Comparison of parallel and distributed implementation of the MST algorithm

Report

Alexis SIRETA

February 7, 2016

Contents

1 Introduction **3**
1.1 Parallel and distributed systems . . . 3
1.2 Comparing the algorithms . . . 3
1.3 Research question . . . 4
1.4 Approach . . . 4
1.5 Report structure . . . 4

2 The MST Algorithm **5**
2.1 Boruvka's algorithm . . . 5
2.2 Parallel and distributed algorims . . . 5
2.3 Bor-el algorithm . . . 7
2.3.1 The algorithm . . . 7
2.3.2 Theoretical Analysis . . . 7
2.4 GHS . . . 11
2.4.1 The algorithm . . . 11
2.4.2 Theoretical analysis . . . 11

3 Comparison **13**
3.1 Theoretical results . . . 13
3.1.1 Experimental setup . . . 13
3.1.2 Graphs used . . . 13
3.1.3 Experimental results . . . 13
3.2 Discussion . . . 14
3.2.1 Communication delay . . . 15
3.2.2 Number of messages sent . . . 15

4 Conclusion **16**

5 Additional Informations **17**
5.1 Annexes . . . 17
5.1.1 Graph generation code . . . 17
5.1.2 Benchmark of communication delay . . . 18
5.2 Acknowledgements . . . 19

Abstract Graph algoritms are widely used in a wide range of scientific fields. From finding the shortest path in a road network to balancing communications in a computer network,graph theory is unavoidable. Very often the size of the considered graph is too big to run the algorithm in a simple sequential way for obvious performance and memory issues. Researchers have therefore tried and managed to implement such algorithms either in a parallel way or a distributed way. The advatage of both techniques is sharing the workload accross different processing units. It is still, however, unclear how these two kinds of implementations relate to each other, how they scale, for which kind of graph they are more performant,can one implementation work on the other architecture and how does it affect performance. This is what we propose to do in this paper.

Chapter 1

Introduction

A graph can be considered as a list of vertices and a list of edges connecting two vertices. Edges can be unidirectionnal or bidirectionnal and can have a wheight to represent virtually any kind of parameter that matters to you. Many problems use graphs as a data representation, However nowadays graphs are getting extremely big and can contain dozens of millions of edges. For instance the graph describing the links between pages in the chinese encyclopedia contains seventeen millions of edges, it is therefore very time consuming to run algorithms on such big graphs. A solution to solve this problem is using parallel or distributed systems. Nevertheless each of these approaches has its own advantages and drawbacks and it is not clear which approcach is the best.
In this paper we will focus on the minimum spanning tree (MST) problem. In a bidirectionnal wheighted graph, finding the minimum spanning tree means finding a subgraph covering all the nodes, with no cycles, and minimizing the sum of the weight of the edges. The minimum spanning tree problem has many uses such as efficient cabling of a street or building road connection between cities. The weight of the edges could for instance be the difficulty of putting a cable or a road between these two nodes. Many sequential algorithms solve the MST problem, the three most famous are Prim [1], Kruskal [2] and Boruvka [3]. While the first two are clearly sequential, Boruvka's algorithm seems to be easily parallelized. In fact most parallel and distributed algorithms are based on Boruvka's idea.

1.1 Parallel and distributed systems

We first have to define what we are talking about when we say parallel and distributed systems. A parallel system is a multi-CPU system with a shared main memroy. Each CPU has its own cache that is accessed in parallel but the main memory is shared between CPUs and this creates memory contention when the graphs get very big. On the other had, distributed systems, the memory is distributed between CPUs and all those machines are connected via a network. Here the bottleneck is mainly communication latency [4].

1.2 Comparing the algorithms

Comparing two algorithms running on the same architecture is simple, the only changing parameter is the efficiency of the algorithm itself. Comparing parallel and distributed algorithm is more challenging since they run on completely different architectures. A shared memory parallel architecture

has memory access as a bottleneck where a distributed architecture has communication latency as a critical factor.
Graph computing is a quite new topic and no previous work tackles the comparison between distributed and parallel approaches. We have therefore no standard framework to do such a comparison. One cannot even compare hardware since having the same CPU for instance will not give you any information on the time computation takes on a distributed architecture. On the other hand, network connection speed is totally irrelevant for a parallel architecture that actually is not even supposed to have one ! One can only for instance compare the computation time algorithm compared with the power used or the money spent on either of the architectures.

1.3 Research question

All that we mentioned before make some questions arise and in this particular research there are three questions that we want to answer.

- can we really compare distributed and parallel algorithms ?
- how do they scale ?
- How do they perform on different architectures ?

1.4 Approach

First we have to choose two implementations of the boruvka's algorithm, one for parallel machines and one for distributed systems.
Then we are planning on studying the algorithms theoretically. Each algorithms have performance costs in term of processing time, memory access and communication. these theoretical values can be found by studying the complexity of each algorithm. It is very important to do this theoretical research prior to experimentations because it allows us to really know where the algorithm and the architecture used did not match the expectations afterwards.
Once we have this theoretical model we can test the algorithms on different architectures.
In the end we will try and see how this theoretical model matches the empirical model and draw conclusions on what kind of elements we might have missed. It is very likely some elements will be overlooked, distributed and parallel systems depend on a high number of parameters, communication latency, competitive memory access, hardware performance are only a few of them.
Nevertheless we can have a first intuition. Our hypothesis would be that a distributed system will run slower that a parallel system for small graphs because of communication latency. But as soon as we consider big graphs, memory access will be very slow for parallel systems and then the distributed approach will outperform it.

1.5 Report structure

In this report we will focus in chapter 2 on the theoretical aspect of each chosen implementations. Then, in chapter 3 we will show the experimental results, in section 3.2 we will discuss these results and in chapter 4 we will conclude our research .

Chapter 2

The MST Algorithm

In this chapter we will introduce the sequential Boruvka's algorithm then we will do a theoretical analysis of both parallel and distributed implementations of this algorithm (Bor-el and GHS).

2.1 Boruvka's algorithm

The principle of the algorithm is fairly simple. It requires to consider the notion of component, which is in fact a sub graph.
pseudocode:
Input: A connected graph G whose edges have distinct weights

1. Initialize a forest T to be a set of one-vertex trees, one for each vertex of the graph.
2. While T has more than one component:
3. For each component C of T:
4. Begin with an empty set of edges S
5. For each vertex v in C:
6. Find the cheapest edge from v to a vertex outside of C, and add it to S
7. Add the cheapest edge in S to T
8. compute the new components

Output: T is the minimum spanning tree of G. [5]

2.2 Parallel and distributed algorims

When investigating the existing implementations of Boruvka's algorithm we had some constraints to take into consideration. The target architectures had to be very specific to each field so that we have a real difference in implementation, and source code should be easily accessible. Therefore we found two algorithms that matched the criterias Bor-el for the parallel architectures and GHS for the distributed ones.

Graph	Components	Explanation
	• A • B • C • D • E • F • G	At the beggining each node is a component.
	• A,B,D,F • C,E,G	Each component find its minimum weight outgoing edge (An edge of minimum weight that goes from one component to another). All components connected by a MWOE become a new component.
	• A,B,C,D,E,F	Iterate with the new components. Then there is only one big component. The algorithm is finished. The MST is the graph with the yellow edges.

Table 2.1: Explanantion of the MST algoritm [5]

2.3 Bor-el algorithm

2.3.1 The algorithm

Described in the research paper ”Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs” by David A. Badera and Guojing Congb Bor-el is a parallel algorithm based on boruvka's approach where the graph is represented by a sorted edge list and operations on this list are made in parallel (each processor accessing a subset of the list). It targets SMP architechure, meaning a multiprocessor shared memory systems. [6]

2.3.2 Theoretical Analysis

Theoretically there are several factors that are significant for the execution time. Since we are talking about a parallel architecture the memory access time is a major time consumer when the graph is very big. The number of computations done at each node cannot be overlooked either since there is a very limited amout of processors to do many operations since the graph is incredibly big. We will first talk about the memory access time by evaluating how long it takes to access memory once and multiplying it with the number of memory accesses. Then we will look at the computation complexity and evaluate how long it takes to make one operation. Then we will take into account the speed of the processor to estimate the execution time.

Size of the graph in memory

In the Bor-el algorithm the graph is described as an edge list, with each edge appears twice (once for each direction). An edge is composed as follow, (node1,node2,weight). The size of the Id of a node in memory depends on the total number of nodes N in the graph. We can say that each node is stored in log(N) bits.
There are two nodes in each edge ,E edges in the graph so 2*E edges in the list. so the size of the node ids in memory is 2*2*E*log(N).
Each weight has to be unique, so there is as many different weights as there are edges. The size of a weight in bits is therefore log(E).There is 2*E edges in the list so the size of the weights in memory is 2*E*log(E) bits.
Finally each processor will only access a part of the whole graph in memory, we must not forget that memory accesses are made in parallel (escept for the main memory but we will talk about it later on). so the final formula for the size of the graph in memory is :

$$S = \frac{4 * E * log(N)}{p} + \frac{2 * E * log(E)}{p} \tag{2.1}$$

Average number of egdes

The number of edges is not constant in memory, the bor-el make it shrink at every iteration by at least N/2 edges. So if E = k*N there is at most 2*k iterations of the algorithm. So the average number of edges is :

$$A = (\sum_{i=0}^{i=2*k} E - i * \frac{N}{2}) * \frac{1}{2 * k} \tag{2.2}$$

$$A = (2k * E - \frac{N}{2} * \sum_{i=0}^{i=2*k} i) * \frac{1}{2 * k} \tag{2.3}$$

$$A = (2k * E - \frac{N}{2} * \frac{(2k)(2k+1)}{2}) * \frac{1}{2*k} \quad (2.4)$$

and E=k*N

$$A = \frac{N}{4}(2k-1) \quad (2.5)$$

Memory access time

The architecture used in the experiments of the paper we used have 400 Mhz processors with 16 kb of L1 cache, 4 Mb of L2 cache and a main shared memory. The formula for memory access time is as follow :

$$MAT = HRL1 * t1 + MRL1 * (HRL2 * t2 + MRL2 * t3) \quad (2.6)$$

- HRL1,HRL2 : Hit rate L1,L2
- MRL1,MRL2 : Miss rate L1,L2
- t1,t2,t3 : time in clock cycles to access L1,L2,L3 memories, respectively 1,10 and 100 clock cycles

now lets create a formula to compute such a Memory access time. lets call S the size of the graph, s1 the size of the L1 memory and s2 the size of the L2 memory.

- HRL1 = min(1,s1/S), MRL1 = 1- HRL1
- HRL2 = min(1,s2/(S-s1)), MRL2 = 1 - HRL2
- For big graphs there is memory contention so we have to multiply L3 by the number of processors.

Lets look ate the final formula :

$$Mat = min(1, \frac{s1}{S}) * t1 + (1 - min(1, \frac{s1}{S})) * (min(1, \frac{s2}{S - s1}) * t2 + (1 - min(1, \frac{s2}{S - s1})) * t3 * p) \quad (2.7)$$

We will plot the curve for this equation in figure 2.1 and 2.2. to show the memory contention happening in a parallel systems. As we ca see in figure 2.1 When the graph is big, the memory access is p*100, meaning most of the memory accesses are done sequentially in the main memory. In figure 2.2 we see that the memory accesses are done in 1 clock cycle for very small graphs and 10 clock cycles for bigger graphs independently of the number of processors. That is because most of the graph is stored in the L1 and L2 cache of the processors so memory accesses are done in parallel.

Number of memory accesses

The research paper [6] provides the number of memory accesses and the computation complexity of the algorithm:

$$NumMemoryAccesses = (\frac{8 * E + N + N * log(N)}{p} + \frac{4mclog(\frac{2m}{plog(z)})}{p}) * log(N) \quad (2.8)$$

$$Complexity = \frac{E}{p} log(E) log(N) \quad (2.9)$$

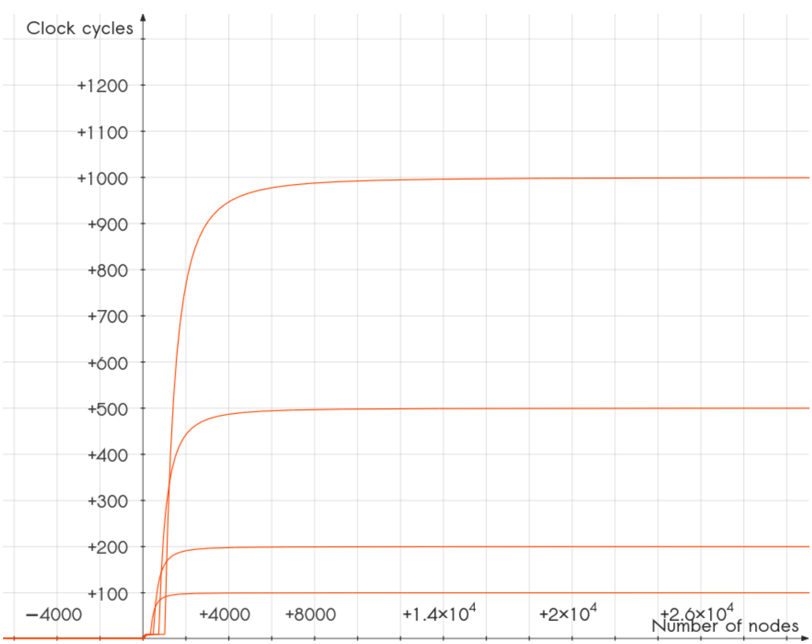

Figure 2.1: Memory access time function of the number of nodes in the graph (with k=N) for p = 1 2 5 and 10

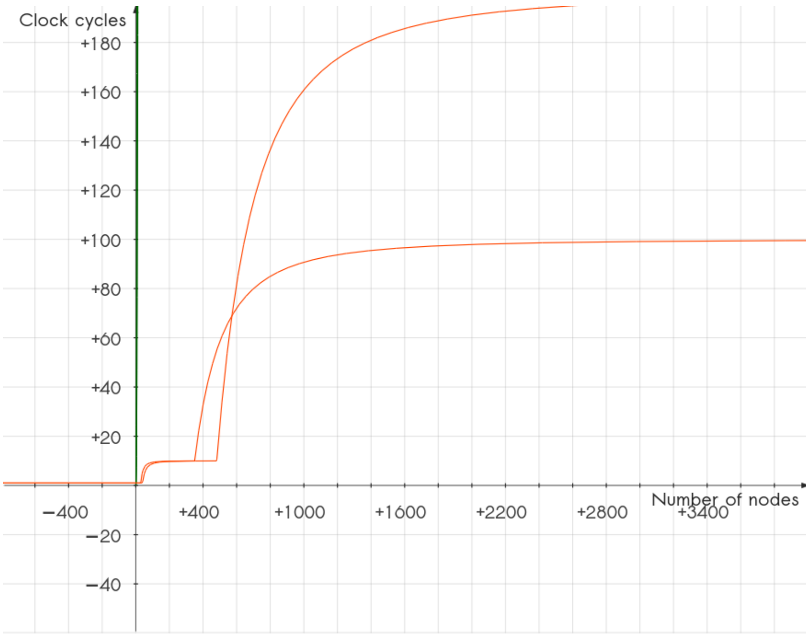

Figure 2.2: Memory access time function of the number of nodes in the graph (with k=N) for p=1 and p=2

The value c/log(z) is unknown but with the experimental results given by [6] we managed to estimate it at 10.

$$ExecTime = \frac{NumMemoryAccesses * MemoryAccessTime}{SpeedOfProcessor} + \frac{Complexity * log(N)}{SpeedOfProcessor} \qquad (2.10)$$

We will plot the memory access time in figure 2.3. We will see that the number of processors lower the execution time but not as much as we could expect from the experimental results. As we can see

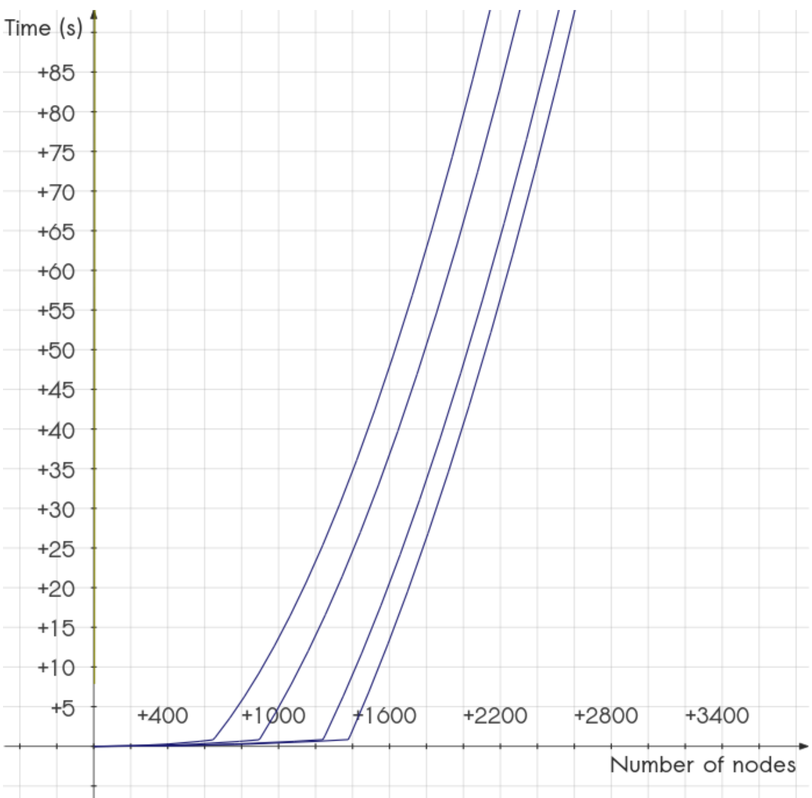

Figure 2.3: Execution time for p = 2 4 8 and 10 function of N with k = N

in figure 2.3 the execution time is better the more processes there is. Nevertheless the theoretical result does not match totally the experimental ones (provided with the paper). We see that for 2000 nodes there is an execution time of 40 seconds for 10 processors when it is experimentally 20 seconds. That is probably is due to a bad evaluation of the memory access time for large graphs. Remember, we said that for large graphs, the memory access time is 100p. But the computer used probably has a smart memory access mechanism that makes the access to ram lower than 100p. If the machine groups some memory accesses together that would make the algorithm run faster for a large amount of processors. This issue still has to be investigated.

2.4 GHS

2.4.1 The algorithm

GHS is a well known distributed boruvka's algorithm in which each node of the graph is a different machine. The target is therefore a truly distributed architecture. In such an architecture, nobody knows the entire graph and each node knows only about its neighbours.The approach is therefore very different from a parallel or sequential algorithm. The principle is that each node sends and receives message from its neighbours to determine their state. In the end each node only knows about its edges that belong to the MST and the edge on the path to the root of the MST. For more information on the algorithm please refer to [7] or [8]

2.4.2 Theoretical analysis

For GHS its a bit simpler. We can estimate that most of the execution time is taken by the time it takes to send and receive a message between two nodes. This approach is quite reductive but we will see in the experimental results that it still is true even if the execution time we will notice is very different. To compute such an execution time we have to know the number of messages sent and the time it takes to send one and wait for an answer.

$$NumMessagesPerNode = \frac{(2E + 5N(log(N) - 1) + 3N)}{N} \tag{2.11}$$

$$MaxSizeOfMessage = log(E) + log(8N) \tag{2.12}$$

$$ConnectionSpeed = 1Gb/s \tag{2.13}$$

$$ExecutionTime = \frac{2(\frac{(2E+5N(log(N)-1)+3N)}{N})(log(E) + log(8N))}{ConnectionSpeed} \tag{2.14}$$

These values are provided by [7]. The execution time is twice the time to send a message because every nodes waits for the message to be answered. In figure 2.4 we will plot the execution time of GHS. We can see in figure 2.4 that this GHS algorithm runs extrememly fast in theory. Nevertheless we will see in the next chapter that several factors made the experimental results very different from what we could expect.

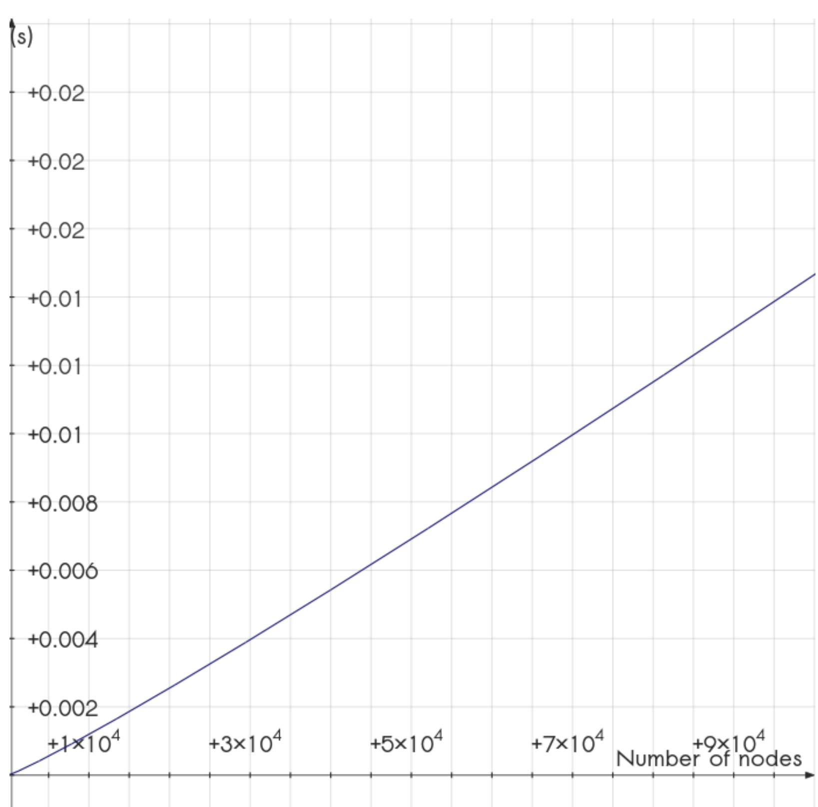

Figure 2.4: Execution time for GHS function of N with k = N

Chapter 3

Comparison

In this chapter we will discuss on the theoretical results we found in the previous chapter. Then we will conduct experiments by running GHS on the Das-5 cluster. finally we will discuss the results and explain the differences between theoretical and experimental analysis.

3.1 Theoretical results

We notice in figure 2.4 that theoretically GHS is always way faster than Bor-el. The results do not even compare.it takes 0.02 seconds for GHS to compute a graph with 1 million nodes where it takes Bor-el 80 seconds to compute a graph with 2000 nodes. We thought in our hypothesis that the distributed algorithm would run slower for small graphs. It is obvious here that it is not the case. However these results are only theoretical and have to be compared with the experiments.

3.1.1 Experimental setup

The experiments are only focused on GHS since for Bor-el we already have experimentals results provided by [6]. We chose an implementation of GHS in C using MPI (Message passing Interface). The experiments were conducted of the uva cluster DAS-5. It has a 1Gb/s connection between nodes and 14 usable nodes at the time of the experiment. Each node has 16 cores which limits the size of the graph to 224 nodes. for more information on this cluster please refer to [9].

3.1.2 Graphs used

The implementation used [10] only works for a specific type of graphs. The weights of each row of the adjacency matrix representing the graph had to be increasingly big. We chose graphs where every node is connected to every other and generated them with a custom python script. You can see in figure 3.1 an example of the graphs used.

3.1.3 Experimental results

We ran the algorithm several time for 16, 32, 64, 128 and 224 nodes in the graph. We distributed the processes as shown in the figure 3.2. We ran each experiment 5 times and computed the average execution time every time. The graph 3.3 shows these results. As we can see the experiments do

0	1	2	3	4
1	0	5	6	7
2	5	0	8	9
3	6	8	0	10
4	7	9	10	0

Figure 3.1: Example of graph used for N = 5

size of graph	number of nodes	processes per nodes
16	1	16
32	2	16
64	4	16
128	8	16
224	14	16

Figure 3.2: Distribution of the algorithm during experiments

GHS execution time function of number of nodes

Figure 3.3: Execution time in seconds for GHS function of N with k = N

not match at all the theoretical results. Here GHS takes 50 seconds for 200 nodes (40 thousand edges) when Bor-el takes 80 seconds for 4 million edges [6]. We thought GHS would run faster for big graphs but now we have results that show that the execution times do not even have the same order of magnitude ! Several factor can cause such a difference and we will see in the next part what they are.

3.2 Discussion

We find that the experimental results are far from what we predicted in theory. Several reasons might have caused such a difference and we will investigate them in this section.

3.2.1 Communication delay

The first cause of such a difference between theoretical and experimental analysis is the communication delays. Theoretically, for our model to match the experimental results we would need to add a delay of 0.05 s every time a message is sent. We benchmarked this value on the Uva cluster with a simple send and receive message script and found a communication delay of 0.025s on average. This is a delay caused by the MPI framework itself and not the cluster. Normally MPI is used with smart mechanism that group the messages and avoid sending too much on the network. In our algorithm each message is sent individually and this delay is therefore multiplied by the number of messages sent.

3.2.2 Number of messages sent

We found a delay of 0.025 which is half what we need to explain the difference between theoretical and practical analysis. We then checked if the implementation we used sent too many messages. you can see the results in figure 3.4. We can see that for the biggest number of graphs the number of

number of nodes	Theoretical value	Experimental value
224	110410	216712
128	37100	56717
64	10250	8200
32	2710	1573

Figure 3.4: Comparison between the theoretical and experimental value of the number of messages sent

messages sent is approximately twice as big as the theoretical value. This can be caused by sloppy programming in the implementation we chose or a bad estimation of the number of messages in [7]. The first option is nevertheless more probable since we already saw that the code does only run properly for a specific kind of graphs. This factor two in the number of messages sent explains the factor two in the theoretical delay we computed. Indeed, if we needed 0.05 seconds delay to explain the difference fo K messages,if there are 2K messages we only need a theoretical delay of O.O25 seconds to explain the difference. This theoretical delay therefore matches the experimental delay we found with our benchmark. There can of course be other factors that explain this difference but we saw that they must not be significant and that most of the bad performances were caused by this communication delay. These other factors might be memory access and computation complexity but they would only be relevant if we are talking about gaining 0.1 of 0.01 second during the execution. Further work could be done to investigate such factors.

Chapter 4

Conclusion

Graph processing is a broad topic and many approaches currently exist to tackle such a problem. Nowadays graphs are extremely big running sequential algorithm on such data is very time consuming. Distributed and parallel computing have been found to be very promising on solving this issue. Nevertheless it is not clear which approcach is the best and in which context since parallel and distributed architectures are very different and the algorithms running on them are even more.

Given the results of this research we cannot state whether bor-el is better than GHS. Indeed, even if the experimental result state that bor-el is infinitly faster, it only is because there has been a bad choice in the implementation of GHS with MPI. Any distributed algorithm using MPI this way will suffer from extremely bad performances. We can see that programming a distributed algorithm leads to making choices that can lead to tremendous performance issues.To compare these two algorithms we would need a GHS implementation that make the most of the DAS-5 cluster of the MPI framework.
The results given in [6] give a quite good scaling capability,nevertheless in theory GHS scales better and gives incredible performance. We can only say that with a proper setup, GHS could process huge graph in no time. The existence of such a setup (with extremely low communication latency) is not certain though.
Finally we now see that GHS runs extremely badly on a frameworks that is not intended for this purpose. One has to be extremely careful when running such an algorithm outside of its target architecture and framework. Nevertheless, on a parallel architecture and for small graphs (one node per core of a processor), GHS would match the theoretical execution time. But It would miss its main goal which is to compute algorithms for very big graphs. In that sense Bor-el scales better on parallel machines.

As stated earlier, we can't conclude this study properly only because of implementation issues. It would be interesting to find (or make) a GHS implementation that makes the most of the Das-5 cluster and either uses MPI smartly or another framework. We could then compare each algorithm at the top of their performance. It would also be interesting to see how bor-el performs on the DAS-5 cluster. It would seem that it would perform quite good given the low number of communication between processes needed.

Chapter 5

Additional Informations

5.1 Annexes

5.1.1 Graph generation code

```
import sys
n = int(sys.argv[1])
adj_mat  = list()
for i in range(0,n):
        lst=[0]*n
        adj_mat.append(lst)

weight = 1
start=0
for i in range(0,n):
        for j in range(start,n):
                if i!= j:
                        adj_mat[i][j]=weight
                        adj_mat[j][i]=weight
                        weight+=1
                else:
                        adj_mat[i][j]=0
                        adj_mat[j][i]=0
        start+=1
for i in range(0,n):
        line = str(adj_mat[i])
        line =line.replace("[","{")
        line =line.replace("]","},")
        print line
```

5.1.2 Benchmark of communication delay

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

/*************************************************************
This is a simple send/receive program in MPI
*************************************************************/

int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs;
    int tag,source,destination,count;
    int buffer;
    double startwtime = 0.0, endwtime;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    tag=1234;
    source=0;
    destination=1;
    count=1;
    startwtime = MPI_Wtime();
    if(myid == source){
      buffer=5678;
      MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
      printf("processor %d  sent %d\n",myid,buffer);
    }
    if(myid == destination){
        MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
        printf("processor %d  got %d\n",myid,buffer);
    }
    endwtime = MPI_Wtime();
    printf("wall clock time = %f\n", endwtime - startwtime);
    MPI_Finalize();
}
```

code taken from [11]

5.2 Acknowledgements

I would like to thank my supervisor Ana Lucia Varbanescu who helped a lot during the whole research and gave extensive feedback during the writing of this report.

Bibliography

[1] Minimum spanning tree and prim's algorithm. http://www.cs.umd.edu/~meesh/351/mount/lectures/extra-mstprim.pdf.

[2] Carl Kingsford. Kruskals minimum spanning tree algorithm and union-find data structures. https://www.cs.cmu.edu/~ckingsf/class/02713-s13/lectures/lec03-othermst.pdf.

[3] Jaroslav Nesetril and Helena Nesetrilova. The origins of minimal spanning tree algorithms boruvka and jarnik.

[4] Rolf Riesen and Ron Brightwell. Differences between distributed and parallel systems.

[5] https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm.

[6] Guojing Cong David A. Bader. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. 2006.

[7] P. A. Humblet R. G. Gallager and P. M. Spira. A distributed algorithm for minimum weight spanning trees.

[8] www.di.univaq.it/~proietti/slide_algdist2010/3-MST.ppt.

[9] http://www.cs.vu.nl/das5/clusters.shtml.

[10] https://github.com/bssilva/ghs-with-mpi.

[11] http://geco.mines.edu/workshop/class2/examples/mpi/c_ex01.c.