1) + 1
end for
```

#### 4.2 Parallel implementation of Browet's algorithm

Let us now see what parts of the Synchronised Louvain Method are parallelisable, without modifying Browet's code. Among the four main steps detailed in section 3, three of them can be parallelised: the assignation step and the positive and maximal corrections.

We can see on Figure 11, obtained by code profiling, that the highest level we can parallelise contains four functions: `find_assignment()`, `extract_components()`, `positive_correction()` and `maximal_correction()` (the first four levels are not). We implemented a parallel version for all of them, except for the `extract_components` step. Indeed, this step was not directly parallelisable. However, the execution time spent in this function is negligible compared to the other functions. In fact, this step would have required a certain amount of computation time, but in this case not. Given that the complexity of this step is growing with the number of edges in the graph and that, after the assignation step, this number of edges is drastically reduced (\( m = n \)), the computation time is really small. Therefore, we did not to implement a parallel version of `extract_components()`.

The **assignation** step of the algorithm can be run in parallel given that the initial assignation of each node is independent of the one of the other nodes. The way the best neighbour of a node is chosen is such that the algorithm only needs the input graph to compute it: the best neighbour of a node does not influence the "computation" of the best neighbour of another node. This allows us to use a parallel processor architecture: each core can handle a set of nodes independently.

Assigning each node to another node allows the corrections to be made in parallel (synchronously). Indeed, if a node has to be switched from a community to another one, we simply have to change its assignation to a node of its new community. This can be done independently of the assignation of the other nodes in the
network (this reassignment does not modify the assignment of the other nodes). The correction on any node is thus independent from the corrections on other nodes.

The positive correction of a community does not depend on the state of the other communities so this step is parallelisable: each split can be assigned to a different core. Indeed, the computation of the local gain of a node only depends on the structure of the community it belongs to and the search for an optimal bisection of a community is only dependent on the structure of this community.

To apply a maximal correction, the algorithm does not use any other information than the current community distribution in order to compute the switching gains and the community structure is updated after all the new assignments have been computed. Here again, communities can be "corrected" simultaneously, each core handling a community.

4.3 OpenMP

In order to parallelise the Synchronised Louvain Method (which was coded in C++), we used OpenMP [3], which is an Application Program Interface (API) that may be used to direct multi-threaded, shared memory parallelism. In the case of shared memory parallelism, all cores share the same memory (see Figure 12), contrary to distributed memory parallel systems where each core has its own memory.

OpenMP uses the fork-join model (shown in Figure 13) of parallel computing. The program begins as a single process: the master thread. The instructions are executed sequentially by the master thread till it encounters a parallel region. At this point, it creates parallel threads (fork), which exist only in the parallel region. Then, the instructions in the parallel region are simultaneously executed by the parallel threads. When all the statements in the parallel region have been executed, they synchronise, terminate and only the master thread does not cease to exist (join) till the next parallel region. All the threads must have finished before the following sequential region is executed by the master thread.
Given that all the threads share the same memory, they all can read and write the data simultaneously and all changes made are visible for all threads. This may raise problems in some cases when different threads try to modify the same variable simultaneously. This is why an important task in the parallelisation of Browet’s algorithm has been to determine which variables (declared before the parallel region but used inside it) were public (all threads have access to them) and which are private (each thread has a copy of this variable and is the only one to have access to it).

If we take a closer look at algorithm 4 (which computes the index of the elements in the vector \( \text{count} \)), one can see that the variable \( \text{iterations} \) has to be public because it counts the total number of iterations of the for-loop. But the variable \( \text{position} \) has to be private. This can be illustrated by this simple example. If a thread is handling the iteration \( (i=2) \), \( \text{position} \) and thus \( \text{count}(2) \) are equal to 2. If another thread is handling iteration \( (i=4) \), \( \text{position} \) and thus \( \text{count}(4) \) are equal to 4. If both threads are executing these operations at the same time, it could be possible that \( \text{count}(2) = 4 \) or that \( \text{count}(4) = 2 \), which is not correct given that the results of the sequential and of the parallel executions of the program are not the same anymore.

**Algorithm 4**

\[
\begin{align*}
\text{count} &= \text{zeros}(1,100) \\
\text{iterations} &= 0 \\
\text{position} &= \text{for } i = 1...100 \text{ do} \\
& \quad \text{position} = i \\
& \quad \text{count}(i) = \text{position} \\
& \quad \text{iterations} += + \\
\text{end for}
\end{align*}
\]

A region is parallelised by inserting directives in the code, for instance

```
#pragma omp parallel for private(position) schedule(dynamic)
[... for loop ...]
```

In this example, the "for" means that the parallel region is a for loop and the "private(position)" that the variable \( \text{position} \) is private. The "schedule" clause is used to define how the iterations of the loop are divided among the threads. If one specifies that they have to be divided on a static way, each thread will execute approximately \( \frac{n}{t} \) (where \( n \) stands for the number of iterations and \( t \) for the number of threads) iterations of the for loop. This is not optimal when all the iterations do not take more or less the same time to be executed. If this is the case, it is a better idea to use the option "dynamic" that ensures that the work is distributed more evenly between the threads (when a thread has finished to execute an iteration, another one is dynamically assigned to it).

Another important task in the parallel implementation of Browet’s algorithm has been to determine if, within a parallel region, a certain region was critical. A critical region is a sequence of instructions that has to be executed by one thread at once. The corresponding directive is

```
#pragma omp critical
[... critical region ...]
```
To parallelise the Synchronised Louvain Method, we added omp directives in the code of Arnaud Browet (in the functions ASSIGN, POSITIVE and MAXIMAL). In order to know where to add them, we had to understand the above mentioned functions, try to see if there were temporal dependencies, if the variables had to be private or public, if some parts of the parallel regions had to be critical, and so on.

5 RESULTS

5.1 Benchmark graphs

To test the performances of the parallelised algorithm, we generated benchmark graphs using the so-called LFR benchmark algorithm [4]. This kind of benchmark graphs have the advantage to get closer to "real-world" networks' topology than graphs generated by other methods. All the graphs we used were weighted and directed because the Synchronised Louvain Method has been written for such graphs. The needed parameters and the ranges we used for each one of them are listed in Table 1.

| Parameter      | Meaning                          | Tested values                  |
|----------------|----------------------------------|--------------------------------|
| N              | Number of nodes                  | $i \in \{1, 2, 4, 6, 8\}, 1e6$ |
| $k$            | Mean degree of nodes             | 50                             |
| $k_{max}$      | Maximum degree of nodes          | 100                            |
| ($\mu_\tau, \mu_w$) | Mixing parameters (topology and weights) | (0.2, 0.1), (0.5, 0.4), (0.8, 0.7) |

Table 1: Used parameters for the generation of benchmark graphs.

The mean and maximum degrees of nodes were chosen constant in order to get the same edges density for each benchmark graph. The mixing parameters ($\in [0, 1]$) influence the way the communities are mixed between each other. The lower they are, the more the communities are distinct and easy to detect (Figure 14). For example, the parameter $\mu_\tau = 0.2$ (resp. $\mu_w = 0.1$) means that the edges density (resp. the weights of the edges) inside each community is (resp. are) way higher than the one(s) between communities. To avoid "ill-posed" graphs, we always chose $\mu_\tau > \mu_w$.

![Figure 14: Influence of mixing parameters. (Source: Browet, A. (2014) Algorithms for community and role detection in networks. Ph.D. thesis in Engineering Sciences, Ecole Polytechnique de Louvain, Université Catholique de Louvain, Louvain-la-Neuve.)](image)
For each set of parameters, we generated two different graphs and computed the mean results. The time results in the following parts are given using "wall-clock" time, i.e. the time that the user really waits for the execution of the algorithm, from the beginning to the end.

5.2 Computation time

Computation time with respect to the number of nodes and depending on the number of threads is represented in Figure 15. Three trends seem to appear on the Figures 15a-15c.

![Computation time vs size of the graph (µt=0.2 µw=0.1)](a)
![Computation time vs size of the graph (µt=0.5 µw=0.4)](b)
![Computation time vs size of the graph (µt=0.8 µw=0.7)](c)

Figure 15: Computation time versus number of nodes.

The first one is that the computation time clearly decreases with the number of threads, as expected. Secondly, by looking at these figures we can see that it seems that using the parallel implementation of Browet’s algorithm is really interesting for large enough graphs. Indeed, for graphs with less than $2 \times 10^3$ nodes, there is no real difference in terms of computation time between sequential and parallel executions. But, the higher the size of the graph, the greater the difference between sequential and parallel (12 threads) times, and thus the higher the gain in computation time.

Finally, these figures illustrate well the influence of the mixing parameters on the computation time. The higher they are, the higher the execution time. Indeed, high values for the mixing parameters mean that the communities are less distinct and thus that more corrections are needed before convergence is reached. And corrections are the most expensive functions in terms of computation time. On the contrary, graphs with
more distinct communities need less corrections before reaching convergence and thus the computation time is smaller.

5.3 Speedup

5.3.1 Expected and real speedup

To evaluate the theoretical maximum speedup we can obtain from our parallel implementation, one way is to refer to Amdahl’s law (see [5]):

\[
S = \frac{1}{(1 - P) + \frac{P}{N}}
\]

where

- \(S\) is the maximum expected speedup
- \(P\) is the portion of parallelised code in terms of wall-clock sequential execution time:

\[
P = \frac{\text{Fraction of time spent in parallelised parts}}{\text{Total execution time}}
\]

- \(N\) is the number of threads being used

The theoretical speedup predicted by Amdahl’s law is illustrated in Figure 16. However, we can notice that the speedup of any parallel program is always bounded given that the execution time of the non parallelisable part of the program can not be reduced.

Even if Amdahl’s law is often used to predict the potential speedup, we need to be careful with this result. Indeed, this law is purely theoretical and does not take into account some factors. One of them, which is critical, is the overhead phenomenon.

The overhead sums up the amount of external processing time needed for the execution of a code. In the case of a parallel program, the overhead is increased by the installation of parallelism, the synchronisation and communication between threads, and so on. In fact, we shall see that, due to increased overhead, our parallel implementation is only useful for large graphs, when the added overhead is sufficiently compensated by the acceleration gained by parallelism. Therefore, we can deduce that the expected speedup is actually lower than predicted by Amdahl’s law, as we can see on Figure 17 where we arbitrarily used \(P = 0.95\). To compute the exact \(P\) for our implementation, we could have used a code profiler, but this was not possible due to technical reasons..

![Amdahl's Law](https://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&oldid=716179061 (accessed May 5, 2016).)
5.3.2 Influence of size, mixing parameters and number of threads

Let us now examine the speedup we actually reach and the influence of the number of threads, the size of the graph and the value of the mixing parameters on it (see Figure 18).

![Empirical and theoretical speed-up vs number of threads](image)

Figure 17: Theoretical (for $P = 95\%$) and empirical speedup.
The first observation that can be made when looking at Figures 18a-18c is that the speedup does not seem to depend on the size of the graph if it is large enough. We have the impression that the speedup asymptotically tends to a constant value (for a constant number of threads), but this phenomenon is not observed for small graphs. Indeed, the computation time for these is smaller so the time “lost” because of overhead is not negligible with respect to the time gained thanks to parallelism. Furthermore, due to a lack of precision in the time measure, any small variation in the computation time can result in an unrealistic great speedup. These two phenomena may explain why the speedup for small graphs seem to be random.

Regarding the influence of the number of threads on the speedup, one can clearly see that the speedup increases with the number of threads, for large enough graphs. This is not always the case for smaller graphs, for the same reasons as above.

Let us now examine the influence of the mixing parameters on the speedup. It seems that speedup is higher for greater values of the mixing parameters (at least for large enough graphs). In graphs in which communities are less distinct, more corrections have to be applied before reaching convergence. Given that we parallelised the positive and maximal correction functions, the influence of parallelism is greater than for graphs with lower mixing parameters in which less corrections need to be made.

To conclude, we can say that the speedup probably depends on the values of the mixing parameters and on the number of threads but not on the size of the graph (for large graphs). It seems that the parallel implementation of Browet’s algorithm better outperforms on large graphs with less distinct communities and when the number of threads is higher. We could obtain an average maximum speedup of approximately 7.7.
with 12 threads on a graph of $10^6$ nodes and characterised by the mixing parameters $(\mu_t = 0.5, \mu_w = 0.4)$. Given that we used only 2 benchmark graph for each set of parameters, the average value we obtain for the speedup are not very representative. Our results are thus mainly qualitative and tests with more benchmark graphs should be carried out in order to have quantitative results.

6 CONCLUSION

The aim of this project was to implement a parallel version of Browet’s community detection algorithm, in order to get better performances in terms of computation time. First, we had to understand and analyse the different parts of the algorithm to spot the main functions to be parallelised. We determined that three of the four functions constituting the highest hierarchical level could be parallelised.

Then, we focused on these functions and their variables. Indeed, to implement the parallel version of Browet’s algorithm we chose to use OpenMP. This API requires special attention for the variable status, since it uses shared memory.

The results, computed using LFR benchmark graphs, confirm the highly parallelisable structure of Browet’s algorithm. Our parallel implementation has an execution time way lower than the sequential one. We also interpreted the role of each parameter involved on the effect of the parallelisation. To conclude, we can clearly consider applying this parallel version of the Synchronised Louvain method on very large networks to highlight their community structure.

FUTURE PROSPECTS The present work sought to give a first approach for the parallelisation of Browet’s algorithm. The results are more qualitative than quantitative in the sense that more performance tests should be executed to get more accurate mean computation times. Another interesting work would be to profile the code (for example, with Intel® VTune™ Amplifier XE) to spot potential parallelisation improvements to be done. We could finally imagine to analyse the classical Louvain method in order to determine if it is parallelisable or not. If so, we could compare performances results of both parallel classical and synchronised Louvain method.
A P P E N D I X

A.1 Pseudocodes of main steps of Browet’s algorithm

Algorithm 5 ASSIGN

function: ASSIGN(G(V,E))
for i ∈ V do
  a(i) = arg max j ΔH(i → j)  ▷ Best neighbour assignment
end for
T ← (V, {(i, a(i)) ∀ i})  ▷ Assignment graph
C_t ← WCC(T)  ▷ Weakly Connected Components
return C_t

Algorithm 6 POSITIVE

function: POSITIVE(C_t,G(V,E))
for all i ∈ V do
  g(i) = −ΔH(c_i → i → {})  ▷ Local gain
  while ∃ i ∈ c_i with g(i) < 0 do
    c_1, c_2 ← SPLIT(c_i)  ▷ Optimal assignment graph bisection
    for all j ∈ c_1 ∪ c_2 do
      g(j) = −ΔH(c_j → j → {})  ▷ Update local gain
    end for
    C_t = C_t \ {c_i} ∪ {c_1, c_2}  ▷ Update communities
  end while
end for
return C_t

Algorithm 7 MAXIMAL

function: MAXIMAL(C_t,G(V,E))
C = C_t
for all i ∈ V do
  c^*_i = arg max_c ΔH(c_i → i → c)  ▷ Best community for node i
end for
for all i ∈ V, if c^*_i ≠ c_i do
  draw p(i) uniform ∈ [0, 1]
  if p(i) < p then
    b(i) = branch(i)  ▷ Nodes in the tail (branch) of i
    if ΔH(c_i → b(i) → c^*_i) > 0 then
      a(i) = arg max_j ΔH(i → j)  ▷ Update assignment of i
      C ← insert(b(i), c^*_i)  ▷ Insert branch b(i)
    end if
  end if
end for
return C
REFERENCES

[1] A. Browet. *Algorithms for community and role detection in networks*. PhD thesis, Université Catholique de Louvain, 2014.

[2] B. Barney. *Introduction to Parallel Computing*, 2016. https://computing.llnl.gov/tutorials/parallel_comp/.

[3] B. Barney. *OpenMP*, 2015. https://computing.llnl.gov/tutorials/openmp/.

[4] A. Lancichinetti, S. Fortunato, and F. Radicchi. *Benchmark graphs for testing community detection algorithms*, 2008. https://sites.google.com/site/santofortunato/inthepress2.

[5] G. Chen. *Validity of the single processor approach to achieving large scale computing capabilities*, 1967. http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf.