Heriot-Watt University

Master Thesis

High-Performance Graph Algorithms

Author:

Adrien Chevrot

H00255927

Supervisor:

Dr Hans-Wolfgang Loidl

Second Reader:

Dr Lynne Baillie

Dissertation submitted in fulfilment of the requirements
for the degree of MSc in Software Engineering

in the

School of Mathematical and Computer Science

August 2017

Declaration of Authorship

I, Adrien Chevrot, declare that that this thesis titled, “High-Performance Graph Algorithms” and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Name:	Date/Signature:
Adrien Chevrot	17 August 2017

Abstract

Over the past decade, our technology has kept improving to become always more powerful and efficient to treat an amount of data that keeps getting bigger too. This data revolution raises major issues. Indeed, our technology has improved, we are able to build better machines year after year, but the data sets that we must deal with keep getting bigger and bigger, more than what we could expect. To face these issues, we can answer with different solutions. This project develops one of these solutions, parallel programming using multi-core machines.

In this project, we focused on the language Haskell. This language is said to be purely functional and well-suited to deliver solutions to parallel programming. We intend to show that we can realize complex tasks and exploit the potential of parallel programming using this functional language.

The main purpose of this project is to build a parallel algorithm capable of solving a Minimum Spanning Tree problem upon a weighted and undirected graph. A Minimum Spanning Tree algorithm returns, from a given graph, a list of edges connecting all the vertices together with the minimum possible total edge weight. This algorithm can find many applications in transportation, telecommunications (reducing cables cost for a company), electrical grid (using the same principle, getting all the components connected using as few wires as possible), image registration using the minimum spanning tree-based segmentation, medical image processing (describing arrangement of nuclei in the epithelium for cancer research) and so on.

Our objectives in this project are to build a graph respecting the DIMACS format and a sequential version. Once this first version completed, we must exploit parallelism using GpH (Glasgow parallel Haskell), an extension of Haskell to get a tuned parallel version. Speedup, runtime and GC percentage graphs must be built to assess performances, and give elements of reflection to improve the program.

Acknowledgements

I would like to express the deepest appreciation to my supervisor, Dr Hans-Wolfgang Loidl for his constant guidance and his support. His patience and the answers he provided to all the questions I had have been very helpful for understanding how far I could go in this research area that I did not know.

Furthermore, I would also like to acknowledge the entire staff of the School of Mathematical and Computer Science of Heriot-Watt University, who gave me the opportunity to follow this Master's programme. Many thanks for giving me the permission to use all required equipment to complete this project successfully.

In addition, I would like to thank the staff of my engineering school ISEN-Toulon, in France, specially Mrs Robert-Inacio, who gave me the opportunity to study abroad for the last year of my engineering course.

Last but not least, I would like to thank my family and friends for their unchanging support.

Table of contents

Chapter 1

1. Table of Figures

Chapter 2

2. Introduction

2.1. Overview

As mentioned earlier in the Abstract of this dissertation, our need of fast computations has grown following the progress of the data revolution we currently are living. It seems fair to say that the more data we can collect, the more data we must treat. The architecture that we are now able to build keeps getting more powerful than previous ones. However, it is difficult to imagine a world relying on programs treating information using single-core processors even if these processors keep improving. Indeed, using a single processor to treat a lot of data would be a waste when we can design programs runnable using several processors. This is where parallel programming on multi-cores machines gets involved.

Running a program on a multi-core machine is a solution to the volume of information that we must treat as explained above in this part. However, running a program on a multi-core machine does not necessarily lead to better performance. The implementation needs to take in account which part of the program can be designed in a way to exploit parallelism in it, otherwise, it would act as a sequential program.

There are many different technologies allowing us to improve the parallelism within our programs. However, as we said in the Abstract of this dissertation, parallel computing can be quite challenging, particularly when the programmer must organize the coordination of parallel threads. To face this, modern programming languages bring solutions for programmers who do not have to struggle on such challenging tasks anymore. Haskell is one of these programming languages. Indeed, GpH (Glasgow Parallel Haskell), an extension of this language, aims to simplify parallelism within a program for the programmer who can focus on major parts instead.

Graphs are, by definition, nothing but vertices connected by edges so they can model a wide range of structures of the real world which is built using the same principle. That is why we can model any network, like a social network or a map representing roads connecting cities using graphs, as we can see on the collection of dataset of Standord [Snap.stanford.edu, 2017].

2.2. Aims and Objectives

In this section of the dissertation, we talk about the objectives for our project. In the first time, we describe the main aim of the project in a single sentence, then, we describe it into more details with sub-tasks.

The main objective for this project is to develop a high-performance graph algorithm, more specifically a Minimum Spanning Tree algorithm, using a high-level parallel programming language, Haskell and its extension GpH, to obtain scalability, performance and usability.

Among this main objective, we can split up the project into several different tasks:

2.2.1. Build a graph

It seems natural to build a graph in order to treat it using a graph algorithm. The first step of this objective is to create a program capable of reading a file text in order to extract the graph. The second step of this objective is to only treat text files respecting the DIMACS graph format. DIMACS (Center for Discrete Mathematics and Theoretical Computer Science) defines a format for undirected graphs. We describe this format in detail in the section 3.2.3 of the Literature Review. The only thing to know for this section is that we can extract all the edges of a graph thanks to it. From all the edges that we extracted of the file text, we must assemble them to create an undirected and weighted graph, which is the third step of this objective. We detail undirected and weighted graphs at the beginning of the section 3.2 of the Literature Review. The final sub-task of this objective is to adapt the program to visualize the graph that we created. There are many software solutions to display graphs and the one advised by my supervisor is GraphTea, a java software designed to work on graphs and social networks. GraphTea receives only specific formats of file. One particular format of file that GraphTea can exploit is the adjacency-matrix notation which is explained in the section 3.2.2 in the Literature Review. The aim here is to create this kind of file from the graph created earlier to have a concrete visualization of the graph.

2.2.2. MST algorithm

This section is dedicated to the sub-tasks of the algorithm that we developed during this project. Before beginning this objective, we assume that the first objective (build a graph), has been correctly accomplished. For this project, we decided to implement the Boruvka algorithm. The Boruvka's algorithm is explained in detail in the section 3.3.4.c of the Literature Review. Among the 3 algorithms solving a MST problem that we detailed in the Literature Review (section 3.3.4.d), it is the only one allowing parallelism in it. The first sub-task of this objective is to visit every vertex of the graph and create a component for each one. This is crucial for the rest of the algorithm since we work with components (set of nodes) instead of vertices. The second step of this objective is to, for each component, here for each vertex, get its minimum edge linked to it and storing it. It implies a function returning the minimum edge from a list of edges. Then the third step of this objective is to arrange the components which share common minimum edge. Finally, we execute these three last steps in a recursive way until the number of total components is equal to one: the whole graph. This is explained in the section 3.4.c of the Literature Review.

This major objective leads to the sequential version of the algorithm. This version must then be parallelized and optimized in the next step of the project which represents the third major objective.

2.2.3. Parallel programming

In this section, we describe the different sub-tasks that we must achieve at this current project status. Once we implemented our sequential version of the algorithm, the objective is to exploit the parallelism in it. The first sub-task of this major objective is to exploit parallelism when we are returning all the minimum edges linked to each component. We can certainly save some time in this task so we must choose the good parallel strategy. The second step is to choose the good parallel strategy for running the arrangement of components which can be treated in parallel. The third step is to refactor the code if needed, in order to make it simpler, it may avoid spending too much time where it is not needed. At the end of these three sub-objectives, we would have created our tuned parallel version. All these sub-objectives can be fulfilled using the different parallel strategies that we detail in the section 3.4 of the Literature Review.

2.2.4. Results

The final objective for this project is to access the performance and scalability of the parallel algorithm, by giving speedup and scalability graphs. It may be useful to display a Garbage Collection percentage using the number of cores used during the experimentations. The final objective is to be able to discuss these results by analysing the different behaviours that the program can adopt. Another objective would be to understand and explain why a specific version of the program is better than another one and in which way it has affected the results.

2.3. Social, Professional, Ethical and Legal Issues

In this part of the dissertation, we talk about the different issues that we may encounter within the project. To begin with, we describe the social issues, then we describe the professional issues. In a third time, we move on to the ethical issues to finish with the legal issues.

2.3.1. Social

The author of this dissertation worked alone in this project. In addition of that, as we said before in the aims and objectives, this project consists of a program implemented by the author, and the conclusion that we drew comes from the tests of performance that we made. Thus, there was not any user involvement. The program is not tested by any experimenter since it is evaluated only using performances.

The project is evaluated using a quantitative evaluation such as speedup, scalability graphs, there is no any user evaluation.

2.3.2. Professional

Concerning the professional issues, there were regular meetings with our supervisor to deliver at each time a version of the project on process. We met each week to get some feedback on which parts of the program should be improved and how we could implement the on-going tasks.

2.3.3. Ethical

Concerning the ethical issues, as we mentioned in the social issues, there should not be any user involvement in the project. There are a lot of steps behind the scenes but it is some programming stuff, nobody is involved in the project. That is the reason why there should not be any ethical issues.

2.3.4. Legal

Finally, there are several rules to respect for this project. The program implemented is open-source and completely available to the public. Plus, any libraries, tools that have been used in this project are open-source such as GpH, an extension of Haskell mentioned above in the dissertation. The program is written in Haskell, so it should respect the rules defined by the School of Haskell, such as the format of comments, naming conventions and programming techniques.

The program that we develop in this project is in compliance with the BCS (British Computer Society), respecting the Code of Conduct explained on the website [Bcs.org, 2017].

The source code of the algorithm that we developed is licensed GNU GPL v3.0 (GNU General Public License) published by the Free Software Foundation [Foundation, 2007].

Concerning the different inputs that we used, the major part of these comes from the database of DIMACS (Center for Discrete Mathematics and Theoretical Computer Science) and represent random graphs. There is not any personal data in this project. Any input graph not taken from the DIMACS website is referenced.

We have the authorization of testing the performance of the program using the Robotarium Cluster of the School of Mathematics and Computer Science of Heriot-Watt. This cluster is composed of 64-core machines so it allows us to test the program using a wide variety of parameters. It is important to precise that we used the batch system to guarantee non-interference of our measurements with other jobs running.

2.4. Project Evaluation

As mentioned in the aims and objectives, the main purpose of this project is to develop a high-performance graph algorithm, more specifically a Minimum Spanning Tree algorithm, using a high-level parallel programming language, Haskell and its extension GpH, to obtain scalability, performance and usability.

We evaluated the project using the results obtained from the tests of performances of the sequential version to begin with. Then, we talk about the performances of other versions to finish with the most improved one, which is the final tuned version. The purpose of all these comparisons is to explain what is the gain of the (n+1)-th version compared to the (n)-th version. By explaining what made us change the program, we are able to highlight the differences of performance (runtime, speedup, garbage collection percentage) between both versions.

Chapter 3

3. Literature Review

In this section of the dissertation, we talk about the topics related to the subject which are mentioned in the literature. As expected, we support our argumentation using references such as books, articles or websites.

As shown in the aims and objectives, the project is divided in different parts since we face many domains. We divided this Literature Review into several pieces as well.

To begin with, we shall explore the language that we used in this project which is Haskell, a functional programming language. We describe its properties and characteristics. Here, we talk about the fact that Haskell is strongly typed, its functional characteristic, its type inference to finish with its laziness. It is crucial to detail the advantages of this language to explain why it is a relevant candidate for our program.

In the second part of the Literature Review, we talk about the graphs and their representations. We see that they can be represented by two tools, the adjacency-list representation and the adjacency-matrix representation. The third part of this section is dedicated to the DIMACS standard notation to represent graphs. This is the notation that we chose to use for extracting graphs from text files so it is important to explain its properties.

I the third section of the Literature Review, we discover different ways of going through a graph, then we shall see some graph algorithms such as the all-pairs shortest paths and the maximum flow to finish with the algorithm that we chose to implement in this project: the minimum spanning tree algorithm. In this specific section, we highlight three algorithms capable of solving a minimum spanning tree problem and we choose between them by identifying the one allowing parallelism.

For the final part of this Literature Review, we talk about the parallel tools that we applied during the implementation of the algorithm, more specifically of GpH (Glasgow parallel Haskell), which is an extension of Haskell aiming to bring parallelism into a program. This section talks about the general strategies that we get to run a program in parallel such as chunking, thresholding and more.

The list of references is located at the end of this dissertation.

3.1. Functional Programming & Haskell

To begin this literature review, we talk about the programming language. We used functional programming which completely differs from what we have studied before. Indeed, functional programming focuses on evaluating mathematical functions.

Iterative languages such as C focuses more on state changes and data mutation while functional programming does not use these features, that is why it relies on applications of functions. The goal of functional programming is to develop in a way that functions can be applied to other functions, to chain these ones.

The language that we are going to use within this project is Haskell. Haskell has been built on the lambda calculus and combinational logic. Its name comes from the mathematician Haskell Brooks Curry. The language has been created in 1990 by a group of searchers in language-theory who were interested in functional programming and lazy evaluation.

The main features of Haskell, as described in the official webpage of the Haskell Organization [Haskell.org, 2017], are described below:

3.1.1. Strongly typed

Every expression in Haskell has a type which is determined during the compilation time. All the types involved in a function must match up with its definition otherwise the program is rejected by the compiler.

```
char = 'a'      :: Char
int = 123       :: Int
fun = isDigit :: Char -> Bool
```

```
isDigit 1                                                    Type error
```

Figure 3.1.1: Each expression has a type in Haskell [Haskell.org, 2017].

As we can see above, the compilation step cannot be fulfilled since a wrong type is applied to a function only accepting a character.

3.1.2. Purely functional

Every function in Haskell is a function in a mathematical sense. There are no statements or instructions, only expressions which cannot mutate variables nor access state like time or random numbers. Functional programming, as indicated by its name, is based on applying functions to other functions.

A function is a mapping that takes one or more arguments and produces a single result, and is defined using an equation that gives a name for the function, a name for each of its arguments, and a body that specifies how the result can be calculated in terms of the arguments [Hutton, 2016].

When a function is applied to actual arguments, the result is obtained by substituting these arguments into the body of the function in place of the argument names. The result is an expression containing other function applications, which must then be processed in the same way to produce the final result. To give an example:

```
    double 3
=       { applying double }
    3 + 3
=       { applying + }
    6
```

Figure 3.1.2: Substituting arguments to a double function [Hutton, 2016].

3.1.3. Type inference

Type do not need to be explicitly provided in a Haskell program since they are inferred by unifying every type bidirectionally.

```haskell
main :: IO ()
main = do line :: String <- getLine
          print (parseDigit line)
  where parseDigit :: String -> Maybe Int
        parseDigit ((c :: Char) : _) =
          if isDigit c
             then Just (ord c - ord '0')
             else Nothing
```

```haskell
main = do line <- getLine
          print (parseDigit line)
  where parseDigit (c : _) =
          if isDigit c
             then Just (ord c - ord '0')
             else Nothing
```

Figure 3.1.3: Type inherence in Haskell [Haskell.org, 2017].

The first part of the Figure 3.1.3 has a type signature for every binding, however, the second part leads to the same result.

3.1.4. Lazy

Functions do not evaluate their arguments which means that program can compose together with the ability to write control such as if/else, just by writing normal functions [Hutton, 2016]. Haskell allows us to chain functions and by doing that, improving the performances. Indeed, since an expression is only evaluated when called within the program, it avoids computing useless calculations. We can illustrate this property with an example taken from [Laziness, 2017].

```haskell
f x y = x + 2
```

Figure 3.1.4: Laziness in Haskell [Laziness, 2017].

If we try to compute, f 5 (29^35792), the second argument is packaged up into a subroutine without doing any computation. Since “f” does not need the second argument, the

subroutine is thrown away by the garbage collector. In a classic strict language, the computation of 29^35792 would be performed even though it is not needed.

Through this first major section of the Literature Review, we saw the different advantages that functional programming has. Functional programming is closer to mathematics than imperative languages in a sense that functions can be seen as relations between elements. Functional languages have excellent abstraction mechanisms that can be applied to both computation and coordination [Hughes, 1989]. Two important abstraction mechanisms are function composition and higher-order functions. Function composition allows complex problems to be decomposed into simpler sub-functions. Higher-order functions, ones that manipulate other functions, allow new control constructs to be defined as required. The principle of abstraction can be carried through to parallel programming, where higher order functions may be used to form the basis of new parallel programming constructs. Typically, parallel functional programs abstract over details such as process placement, the timing and volume of communication, and synchronisation issues [Loidl et al., 2002]. Finally, as explained in the same reference, values do not change once they have been computed, so the language is free from unnecessary dependencies. Haskell is characterized as architecture-independent language due to the use of high order functions and polymorphism and most importantly, Haskell is side-effect free.

3.2. Graph Representations

Before talking about graph algorithms, we talk about graphs, their structure and their possible representations. If we desire to go through an entire graph and be able to treat it using an algorithm, we have to know their best representation to easily understand how the program must deal with it.

A graph is composed of vertices linked by edges. These edges can be weighted, meaning that they link two vertices (or one) together with a value. In this case, the graph is called weighted. A graph can also be directed which means that the edges have a direction from a vertex to another one. Some of the algorithms that we study in the Graph Algorithms of this Literature review are only valid if applied to oriented/directed or inversely.

Graph theory is globally detailed in the well-known textbook from [Bondy and Murty, 2010].Graphs can be qualified of dense or sparse, depending on how high is their density. We can calculate the density of a graph using its number of edges/vertices with this expression:

$$D = \frac{2\,|E|}{|V|.\,(|V|-1)}$$

Considering D as the density, E the number of edges and V the number of vertices. The numerator is $2\,|E|$ since we are counting two times the number of edges since each edge is a link between 2 vertices. The denominator is $|V|.\,(|V|-1)$ instead of $|V|^2$ because we assume that a vertex is not linked to itself.

A graph is said to be sparse if the number of edges is much less than the possible number of edges [Black, 1998], so if the density is near 0 while a dense graph is characterized by a high number of edges and a density near 1 as we can see below:

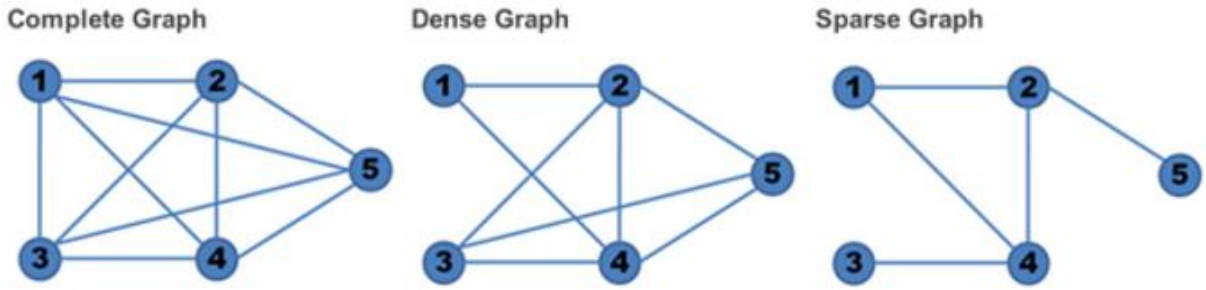


Figure 3.2.1: Density of a graph [Black, 1998].

The density of the graph is crucial in this project because, as we explain it later in the section 3.3.4, the performances of a Minimum Spanning Tree algorithm mainly depends on the number of edges, more than the number of nodes. Our program applied on a dense graph spends way more time than applied on a sparse one.

3.2.1. Adjacency-list representation

Adjacency-list representation provides a compact way to represent sparse graphs (explained above) [Cormen et al., 2014].

The adjacency-list representation of a graph G = (V, E) consists of an array Adj of |V| lists, one for each vertex in V. For each u ∈ V, the adjacency list Adj[u] contains all the vertices v such that there is an edge (u,v) ∈ E.

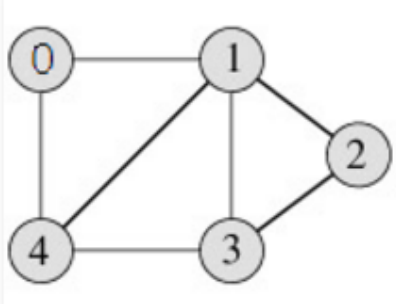


Figure 3.2.2: Default graph to study [GeeksforGeeks, 2017].

We assume studying this graph as an example:

Its adjacency-list representation would be:

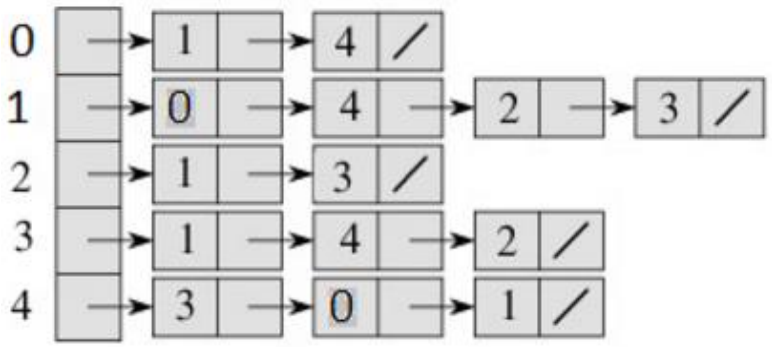


Figure 3.2.3: Adjacency-list representation of the default graph [GeeksforGeeks, 2017].

The list clearly enumerates all the vertices linked to each vertex.

We can adapt this kind of representation to weighted graphs thanks to a weight function. This representation is convenient in a way that it can support many graph variants.

However, a potential disadvantage of the adjacency-list representation is that it is quite long to determine whether a given edge (u,v) exists. The only way to do it is to search 'v' in the list of 'u'. A good mean to remedy this problem is to use the adjacency-matrix representation that we develop below.

3.2.2. Adjacency-matrix representation

The adjacency-matrix representation is a good choice when we are dealing with a dense graph (explained in Graph Representations [Cormen et al., 2014].

As an example, we describe the adjacency-matrix representation of the graph that we previously studied in the 2.a) part of the same section.

Its adjacency-matrix representation would give:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Figure 3.2.4: Adjacency-matrix representation of the default graph [GeeksforGeeks, 2017].

As described in "Introduction to algorithms" [Cormen et al., 2014], to draw an adjacency-matrix representation of a graph G = (V, E), we assume that the vertices are numbered 1, 2..., |V| in some arbitrary manner. Then the matrix consists of a |V|*|V| matrix $A = (a_j^i)$ such as

$$A = (a_j^i) = \begin{cases} 1 \; if \; (i,j) \in \mathrm{E}, \\ 0 \; otherwise \end{cases}$$

Since in an undirected graph, (u,v) and (v,u) represent the same edge, the adjacency-matrix A of an undirected graph is its own transpose: $A = A^T$.

As mentioned in the 2.a) part, the adjacency-matrix representation solves the problem that it can be long to identify if a given edge exists. Indeed, we only need to check the value between the first vertex in the matrix's rows and the second vertex in the matrix's columns.

However, a disadvantage of this solution is that it would cost asymptotically more memory.

3.2.3. DIMACS graph notation

As mentioned in the section 2.2.1 of this dissertation, a sub-objective in this project is to treat text files respecting the DIMACS format for graphs, as detailed in [Dimacs.rutgers.edu, 2017]. DIMACS is the Center for Discrete Mathematics and Theoretical Computer Science. It was founded in 1989 as an NSF (National Science Foundation) funded Science and Technology Center (STC) and one of the New Jersey Commission on Science and Technology's Advanced Technology Centers. The broad mission of DIMACS is to catalyse and conduct research and education in

mathematical, computational, and statistical methods, algorithms, modelling, analysis, and applications [Dimacs.rutgers.edu, 2017].

We used the DIMACS input graph format which is described below using the webpage [Prolland.free.fr, 2017]. In this format, nodes are numbered from 1 up to *n* edges in the graph.

Files are assumed to be well-formed and internally consistent: node identifier values are valid, nodes are defined uniquely, exactly *m* edges are defined, and so forth.

- Comment lines give human-readable information about the file and are ignored by programs. Comment lines can appear anywhere in the file. Each comment line begins with a lower-case character **c**.
- Problem line. There is one problem line per input file. The problem line must appear before any node or arc descriptor lines. For network instances, the problem line has the following format.

  The lower-case character p signifies that this is the problem line. The FORMAT field is for consistency with the previous Challenge, and should contain the word ``edge''. The NODES field contains an integer value specifying *n*, the number of nodes in the graph. The EDGES field contains an integer value specifying *m*, the number of edges in the graph.

- Edge Descriptors. There is one edge descriptor line for each edge the graph, each with the following format. Each edge *(v,w)* appears exactly once in the input file and is not repeated as *(w,v)*. The lower-case character e signifies that this is an edge descriptor line. For an edge *(w,v)* the fields W and V specify its endpoints.

3.3. Graph Algorithms

In this section of the Literature Review, we describe different graph algorithms. Each of them is specific to one kind of graph and computes completely different tasks. Among these algorithms, we chose the one that we implemented during the project depending on its interest.

3.3.1. Searching

3.3.1.a. Breadth-first search

The breadth-first search is one of the simplest way to go through an entire graph and is at the basis of many graph algorithms. We can use the breadth-first search algorithm on both directed and undirected graphs.

If we use as an example a graph G = (V, E) and a source vertex s, as explained in [Cormen et al., 2014], 'breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex'.

It also produces a breadth-first tree with root s that contains all reachable vertices. For any vertex v reachable from s, the simple path in the breadth-search tree from s to v corresponds to a "shortest-path" from s to v in G, that is, a path containing the smallest number of edges.

Breadth-first search is so called because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance k + 1 as we can see below:

In this example, we take the vertex 'A' as the source vertex. From this vertex, we can reach B, C and D that we store in a queue as visited vertices. After that, we discover E, G and F which belong to the "level 2".

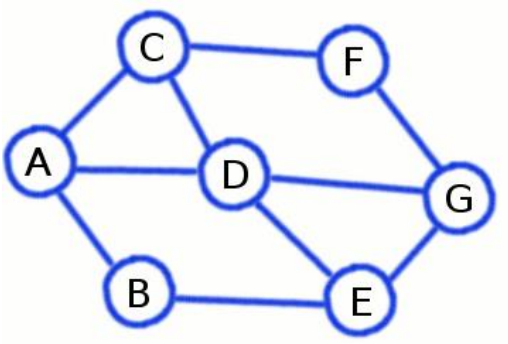


Figure 3.3.1: Random graph for breadth-first search [Gorkovenko, 2017].

The reason why we study this search algorithm its principle is used within the algorithms of minimum spanning tree and single-source shortest-paths.

3.3.1.b. Depth-first search

In the depth-first search, we are searching in priority deeper in the graph, which implies that instead of first looking which vertices are reachable from the source at the same distance, we want here to discover all the vertices undiscovered from a reached vertex. Once we have discovered all the vertices reachable from a vertex v, the algorithm goes back to a visited vertex from where we can reach undiscovered vertices.

The algorithm of Depth-first search uses a stack (a list in which we can insert or remove at the front).

As we can see here, we go from the source 1, we reach 2, and instead of going to 7 and 8 as would do a breadth-first search algorithm (3.1.a part), we go deeper until 4. Since from 4 we cannot reach another vertex anymore we go back to 3 and discover 5, and so on to discover the other vertices.

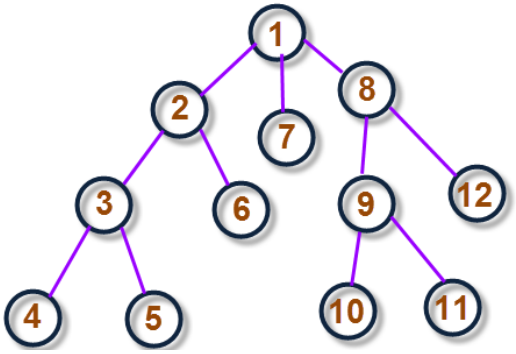


Figure 3.3.2: Random graph for depth-first search [Rallabhandi, 2017].

In the next parts of this Graph Algorithm's section, we shall use the Big O notation to describe how the runtime of an algorithm is. To know how long takes an algorithm to run, with this notation, we take in account, how the runtime grows and using parameters relative such as the number of edges/vertices here in our example. To give an example, if we say that the runtime of an algorithm is $O(V^2)$, it means that the runtime mainly depends on the square of the number of vertices.

3.3.2. All-Pairs Shortest Paths

In this part, we describe how we can, from a given vertex u, go to a given vertex v, using the shortest path that we can find. This kind of application can be applied on both directed and undirected graphs but also on weighted or unweighted graphs.

This kind of problem can be applied for example in a GPS (Global Positioning System) to link two cities with the lower distance possible. The shortest path is calculated by adding the weights of the edges which belong to the paths.

There are different algorithms that can compute this kind of problem like the Bellman-Ford algorithm or Dijkstra algorithm but the one that we focus on is the Floyd-Warshall algorithm. A reason why we focus on this algorithm is its particularity which is not involved in Bellman-Ford or Dijkstra algorithms. Indeed, these two algorithms both are single-source shortest path algorithms so it computes the shortest path between two given vertices.

The Floyd-Warshall algorithm can get the same result but can also compute the shortest distances between every pair of vertices. Indeed, the principle of this algorithm is to split up the work between subparts, and once all the subparts are solved, the algorithm combines all the parts together to solve the entire problem.

The idea of the algorithm is that the shortest path between two given vertices 'S 'and 'T' is either the shortest path found so bath between S and T or the shortest path between 'S' and an intermediary vertex 'I' added to the shortest path between 'I' and 'T'.

As we said in the previous paragraph, it looks either for the quicker path between two vertices or the addition of two short distances using an intermediate. This is what differentiate this algorithm from the two others that we talked about just a few lines earlier, no matter the vertex you are currently in, it calculates the shortest path to every other vertex.

In the computation of the algorithm, we use the adjacency-matrix representation, as seen in 2.b) part of this section, because we must multiply these matrixes.

We do not describe the whole pseudo-code of the algorithm but taking in mind this property, we know that there should be three for-loops, one for the source, one for an intermediate and the last one for the destination. That is the reason why the runtime of this specific algorithm is $O(V^3)$.

3.3.3. Maximum Flow

In a maximum flow problem, we want to calculate the maximum rate at which we can move some material (it can be materialized by some water moving into pipes), from a vertex source to a destination vertex.

This kind of problem can be applied to:

- Directed graphs because edges must have a direction to transport some material from a vertex to another.

- Weighted graphs because each edge must have a maximum capacity (for the water example, a pipe is does not have an infinite rate).

However, there are two rules to keep in mind when we are implementing the algorithm:

1 - The rate of water (to keep up with this analogy) cannot be greater than the maximal rate of the pipe.

2 - if we consider vertices as junctions between pipes, the rate at which water enters the junctions must be equal to the rate that leaves the junction, it is called “flow conservation”.

To illustrate this problem, we can analyse the picture below:

We assume that the source vertex is ‘S’ and the destination is ‘T’. Here the maximum flow is 15 since we can pass 10 out of 10 from ‘S’ to ‘U’ and 5 out of 5 from ‘S’ to ‘V’. Then we ship 5 out of 15 from ‘U’ to ‘V’, 5 out of 5 from ‘U’ to ‘T’ and 10 out of 10 from ‘V’ to ‘T’.

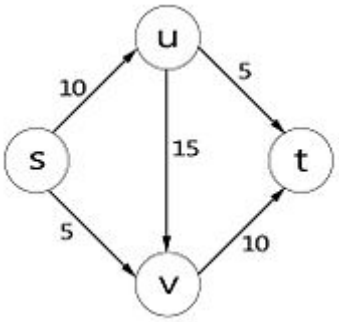


Figure 3.3.3: Random graph for maximum flow problem [En.wikipedia.org, 2017].

If we try to increment the input rate, we put 10 out of 10 from ‘S’ to ‘U’ and we are forced to ship 6 out of 5 from ‘S’ to ‘V’ which violates the first rule stated earlier.

We can solve this maximum flow problem using the Fold-Fulkerson method. As explained in [Even and Even, 2012], the Fold-Fulkerson method uses a simple main part, we initialize the flow to 0, and while we can find an augmenting path in the network, we increment the flow. At the end, we obtain the maximum flow which can go through the graph. Obviously, behind this main part, there are many additional concepts to determine if can find a path within the graph. However, we do not go into more complex details for this method since we did not develop this algorithm in the project.

3.3.4. Minimum Spanning Tree

In this part of the Literature Review, we describe the graph algorithm that we decide to study for our project. The principle of a MST algorithm is to find the shortest path, in a graph, to connect all the vertices of a graph using the weights of the edges. In our case, we shall use this this algorithm in a weighted and undirected graph. The graph must also be itself connected, meaning that it has exactly only one component, consisting of the whole graph. In other words,

you can access any vertex of the graph from any vertex. We cannot solve a minimum spanning tree problem in a graph in which a particular vertex cannot be reached.

This algorithm is commonly used in a TSP problem (Travelling salesperson problem). As explained in [Davendra, 2010], 'with a given set of cities all connected by roads, the aim of a Travelling salesperson algorithm is to answer the following question: What is the shortest route that visits every city and returns to the starting place'. A minimum spanning tree algorithm I capable of solving this problem but can have other applications.

Indeed, this algorithm is also used in many domains such as biology with the DNA sequencing in which distance represents measure between DNA fragments but also in astronomy when astronomers want to minimize the time spent moving the telescope between the sources. Finally, it is also applied during the manufacturing of electronic circuit in which we want to interconnect a set of pins. To do this we use a MST algorithm to use the least amount of wire which is preferable.

The idea for solving this type of problem would be check every round-trip route to find the shortest one, but this would be too time time-consuming to proceed like this. Indeed, as E.Klarreich explains it in [Klarreich et al., 2017], 'as the number of cities grows, the corresponding number of round-trips to check quickly outstrips the capabilities of the fastest computers', we understand here than even a small graph would lead to heavy computation so we must change our approach.

We call this algorithm the minimum spanning tree algorithm since by linking all the vertices together, we create a subgraph that forms a tree. Within a graph, we can create many subgraphs which form trees but the objective in this algorithm is to find the minimum one. As shown in the Figure 3.3.4, we have two different subgraphs, two different spanning trees.

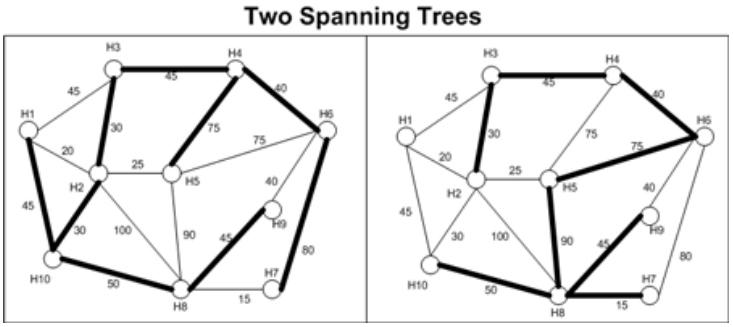


Figure 3.3.4: Two different spanning trees [Msdn.microsoft.com, 2017].

In order to solve this problem, we have three different algorithms that we describe just below.

3.3.4.a. Kruskal's algorithm

In this section of the MST part, we describe the Kruskal's algorithm which has been designed by Joseph Kruskal in 1956. Its principle is, as explained in [Cormen et al., 2014], 'to find a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) of least weight'.

Kruskal's algorithm is qualified as a greedy algorithm since it adds to the forest previously created an edge of least possible weight at each step. A greedy choice, as explained in [GeeksforGeeks, 2017], is to take the smallest weight and add it to the current tree that does not create a cycle within this tree.

To understand what is a cycle within a graph, it is detailed in [L.Gross and Yellen, 2006] that a cycle within a graph form a closed path in it. We can reach a vertex from itself using this path.

Here is an idea of how it works:

- We sort all the edges in a decreasing order of their weight
- We pick the smallest edge.
- We add the smallest edge to the forest constructed so far. Obviously, if we are in the first step of the program, we create this forest.
- If the forest has become a cycle, then we cancel the previous step by removing the edge that we just added. Indeed, as we said previously in this part, we do not want to create a cycle within the graph, we want to build an acyclic tree. We take a new edge in the sorted list that we created in the first step and try to do the same operation. However, if the forest built in this step does not form a cycle, we go back to the first step.

We keep doing these operations while the forest has less components than the number of vertices minus 1. Indeed, if the graph contains a number 'V' of vertices, the minimum spanning tree must have necessarily V-1 components.

By using binary heaps, the authors of [Cormen et al., 2014] estimate that we could run the Kruskal's algorithm in time O(E lg(V)) using E as the set of possible interconnections between two vertices, V is the set of vertices and lg being the logarithm function (base-10).

3.3.4.b. Prim's algorithm

In this section of the MST part, we describe the Prim's algorithm which has been developed in 1930 by V.Jarnik and rediscovered in 1959 by R.C.Prim and E.W.Dijkstra. This algorithm operates like the Dijkstra's algorithm that we mentioned in the 'All Pairs Shortest paths' part of this Literature Review.

In a way, we could think that this algorithm behaves like the Kruskal's algorithm that we just saw since it adds to the tree an edge that contributes the minimum amount possible to the tree's weight at each step, but it differs in a point: in the Kruskal's algorithm we build a tree from a sorted list of edges. Each added edge contributes to the forest but without knowing that other edges have been selected.

Here, in the Prim's algorithm, we start the tree by an arbitrary root, from this root we create a list of all known edges (the ones linked to the source vertex) and we add to the tree the edge that has the smallest weight. We arrive to the second vertex and we redress a list of possible edges. If the second vertex offers edges that are more weighted than the ones already known from the source vertex, then we include another edge from the source vertex to the tree.

The principle is to, at each step, dress a new list of all reachable edges, and pick the most interesting one. If, by including it to the tree, we create a cycle, then we try with the second most interesting edge and so on until obtaining a forest composed of 'V-1' edges, V being the number of vertices.

As we said previously, the Prim's algorithm is a greedy algorithm, like the Kruskal's algorithm but their respective runtime can be different. The authors of [Cormen et al., 2014] estimate than, by using Fibonacci heaps, we could run the algorithm in time O(E + V*lg(V)), which can be a good improvement compared to the Kruskal's algorithm if V is much smaller than E. Considering the fact we build our forest of edges by adding at each step one edge, the algorithm could only be treated sequentially, which is not relevant for our project.

3.3.4.c. Boruvka's algorithm

In this section of the MST, we describe the Boruvka's algorithm which has been the first minimum spanning tree algorithm discovered in 1926 by Otakar Boruvka. This algorithm does not work like the previous ones that in saw in the MST part, it uses the merging of disjoint components.

The principle of this algorithm is to visit each vertex once. For each vertex, we pick the smallest edge attached to the specific vertex. Once we have done that for each vertex, we obtain either directly the minimum spanning tree or several spanning trees. We could illustrate the second case with the Figure 3.3.5:

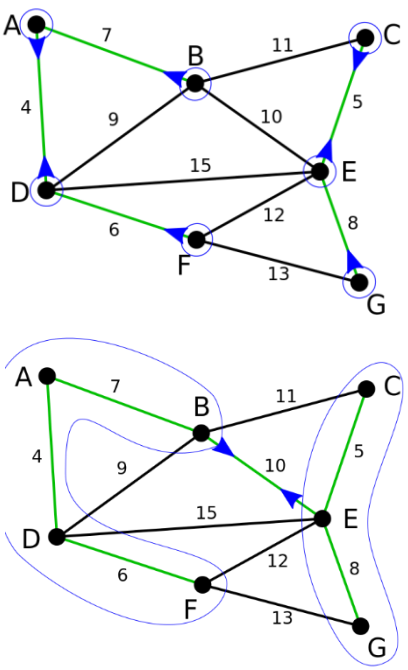


Figure 3.3.5: First and second step of a boruvka algorithm [En.wikipedia.org, 2017].

Now that we have two distinct trees which are: {A, B, D, F} and {C, E, G}, we can contract these sets to keep only two new vertices. We can imagine that the first vertex would be X, the merging of the first tree, and the second would be Y, the merging of the second tree. By doing that we only have to deal with these two vertices and their edges which are 11, 10, 15, 12 and 13. We do the same operation, we visit the two vertices and pick the edge which has the smallest weight, here 10. Finally, we build the new tree with the results obtained, as shown in the Figure 3.3.5:

We obtain our final minimum spanning tree shown in green in the second part of Figure 3.3.5.

In the example taken here, we have a very small number of vertices but in a bigger graph, we would probably obtain way more sub-trees that we could contract in unique vertices.

The main advantage of this algorithm is that, unlike Kruskal & Prim's algorithms, we obtain subtrees that can be treated independently. By doing that, we avoid dependencies in the program and we can allow parallelism within the algorithm, which is extremely relevant for our project.

A drawback of this algorithm is that all the weights must be different, otherwise, the algorithm is not able to choose a particular edge during the first step of the execution.

3.3.4.d. Parallelism

In this section of the MST part, we talk about how we could include parallelism within the chose algorithm. We need here to parallelize an algorithm meaning that we have to ensure that within this algorithm, there are tasks without any dependencies to execute them in parallel.

If we analyse in a first time the Kruskal's algorithm, we know that we must dress a list of all the less weighted edges and includes one after another to the forest, taking in account that it must not form a cycle in the tree. I think this is the worst algorithm to parallelize because each step must be done sequentially and we must know the results of each iteration to move forward. Including parallelism within it would be a challenging and inefficient task.

If we analyse in a second time the Prim's algorithm, we know that we begin with an arbitrary source vertex and from this source, we analyse which edge is the less weighted and we include the edge if the built forest does not form a cycle. We repeat this step with at each step the list of available edges that keeps growing. I think it would not be efficient to parallelize this algorithm too because steps must be done sequentially.

We finally talk about the Boruvka's algorithm which has, to me, a more general approach. Indeed, it is an algorithm that could include parallelism since as a first step, you visit all the vertices of the graph and, for each vertex you pick the less weighted edge attached to this specific vertex. Obviously, this is the part that we should make in parallel. Indeed, there are no dependencies in this task. Each vertex can be treated independently from others so that is how I plan to include parallelism within the algorithm.

In order to include parallelism, we shall use techniques and tools from the different extensions of Haskell that we describe below in this Literature Review.

3.4. Glasgow parallel Haskell

In this section of the Literature Review, we focus on GpH which is an extension of the language Haskell, aiming to bring parallelism into a program. It is a high-level parallel programming extension which has a more user-friendly approach since the runtime system does the major part of the work. Indeed, the user must only identify where parallelism can be included within the program. Then the user indicates how the task must be parallelized thanks to strategies and the runtime system takes care of the rest.

As we said previously in the Haskell part of this Literature Review, Haskell is a language that allows us to write complex tasks within a few lines. As expected, this principle can be applied with GpH. As S.Marlow explains it in [Marlow, 2013], adding a small notation to your program can suddenly make it run several times faster on a multicore machine.

The approach used in GPH is intermediate between purely implicit and purely explicit approaches. The runtime system manages most of the parallel execution, only requiring the programmer to indicate those values that might usefully be evaluated by parallel threads and, since our basic execution model is a lazy one, perhaps also the extent to which those values should be evaluated. We term these programmer-specified aspects the program's dynamic behaviour [Hans-Wolfgang Loidl, 2001].

3.4.1. Evaluation Strategies

In the GpH technology, we use strategies to run tasks in parallel. This parallelism is expressed using the Eval monad. We can use the keyword 'rpar' to indicate to the system that an argument should be evaluated in parallel while the keyword 'rseq' indicates to the runtime system should evaluate sequentially the argument meaning evaluating this argument and wait for the result.

However, it is possible that an argument gets partially evaluated, Haskell being a lazy language (clarifications in the Haskell part of the L.R), we talk here of a Weak Head Normal Form (WHNF). In order to fully evaluate an argument, we call the force the full evaluation with the 'deepseq' keyword which, by definition: $rdeepseq\ x = x\ `deepseq`\ Done\ x$, fully do the evaluation of x.

3.4.2. Data-oriented Parallelism

Data-oriented parallelism is an approach where elements of a data structure are evaluated in parallel. A parallel map is a useful example of data-oriented parallelism; for example, the 'parMap' function defined below applies its function argument to every element of a list in parallel.

```
parMap strat f xs = map f xs `using` parList strat
```

Figure 3.4.1: Data-oriented parallelism with 'parMap' [Marlow et al., 2010].

The parMap function takes a strategy strat, a function f, and a list xs as arguments and maps the function f over the list in parallel, applying strat to every element. The construction of the result with map, on the left of using, is separated from the specification of the parallelism, on the right [Marlow et al., 2010].

In our project, as mentioned in the section 3.3.4.d, we chose to implement the Boruvka's algorithm because it is the only algorithm, among the different ones solving a minimum spanning tree problem, in which we easily find a task to do in parallel. Indeed, for each vertex, we must find its less weighted edge. There are no dependencies in this task so I dress a list of vertices and apply the 'parMap' keyword to this list using a function which finds for each vertex its less weighted edge.

3.4.3. Control-oriented Parallelism

Control-oriented parallelism is typically expressed by a sequence of strategy applications composed with 'par' and 'seq' that specifies which subexpressions of a function are to be evaluated in parallel, and in what order.

```
quicksortS (x:xs) = losort ++ (x:hisort) `using` strategy
                    where
                      losort = quicksortS [y|y <- xs, y < x]
                      hisort = quicksortS [y|y <- xs, y >= x]
                      strategy result = rnf losort `par`
                                        rnf hisort `par`
                                        rnf result
```

Figure 3.4.2: Quicksort function with control-oriented parallelism [Trinder et al., 1998].

As we can see above in Figure 3.4.2, we apply our control-oriented strategy with the keyword 'using'. We exploit parallelism here by selecting 'losort' and 'hisort' for parallel evaluation [Trinder et al., 1998].

3.4.4. Thresholding mechanism

A common strategy to obtain better performances is to use thresholds. The basic idea of this principle is to avoid creating too many threads for small tasks. Indeed, treating these small tasks sequentially leads to better performances.

```
-- thresholding version
pfactThresh' :: Integer -> Integer -> Integer -> Integer
pfactThresh' m n t
  | (n-m) <= t = product [m..n]    -- seq solve
  | otherwise = (left * right) `using` strategy
      where mid    = (m + n) `div` 2
            left   = pfactThresh' m mid t
            right = pfactThresh' (mid+1) n t
            strategy result = do
                                rpar left
                                rseq right
                                return result
```

Figure 3.4.3: Example of a thresholding strategy [Loidl, 2017].

As we can see above, the select 'left' and 'right' to be evaluated in a parallel manner if the difference of the two given parameters is above a specific threshold 't'. It avoids doing too much small tasks in this divide and conquer program by solving it sequentially.

3.4.5. Chunking Data Parallelism

Another well-known strategy to get better performance is to chunk a data-structure. Indeed, if we must evaluate every single element of a list, instead of creating sparks for each element, it would be more efficient to evaluate many elements using a single thread. The number of elements (the size of the chunks) can be tuned using for example the number of threads that we can provide, to give good performance.

```
-- print (map Main.factorial [12..30] `using` (Control.Parallel.Strategies.parListChunk 2 rdeepseq))
```

Figure 3.4.4: Example of a chunking strategy [Loidl, 2017].

As we can see above in Figure 3.4.4, we use the 'parListChunk' function in order to create a thread and execute factorial function for each chunk of size 2. Obviously, setting the chunk size to 1 and using 'parList' would act the same way. In our project, as mentioned in the section

3.4.2, we planned to use the ‘parMap’ keyword to exploit parallelism. That would be more effective to use a chunking strategy instead and discuss the influence of our chunking size.

Chapter 4

4. Implementation & Results

This section of the dissertation details all the technical work performed during this project. We go through all the different steps presenting the versions of the project. We highlight the main decisions that the we had to make to move forward in this task and why we took these decisions.

The first part of this major section gives details about the sequential implementation and all the stuff that have been realised before this sequential version which is a major objective as we saw in the section 2.2 of this dissertation. The second part of this section highlights the first parallel version that we produced, the final tuned parallel version and each version between these two. For each version, we detail why we went from the previous to the new one to explain the differences in performance. In the final part of this section, we give details about the different results to compare the different versions and draw evaluations of our project using these graphs (speedup, runtime, garbage collection percentage).

4.1. Sequential Implementation

In this section, we detail all the steps we went through to reach our first sequential version of the Minimum Spanning Tree algorithm, and by sequential version, we talk here about the Boruvka's algorithm.. To begin with, we describe the problem situation for building our sequential version, then, we explain which data structure we decided to use and how we built graphs from text files respecting the DIMACS format. We detail how we made the use of GraphTea possible using our program. Then, we go through the implementation of the Dijkstra algorithm which helped us to build the Prim algorithm to reach the Boruvka algorithm. We end this section with the different results that we can extract from this sequential version of the Minimum spanning tree algorithm.

4.1.1. Problem description

In this project, we are asked to develop a parallel version of a graph algorithm. We chose to solve the minimum spanning tree problem, in a functional environment. The problematic to solve here is how to extract parallelism in a minimum spanning tree algorithm. As we know from [Macs.hw.ac.uk, 2017], GpH is a modest conservative extension of Haskell realising thread-based semi-explicit parallelism. It provides a single primitive to create a thread, but thereafter threads are automatically managed by a sophisticated runtime system. By precising that, we know that our parallel program must be created from the implementation of a sequential version. We do not have to think how we deal with our different threads and their communication since our runtime system automatically handles it. That is why we must focus on a sequential version in which there are few dependencies but without knowing exactly how we implement the threads of the parallel version since all coordination is implicit.

4.1.2. Data structure

During the first days of the project, we made the choice to use our own data structures. The idea was to considerate nodes as integers values, a 200-nodes graph would have node 1, node 2, until the node 200. Concerning the edges, they are simply expressed using the two nodes they rely, and its weight, which is a floating-point value.

Finally, we consider graphs as a simple list of edges. We can match this idea with the adjacency-matrix representation described in the section 3.2.1 in which we describe a graph by listing, for each node, all the edges linked it. Basically, we can resume a graph as a list of edges since we have all the data that we need with the number of nodes and weights. The Figure 4.1.1 below describes in Haskell which data structure we chose to use:

```
type Node = Int
data Edge = Edge (Node, Node) deriving (Show, Eq)
data Wedge = Wedge (Edge, Float) deriving (Show, Eq)
data Wgraph = Wgraph [Wedge] deriving (Show)
```

Figure 4.1.1: Data structure to define a graph.

As we can see in the Figure 4.1.1 above, we defined nodes as Integer numbers. From this first definition, we can create other data structure: in a first time, we define an Edge using

a pair of 2 nodes. Then we define a weighted edge by combining an Edge and a floating-point value which is the weight. Finally, we define a graph using a list of weighted edges.

4.1.3. Build a graph

In this section, we explain how we built graph from text files. It is important to keep in mind for this section that an edge between two nodes is specified only once in a text file respecting the DIMACS standard format. To give an example, if there is an edge between the first and the third node, there is not any edge specified between the third node and the first one.

To build a graph, we start reading the data file and take all the lines of the file. This is generic to all graph algorithms, not only the one that we implement here. The time spent for building the edge is not included in the measured time. For each line, we apply a function to build a weighted edge. From these steps, we obtain a list of weighted edges that we turn into a graph, as detailed in the Figure 4.1.2 below

```
-- Build a weighted edge
buildWedge :: [String] -> Wedge
buildWedge [e, n1, n2, d] = Wedge (Edge (read n1 :: Node, read n2 :: Node), read d :: Float)
buildWedge [e, n1, n2] = Wedge (Edge (read n1 :: Node, read n2 :: Node), 1)

-- From a DIMACS format text file, keeps only the edge while removing the rest
keepEdges :: [String] -> [String]
keepEdges str = filter (\e -> head e == 'e') str

-- Build a list of weighted edges from a list of String
fromLinestoWedge :: [String] -> [Wedge]
fromLinestoWedge str = map buildWedge (map words str)

-- Build a weighted graph from a list of weighted edges
buildGraph :: [Wedge] -> Wgraph
buildGraph wedges = Wgraph (wedges)
```

```
let graph = buildGraph (fromLinestoWedge (keepEdges linesOfFiles))
```

Figure 4.1.2: Building a graph using a text file.

As we can see above, we get all the content of the text file with the value ‘linesOfFiles’ which is a list of String, one String for each line. Then, we use the ‘map’ keyword that we saw in the section 3.4.2 of the dissertation to apply the ‘buildWedge’ function to every line. This function uses a powerful tool of Haskell, the pattern matching. Using pattern matching, you can

specify different function bodies for different arguments. When you call a function, the appropriate body is chosen by comparing the actual arguments with the various argument patterns [Haskell.org, 2017].

The 'buildWedge' function identifies if a weight is specified, otherwise it defines the weight of the edge to 1 as a standard. We use this kind of pattern matching because the major part of the DIMACS graph text files does not specify the weights of the edges. Setting the default weight to 1 is not very relevant for the minimum spanning tree but as long as we know the algorithm works with normal graphs (different weights), we know the program behaves correctly even with weights equal to 1. We work here to compare performances instead of comparing the value of the spanning tree obtained.

One thing that I did not mentioned yet in this section is that before using the 'buildWedge' function upon all the lines, we keep only the lines of a text beginning by the letter 'e' because it is in this way that an edge is expressed according to the DIMACS standard, as we can see in the example below:

```
c edge density          : 0.900216
c max degree            : 1848
c avg degree            : 1799.53
c min degree            : 1751
p col 2000 1799532
e 2 1
e 3 1
e 3 2
e 4 1
e 4 2
e 4 3
e 5 1
e 5 2
e 5 3
```

Figure 4.1.3: Example of a DIMACS graph text file

The standard input text file is explained in the section 3.2.3 of the dissertation.

4.1.4. GraphTea

In this section, we describe what is GraphTea, the different steps we had to go through in order to make possible the use of GraphTea.

It is an open-source software, created for high quality standards and released under GPL license, as explained in the official website [Graphtheorysoftware.herokuapp.com, 2017]. This tool is very helpful when it comes to generate graphs, get information about drown graphs, visualize them as well. It is also possible to run classic graph algorithms such as the MST algorithm, or a Maximum flow algorithm, described in the section 3.3.3. It has been helpful at the beginning of the project to check is our sequential version produced the correct answer for the MST problem, and for representing simple graphs.

GraphTea can build graphs only if the provided file gives the adjacency-matrix representation of the graph (explained in the section 3.2.2 of this dissertation).

The idea to build the adjacency-matrix notation is to create a String for each node. The first String is devoted to the first node, the second String to the second node, and so on. For each node, we check if the first node shares a weighted edge with it. If yes, we put the weight into the String, if not, we put '0' into the String. Then we do the same thing with the second node, and so on. We can see this feature of our program with the Figure 4.1.4 below, pointing out a random graph from [GeeksforGeeks, 2017] and its adjacency-matrix notation created:

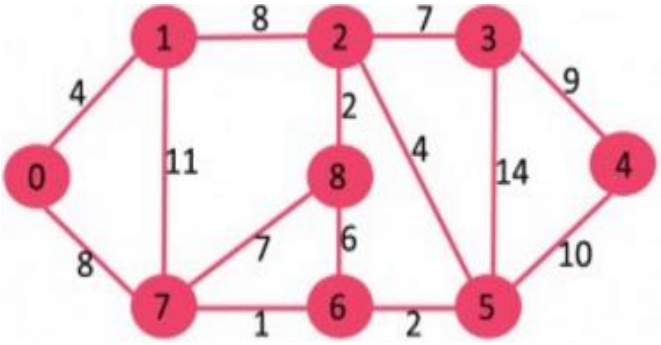


Figure 4.1.4: Random graph and its adjacency-matrix notation.

4.1.5. Algorithms

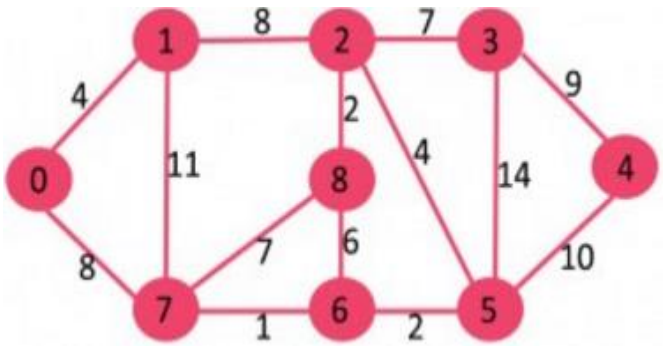
In this section, we describe the different algorithms we implemented before reaching the Boruvka algorithm which is our second major objective described in the section 2.2.2 of the dissertation.

At the beginning of the project, my supervisor told be to implement, as a warm-up exercise, a shortest path algorithm. After implementing that, it would be easier to get a

minimum spanning tree algorithm. I decided to implement the Dijkstra's algorithm since it is the most popular.

In the Dijkstra's algorithm, we take a node as the start (for the rest of the details, we call it 'node 0'). At the first step of the algorithm, we know that the distance between 'node 0' and itself is equal to 0 while we put the infinity as distances between 'node 0' and all other nodes. Then we look at the edges linked to 'node 0' and we update the distance between them and 'node 0' as their weight. Then we take the closer node (we call it 'node 1') to 'node 0' and we update all the distances between the linked edges to 'node 1' and 'node 0' with the sum: distance (from 'node 0') = distance (between 'node 0' & 'node 1') + weight (between 'node 1' & current node).

If this sum is less than the previous distance (at first step infinity), this distance replaces the previous one. And we repeat these steps until we know the distance to reach any node in the graph. In the Figure 4.1.5 below, we take the Node 0 as the start node and we look the shortest path to get to Node 4 & Node 8. The graph is taken from [GeeksforGeeks, 2017].




```
Start node ?
0
End node ?
4
Path: [0,7,6,5,4]
Distance: 21.0
End node ?
8
Path: [0,1,2,8]
Distance: 14.0
End node ?
```

Figure 4.1.5: Application of the Dijkstra's algorithm

At this stage of the project, we had a correct version of a Shortest Path Algorithm. The goal here was to go from this Dijkstra's algorithm to a Minimum Spanning Tree algorithm. One simple way to achieve that was to implement the Prim's algorithm explained in the section 3.3.4.b. Indeed, the Prim's algorithm has a behaviour very close to the Dijkstra's algorithm. Even if, as we said before, the Prim's algorithm is not an algorithm in which we can exploit a lot of parallelism. But the purpose behind this was to have at least a correct MST algorithm. It has helped me to understand how a MST algorithm could work and more particularly how I could

design the final desired version (Boruvka's algorithm) since I knew where I could extract parallelism from it compared to the Prim algorithm.

The principle behind the Prim's algorithm is the same than the Dijkstra's algorithm, apart from the fact that instead of updating a distance from a start node, we update the distance from the node we are visiting. At each step, we increase the number of known vertices, until all the vertices have been visited. Then we stop the program and we return the edges that have been selected at each step.

Now that we had an efficient algorithm for a Minimum Spanning Tree problem, we had to implement the one that could be run in parallel. The principle of the Boruvka's algorithm is explained in the section 3.3.4.c of this dissertation so we directly give details about our implementation.

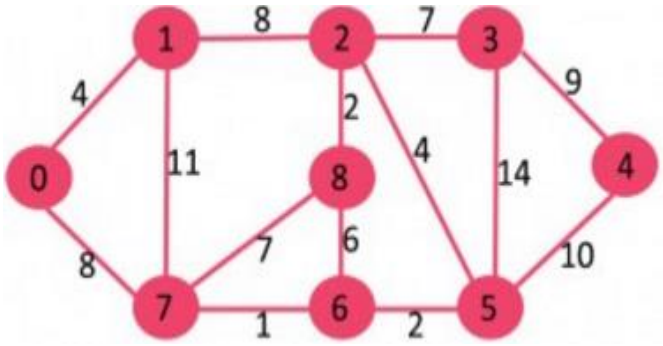
```
-- Launch the recursive boruvka algorithm
boruvka :: Wgraph -> [Wedge]
boruvka g =
  let wedges = []
      components = getComponents g
      (_, _, wedges') = boruvkaAlg g components wedges
      in wedges'

-- Recursive boruvka algorithm which ends when the list of component is only composed of one component
boruvkaAlg :: Wgraph -> [[Node]] -> [Wedge] -> (Wgraph, [[Node]], [Wedge])
boruvkaAlg g components wedges | components == [getNodes g] = (g, [], wedges)
                               | otherwise =
      let
          currwedges' = nub $ map (minLinkedComponent g) components
          --currwedges' = nub (map (minLinkedComponent g) components)
          wedges' | length components == 2 = wedges ++ [head currwedges']
                  | otherwise = wedges ++ currwedges'
          midcomponents = fromWedgesToNodes currwedges'
          midcomponents' = rearrangeComponentFinal midcomponents
          components' = rearrangeComponentFinal (components ++ midcomponents')
      in boruvkaAlg g components' wedges'
```

Figure 4.1.6: Recursive Boruvka's algorithm

As we can see on the Figure 4.1.6 above, we apply the 'boruvka' function upon a graph and it returns a list of weighted edges. As we said before, a list of weighted edges is a graph. The minimum spanning tree that we obtain represent a sub-graph of the graph. As described in the Figure 4.1.6, the 'boruvka' function is the top-level function which launches the recursive algorithm below.

The ‘boruvkaAlg’ function is the function executed at each step of the program. The parameters as inputs are the graph that we are using, a list of lists of nodes which corresponds to a list of components, and a list of weighted edges, which increases at each step since it is in this parameter that we store our spanning tree. In the function, we first check the value of our components. For a graph of 10 nodes, the ‘getNodes’ function would give [1,2,3,4,5,6,7,8,9,10]. That is why, if our parameter ‘components’ is equal to the list of all the nodes, it implies that all the spanning tree managed to rely all the nodes, thus, the algorithm has finished so we set the components as an empty list to end the recursive algorithm.

In the body of the recursive algorithm, we execute the boruvka’s algorithm as we explained it in the section 3.3.4.c of the dissertation. We start looking for each component its minimum edge liked to it. Then we use the function’fromWedgeToNodes’ which for example takes ‘Wedge(Edge(3,5),1)’ as input and gives [3,5] as output. Then we rearrange our components with the ‘reArrangeComponentFinal’ function. This function merges components if they share elements. To give an example, if we have two components [1,2,3,4] & [5,6,7], they do not share elements so the function does not affect these components. However, if the second component is [3,6,7], they have Node 3 in common so the result of the merging would be [1,2,3,4,6,7].




```
[Wedge (Edge (0,1),4.0),Wedge (Edge (2,8),2.0),Wedge (Edge (2,3),7
.0),Wedge (Edge (3,4),9.0),Wedge (Edge (5,6),2.0),Wedge (Edge (6,7
),1.0),Wedge (Edge (0,7),8.0),Wedge (Edge (2,5),4.0)]
"Length of the Spanning Tree : 37.0"
```

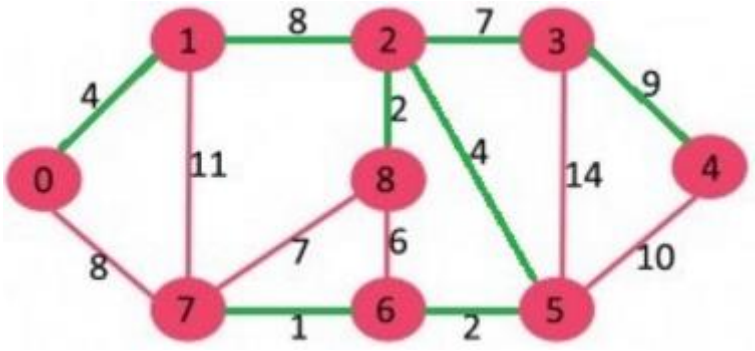


Figure 4.1.7: Demonstration of our sequential implementation of the Boruvka's algorithm [GeeksforGeeks, 2017]

The first part of the Figure 4.1.7 describes us a random graph, the second part corresponds to the output of our program while the third part gives the correction of the website [GeeksforGeeks, 2017]. As we can see, our program gets us the correct minimum spanning tree. There are two different minimum spanning trees in this specific graph. The first one includes the weighted edge between node 1 and 2 (3rd part of the Figure 4.1.7) while the second one includes the weighted edge between 0 and 7. Both edges gives a total length of 37 for the minimum spanning tree since they both have a weight of 8.

4.1.6. Implementation complexity

Before moving up to the results, we must talk here about the complexity of implementing this sequential version using this particular language. Actually, it has been less painful than I expected, although there have been frustrating moments during the project. Sometimes, I took afternoons to do a particular task that I knew it would take me about 1 hour in a language like C++ which I know better. Even if studied it during the year, I could not say that I had a solid background in Haskell, it is sometimes painful to write functions using a syntax difficult to take charge at the beginning.

It is difficult for me to say that it would be easier to implement the algorithm using another such as Java or C++, because on one hand, I feel more comfortable using this kind of languages, while on the other hand, now that I have experienced the functional programming with Haskell, the program does not seem so complex and it takes less source code than it would take using Java or C++. Another advantage I would give away to Haskell is the fact that we treat

our data structure, in a more mathematical manner, which for this kind of application, seems more natural.

4.1.7. Sequential Results and Performance Analysis

In this section, we describe the different results that we could get from this sequential implementation of the Boruvka's algorithm. We describe the runtime analysis in a first time and the heap profile in a second time.

4.1.6.a. Sequential time profiling

In this section, we detail the time analysis of our sequential program. The sequential time profiling is a tool used to identify which parts of the program take the most time. We call these parts the "big eaters". This step is important for the rest of the project because we know that in order to make our program parallel, we have to handle first the top-level functions and then to introduce adequate parallelism in these functions called "big eaters".

To do this, we compile our sequential program using first '-O2' to increases both compilation time and the performance of the generated code. We use '-prof' for enabling heap profiling during runtime -auto-all.

Then we execute the program with the following flags: +RTS (runtime system, which handles storage management, thread scheduling, and so on), -pT for generating standard report and -hC for heap profiling. The results displayed below for profiling have been edited using a graph of 1000 vertices and 246708 edges from [Info.univ-angers.fr, 2017].

				individual		inherited	
COST CENTRE	MODULE	no.	entries	%time	%alloc	%time	%alloc
minLinkedComponent	Main	150	1002	0.0	0.0	65.8	66.7
minWeight	Main	161	1002	0.0	0.2	0.1	0.2
weight	Main	162	715852	0.0	0.0	0.0	0.0
findWedgefromWeight	Main	158	1002	0.0	0.0	0.0	0.0
findWedgefromWeight.\	Main	159	1002	0.0	0.0	0.0	0.0
weight	Main	160	1002	0.0	0.0	0.0	0.0
linkedComponent	Main	151	2004	0.1	0.1	65.8	66.6
linkedComponent.\	Main	156	988326	5.0	0.1	5.0	0.2
twoNodes	Main	157	1618721	0.0	0.1	0.0	0.1
linkedWedges	Main	152	3003	11.1	0.1	60.7	66.3
linkedWedges.\	Main	154	575696353	38.4	0.0	49.5	66.2
twoNodes	Main	155	575696353	11.1	66.2	11.1	66.2

rearrangeComponentFinal	Main	143	2	0.0	0.0	14.3	16.1
rearrangeComponent	Main	184	3	0.0	0.0	14.3	16.1
checkComponentDuplicate	Main	185	1009	6.6	0.5	14.3	16.1
quickSort	Main	189	1105333	0.0	0.1	2.7	14.8
quickSort.biggerSorted	Main	192	555054	2.0	14.6	2.0	14.6
quickSort.smallerSorted	Main	190	555054	0.7	0.1	0.7	0.1

Figure 4.1.8: Sequential time profiling for a graph of 1000 nodes

This is just an overview of the time profiling, the whole list of functions is available in the Appendix. We only put here in the Figure 4.1.8 above the most important points to highlight. As we can see, the major part of the runtime is taken by the 'minLinkedComponent' which is used to find the minimum weighted edge linked to a Component. The "big eater" within this function is obviously the 'linkedWedges' function since it represents 75,4% of the 'minLinkedComponent' function. However, this amount of time is due to the number of times the function has been called. It is a good thing to have the 'minLinkedComponent' as the most time-consuming function since we plan to make it run in parallel in our future versions.

We hope also to exploit some parallelism in the 'reArrangeComponentFinal' function which is responsible for 14,3% of the whole runtime. Within this function, the 'checkComponentDuplicate' is responsible of this percentage of runtime. Hopefully, we extracted parallelism in this function.

4.1.6.b. Heap profiling

In this section, we explain how our memory has been used over time. It is a very useful tool to identify potential memory overflow.

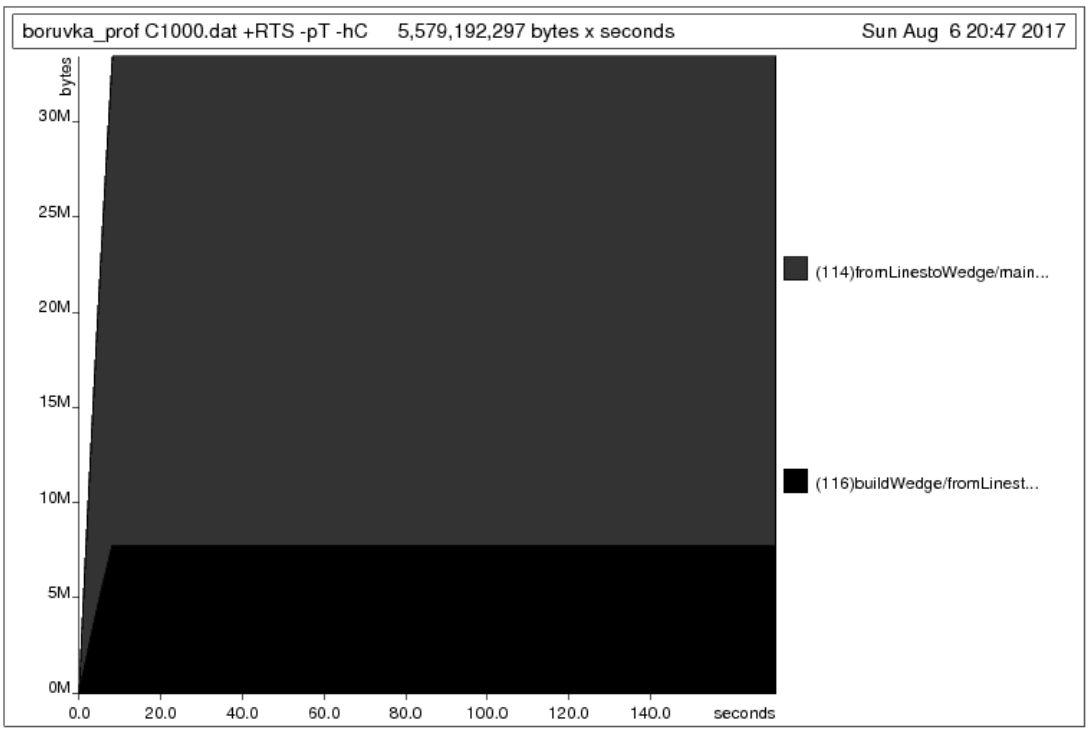


Figure 4.1.9: Heap profiling for a 1000-nodes graph.

The only remarks that we could make from this Figure 4.1.9 above is that the whole consumption of the memory is taken by the reading of our data file. Indeed, the function fromLinesToWedge uses a as many times the function buidWedge as the graph contains an edge.

We can see that the whole consumption of the memory is reached in the first 5 seconds of the execution, as expected since the I/O part of the program is 5 seconds long for this 1000-nodes graph from [Info.univ-angers.fr, 2017].

4.1.6.c. Input size influence

This section displays a table resuming the influence of the input graph:

Input graph	125 nodes – 6963 edges	500 nodes – 112332 edges	1000 nodes – 246708 edges	2000 nodes – 1999732 edges	4000 nodes – 4000268 edges
Runtime (seconds)	0,24	15,2	76,4	915,9	4853,7

GC percentage	5,2	3,5	2,1	3,2	4,3

Table 4.1-1: Influence of the input graph

As we can see in the table above, the runtime increases very quickly as soon as the input graphs gets bigger. A Boruka's algorithm runs in time O(E log(V)) where E is the number of edges while V is the number of vertices. It is crucial to highlight that the runtime depends a lot upon the number of edges, way more than the number of vertices. Indeed, the number of edges of the second graph is 16 times bigger than the one from the first graph and there is a ratio of 63 when we divide the two runtimes. However, the number of edges from the third graph is only 2,2 times bigger than the one from the second graph and the runtime gets less affected since when we divide the two runtimes (from 3rd graph and 2nd graph), the ratio is equals to 5, which is way less than 16.

A good point for this sequential implementation is its garbage collection percentage since it remains at an acceptable value, below 6%, regardless of the input graph. Unfortunately, this percentage increased when we introduced parallelism in the program, which is the next section of the dissertation.

4.2. Parallel Implementation

Now that we have implemented our sequential version of the Boruvka's algorithm, it is time to introduce some parallelism within the program. Thanks to the profiling, we know that we must do something for the 'minLinkedComponent' function and for the 'reArrangeComponentFinal' function since they both represent the two main parts of the execution time. We are not able to improve the reading of the input file but it does not matter since from the total execution time, we are able to extract only the time spent for the recursive algorithm. We can also calculate the garbage collection percentage only for the recursive algorithm which is nice because I/O leads to a heavy use of garbage collection.

In this section, we go through the different parallel versions of our program with their results in terms of runtime & GC percentage while explaining the reasons to go from a specific version to the next one.

The term speedup defines the ratio between the computation time using 1 processor and the computation time using 'n' processors.

4.2.1. 1st parallel version: Parallel Mapping

In this section, we explain what we did to obtain our first parallel version of the Boruvka's algorithm, then we give the results of this version in terms of runtime, speedup and GC percentage to give a standard for building comparisons with other parallel versions.

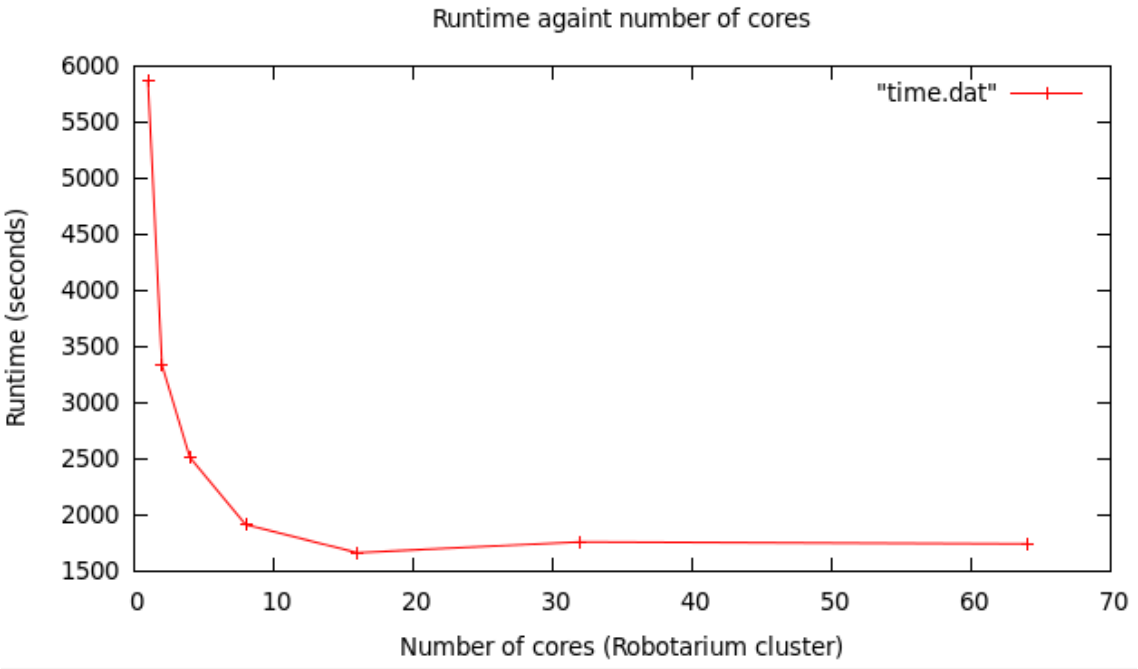
As we said above in the sequential implementation, we have to make things run in parallel for searching the minimum edge linked to a component and for rearranging our components. To do this, we used the naivest way to introduce parallelism in the program. We used the strategy the data-oriented parallelism that we saw in the section 3.4.2 of this dissertation. We use the keyword 'parMap' instead of a classical 'map' in order to create parallel threads where we are able to create them, as shown in the Figure 4.2.1 below:

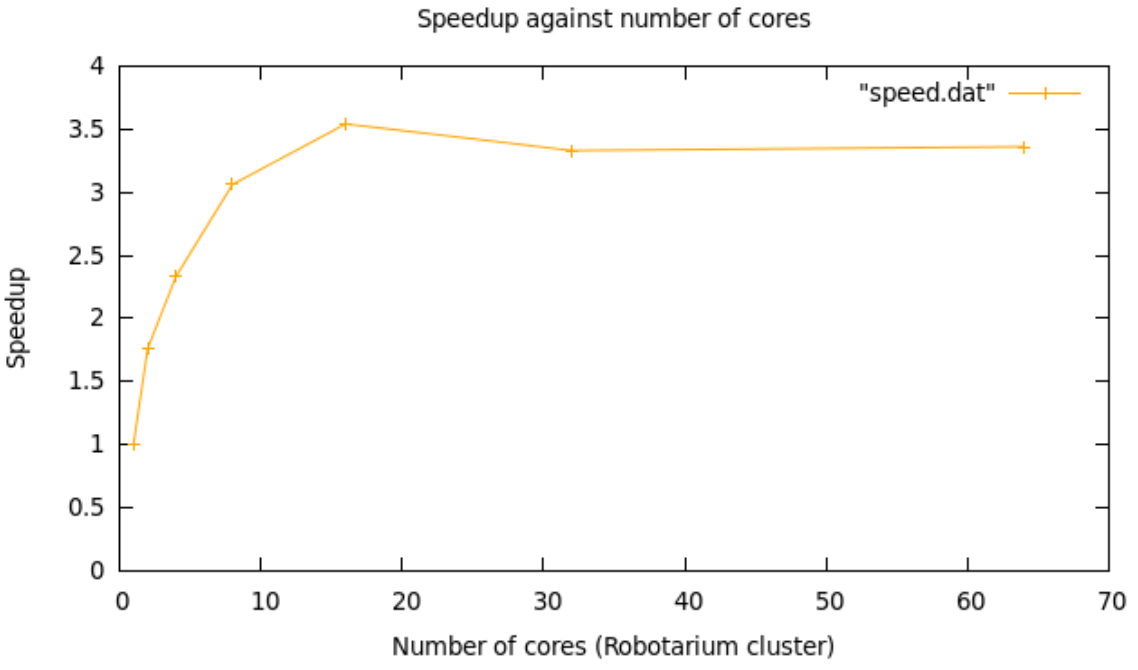
```
-- Recursive boruvka algorithm which ends when the list of component is only composed
--                                                          of one component (all the nodes)
boruvkaAlg :: Wgraph -> [[Node]] -> [Wedge] -> (Wgraph, [[Node]], [Wedge])
boruvkaAlg g components wedges | components == [getNodes g] = (g, [], wedges)
                               | otherwise =
      let
          currwedges' = nub $ parMap rdeepseq (minLinkedComponent g) components
          wedges' | length components == 2 = wedges ++ [head currwedges']
                  | otherwise = wedges ++ currwedges'
          midcomponents = fromWedgesToNodes currwedges'
          midcomponents' = rearrangeComponentFinal midcomponents
          components' = rearrangeComponentFinal (components ++ midcomponents')
      in boruvkaAlg g components' wedges'
```

```
-- Rearrange the components modified by the boruvka algorithm at each step
rearrangeComponent :: [[Node]] -> [[Node]]
rearrangeComponent listoflist = nub (parMap rdeepseq (checkComponentDuplicate listoflist) listoflist)

-- Rearrange the components (top-level function)
rearrangeComponentFinal :: [[Node]] -> [[Node]]
rearrangeComponentFinal listoflist | length listoflist == 1 = listoflist
                                   | otherwise = until (not . isThereStillDuplicates) (rearrangeComponent) listoflist
```

Figure 4.2.1: Modifications made to obtain the first parallel version.

As we can see in the first part of the Figure 4.2.1 above, we create parallel threads to search the minimum edge linked to a Component and we also create parallel threads to rearrange components in a shorter time. However, even if we run these parts of the program in parallel, it is a naïve implementation because there is a lot of data to treat at each part of the program and creating a thread for each single data is naïve. Indeed, it leads to a bad use of our processors because each one of them must deal with a tiny piece of work. Here are the results that we obtained from this version using a graph of 4000 vertices and 4000268 edges (runtime and speedup) and 1000 nodes and 246708 edges (GC) from [Info.univ-angers.fr, 2017]:





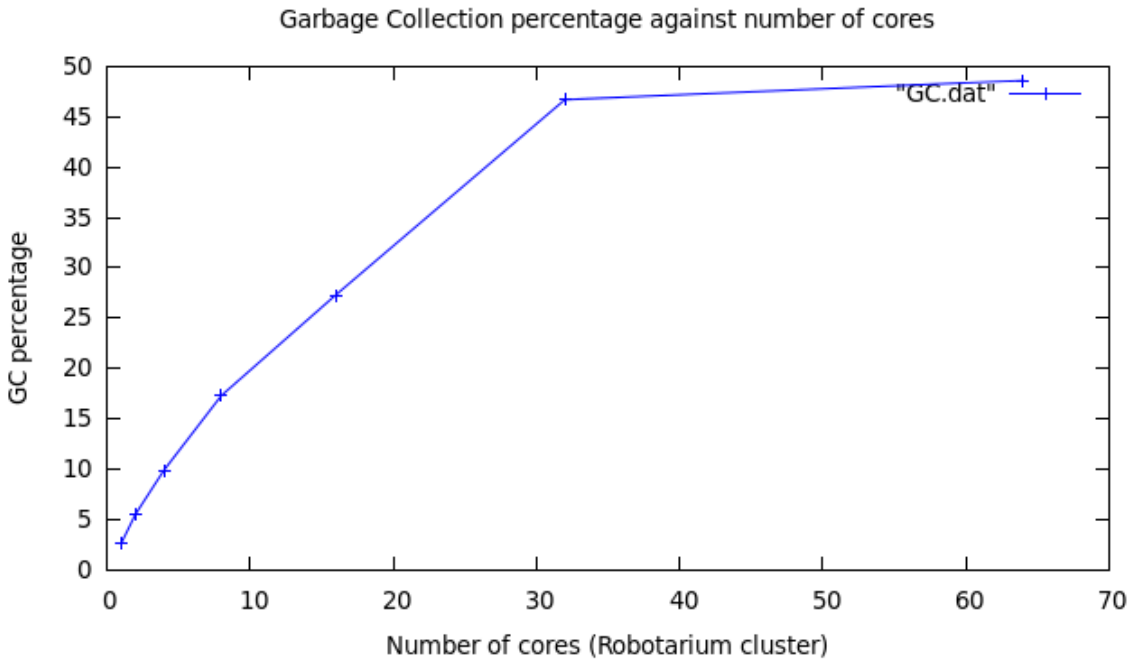


Figure 4.2.2: Runtime, speedup & GC percentage of the first parallel version.

When we take a look at the first and second part of the Figure 4.2.2 above which are respectively the runtime and the speedup for this parallel version, we can notice that when we increase slightly the number of nodes to use for the computation, the runtime is quickly affected, in a good way. However, the disturbing part for these tests is clearly when we use more than 16 cores, the improvement in runtime/speedup is really bad. Indeed, the speedup gets at its maximum value when we are using 16 cores, and decreases a bit with 32 cores to remain at the same value for 64 cores.

The most disturbing graph is the one about garbage collection. Indeed, the garbage collection percentage gets too high from a number of cores of 8 which is low. On top of that, this value gets even more high from 16 cores, to reach values that should never be reached. With 64 cores, we almost got half of the computation dedicated to garbage collection, which is terrible.

We can suppose that this percentage of garbage collection is due to the number of sparks created within the program. There are too many threads hanging on to data in the heap which needs to be treated. Introducing a strategy of chunking/thresholding would lead to better runtime by increasing the amount of work within a thread and would decrease the percentage of GC with the same reason, this is our next version.

4.2.2. 2nd parallel version: Chunking & Thresholding

In this section, we explain why we decided to improve our first parallel version described above, and what modifications we did to get better results.

As we said before in the first parallel version, we had problems with our runtimes from a number of cores of 16, which is not high enough. Another huge trouble we encountered is the garbage collection percentage which reached almost 50% which seems completely inconceivable. As explained before, this high percentage is due to the number of threads that we create which is way too high. Indeed, we create a thread for each single piece of data to treat.

A solution to fix this would be to use a chunking strategy. This strategy is explained in the section 3.4.5 of this dissertation. Instead of creating a thread for each single piece of data, we gather elements together using 'chunks' and create a thread for one chunk. By using this, we force the processor to deal with several pieces of data instead of one.

We decided to use the 'parListChunk' keyword in order to arrange the elements using a specific size of chunk and treat this chunk with one thread. We used this strategy upon the two major functions 'minLinkedComponent' ad 'reArrangeComponentFinal', as we can see below:

```
-- Recursive boruvka algorithm which ends when the list of component is only composed of one component (all the nodes)
boruvkaAlg :: Wgraph -> [[Node]] -> [Wedge] -> (Wgraph, [[Node]], [Wedge])
boruvkaAlg g components wedges | components == [getNodes g] = (g, [], wedges)
                               | otherwise =
      let
          currwedges' | length components < 16 = nub $ map (minLinkedComponent g) components
                      | otherwise = nub (map (minLinkedComponent g) components `using` parListChunk 8 rdeepseq)
```

```
-- Rearrange the components modified by the boruvka algorithm at each step
rearrangeComponent :: [[Node]] -> [[Node]]
rearrangeComponent listoflist | length listoflist <= 32 = nub (map (checkComponentDuplicate listoflist) listoflist)
                              | otherwise = nub (map (checkComponentDuplicate listoflist) listoflist `using` parListChunk 8 rdeepseq

-- Rearrange the components (top-level function)
rearrangeComponentFinal :: [[Node]] -> [[Node]]
rearrangeComponentFinal listoflist | length listoflist == 1 = listoflist
                                   | otherwise = until (not . isThereStillDuplicates) (rearrangeComponent) listoflist
```

Figure 4.2.3: Modifications made to get the 2nd parallel version.

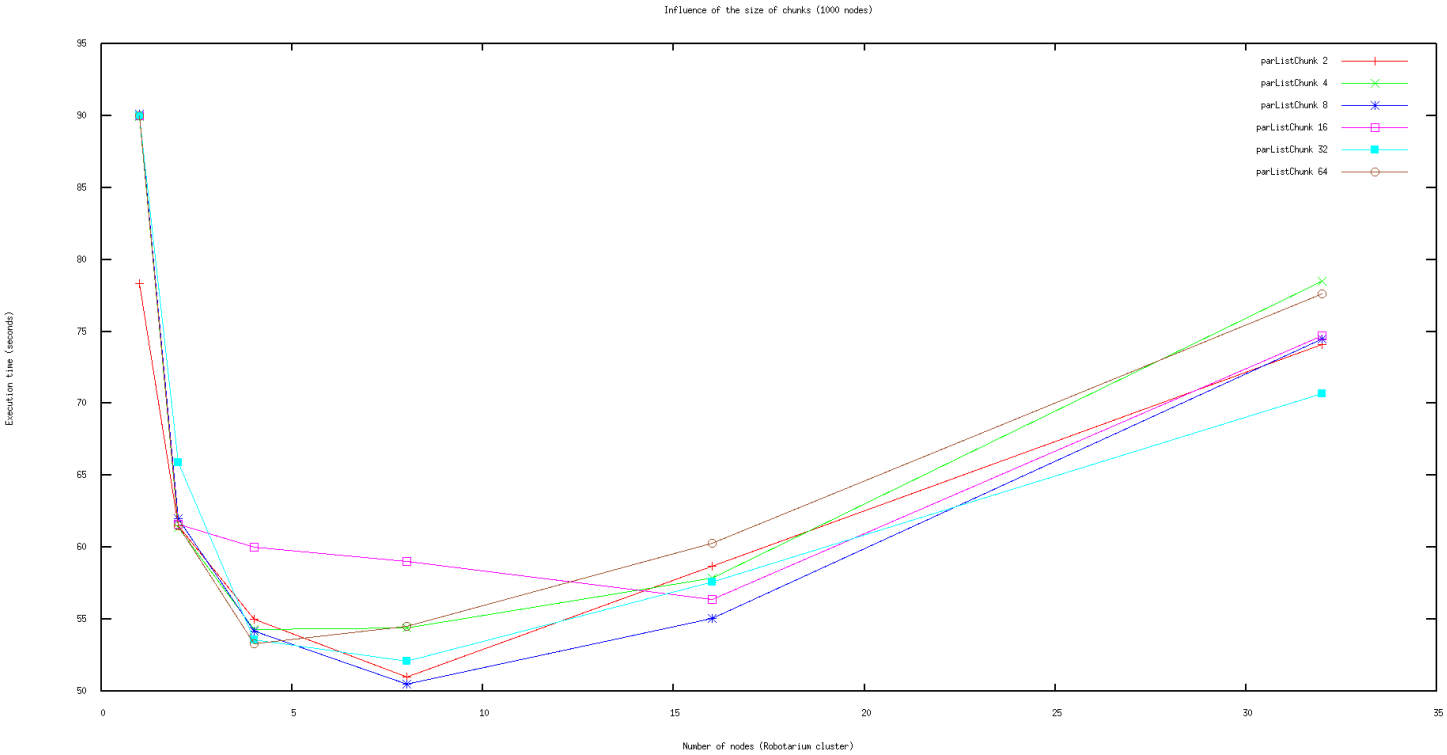


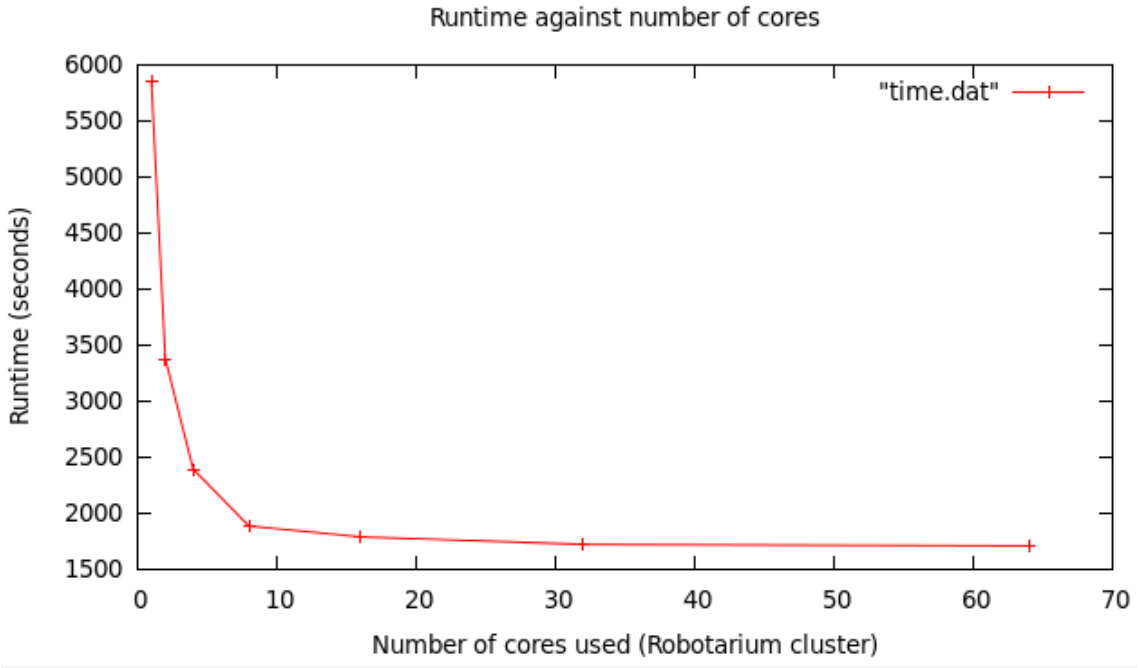
Figure 4.2.4: Runtime against number of cores using different sizes of chunks.

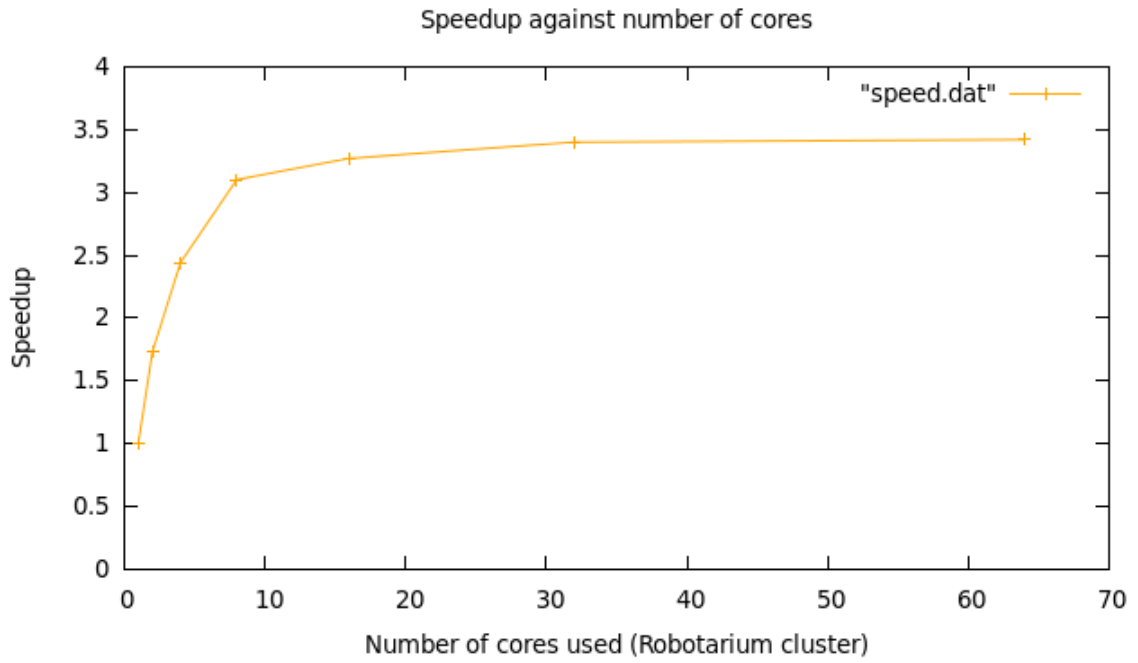
parListChunk 2
parListChunk 4
parListChunk 8
parListChunk 16
parListChunk 32
parListChunk 64

As we mentioned in the section 3.4.5 of the dissertation, that would be interesting to test the influence of the size of the chunks to keep the one that leads to the best results. The results below have been extracted using a graph of 500 nodes and 112 332 edges from [Info.univ-angers.fr, 2017].

As we can see in the Figure 4.2.4 above, a size of chunk of 2 or 4 would not be enough since these sizes lead to the worst results. A chunking size of 16 would lead to decent results when we are using a lot of cores but bad results below 16 cores. A chunking size of 64 is too large for getting good results. Between a chunking size of 8 and 32, the best results are obtained globally for 8 cores so we decided for the rest of this version to use a chunking size of 8.

We then introduced the concept of thresholding which is explained in the section 3.4.4 of this dissertation. Below a specific threshold, we stop running functions in parallel in order to stop creating threads where it is not necessary. As a default value, we set the threshold to 16 considering a chunk size of 8. If we have less than 16 components to treat, we do it sequentially which improves a bit the runtime but the difference is very small. Here are the results we obtained for this second parallel version using our graph of 4000 vertices (for runtime & speedup) and 1000 vertices (for GC) from [Info.univ-angers.fr, 2017].


Runtime against number of cores
"time.dat"
Runtime (seconds)
6000
5500
5000
4500
4000
3500
3000
2500
2000
1500
0
10
20
30
40
50
60
70
Number of cores used (Robotarium cluster)


Speedup against number of cores
"speed.dat"
Speedup
4
3.5
3
2.5
2
1.5
1
0.5
0
0
10
20
30
40
50
60
70
Number of cores used (Robotarium cluster)

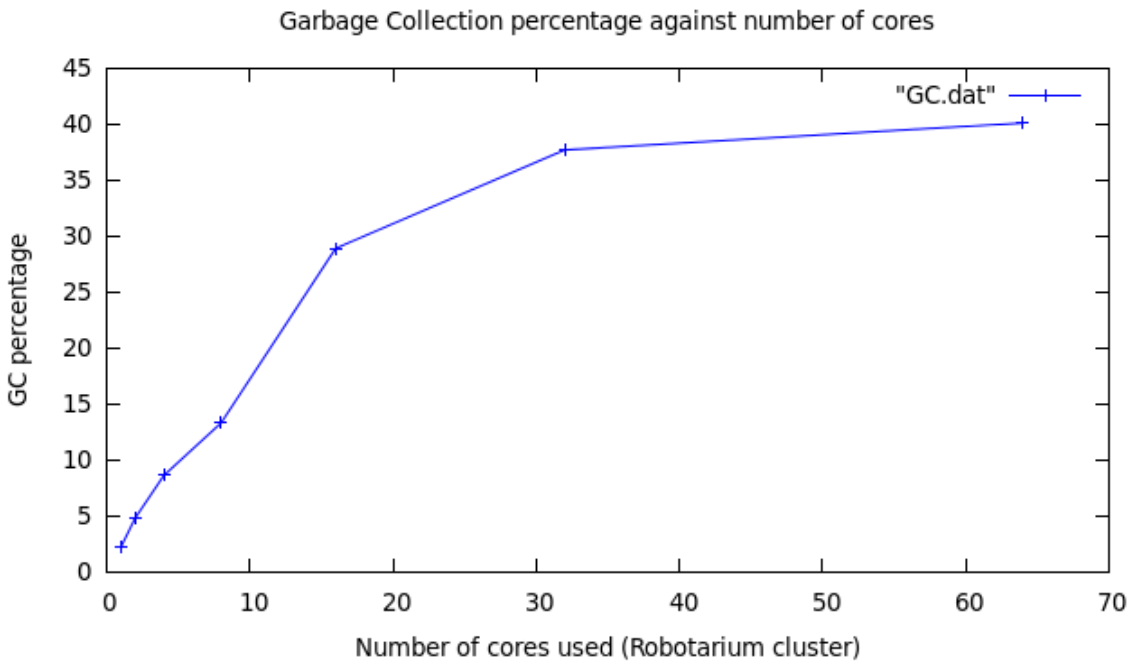


Figure 4.2.5: Results for the second parallel version (4000 vertices – 4000268 edges)

We can see in the first part of the Figure 4.2.5 above that the runtime from this 2nd parallel version is better than the one obtained from our first parallel version. Concerning the speedup, we do not notice a better speedup with 16 cores than in the first version. It is still disturbing to see that from 16 cores, its value almost does not get higher than its previous one.

We can observe a decrease of the garbage collection percentage using this version, as shown in the third part of the Figure 4.2.5. Indeed, it is lower than in the first version but still way too much since it still reaches about 40% using 64 cores.

We tried a classic data-oriented parallelism suing the keyword 'parMap', we tried to use a chunking strategy, which improved a bit the program, but still, it is not enough. We are spending way too much time on 'reArrangeComponentFinal' function. Actually, in this function we keep repeating the 'reArrangeComponent' function until we cannot find duplicates anymore. In the 'reArrangeComponent' function, for each element, we compare it with all other elements. It seems fine to do this when we have to deal with small graphs, but as soon as the graph gets bigger, there are a lot of components and comparing each element with all other elements is too time-consuming. We thought the keyword 'parListChunk' could help with this point but it did not give the expected result so we decided to refactor the code for the rearrangement, which our 3rd parallel version.

4.2.3. 3rd parallel version: Rearranging components

In this section, we discuss the modifications we made to build our 3rd parallel version. As mentioned earlier, we needed to fix this function of rearranging our components which consumed too much time.

We explain how we proceeded to make this version way more efficient than the previous one. In our previous version of the 'reArrangementComponent' function, we take one components, and we check with all other components if it shares common nodes. If yes, we merge the components. We illustrate this using the Figure 4.2.6 below:

[2	3	5	7]	[11	13	17	19]
[23	29	31	37]	[41	43	47	53]
[59	61	67	71]	[73	79	83	89]
[97	101	103	107]	[109	113	127	131]
[137	139	149	151]	[157	163	167	173]
[179	181	191	193]	[197	199	211	223]
[227	229	233	239]	[241	251	257	263]
[269	271	277	281]	[283	293	307	311]

Figure 4.2.6: Rearranging components before the 3rd parallel version.

In this rearrangement, we take the first list of nodes (first component) and we check with the second list (second component) if they share elements, then with the 3rd, and so on. Then we take the second component (2nd list), and we compare this list to the first one, the third one, and so on. And we repeat until the last component. It seems natural, that if the list of components is way bigger than this one, the runtime is considerably affected.

To fix this, instead of using the 'parListChunk' strategy as in the 2nd parallel version, we chunk in a first time the list of components, and we execute the same function than before, but here within the chunk, as described below:

[2	3	5	7]	[11	13	17	19]
[23	29	31	37]	[41	43	47	53]
[59	61	67	71]	[73	79	83	89]
[97	101	103	107]	[109	113	127	131]
[137	139	149	151]	[157	163	167	173]
[179	181	191	193]	[197	199	211	223]
[227	229	233	239]	[241	251	257	263]
[269	271	277	281]	[283	293	307	311]

Figure 4.2.7: Rearranging components in the 3rd parallel version.

In the Figure 4.2.7 above, we chunk the list of components using a size of 4, to keep 4 components in the first chunk, 4 components in the second chunk, and so on. When we introduce parallelism in this function, what we do is to give the first chunk to core 1, the second to core 2, and so on. In each chunk, the first component is compared to other components as like before, but it is way faster. Once all the chunks have been treated, we get a new list of components. If there are still duplicates in it, we chunk again the list to repeat the operation.

This modification in the code corresponds to:

```
-- Rearrange the components modified by the boruvka algorithm at each step
rearrangeComponent :: [[Node]] -> [[Node]]
rearrangeComponent listoflist | length listoflist == 1 = listoflist
                              | length listoflist < 32 = nub $ map (checkComponentDuplicate listoflist) listoflist
                              | otherwise = nub $ concat $ parMap rdeepseq rearrangeComponent (chunk 8 listoflist)

-- Rearrange the components (top-level function)
rearrangeComponentFinal :: [[Node]] -> [[Node]]
rearrangeComponentFinal listoflist | length listoflist == 1 = listoflist
                                   | otherwise = until (not . isThereStillDuplicates) (rearrangeComponent) listoflist
```

Figure 4.2.8: Modifications made for rearranging components.

As described in the definition of the 'reArrangeComponent' function, instead of using the function upon the list of components, we apply the function upon the list previously chunked. The results that we obtained using a graph of 4000 nodes (runtime) and 1000 nodes (GC) from [Info.univ-angers.fr, 2017] are:

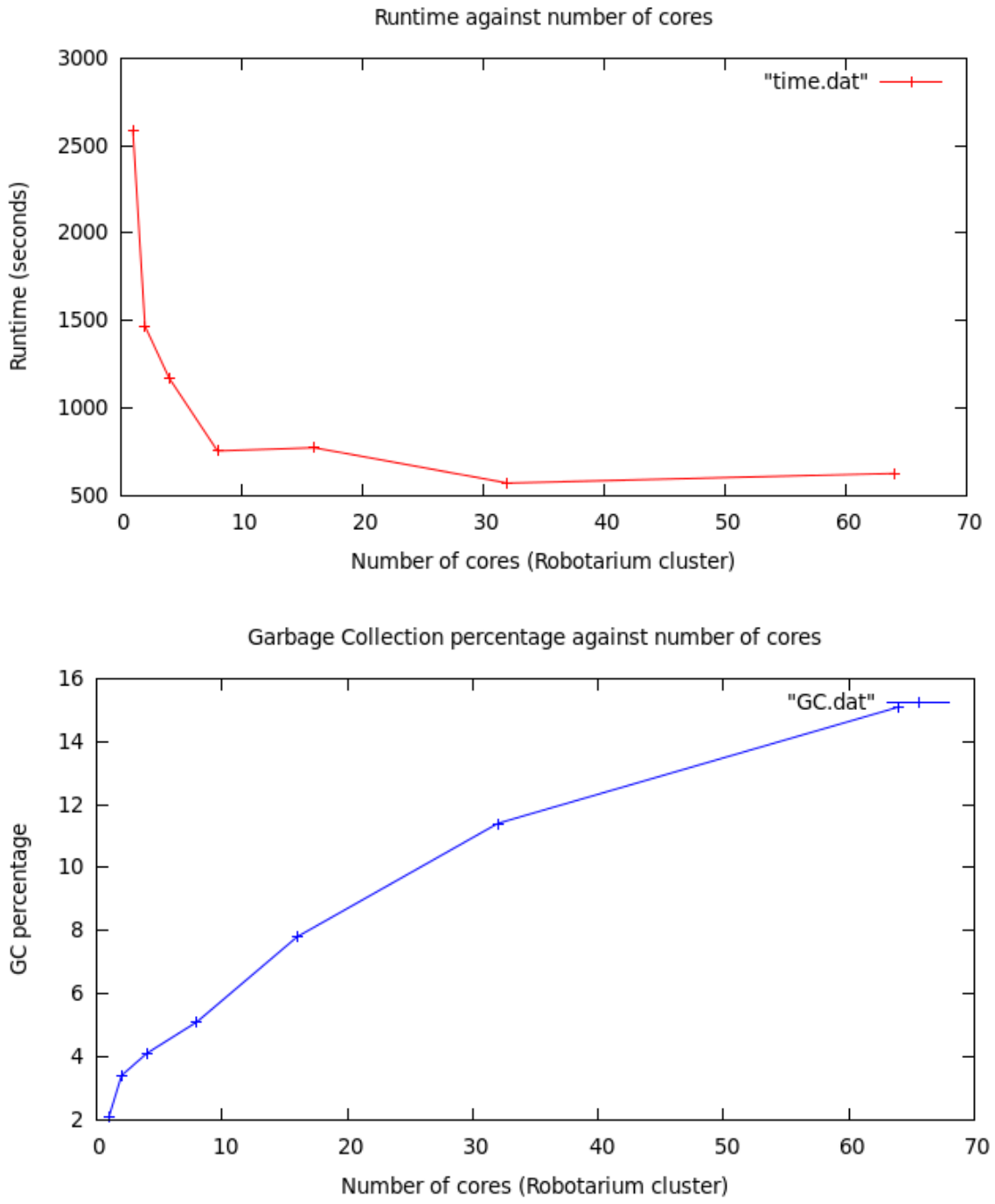


Figure 4.2.9: Runtime and GC percentage of the 3rd parallel version.

This version is a real improvement compared to the previous version. Indeed, the best runtime for the previous version was reached using 64 cores. With only cores, we can do better than this runtime using this version, which is nice. Besides, the Garbage Collection percentage has been considerable improved with this version, as shown in the second part of the Figure 4.2.9. In the last version, the worst case was almost 40% using 64 cores while here, it is “only” 15% of the computation.

Now that we improved our way of rearranging the components, we focus on how to improve the other major part of the program, which is the search of the minimum edge linked to a component, as explained in the section 4.1.6 of this dissertation.

4.2.4. 4th parallel version: Minimum Edge

In this section, we explain how we improved this major part of the computation, which is how to get the edge with has the minimum weight linked to each component. In the Figure 4.2.10 below, we have two code sources: the first one describes the functions used in this task of getting the minimum edge while the second shows the modifications to avoid wasting time for our 4th parallel version.

```
-- Returns a list of edges with the same weight than given
findWedgefromWeight :: Wgraph -> Float -> [Wedge] -> [Wedge]
findWedgefromWeight graph value edges = filter (\e -> weight e == value) edges

-- Returns a list of weighted edges linked to the given node in a graph
linkedWedges :: Wgraph -> Node -> [Wedge]
linkedWedges graph n = filter (\e -> n `elem` twoNodes e) (getEdges graph)

-- Returns a list of weighted edges linked to a given component within a graph
linkedComponent :: Wgraph -> [Node] -> [Wedge]
linkedComponent g nodes = filter (\w -> not (((head (twoNodes w)) `elem` nodes) &&
                                            (((twoNodes w)!!1) `elem` nodes)))
                                                (concat (parMap rdeepseq (linkedWedges g) nodes))

-- Returns the less weighted edge linked to a given node
minLinkedWedges :: Wgraph -> Node -> Wedge
minLinkedWedges graph n = head (findWedgefromWeight graph
                                        (minWeight (linkedWedges graph n)) (linkedWedges graph n))

-- Returns the less weighted edge linled to a given component
minLinkedComponent :: Wgraph -> [Node] -> Wedge
minLinkedComponent graph component = head (findWedgefromWeight graph
                        (minWeight (linkedComponent graph component)) (linkedComponent graph component))
```

```
-- Gives the less weighted edge
minWeight :: [Wedge] -> Wedge
minWeight edges = minimumBy (compare `on` weight) edges

-- Returns a list of weighted edges linked to the given node in a graph
linkedWedges :: Wgraph -> Node -> [Wedge]
linkedWedges graph n = filter (\e -> n `elem` twoNodes e) (getEdges graph)

-- Returns a list of weighted edges linked to a given component within a graph
linkedComponent :: Wgraph -> [Node] -> [Wedge]
linkedComponent g nodes = filter (\w -> not (((head (twoNodes w)) `elem` nodes) &&
                                            (((twoNodes w)!!1) `elem` nodes)))
                                                (concat (parMap rdeepseq (linkedWedges g) nodes))

-- Returns the less weighted edge linked to a given node
minLinkedWedges :: Wgraph -> Node -> Wedge
minLinkedWedges graph n = minWeight (linkedWedges graph n)

-- Returns the less weighted edge linked to a given component
minLinkedComponent :: Wgraph -> [Node] -> Wedge
minLinkedComponent graph component = minWeight (linkedComponent graph component)
```

Figure 4.2.10: Modifications made for getting the minimum edge

As we know from the sequential profiling, the function ‘minLinkedComponent’ is largely used in the program. We have seen that for a graph of 1000 nodes, the function has been called 1002 times which is not necessarily a lot but it is quite a time-consuming function because in this function, we use the ‘linkedComponent’ function. This function represents the biggest part of the computation of the task since its purpose is to give all the weighted edges which are linked to a specific component, without the edges which are intern to a component, meaning all the edges relying node A & node B while A & B both are in the component. This spends a lot of time because for each node of the component, we check at each weighted edge linked to it if the other node sharing the edge is not in the component.

How I get the minimum edge liked to a component was a bit complex before the modifications I made in this new version. Indeed, as shown in the first part of the Figure 4.2.10, I always use the ‘findWedgeFromWeight’ function which is not very convenient because: Firstly, it is not relevant, because I find the minimum weight of all the edges linked to a component and then I search for all the edges linked to the component which have this weight, and I take the head of the list. All this process consumes resources. Secondly, besides the fact that it consumes a lot of resources, it does not make sense to get a minimum weighted edge by getting only its weight and then, do a new search to look at all the edges linked to the component which have this weight, and take the first one coming up. It was using this way that I returned the minimum edge linked to a node, and when I introduced the concept of component, I kept this principle while it was not relevant.

In the second part of the Figure 4.2.10, we can see that I directly return a weighted edge instead of a weight. Indeed, I use the function ‘minWeight’ to return the less weighted edge among all the edges linked to the component. It seems more natural and simpler, as we can see with the results using a graph of 4000 nodes (runtime) and 1000 nodes (GC) from [Info.univ-angers.fr, 2017]:

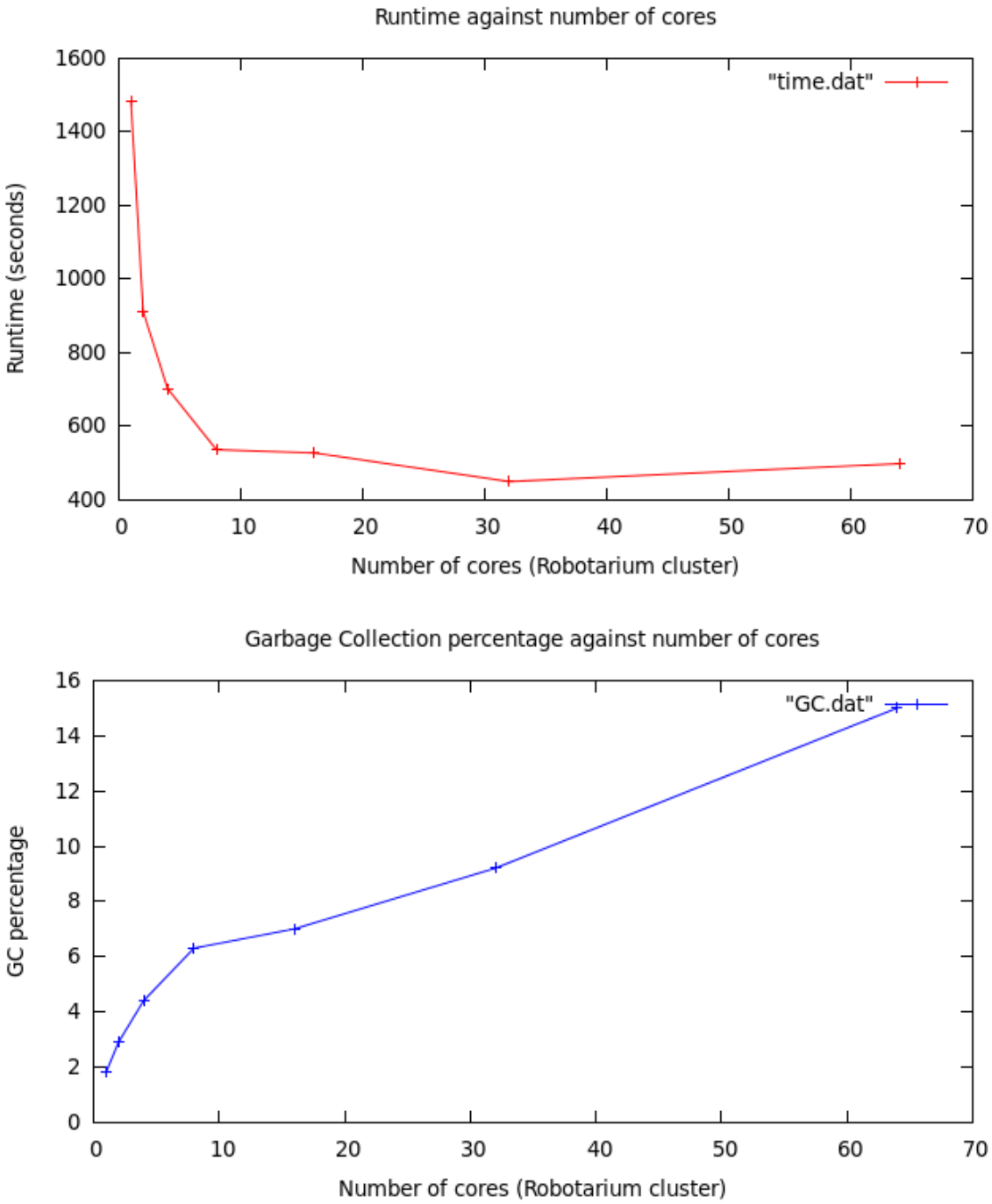


Figure 4.2.11: Runtime and GC percentage for the 4th parallel version.

As we can see in the Figure 4.2.11 above, the runtime is still improving thanks to the modifications made during this version. Indeed, we divided the runtime of the previous version by 1.6 which is a good improvement. We still have a problem since the speedup is not as we would like it to be. From a specific number of cores, the runtime does not improve anymore. The best runtime is reached using. I think 64 cores is maybe too much for this kind of graph.

Concerning the garbage collection percentage, we cannot see a real difference between its value from the previous version and the one from this version. The modifications we brought to the program was mostly dedicated to decrease the runtime but we did not expect an improvement in GC since we did not fix here problems of heavy computation such as in the previous version with the rearrangement of the components.

4.2.5. 5th parallel version: Adapting parallelism

In this fifth and final section of this part of the dissertation, we introduce the last modifications we made to our implementation. As stated above in the results of the 4th parallel version, we managed to improve the runtime but the garbage collection percentage was not really affected by the modifications. We try here to reduce this garbage collection percentage and our runtime as well, if it is possible.

To do this, our supervisor advised us to adapt the parallelism to the number of cores that we are using for the computation. We are talking here about changing the size of the chunks when we use the data-oriented parallelism.

Indeed, instead of creating several threads for each chunk of size 8 as we saw before, we used 'parListChunk' with the chunking size that we actually need.

To give a simple example, we imagine that we have 1000 elements to treat and 10 cores provided to do the computation. The idea is to divide the number of elements by the number of cores which gives the chunking size to adopt, which would give 100. Here is the new source code for the version (the whole implementation is in the Appendix):

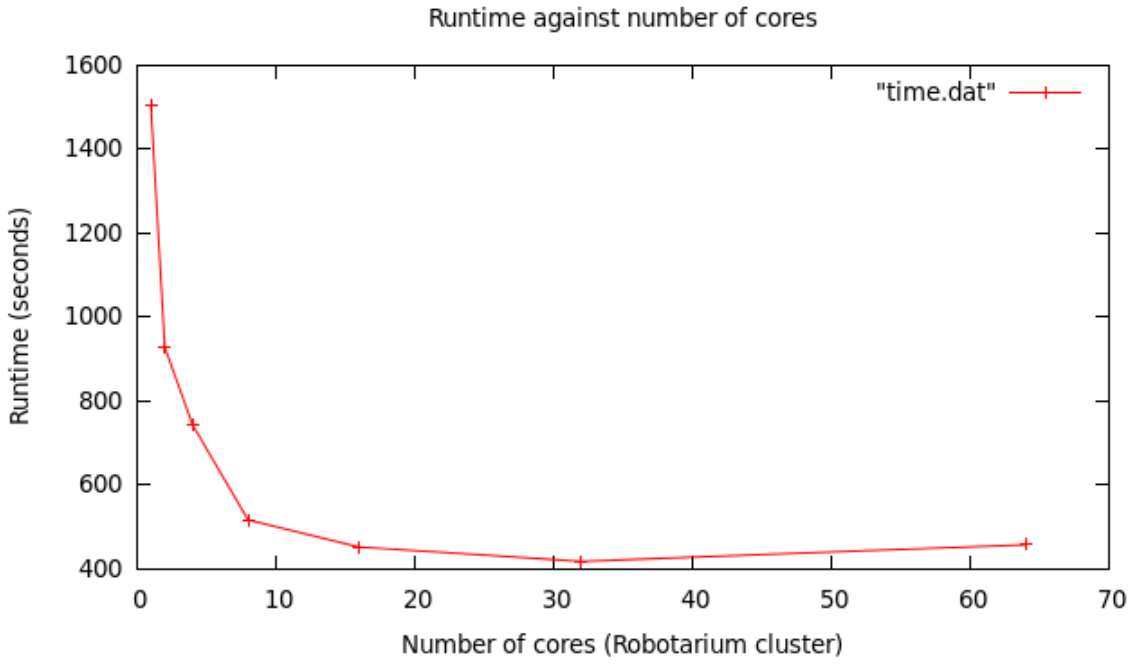
```
cores = fromIntegral numCapabilities
```

```
-- Recursive boruvka algorithm which ends when the list of component is only composed of one component (all the nodes)
boruvkaAlg :: Wgraph -> [[Node]] -> [[Node]] -> [Wedge] -> (Wgraph, [[Node]], [[Node]], [Wedge])
boruvkaAlg g components real_comp wedges | real_comp == [getNodes g] = (g, [], [], wedges)
                                         | otherwise =
    let
        currwedges' | wedges == [] && length components < cores = nub $ parMap rdeepseq (minLinkedComponent g) components
                    | wedges == [] && length components >= cores = nub (map (minLinkedComponent g) components `using`
                                                                        parListChunk (length components `div` cores) rdeepseq)
                    | wedges /= [] && length real_comp < cores = nub $ parMap rdeepseq (minLinkedComponent g) real_comp
                    | wedges /= [] && length real_comp >= cores = nub (map (minLinkedComponent g) real_comp `using`
                                                                        parListChunk (length real_comp `div` cores) rdeepseq)
```

Figure 4.2.12: Modifications made to get the last parallel version.

The first part of the Figure 4.2.12 above is the source code to get the number of cores that we are using to do the computation. In the second part of the Figure 4.2.12, we introduce the parallelism strategy that we stated earlier by taking the length of the list of components and divide it by this number of cores provided for the computation. We have then the correct number of cores treating a correct number of chunks. The same principle is applied to the rearrangement of the components as well.

By doing that we use as less threads as possible and the workload handled by each processor is bigger than in the previous versions. It avoids creating too much threads and this leads to a better garbage collection percentage, as shown in the results below (obtained using a graph of 4000 nodes (runtime & speedup) and 1000 nodes (GC) from [Info.univ-angers.fr, 2017]:



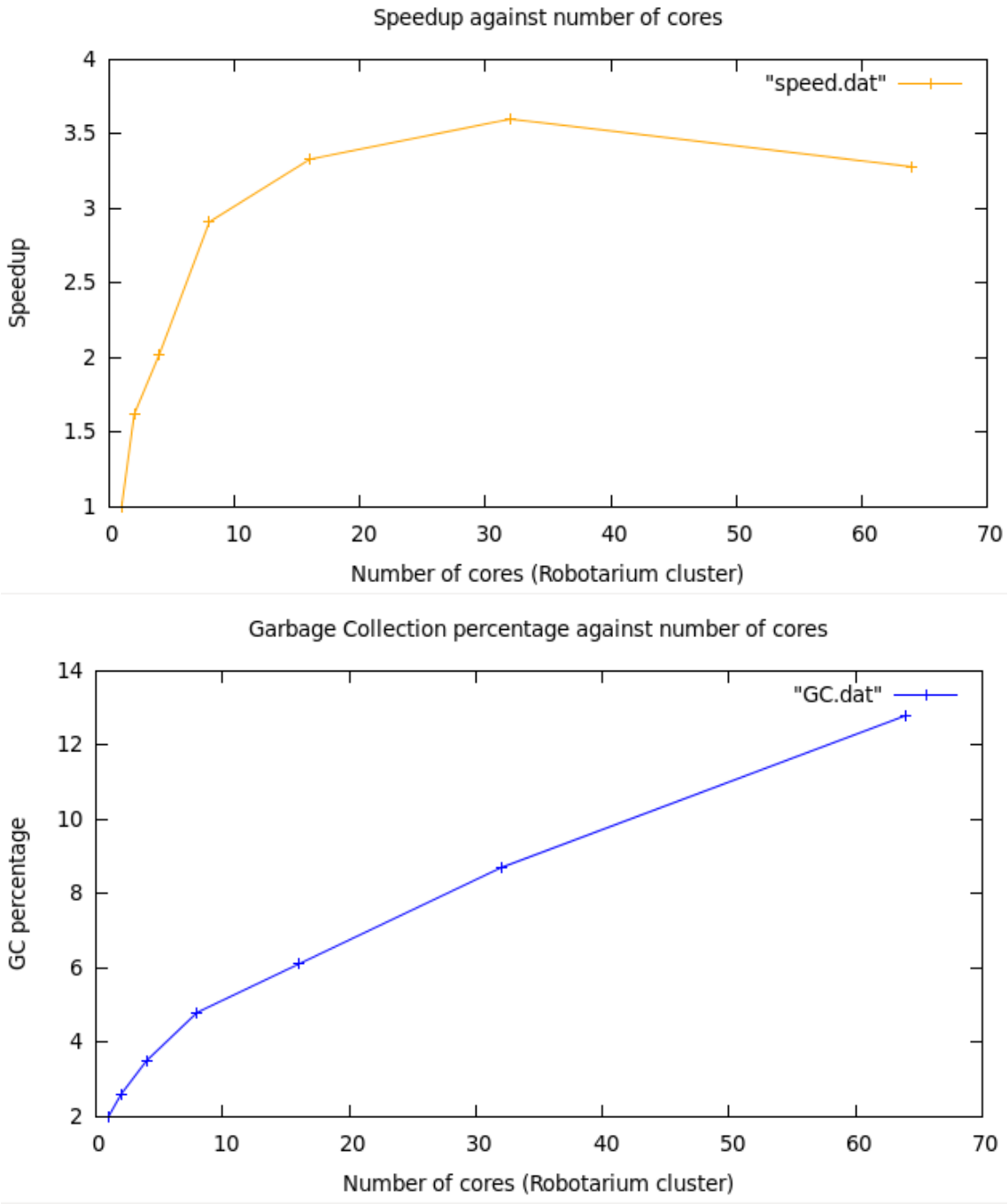


Figure 4.2.13: Runtime, speedup & GC percentage graphs of the last parallel version.

As we can see in the first part of the Figure 4.2.13, showing us the runtime against the number of cores, this modification slightly improved the runtime compared to the previous version when we look at it generally. It is relevant to highlight the fact that even if the previous version is better using 1,2 and 4 cores, this version is better using 8, 16, 32 and 64 cores which is nice since we aim to improve our parallelism strategy.

Like the previous versions, we get the best speedup using 32 cores which seems to be an ideal number of cores to use. Maybe a way bigger graph would set 64 cores as the ideal number. We can notice a slight improvement in the second part of the Figure 4.2.13, our speedup, which does not seem to remain at the same value from 8 cores to 16, even if our speedup does not reach good values for these number of cores.

Finally, a good improvement is the decrease of the garbage collection percentage in the third part of the Figure 4.2.13. As expected with the modifications we made, we managed to make it slow down until 12% which is very good for 64 cores when we compare with other previous versions.

It is now time for us to get an overview of all the results we made, by merging our versions and see what conclusions we can draw.

4.3. Results

To begin this section, we introduce the specifications of the machines we used for our tests and all the results. The results that we obtained for the sequential implementation and for the different parallel versions have been calculated using the machines available in the Robotarim cluster of the School of Mathematical and Computer Science of Heriot-Watt. Among the different machines available, we entirely used 4 processors of this kind in our computations:

CPU Model Name	AMD Opteron 6376
Frequency	2300 MHz
Turbo frequency	2600 MHz
Bus speed	3200 MHz HyperTransport links (6.4 GT/s per link)
Socket type	Socket G34
Data width	64 bit
CPU cores	16

Threads	16
Level 1 cache size	8*64 KB 2-way set associative shared instruction caches 16*16 KB 4-way set associative data caches
Level 2 cache size	8*2 MB 16-way set associative shared exclusive caches
Level 3 cache size	2*8 MB shared caches

Table 4.3-1: Specifications of the CPU used for the computations

4.3.1. Comparison Sequential/Parallel implementation

This section is dedicated to build a comparison between our sequential implementation and our first parallel version to show what we can obtain from a first parallel version using very basic strategies.

We first compared the runtimes that we got from these two versions, using a graph of 4000 nodes from [Info.univ-angers.fr, 2017]:

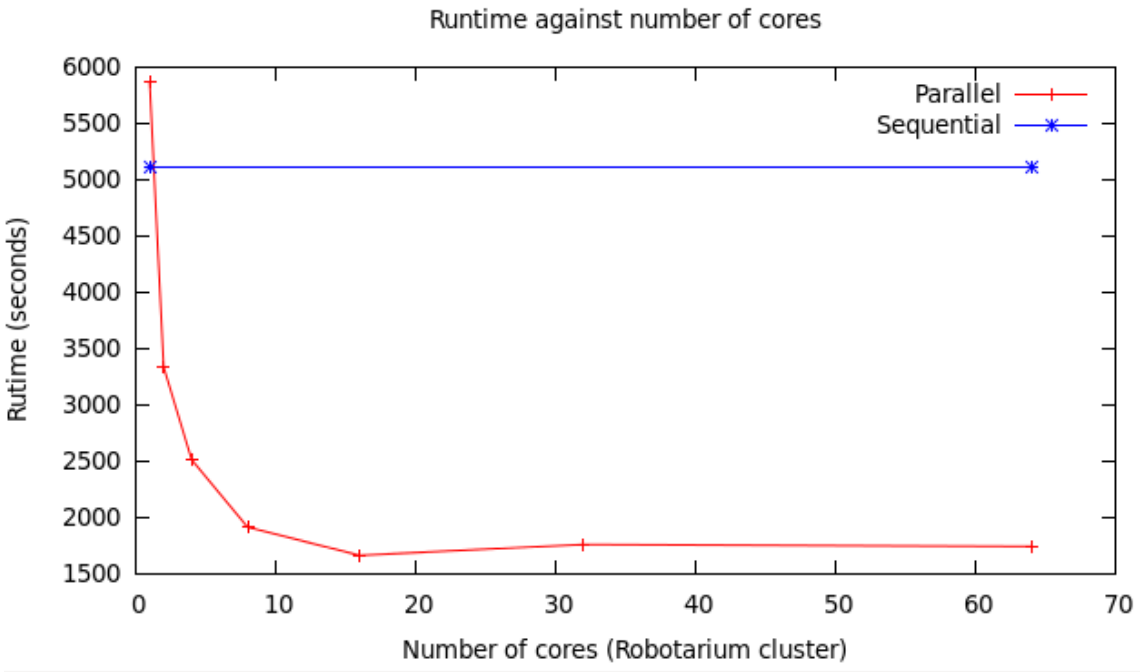


Figure 4.3.1: Comparison between sequential and first parallel version.

As we can see in the Figure 4.3.1, even with very basic strategies applied to a sequential implementation, we can obtain, as expected, decent runtimes compared to the one of the sequential version. Obviously, the runtime of the sequential version does not change against the number of cores since the program is not designed to take in account more than 1 processor.

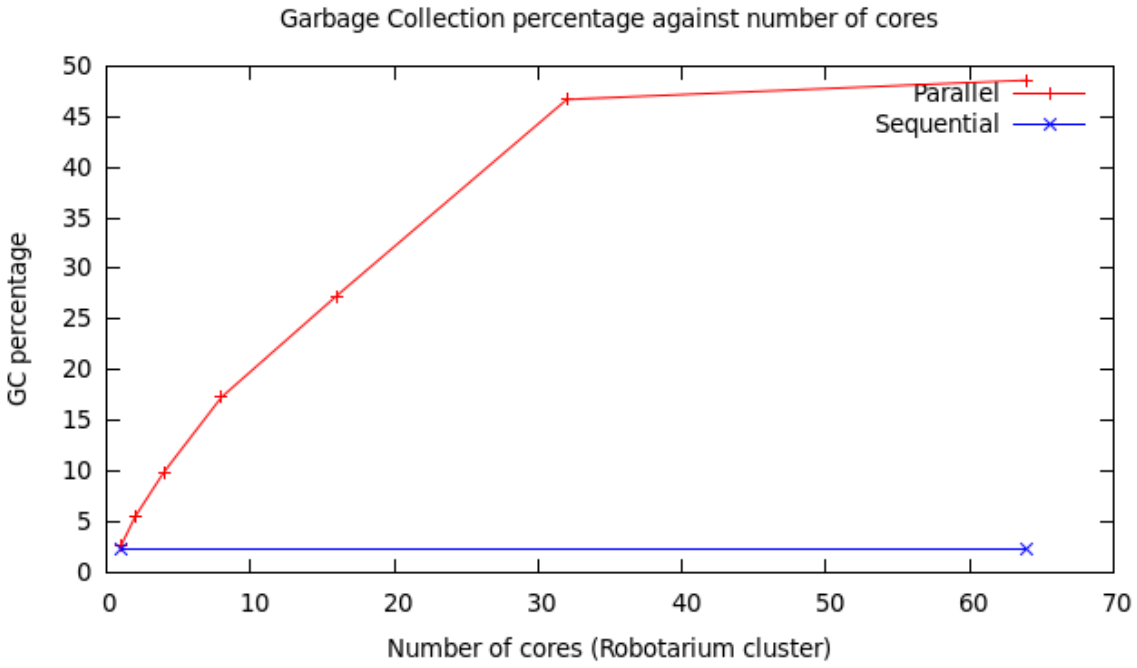


Figure 4.3.2: Comparison GC percentage between parallel and sequential version

Wee in the Figure 4.3.2 that we have a clear problem of garbage collection using basic strategies, our garbage collection for both versions using 1 core are similar. However, as soon as we increase the number of cores to do the computation, the percentage gets higher too quickly. It almost seems that doubling the number of cores leads to doubling the percentage using this first parallel version. To fix this, we developed other parallel versions that we compare below in the dissertation.

4.3.2. Comparison of Parallel implementations

In this section, we give details about the performance of our different parallel implementations. As stated in the project evaluation, the project is evaluated by accessing the

performances of the program. To access performance, we compared the different implementations using runtime, speedup and GC percentage graphs.

4.3.2.a. Runtime

In this section, we compare the different parallel algorithms we developed during the project, basing our discussion on the performances that we obtained from each one.

To assess these performances, we drawn all the runtimes from each version on the same graph to see the progression through the versions, using a graph of 4000 nodes from [Info.univ-angers.fr, 2017]:

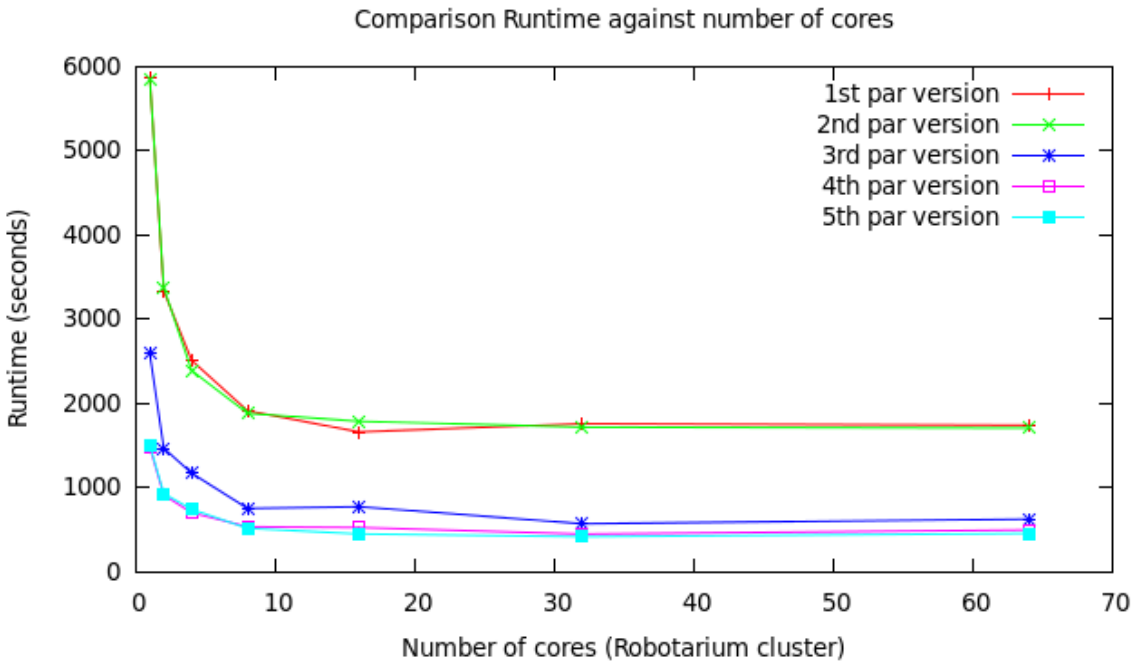


Figure 4.3.3: Comparison of runtimes between parallel versions

As shown in the Figure 4.3.3, we managed to reduce our runtime significantly from the first parallel version to the last one. This figure clearly highlights the different versions and their purposes. Indeed, we can see that there is not a big difference in terms of runtime between the first version and the second one. This result was expected since, as we said in the section 4.2.2 of this dissertation, we developed the second parallel version to reduce a bit our runtime for sure but above all to fix the garbage collection problem we encountered (creating too many threads hanging on data in the heap), as we will see later using the GC percentage graph.

The best modification the algorithm has known is obviously from the second version to the third one. As we explained it in the section 4.2.3 of this dissertation, instead of using a data-oriented parallelism using chunking strategy, we chunked first the dataset, to use a data-oriented parallelism after, which makes a big difference. Thanks to this modification, we managed to divide the runtime of the 2nd version by 2.3 in average which is a good ratio.

Then, the modification made during the 4th parallel version about getting more quickly the minimum edge linked to a given Component saved us some more time. As we explained in the section 4.2.4, we did not expect any improvement in terms of garbage collecting, but we did expect an improvement in runtime, which, as we can see in Figure 4.3.3, is not negligible. Even though it is hard to notice it using the Figure 4.3.3, we managed to divide the runtime of the 3rd version by 1.4 in average to get the runtime of the 4th version.

Finally, the last version is not quicker than the 4th version because the modifications we made were not meant to reduce the speedup, so this runtime was expected. However, we might see a change in our garbage collection percentage, which is our next part.

4.3.2.b. Garbage Collection

In this section, we display the different results of our tests dedicated to garbage collecting.

To access the performances of our parallel implementations in terms of garbage collecting, we draw all the GC percentages we collected, using a graph of 1000 nodes from [Info.univ-angers.fr, 2017]:

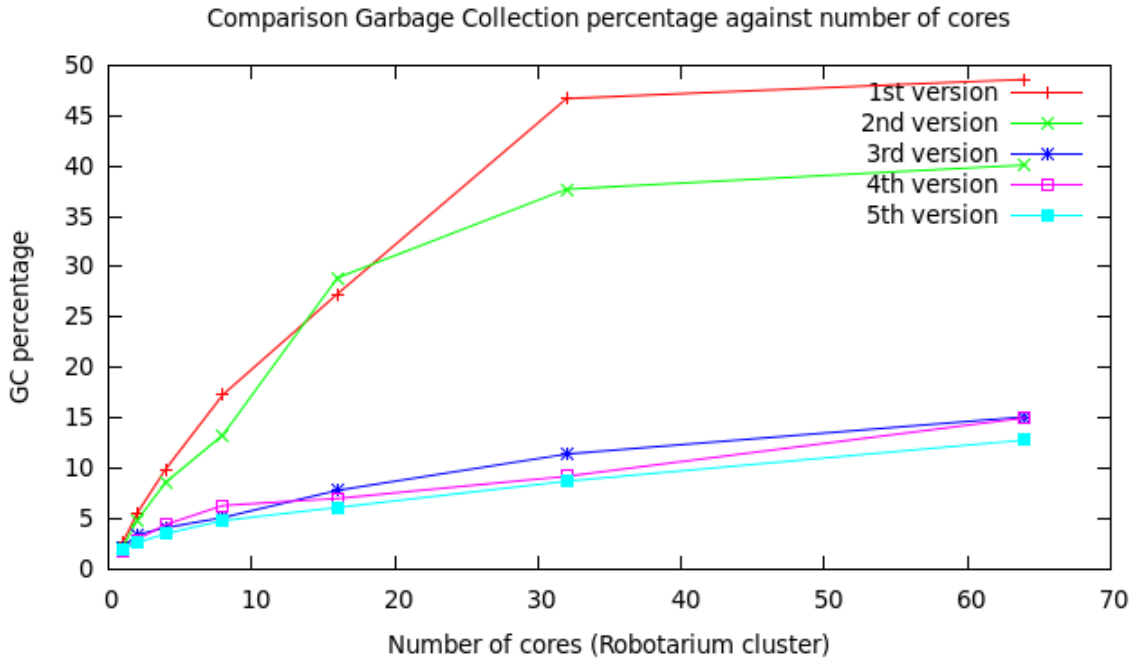


Figure 4.3.4: Comparison of garbage collection percentages between parallel versions.

Using the Figure 4.3.4, we see that we managed to considerably reduce the Garbage Collection in our program. Actually, it is easier to reduce its proportion when its starts from a very high value. In the first parallel version, almost half of the computation time is dedicated to garbage collection, using 32 and 64 cores, which is way too high. As explained in the section 4.2.2, the objective of the second version was to reduce a bit the runtime, which was not so successful, but more principally to reduce this garbage collection percentage.

Even though the garbage collection percentage for the second parallel version has slightly decreased using a small number of cores (below 16 cores), we can see that from 32 cores, the percentage gets knows a clear improvement. As we explained in the section 4.2.2, this is due to the fact that instead of creating too many threads like in the first version, we arrange the workload for each thread in order to hang less on data.

Obviously, the main modification has been made with the 3rd parallel version. We have seen this using the runtime graph, but it is still the case for garbage collecting. Indeed, the worst percentage in the 2nd version is 40% for 64 cores while the worst percentage in the 3rd version is 15% also for 64 cores. As we explained in the section 4.2.3, since we chunk the list of components and then we execute recursive functions upon it, it saves a lot of resources. Indeed, the amount of work to be done before was huge, particularly if the graph input gets bigger, while

in the third version, the runtime is way less affected. The heap is not overloaded, and since we reduce our garbage collection percentage, automatically, the runtime reduces.

Concerning the 4th version, we focused here on runtime improvements since we simplified the way we get the minimum edge linked to a Component. The heap is not a lot affected here, so as expected, there is not a big difference in terms of garbage collection between the two versions.

Regarding the last parallel version, as we stated in the section 4.2.5, we did not focus here not on the runtime, but on GC instead since we tend to adapt the parallelism to the number of cores we are using for the computation. By doing that, we create as many threads as we really need, so the heap is a bit alleviated.

4.3.2.c. Speedup

In this section, we explain the different results we obtained from the tests of speedup made using the parallel versions.

Like in the comparison of the runtime (section 4.3.2.a), we used a graph of 4000 nodes from [Info.univ-angers.fr, 2017] to access the versions. Here are the results using different number of cores:

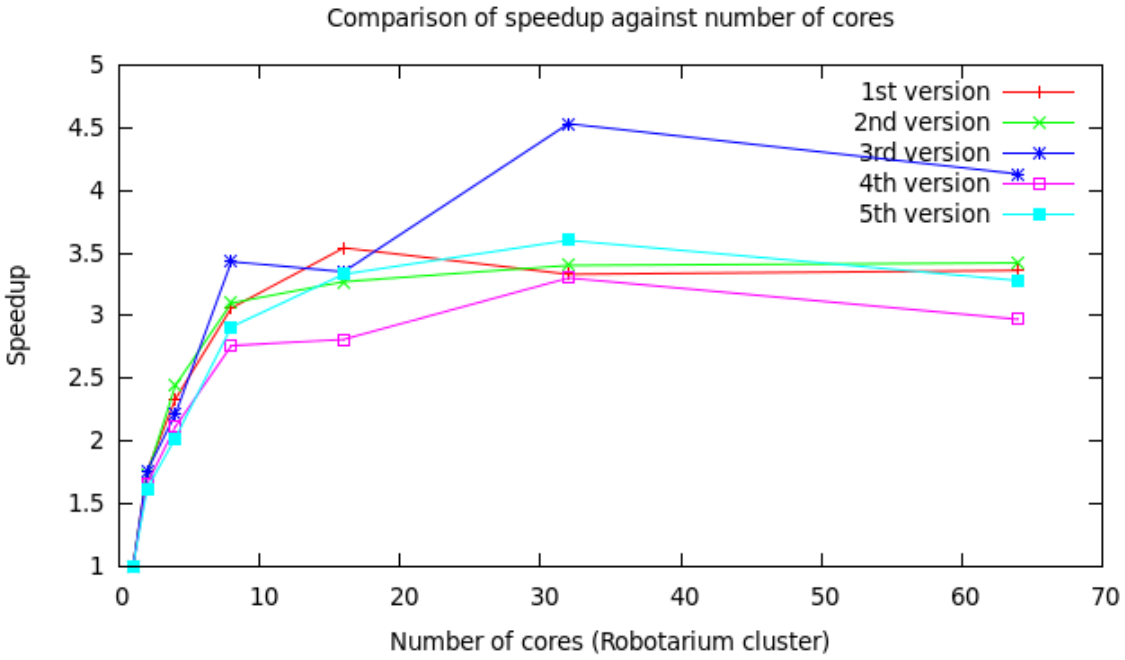


Figure 4.3.5: Comparison of speedup for different parallel versions.

As shown in the Figure 4.3.5 above, one of the worst speedup graph that we can obtain is the one from the 4th version. This was expected, because as we explained in the section 4.2.4, the modification we made in this fourth version concerned sequential tasks. Indeed, we improved the way of getting the minimum edge linked to a Component. Thanks to this modification, we saved some time, but this time was not included in the parallel time. Since we do not save some time in a parallel task, the sequential time decreases, while the parallel time remains at the same value. That is why the runtime decreases and the speedup decreases too.

Concerning the first and second versions, there is not a clear distinction between the two speedup graphs. This is due to the fact that the second version was mainly implemented to fix this massive problem of data in the heap. We hoped to get a distinction in runtime but it did not happen. However, we did get an improvement in Garbage Collecting but it does not change the speedup.

Obviously, the best speedup graph from the five parallel algorithms is the one from the 3rd version. We managed to get 4,5 using 32 cores and about 4 using 64 cores, which is not a good speedup though but it becomes a good one compared to the other versions.

Finally, for our last version, we made modifications to adapt the parallelism. Therefore, the sequential time of the algorithm has not been modified compared to the parallel one which

has been slightly improved. That is why we have a slight improvement in the speedup of the 5th version compared to the 4th version.

4.3.3. Recap & Reflection

In this section, we tried to summarize all the results we obtained from our different tests. Then, we made a statement of our opinion basing our reflecting on the results and our experience in the project to step back and look at the overall picture of what worked and what did not.

To give an overview of what we have produced in the project, we built a table below summarizing the best runtimes, speedups and garbage collection percentages that we could obtain using all the sequential & parallel versions we developed within the project. The runtime and speedup were measured using a graph of 4000 nodes while the GC percentage was measured using a graph of 1000 nodes, both graphs were taken at [Info.univ-angers.fr, 2017].

Versions	Runtime (seconds)	Speedup	Garbage Collection percentage (using 16 cores)
Sequential	5111,7	1	
Parallel Mapping	1662,5	3,54	27,3
Chunking/Thresholding	1707,6	3,42	28,9
Rearranging Components	570,8	**4,53**	7,8
Minimum Edge	449,3	3,30	7,0
Adapting Parallelism	**417,7**	3,60	**6,1**

Table 4.3-2: Recap of the performances from all versions.

As we have seen through the sub-sections of this major part, we went through many different versions in this project. We saw that each version was developed in order to fix something in the project, as we will develop later in the conclusion of this dissertation.

The table 4.3-1 above displays the different performances of each version. We can see that from a runtime of 5111,7 seconds for a graph of 4000 nodes and about 4000000 edges, we managed to turn it into 417,7 seconds which is a nice improvement.

Concerning the speedup, I must admit that the measurements do not satisfy me. Indeed, even the max speedup that we measured during the project, 4.53 for the 3rd parallel version using 32 cores, is not enough. A speedup of 4.53 would be satisfying using 8 cores, but not 32. As we have seen through the different versions, from a particular number of cores, we had difficulties to not remain at the same speedup. Go beyond this value is the missing piece of this project.

I consider our improvement in garbage collecting as one of the major best successes within the project. The garbage collection percentage of the first parallel version was measured at 48,6%. Almost half of the computation time dedicated to garbage collecting is terrible in terms of performance. Thanks to all the modifications we brought to the implementations, specially the 2nd, 3rd and 5th parallel versions, we managed to turn this percentage into exactly 12,8% using 64 cores. Considering the fact that the program starts to get a bad behaviour using 64 cores, measuring 12,8% is an achievement to me, since it remains at a reasonable value.

Even though the 3rd version gave the best speedup graph, the best overall version was the 5th one since we measured the best runtime with this one. Since we add one feature at each version, it seems natural that the last one is the most effective. We managed to seriously reduce the runtime of the sequential version and the GC percentage of the first parallel version. However, it is a bit frustrating to fail in increasing the speedup. Here is a thread profile of the last version using a graph of 1000 nodes and 1799532 edges from [Info.univ-angers.fr, 2017]:

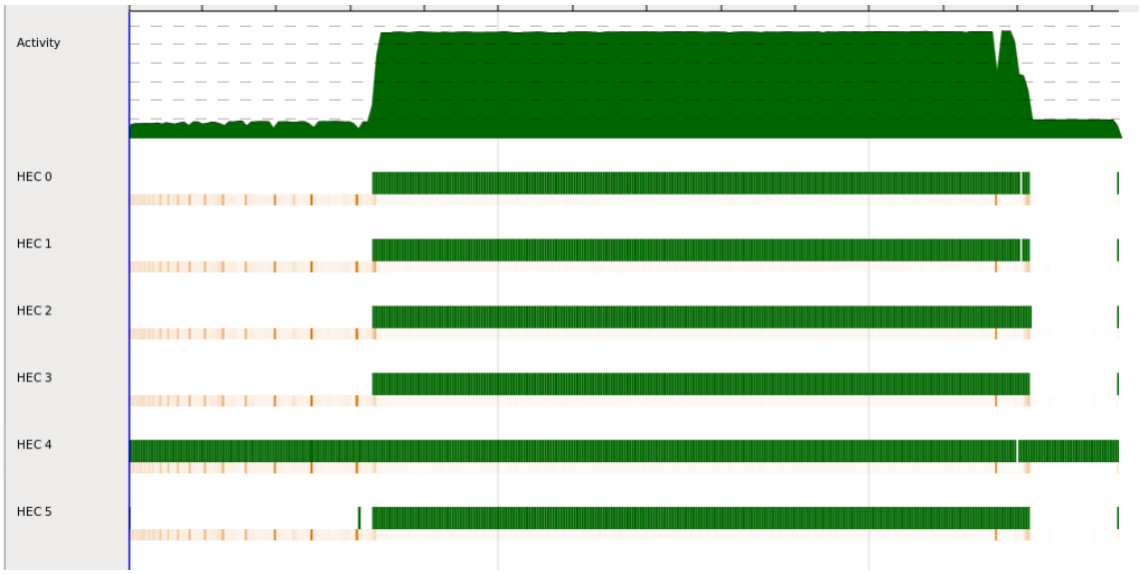


Figure 4.3.6: Thread profile using 6 cores.

The thread profile here in the Figure 4.3.6 is composed of two different parts: the sequential part is dedicated to the Input/Output, in which we create the graph using the text file provided in input while the parallel part represents the Boruvka's algorithm. In the figure, the parallel part seems to work fine. Indeed, there is a good balance in the workload of each core. That would be perfect if we could reproduce this kind of profile using more than 32 cores, which is the missing piece of the puzzle. There are still improvements to be done in this project in terms of granularity, which will be discussed in the Future work.

Chapter 5

5. Conclusion

5.1. Summary

This project gave us the opportunity to discover new fields such as graphs algorithms and parallel programming in a functional environment. We had the chance to go deep in the understanding of graph theory which is a complex but promising field. We also had the chance to explore the properties of a functional language, Haskell, and to use the tools provided by GpH an extension of Haskell, making parallelism possible.

In this dissertation, we covered many different subjects. First of all, we introduced the subject giving an overview of the project, the objectives to reach in terms of deliverable. Then, we gave details about the different issues that we may encounter such as the tools we used and the license of the source code created. Finally, we explained how we evaluated the project, using speedup graphs as an example.

Secondly, the Literature Review helped us to have a clear vision of what already existed in the Literature about our fields. We first understood the main characteristics of the functional programming language that we used. Then, we gave detail about the principles of graph theory that was involved in the project, which was helpful to understand how the algorithm worked and how we could exploit it. Finally, we gave the reader an overall view of how to use Glasgow parallel Haskell, which actually helped us later in the implementation phase.

Thirdly, we went through the different versions of the algorithms. We described in a first time the sequential version and its results. Then, we explained each parallel version that we developed with their results to show the improvements in our performances. We finished this section by displaying the performances we measured. The objective was to compare our versions and discuss the results.

By looking at the overall picture, I would say that this project was quite challenging since I had to face difficulties I have never experienced before. Indeed, I had a poor background in

functional programming so it forced me think differently than in my past projects. As I previously said, I sometimes took afternoons to code a function that I would have coded in 20 minutes in an object-oriented language like Java, because these two ways of programming are completely separated. I could not think as I was used to think, but now I am satisfied with this discovery of the functional programming because, once I got over the syntax, Haskell and other programming languages, seem maybe more natural to write programs, specially algorithms close to mathematics like graph algorithms.

This project made me familiarize with another strange field to me, graph theory. I knew what could model this kind of structure, and some of the basic algorithms that could be run upon these graphs. However, I had not experienced a project in the past in which I had to choose a specific problem, study the different algorithms capable of solving this problem and choose to implement one among other for a specific reason. Here I chose to solve the MST problem because, among the other ones I presented in the Literature Review, it was the most concrete to me. Then I chose to implement the Boruvka's algorithm because I could exploit its parallel potential and that was the purpose of the project.

Finally, we discovered the opportunities that GpH could offer in terms of parallel programming in Haskell. There are a lot of different strategies providing efficient ways for parallelisation. Developing this program with functional languages such as Haskell seems now to be a good choice in terms of ease of parallelisation as we have seen with the many benefits of the semi-explicit parallelism of GpH. This semi-explicit parallelism was a tremendous help, threads are automatically managed by the runtime system, even though the programmer still needs to specify how a task can be run in parallel, which is convenient to tune the program's behaviour. I have used a lot of tools during this project to tune the versions such as 'threadscope' which offers a good way to see each core working, or not working in some cases, and make the necessary amendments.

To conclude on this experience, I would say that I earned more knowledge than I thought in the fields I mentioned above, but also in the research area. What I found meaningful, especially in this kind of project in which the objective is to maximise the performance, was to always adapt to the situation. As we have seen, I have been through a lot of different versions. At every moment, I wondered what was not working, or what I had to improve, so I tried different solutions. If it worked, there were always new improvements to make, and if it did not

work, I had to find another solution. It was interesting to always take a new look at the program and act accordingly.

5.2. Evaluation

In this section, we evaluate our project basing our discussion on the aims and objectives we mentioned at the beginning of this dissertation. We said that our project would be evaluated using the performance measured of the parallel algorithm solving a MST problem. To do this, we distinguished 4 major objectives:

- Build a graph

As we have seen in the sequential implementation, we managed to build a graph using our own data structures and text files respecting the DIMACS graph format. We then adapted the program to make graphs visible with GraphTea to visualize what we built.

- MST algorithm

Also explained in the sequential version, we have been through 3 different algorithms to reach the good one. Indeed, we started by a Dijkstra's algorithm to get the shortest way between two vertices. Then, we implemented the Prim's algorithm, very close to the Dijkstra's but here to have a first version of a MST algorithm. Finally, we developed the Boruvka's algorithm, with the possibility of running it in parallel.

- Parallel programming

As we have seen in the different parallel versions implemented, we managed to speed things up in the program. Indeed, we had 2 tasks to parallel. We managed to introduce parallelism in the search of the minimum edge linked to a Component and in the rearrangement of the Components. We used the different strategies offered by GpH to build a tuned parallel version.

- Results

We have developed many parallel versions of the algorithm. Each one had an improvement compared to the previous one. We managed to produce speedup and scalability graphs based on these implementations. We used the number of cores used in the Robotarium cluster as an input for our tests. We suppose than the measures we got were accurate enough since we used for example a graph of 4000 nodes and about 4000000 edges which represents a large input.

As shown in the section about reflection, the 5th version seems to be the most elaborated version since we managed to get the lowest runtime using this version and the best garbage collecting as well. However, we also saw that the 3rd parallel version is the best when it comes to speedup. We managed to get 4.5 which is good compared to the other speedups we measured. The same recurrent problem is that from a specific number of cores, which seems to be 8 or 16 depending on each graph, the speedup seems to remain close to the same value. A speedup of 4.5 is not very satisfying when using 32 cores. We still managed to considerably reduce the runtime and the garbage collection percentage which reached, with the last parallel version, a reasonable value.

5.3. Future work

Despite all the modifications we made to our program, there are still improvements to be made in this project. These improvements could be made upon the parallel version in order to improve our thread granularity, as we saw, we have to fix this problem of speedup. But we could also operate in the sequential version. Indeed, we should be able to write the recursive algorithm in a way that simplifies our parallelism or at least write more effective functions to reduce the overall runtime.

As we know it from the 3rd parallel version, the major part of the computation comes from the rearrangement of the Components. Obviously, we must put some effort into this task. This is a difficult function to handle because we must treat a lot of data which has a huge impact upon the heap. A possible solution would be work using data structures which would be faster to treat. In a general way, the best possible option would be to build a new function for rearranging the Components, I have difficulties to imagine how we could get way better results than I obtained using the same basis for this function.

As we have seen in the introduction of this dissertation, graph theory is a very promising area. We already know that it is useful in classic fields such as transport but there are more and more articles about using graph theory like MST algorithms in more complex fields such as brain functional networks, finance, weather prediction and so on. This program that we developed all along this project could be helpful in this field that just keeps on growing. There must be plenty of MST implementations but these versions are maybe not suited for multi-core machines.

Chapter 6

6. References

[Bcs.org, 2017]: Bcs.org. (2017). *Code of conduct | Membership | BCS - The Chartered Institute for IT*. [online] Available at: http://www.bcs.org/category/6030 [Accessed 9 Aug. 2017].

[Black, 1998]: Black, P. (1998). *Dictionary of algorithms, data structures, and problems*. [Gaithersburg, Md.]: National Institute of Standards and Technology.

[Bondy and Murty, 2010]: Bondy, J. and Murty, U. (2010). *Graph theory*. New York: Springer.

[Boost.org, 2017]: Boost.org. (2017). *Parallel BGL Minimum Spanning Tree - 1.59.0*. [online] Available at: http://www.boost.org/doc/libs/1_59_0/libs/graph_parallel/doc/html/dehne_gotz_min_spanning_tree.html [Accessed 31 Jul. 2017].

[Computing.llnl.gov, 2017]: Computing.llnl.gov. (2017). *Message Passing Interface (MPI)*. [online] Available at: https://computing.llnl.gov/tutorials/mpi/ [Accessed 31 Jul. 2017].

[Cormen et al., 2014]: Cormen, T., Leiserson, C., Rivest, R. and Stein, C. (2014). Introduction to algorithms. 1st ed. Cambridge, Massachusetts: The MIT Press, pp.547-748.

[Davendra, 2010]: Davendra, D. (2010). Traveling salesman problem. Rijeka, Croatia: InTech.

[Dimacs.rutgers.edu, 2017]: Dimacs.rutgers.edu. (2017). *DIMACS*. [online] Available at: http://dimacs.rutgers.edu/ [Accessed 30 Jul. 2017].

[En.wikipedia.org, 2017]: En.wikipedia.org. (2017). *Borůvka's algorithm*. [online] Available at: https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm [Accessed 1 Aug. 2017].

[En.wikipedia.org, 2017]: En.wikipedia.org. (2017). *Maximum flow problem*. [online] Available at: https://en.wikipedia.org/wiki/Maximum_flow_problem [Accessed 1 Aug. 2017].

[Even and Even, 2012]: Even, S. and Even, G. (2012). Graph algorithms. Cambridge: Cambridge University Press.

[Foster, 2015]: Foster, I. (1995). Designing and building parallel programs. 1st ed. Reading, Mass. [u.a.]: Addison-Wesley.

[Foundation, 2007]: Foundation, F. S. (2007). GNU General Public License.

[GeeksforGeeks, 2017]: GeeksforGeeks. (2017). *Graph and its representations - GeeksforGeeks*. [online] Available at: http://www.geeksforgeeks.org/graph-and-its-representations/ [Accessed 1 Aug. 2017].

[GeeksforGeeks, 2017]: GeeksforGeeks. (2017). *Greedy Algorithms | Set 2 (Kruskal's Minimum Spanning Tree Algorithm) - GeeksforGeeks*. [online] Available at: http://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/ [Accessed 31 Jul. 2017].

[Gorkovenko, 2017]: Gorkovenko, R. (2017). *Breadth First Search - CodeAbbey*. [online] Codeabbey.com. Available at: http://www.codeabbey.com/index/task_view/breadth-first-search [Accessed 1 Aug. 2017].

[Graphtheorysoftware.herokuapp.com, 2017]: Graphtheorysoftware.herokuapp.com. (2017). *GraphTea*. [online] Available at: http://graphtheorysoftware.herokuapp.com/ [Accessed 9 Aug. 2017].

[Haskell.org, 2017]: Haskell.org. (2017). Haskell Language. [online] Available at: https://www.haskell.org/ [Accessed 30 Mar. 2017].

[Hughes, 1989]: Hughes, J. (1989). Why Functional Programming Matters. *The Computer Journal*, 32(2), pp.98-107.

[Hutton, 2016]: Hutton, G. (n.d.). Programming in Haskell. 2nd ed.

[Ics.uci.edu, 2017]: Ics.uci.edu. (2017). *Minimum spanning trees*. [online] Available at: https://www.ics.uci.edu/~eppstein/161/960206.html [Accessed 31 Jul. 2017].

[Info.univ-angers.fr, 2017]**:** Info.univ-angers.fr. (2017). *DIMACS Graphs and Best Algorithms*. [online] Available at: http://www.info.univ-angers.fr/pub/porumbel/graphs/ [Accessed 31 Jul. 2017].

[Klarreich et al., 2017]: Klarreich, E., Klarreich, E., Molteni, M., Scoles, S., Stockton, N., Wolchover, N., Rogers, A. and Allain, R. (2017). *Computer Scientists Find New Shortcuts for Infamous Traveling Salesman Problem*. [online] WIRED. Available at: https://www.wired.com/2013/01/traveling-salesman-problem/ [Accessed 31 Jul. 2017].

[Kleinberg and Tardos, 2014]: Kleinberg, J. and Tardos, E. (2014). *Algorithm Design*. Harlow: Pearson Education.

[L.Gross and Yellen, 2006]: Gross, J. and Yellen, J. (2006). Graph theory and its applications. Boca Raton, Fla: Chapman & Hall.

[Laziness, 2017]: Laziness, 6. (2017). 6: Laziness - School of Haskell | School of Haskell. [online] Schoolofhaskell.com. Available at: https://www.schoolofhaskell.com/school/starting-with-haskell/introduction-to-haskell/6-laziness [Accessed 4 Mar. 2017].

[Hans-Wolfgang Loidl, 2001]: Hans-Wolfgang Loidl, P. W. T. (2001). A Gentle Introduction to GPH.

[Loidl et al., 2002]: Loidl, H.-W., Rubio, F., Scaife, N., Hammond, K., Horiguchi, S., Klusik, U., Loogen, R., Michaelson, G. J., Pe˜na, R., Priebe, S., Portillo, A. J. R., and Trinder, P. W. (2002). Comparing Parallel Functional Languages: Programming and Performance.

[Loidl, 2017]: Loidl, H. (2017). Course F21DP: Parallel and Distributed Technology. [online] Macs.hw.ac.uk. Available at: http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/ [Accessed 4 Apr. 2017].

[Macs.hw.ac.uk, 2017]: Macs.hw.ac.uk. (2017). *Glasgow Parallel Haskell*. [online] Available at: http://www.macs.hw.ac.uk/~dsg/gph/ [Accessed 3 Aug. 2017].

[Macs.hw.ac.uk, 2017]: Macs.hw.ac.uk. (2017). *GpH - A Parallel Functional Language*. [online] Available at: http://www.macs.hw.ac.uk/~dsg/gph/docs/Gentle-GPH/sec-gph.html [Accessed 31 Jul. 2017].

[Marlow, 2013]: Marlow, S. (2013). Parallel and concurrent programming in Haskell. 1st ed. Sebastopol, CA: O'Reilly.

[Marlow et al., 2010]**:** Marlow, S., Maier, P., Loidl, H., Aswad, M. and Trinder, P. (2010). Seq no more. *ACM SIGPLAN Notices*, 45(11), p.91.

[Msdn.microsoft.com, 2017]**:** Msdn.microsoft.com. (2017). *Part 5: From Trees to Graphs*. [online] Available at: https://msdn.microsoft.com/en-us/library/aa289152(v=vs.71).aspx [Accessed 1 Aug. 2017].

[O'Sullivan, Goerzen and Stewart, 2009]: O'Sullivan, B., Goerzen, J. and Stewart, D. (2009). *Real world Haskell*. Sebastopol, CA: O'Reilly.

[Prolland.free.fr, 2017]: Prolland.free.fr. (2017). *Coloring Problems:DIMACS Graph Format*. [online] Available at: http://prolland.free.fr/works/research/dsat/dimacs.html [Accessed 31 Jul. 2017].

[Rallabhandi, 2017]: Rallabhandi, L. (2017). *Techfinite*. [online] Techfinite.blogspot.co.uk. Available at: http://techfinite.blogspot.co.uk/ [Accessed 1 Aug. 2017].

[Sedgewick, 2009]: Sedgewick, R. (2009). *Graph algorithms*. Boston [u.a.]: Addison-Wesley.

[Snap.stanford.edu, 2017]**:** Snap.stanford.edu. (2017). *Stanford Large Network Dataset Collection*. [online] Available at: https://snap.stanford.edu/data/ [Accessed 9 Aug. 2017].

[Trinder et al., 1998]: Trinder, P., Hammond, K., Loidl, H. and Peyton Jones, S. (1998). Algorithm + strategy = parallelism. *Journal of Functional Programming*, 8(1), pp.23-60.

Chapter 7

7. Appendices

Sequential profile of the Boruvka's algorithm:

COST CENTRE	MODULE	no.	entries	%time	%alloc	%time	%alloc
MAIN	MAIN	52	0	0.0	0.0	100.0	100.0
main	Main	105	0	0.1	1.2	100.0	100.0
getDistance	Main	197	1	0.0	0.0	0.0	0.0
weight	Main	198	999	0.0	0.0	0.0	0.0
main.wedges	Main	122	1	0.0	0.0	95.3	91.0
boruvka	Main	123	1	0.0	0.0	95.3	91.0
boruvka.wedges'	Main	193	1	0.0	0.0	0.0	0.0
boruvka.(...)	Main	132	1	0.0	0.0	94.1	90.9
boruvkaAlg	Main	133	3	0.0	0.0	94.1	90.9
boruvkaAlg.wedges'	Main	194	2	0.0	0.0	0.0	0.0
boruvkaAlg.midcomponents'	Main	148	2	0.0	0.0	10.4	7.7
rearrangeComponentFinal	Main	149	2	0.0	0.0	10.4	7.7
rearrangeComponent	Main	171	2	0.0	0.0	10.4	7.7
checkComponentDuplicate	Main	172	1014	8.2	0.5	10.4	7.7
quickSort	Main	176	674906	0.0	0.1	1.2	6.5
quickSort.biggerSorted	Main	179	340322	0.9	6.3	0.9	6.3
quickSort.smallerSorted	Main	177	340322	0.3	0.0	0.3	0.0
modifyIfDuplicate	Main	174	995246	0.1	0.0	0.9	0.7
add	Main	178	329520	0.4	0.3	0.4	0.3
compareList	Main	175	995246	0.5	0.3	0.5	0.3
checkComponentDuplicate.new_mat	Main	173	1014	0.1	0.1	0.1	0.1
isThereStillDuplicates	Main	165	4	0.0	0.0	0.0	0.0
isThereDuplicatesNode	Main	166	6	0.0	0.0	0.0	0.0
compareList	Main	170	6	0.0	0.0	0.0	0.0
isThereDuplicatesNode.new_mat	Main	167	6	0.0	0.0	0.0	0.0
boruvkaAlg.currwedges'	Main	147	2	0.1	0.0	65.9	66.7
==	Main	163	497508	0.0	0.0	0.0	0.0
==	Main	164	497508	0.0	0.0	0.0	0.0

minLinkedComponent	Main	150	1002	0.0	0.0	65.8	66.7
minWeight	Main	161	1002	0.0	0.2	0.1	0.2
weight	Main	162	715852	0.0	0.0	0.0	0.0
findWedgefromWeight	Main	158	1002	0.0	0.0	0.0	0.0
findWedgefromWeight.\	Main	159	1002	0.0	0.0	0.0	0.0
weight	Main	160	1002	0.0	0.0	0.0	0.0
linkedComponent	Main	151	2004	0.1	0.1	65.8	66.6
linkedComponent.\	Main	156	988326	5.0	0.1	5.0	0.2
twoNodes	Main	157	1618721	0.0	0.1	0.0	0.1
linkedWedges	Main	152	3003	11.1	0.1	60.7	66.3
linkedWedges.\	Main	154	575696353	38.4	0.0	49.5	66.2
twoNodes	Main	155	575696353	11.1	66.2	11.1	66.2
getEdges	Main	153	3003	0.0	0.0	0.0	0.0
boruvkaAlg.midcomponents	Main	145	2	0.0	0.0	0.0	0.0
fromWedgesToNodes	Main	146	2	0.0	0.0	0.0	0.0
fromWedgesToNodes.\	Main	168	1000	0.0	0.0	0.0	0.0
twoNodes	Main	169	1000	0.0	0.0	0.0	0.0
boruvkaAlg.components'	Main	142	2	0.0	0.0	14.3	16.1
rearrangeComponentFinal	Main	143	2	0.0	0.0	14.3	16.1
rearrangeComponent	Main	184	3	0.0	0.0	14.3	16.1
checkComponentDuplicate	Main	185	1009	6.6	0.5	14.3	16.1
quickSort	Main	189	1105333	0.0	0.1	2.7	14.8
quickSort.biggerSorted	Main	192	555054	2.0	14.6	2.0	14.6
quickSort.smallerSorted	Main	190	555054	0.7	0.1	0.7	0.1
modifyIfDuplicate	Main	187	1003020	0.0	0.0	5.0	0.6
add	Main	191	2014	4.9	0.3	4.9	0.3
compareList	Main	188	1003020	0.1	0.3	0.1	0.3
checkComponentDuplicate.new_mat	Main	186	1009	0.0	0.1	0.0	0.1
isThereStillDuplicates	Main	180	5	0.0	0.0	0.0	0.0
isThereDuplicatesNode	Main	181	6	0.0	0.0	0.0	0.0
compareList	Main	183	5	0.0	0.0	0.0	0.0
isThereDuplicatesNode.new_mat	Main	182	6	0.0	0.0	0.0	0.0
getNodes	Main	135	3	0.0	0.0	3.6	0.4
quickSort	Main	139	2003	0.0	0.0	0.0	0.1
quickSort.biggerSorted	Main	141	1002	0.0	0.1	0.0	0.1
quickSort.smallerSorted	Main	140	1006	0.0	0.0	0.0	0.0
getEdges	Main	137	3	0.0	0.0	0.0	0.0
nodesForEdges	Main	136	3	3.5	0.2	3.6	0.3
twoNodes	Main	138	740124	0.0	0.1	0.0	0.1
boruvka.components	Main	124	1	0.0	0.0	1.2	0.2
getComponents	Main	125	1	0.0	0.0	1.2	0.2
listOfNode	Main	134	1000	0.0	0.0	0.0	0.0
getNodes	Main	126	1	0.0	0.0	1.2	0.2
quickSort	Main	130	1995	0.0	0.0	0.0	0.1
quickSort.biggerSorted	Main	144	1000	0.0	0.1	0.0	0.1
quickSort.smallerSorted	Main	131	1000	0.0	0.0	0.0	0.0
getEdges	Main	128	1	0.0	0.0	0.0	0.0
nodesForEdges	Main	127	1	1.2	0.1	1.2	0.1
twoNodes	Main	129	246708	0.0	0.0	0.0	0.0
main.linesOfFiles	Main	112	1	0.2	0.7	0.2	0.7
main.graph	Main	109	1	0.0	0.0	3.0	6.9
fromLinestoWedge	Main	114	1	0.3	0.9	3.0	6.8
buildWedge	Main	116	246708	2.6	6.0	2.6	6.0
keepEdges	Main	111	1	0.0	0.0	0.0	0.0
keepEdges.\	Main	113	246723	0.0	0.0	0.0	0.0
buildGraph	Main	110	1	0.0	0.0	0.0	0.0
getNodes	Main	106	1	0.0	0.0	1.4	0.2
quickSort	Main	117	1995	0.0	0.0	0.0	0.1
quickSort.biggerSorted	Main	120	1000	0.0	0.1	0.0	0.1
quickSort.smallerSorted	Main	118	1000	0.0	0.0	0.0	0.0
getEdges	Main	108	1	0.0	0.0	0.0	0.0
nodesForEdges	Main	107	1	1.4	0.1	1.4	0.1
twoNodes	Main	115	246708	0.0	0.0	0.0	0.0
CAF	Main	103	0	0.0	0.0	0.0	0.0
cores	Main	199	1	0.0	0.0	0.0	0.0
boruvka	Main	195	0	0.0	0.0	0.0	0.0
boruvka.wedges	Main	196	1	0.0	0.0	0.0	0.0
quickSort	Main	121	1	0.0	0.0	0.0	0.0
buildWedge	Main	119	0	0.0	0.0	0.0	0.0
main	Main	104	1	0.0	0.0	0.0	0.0
CAF	GHC.Conc.Sync	102	0	0.0	0.0	0.0	0.0
CAF	GHC.IO.Handle.FD	99	0	0.0	0.0	0.0	0.0
CAF	GHC.Conc.Signal	96	0	0.0	0.0	0.0	0.0
CAF	GHC.IO.Encoding	92	0	0.0	0.0	0.0	0.0
CAF	Text.Read.Lex	80	0	0.0	0.0	0.0	0.0
CAF	GHC.Float	79	0	0.0	0.0	0.0	0.0
CAF	GHC.IO.Encoding.Iconv	78	0	0.0	0.0	0.0	0.0
CAF	GHC.IO.FD	76	0	0.0	0.0	0.0	0.0

Risk assessment:

Risk	Likelihood	Impact
Supervisor illness	Low	Low
Supervisor relocation	Low	Medium
Author illness	Low	Medium
Software bugs	High	Medium
Project plan changes	High	Low
Complexity of the algorithm	Medium	High
Performances not enough satisfying	Medium	Medium
Requirement changes	Low	High
Dataset not big enough	Medium	Low
Machine breakdown (Linux machines)	Low	Low
Robotarium breakdown	Low	Medium

Source code of the Sequential Boruvka's algorithm:

We import the Data module to use the list we create. We define nodes as integer values, edges as a pair of nodes, a weighted edge as an edge and a weight, a graph as a list of weighted edges.

```
-------------------------------------------------------------------------
-- File created as part of the MSc Softare Engineering in
-- Heriot-Watt University.
-- This project is realized by Adrien Chevrot (adrien.chevrot83@gmail.com)
-- and supervised by Dr Hans-Wolfgang Loidl (h.w.loidl.hwu@gmail.com)
-- 17 August 2017
-- Implementation of a boruvka's algorithm (Minimum Spanning Tree)
-------------------------------------------------------------------------


module Main where

import System.Environment
import System.IO
import Control.Monad
import Data.List
import Data.Bool
import Data.Maybe
import Data.Function
import Control.Parallel
import Control.Parallel.Strategies
import Control.DeepSeq
import Data.Time
-------------------------------------------------------------------------
------------------------DEFINING DATA STRUCTURES-------------------------
-------------------------------------------------------------------------

type Node = Int
data Edge = Edge (Node, Node) deriving (Show, Eq)
data Wedge = Wedge (Edge, Float) deriving (Show, Eq)
data Wgraph = Wgraph [Wedge] deriving (Show)

instance NFData Wedge where
    rnf w = w `seq` ()
```

To build a graph, we read all the lines from a text file, we keep only the edges, and we apply the buildWedge function upon each line remaining. Then we build the graph from the edges.

```
------------------------------------------------------------------------
----------------------------BUILD GRAPH---------------------------------
------------------------------------------------------------------------

-- Build a weighted edge
buildWedge :: [String] -> Wedge
buildWedge [e, n1, n2, d] = Wedge (Edge (read n1 :: Node, read n2 :: Node), read d :: Float)
buildWedge [e, n1, n2] = Wedge (Edge (read n1 :: Node, read n2 :: Node), 1)

-- From a DIMACS format text file, keeps only the edge while removing the rest
keepEdges :: [String] -> [String]
keepEdges str = filter (\e -> head e == 'e') str

-- Build a list of weighted edges from a list of String
fromLinestoWedge :: [String] -> [Wedge]
fromLinestoWedge str = map buildWedge (map words str)

-- Build a weighted graph from a list of weighted edges
buildGraph :: [Wedge] -> Wgraph
buildGraph wedges = Wgraph (wedges)
```

Functions implemented before the MST algorithm. Only used to do basic operations upon a graph such as returning all the nodes of a graph.

```
------------------------------------------------------------------------
----------------------RETURNS EDGES/NODES/WNODES------------------------
------------------------------------------------------------------------

-- Gives all the weighted edges within a graph
getEdges :: Wgraph -> [Wedge]
getEdges (Wgraph graph) = graph

-- Gives the two nodes linked by a weighted edge
twoNodes :: Wedge -> [Node]
twoNodes (Wedge (Edge (n1, n2), f)) = [n1, n2]

-- Gives the same thing than twoNodes but with several weighted edges
nodesForEdges :: [Wedge] -> [Node]
nodesForEdges wedges = nub (concat (map twoNodes wedges))

-- Gives all the nodes within a graph
getNodes :: Wgraph -> [Node]
getNodes graph = quickSort (nodesForEdges (getEdges graph))

-- From Node to [Node]
listOfNode :: Node -> [Node]
listOfNode node = [node]

-- Gives all the components within a graph
getComponents :: Wgraph -> [[Node]]
getComponents graph = map listOfNode (getNodes graph)

-- Tries to find an edge between two given nodes, returns it if it exists
tryGetWedge :: Wgraph -> Node -> Node -> Maybe Wedge
tryGetWedge (Wgraph g) n1 n2  = find (\x -> [n1, n2] \\ twoNodes x == []) g
```

Function to build the adjacency-matrix notation of a graph, to be usable with GraphTea.

```
------------------------------------------------------------------------
------------------------MATRIX REPRESENTATION---------------------------
------------------------------------------------------------------------

-- Returns the matrix representation within strings
buildMatrixNotation :: Wgraph -> [String]
buildMatrixNotation graph = map (makeLineWedge graph (getNodes graph)) (getNodes graph)

-- Creates a String listing all the linked edges to a given node
makeLineWedge :: Wgraph -> [Node] -> Node -> String
makeLineWedge graph nodes n = intercalate "" (map (getWeightOrZero graph n) nodes)

-- Gets the weight of an wedge if it exists, 0 otherwise
getWeightOrZero :: Wgraph -> Node -> Node -> String
getWeightOrZero graph n n2 | wedge == Nothing = "0 "
                           | otherwise = show (round (weight (fromJust wedge))) ++ " "
                           where
                             wedge = tryGetWedge graph n n2
```

Functions determining whether a graph has self-connected nodes (so incompatible with the Boruvka's algorithm), which is actually not used in the program since our input data does not include self-connected nodes.

```
------------------------------------------------------------------------
------------------------SELF CONNECTED NODES----------------------------
------------------------------------------------------------------------

-- Returns True if the given edge is self connected
loopWedge :: Wedge -> Bool
loopWedge (Wedge (Edge (n1, n2), f)) | n1 == n2  = True
                                     | otherwise = False

-- Returns False if a given graph has no self-connected edges, false otherwise
noLoopGraph :: Wgraph -> Bool
noLoopGraph graph | find (==True) (map loopWedge (getEdges graph)) == Just True = True
                  | otherwise                                                   = False
```

All the functions we used to get the minimum edge linked to a Component.

```
--------------------------------------------------------------------------
---------------------------------LINKED-----------------------------------
--------------------------------------------------------------------------

-- Returns the weight of a weighted edge
weight :: Wedge -> Float
weight (Wedge (_, w)) = w

-- Gives the less weighted edge
minWeight :: [Wedge] -> Float
minWeight edges = minimum (map weight edges)

-- Returns a list of edges with the same weight than given
findWedgefromWeight :: Wgraph -> Float -> [Wedge] -> [Wedge]
findWedgefromWeight graph value edges = filter (\e -> weight e == value) edges

-- Returns a list of weighted edges linked to the given node in a graph
linkedWedges :: Wgraph -> Node -> [Wedge]
linkedWedges graph n = filter (\e -> n `elem` twoNodes e) (getEdges graph)

-- Returns a list of weighted edges linked to a given component within a graph
linkedComponent :: Wgraph -> [Node] -> [Wedge]
linkedComponent g nodes = filter (\w -> not (((head (twoNodes w)) `elem` nodes)
                                        && (((twoNodes w)!!1) `elem` nodes)))
                                              (concat (map (linkedWedges g) nodes))

-- Returns the less weighted edge linked to a given node
minLinkedWedges :: Wgraph -> Node -> Wedge
minLinkedWedges graph n = head (findWedgefromWeight graph
                        (minWeight (linkedWedges graph n)) (linkedWedges graph n))

-- Returns the less weighted edge linled to a given component
minLinkedComponent :: Wgraph -> [Node] -> Wedge
minLinkedComponent graph component = head (findWedgefromWeight graph
                (minWeight (linkedComponent graph component)) (linkedComponent graph component))

-- Returns a list of components from a given list of edges
fromWedgesToNodes :: [Wedge] -> [[Node]]
fromWedgesToNodes wedges = map (\w -> twoNodes w) wedges
```

The Boruvka's algorithm including the launch of the recurive algorithm and what is inside. In this function, we first get the minimum edge linked to each component that we store in the spanning tree. We then include the nodes connected by the edges to the list of components. Finally, we rearrange these components. We repeat this operation until the list of components is equal to the nodes of the graph (all the nodes are linked by the spanning tree).

```
-------------------------------------------------------------------------
-------------------------BORUVKA ALGORITHM-------------------------------
-------------------------------------------------------------------------

-- Launch the recursive boruvka algorithm
boruvka :: Wgraph -> [Wedge]
boruvka g =
  let wedges = []
      components = getComponents g
      (_, _, wedges') = boruvkaAlg g components wedges
      in wedges'

-- Recursive boruvka algorithm which ends when the list of component is only
--                                  composed of one component (all the nodes)
boruvkaAlg :: Wgraph -> [[Node]] -> [Wedge] -> (Wgraph, [[Node]], [Wedge])
boruvkaAlg g components wedges | components == [getNodes g] = (g, [], wedges)
                               | otherwise =
      let
          currwedges' = nub $ map (minLinkedComponent g) components
          wedges' | length components == 2 = wedges ++ [head currwedges']
                  | otherwise = wedges ++ currwedges'
          midcomponents = fromWedgesToNodes currwedges'
          midcomponents' = rearrangeComponentFinal midcomponents
          components' = rearrangeComponentFinal (components ++ midcomponents')
      in boruvkaAlg g components' wedges'
```

Returns the total weight of the MST.

```
-------------------------------------------------------------------------
---------------------------------LENGTH----------------------------------
-------------------------------------------------------------------------

-- Returns the length of a graph
getDistance :: [Wedge] -> Float
getDistance wedges = sum (map weight wedges)
```

Basic quicksort function to get a sorted list of nodes.

```
-----------------------------------------------------------------------
----------------------------------SORT---------------------------------
-----------------------------------------------------------------------

-- Basic quicksort function
quickSort :: (Ord a) => [a] -> [a]
quickSort [] = []
quickSort (x:xs) =
    let smallerSorted = quickSort [a | a <- xs, a <= x]
        biggerSorted = quickSort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```

Functions needed to rearrange the components.

'reArrangeComponentFinal' is the top-level function, it executes the 'reArrangeComponent' function until we cannot find duplicates anymore in the list of component. In this function, we execute the function 'checkComponentDuplicate' for each component. This function'checkComponentDuplicate' takes the Component as input and compare for each other component if the input share common elements. If yes, we merge the components.

```
-----------------------------------------------------------------------
----------------------------------LIST---------------------------------
-----------------------------------------------------------------------

-- Chunk a list using the first parameter
chunk :: Int -> [[Node]] -> [[[Node]]]
chunk _ [] = []
chunk n xs = y1 : chunk n y2
        where
                (y1, y2) = splitAt n xs

-- Concatenate two lists such that the elements in the resulting list occur only once
add :: [Node] -> [Node] -> [Node]
add l1 l2 = nub (l1 ++ l2)

-- Compare two given lists and returns true if some elements of the first are in the second
compareList :: (Eq a) => [a] -> [a] -> Bool
compareList a = not . null . intersect a

-- Modifies a list if two lists share elements
modifyIfDuplicate :: [Node] -> [Node] -> [Node]
modifyIfDuplicate l1 l2 | compareList l1 l2 = add l1 l2
                        | otherwise = l1

-- Checks for duplicates in the list of lists for a given list
checkComponentDuplicate :: [[Node]] -> [Node] -> [Node]
checkComponentDuplicate mat nodes =
   let new_mat = delete nodes mat
       in quickSort (nub (concat (map (modifyIfDuplicate nodes) new_mat)))
```

```
-- Rearrange the components modified by the boruvka algorithm at each step
rearrangeComponent :: [[Node]] -> [[Node]]
rearrangeComponent listoflist = nub (map (checkComponentDuplicate listoflist) listoflist)

-- Rearrange the components (top-level function)
rearrangeComponentFinal :: [[Node]] -> [[Node]]
rearrangeComponentFinal listoflist | length listoflist == 1 = listoflist
                                   | otherwise = until (not . isThereStillDuplicates)
                                                       (rearrangeComponent) listoflist

-- Checks whether a list of nodes has duplicates into a list of list
isThereDuplicatesNode :: [[Node]] -> [Node] -> Bool
isThereDuplicatesNode mat node | find (==True) (map (compareList node) new_mat) == Just True = True
                               | otherwise = False
                               where new_mat = delete node mat

-- Checks whether the list of lists of nodes still has duplicates
isThereStillDuplicates :: [[Node]] -> Bool
isThereStillDuplicates listoflist | find (==True) (map (isThereDuplicatesNode listoflist) listoflist)
                                                                        == Just True = True
                                  | otherwise = False
```

Main function in which we read the text file to create the graph. Then, we applied the boruvka's function upon it and print the weight of the Minimum Spanning Tree (we can also print the sub-graph created or create the adjacency-matrix notation of the MST to display it with GraphTea).

```
------------------------------------------------------------------------
----------------------------------MAIN----------------------------------
------------------------------------------------------------------------

main = do
                        args <- getArgs
                        content <- readFile (args !! 0)
                        let file = (args !! 0)
                        let linesOfFiles = lines content
                        let graph = buildGraph (fromLinestoWedge (keepEdges linesOfFiles))



                        t0 <- getCurrentTime
                        let wedges = boruvka graph


                        --print (wedges)
                        print (getDistance wedges)

                        t1 <- getCurrentTime
                        print (diffUTCTime t1 t0)
```

Modifications for 1st parallel version:

```
-- Returns a list of weighted edges linked to a given component within a graph
linkedComponent :: Wgraph -> [Node] -> [Wedge]
linkedComponent g nodes = filter (\w -> not (((head (twoNodes w)) `elem` nodes)
                                          && (((twoNodes w)!!1) `elem` nodes)))
                                         (concat (parMap rdeepseq (linkedWedges g) nodes))
```

```
-- Recursive boruvka algorithm which ends when the list of component is only composed
--                                                    of one component (all the nodes)
boruvkaAlg :: Wgraph -> [[Node]] -> [Wedge] -> (Wgraph, [[Node]], [Wedge])
boruvkaAlg g components wedges | components == [getNodes g] = (g, [], wedges)
                               | otherwise =
      let
          currwedges' = nub $ parMap rdeepseq (minLinkedComponent g) components
          wedges' | length components == 2 = wedges ++ [head currwedges']
                  | otherwise = wedges ++ currwedges'
          midcomponents = fromWedgesToNodes currwedges'
          midcomponents' = rearrangeComponentFinal midcomponents
          components' = rearrangeComponentFinal (components ++ midcomponents')
      in boruvkaAlg g components' wedges'
```

```
-- Checks for duplicates in the list of lists for a given list
checkComponentDuplicate :: [[Node]] -> [Node] -> [Node]
checkComponentDuplicate mat nodes =
   let new_mat = delete nodes mat
       in quickSort (nub (concat (parMap rdeepseq (modifyIfDuplicate nodes) new_mat)))

-- Rearrange the components modified by the boruvka algorithm at each step
rearrangeComponent :: [[Node]] -> [[Node]]
rearrangeComponent listoflist = nub (parMap rdeepseq (checkComponentDuplicate listoflist) listoflist)
```

Modifications for 2nd parallel version:

```
-- Recursive boruvka algorithm which ends when the list of component is only composed
--                                                    of one component (all the nodes)
boruvkaAlg :: Wgraph -> [[Node]] -> [Wedge] -> (Wgraph, [[Node]], [Wedge])
boruvkaAlg g components wedges | components == [getNodes g] = (g, [], wedges)
                               | otherwise =
      let
          currwedges' | length components < 16 = nub $ map (minLinkedComponent g) components
                      | otherwise = nub (map (minLinkedComponent g) components
                                             `using` parListChunk 8 rdeepseq)
          --currwedges' = nub (map (minLinkedComponent g) components)
          wedges' | length components == 2 = wedges ++ [head currwedges']
                  | otherwise = wedges ++ currwedges'
          midcomponents = fromWedgesToNodes currwedges'
          midcomponents' = rearrangeComponentFinal midcomponents
          components' = rearrangeComponentFinal (components ++ midcomponents')
      in boruvkaAlg g components' wedges'
```

```
-- Rearrange the components modified by the boruvka algorithm at each step
rearrangeComponent :: [[Node]] -> [[Node]]
rearrangeComponent listoflist | length listoflist <= 32 = nub (map (checkComponentDuplicate listoflist) listoflist)
                              | otherwise = nub (map (checkComponentDuplicate listoflist) listoflist
                                                     `using` parListChunk 8 rdeepseq)
```

Modifications for 3rd parallel version:

```
-- Rearrange the components modified by the boruvka algorithm at each step
rearrangeComponent :: [[Node]] -> [[Node]]
rearrangeComponent listoflist | length listoflist == 1 = listoflist
                              | length listoflist < 32 =
                                      nub $ map (checkComponentDuplicate listoflist) listoflist
                              | otherwise =
                          nub $ concat $ parMap rdeepseq rearrangeComponent (chunk 8 listoflist)
```

Modifications for 4th parallel version:

```
-- Gives the less weighted edge
minWeight :: [Wedge] -> Wedge
minWeight edges = minimumBy (compare `on` weight) edges

-- Returns a list of weighted edges linked to the given node in a graph
linkedWedges :: Wgraph -> Node -> [Wedge]
linkedWedges graph n = filter (\e -> n `elem` twoNodes e) (getEdges graph)

-- Returns a list of weighted edges linked to a given component within a graph
linkedComponent :: Wgraph -> [Node] -> [Wedge]
linkedComponent g nodes = filter (\w -> not (((head (twoNodes w)) `elem` nodes) &&
                                             (((twoNodes w)!!1) `elem` nodes)))
                                    (concat (parMap rdeepseq (linkedWedges g) nodes))

-- Returns the less weighted edge linked to a given node
minLinkedWedges :: Wgraph -> Node -> Wedge
minLinkedWedges graph n = minWeight (linkedWedges graph n)

-- Returns the less weighted edge linked to a given component
minLinkedComponent :: Wgraph -> [Node] -> Wedge
minLinkedComponent graph component = minWeight (linkedComponent graph component)
```

Modifications for 5th parallel version:

```
-- Recursive boruvka algorithm which ends when the list of component is only composed of one component (all the nodes)
boruvkaAlg :: Wgraph -> [[Node]] -> [[Node]] -> [Wedge] -> (Wgraph, [[Node]], [[Node]], [Wedge])
boruvkaAlg g components real_comp wedges | real_comp == [getNodes g] = (g, [], [], wedges)
                                         | otherwise =
      let
          currwedges' | wedges == [] && length components < cores = nub $ parMap rdeepseq (minLinkedComponent g) components
                      | wedges == [] && length components >= cores = nub (map (minLinkedComponent g) components `using`
                                                                             parListChunk (length components `div` cores) rdeepseq)
                      | wedges /= [] && length real_comp < cores = nub $ parMap rdeepseq (minLinkedComponent g) real_comp
                      | wedges /= [] && length real_comp >= cores = nub (map (minLinkedComponent g) real_comp `using`
                                                                             parListChunk (length real_comp `div` cores) rdeepseq)
          wedges' | length real_comp == 2 = wedges ++ [head currwedges']
                  | otherwise = wedges ++ currwedges'
          midcomponents = fromWedgesToNodes currwedges'
          midcomponents' = rearrangeComponentFinal midcomponents
          real_comp' = rearrangeComponentFinal (real_comp ++ midcomponents')
      in boruvkaAlg g [] real_comp' wedges'


-- Rearrange the components modified by the boruvka algorithm at each step
rearrangeComponent :: [[Node]] -> [[Node]]
rearrangeComponent listoflist | length listoflist == 1 = listoflist
                              | length listoflist < 32 = nub (map (checkComponentDuplicate listoflist) listoflist `using`
                                                                  parListChunk (length listoflist `div` cores) rdeepseq)
                              | otherwise = nub (concat (map rearrangeComponent (chunk 8 listoflist) `using`
                                                                  parListChunk (length listoflist `div` cores) rdeepseq))
```