Final Project

Arthur Martins Braga

SLR207 - Technologies de calcul parallèle à grande échelle

June 28, 2024

1 Introduction

Provide a brief introduction to the project, including its objectives and scope.

2 Building, Executing, and Deploying the Project

To successfully build, execute, and deploy this distributed computing project, follow the steps below:

2.1 Prerequisites

- Ensure that Maven (`mvn`) and Python are installed on your local machine for building and deploying the project.

2.2 Building the Project

Navigate to the rood directory (the one that contain the folder of all projects: `core`, `master`, `server`) and run the Maven command to build the project. This command will compile all projects with all dependencies included.

```
mvn clean compile assembly:single
```

2.3 Preparing for Execution and Deployment

Login Credentials:

In the `SendDeploy.py` script, the login and password are prompted at runtime. Therefore, you do not need to change the script for different logins.

Update Machine Addresses:

In the same `SendDeploy.py` script, update the `computers` list and the `master` variable with the addresses of the new target machines. This list is used to deploy the project across multiple machines.

```
computers = [
    "new-machine-1", "new-machine-2", # Add more as needed
]
master = "new-master-machine"
```

2.4 Deploying and running the Project

Run the `SendDeploy.py` script. This script automates the process of deploying the JAR files to the specified machines. It will:

- Kill any existing instances of the project.
- Clear the deployment directory.
- Copy the new JAR files to the target machines.
- Execute the project on each machine.
- Prompt the output of the master machine.
- Collect the metrics generated from the master machine.

Before running the script, ensure you are connected to the Télécom VPN.

```
python SendDeploy.py
```

Enter your login and password when prompted.

The execution of the project on the master and server machines is handled by the `SendDeploy.py` script as part of the deployment process.

2.5 Generated artifacts

In each node, the final processed files will be stored in the directory: `/dev/shm/braga-23/fina`
The master generates a `metrics.csv` file, which will be automatically downloaded to your machine when `SendDeploy.py` completes its execution.

2.6 Other Files Available

In the repository, you can also find the following files:

- `Plots/notebook.ipynb`: This Python code was used to generate the graphs included in this report.
- `Plots/images/`: This directory contains several generated graphs.
- `report_results.csv`: This file contains the data used to produce this report.

These additional files provide supporting materials used in the analysis and preparation of this report.

3 Achievements and Issues

3.1 Achievements

The project successfully implemented a distributed computing system capable of processing significant amounts of data using a MapReduce-like framework. The system efficiently handled datasets up to 660 MB and demonstrated substantial performance improvements with the integration of various optimizations, such as memory-based file storage and multi-threading. By employing both FTP for file transfers and sockets for message communication, the project achieved reliable data exchange between nodes.

3.2 Remaining Issues

Regarding the issues present in the code, the most significant problem is the difficulty in handling very large files. The current implementation reads files from CommonCrawl and stores all the content in different lists in Java. This approach creates a significant overhead and requires a substantial amount of RAM, as everything is stored in memory. Consequently, testing was only possible with two CommonCrawl files, averaging 660 MB, which was the maximum amount of data the master could transfer to the nodes without issues. This problem would require more time to address and resolve properly.

Another issue encountered was that the code, initially, did not function with more than 20 nodes, which appeared to be a limitation on the number of simultaneous connections in a Java FTP server. This problem was improved by optimizing the connection handling, ensuring that nodes only connect when sending files and then disconnect immediately afterward. While this optimization may increase the time taken to process the data due to the time required to initialize a connection, it enabled stable connections with up to 40 machines. However, beyond this number, the program becomes unstable, and the exact reason for this instability remains unknown.

4 Dataset and Computation

4.1 Maximum Dataset Size

The largest dataset processed consisted of two files from the CommonCrawl directory, totaling 660 MB.

4.2 Present Phases

The project implements a MapReduce-like process with multiple phases. Here are the phases present in this project:

Split Phase

This initial phase involves reading files and splitting their content for distribution to the nodes. The maximum amount of data this program can handle is approximately 990 MB, equivalent to three files from CommonCrawl. The amount of data processed is determined by the second argument of the Java program, a float between 0 and 1 representing the percentage of total data to distribute across the network. All files are read, and each line has a probability defined by "percentage" to be included in the distribution. Subsequently, the code performs an operation `lineNumber % numberOfNodes`, which determines the index representing the node to which each line should be sent. To speed up the file division process, n threads are created, where n equals the number of nodes.

First MapReduce Phases

- **Map1**: Processes the file sent by the master containing the splits, removes special characters such as commas, converts all letters to lowercase, and separates words by spaces. It stores them in a hashmap where key = word and value = count. If the word repeats, it increments the count.
- **Shuffle1**: Converts the key into a hash and performs a modulo operation based on the number of nodes. This determines which node will receive the file. Files are then sent to respective nodes.
- **Reduce1**: After all nodes complete the shuffle, the reduce phase aggregates all received shuffle files in memory, summing words into a single hashmap.

Group Phase

With the hashmap obtained in Reduce1, nodes inform the master of the highest and lowest word frequencies found. The master waits for responses from all nodes, calculates the frequency range, generates frequency intervals for each node, and distributes information to nodes about which words to send with specific frequencies.

Second MapReduce Phases

- **Map2**: Nodes receive frequency intervals and separate data in memory according to their respective intervals. This results in a list of hashmaps representing data to be sent.
- **Shuffle2**: During shuffle, nodes transfer separated data to respective nodes based on defined intervals.

- **Reduce2**: After all nodes finish shuffle, reduce processes all received files into a single output file named `finalResult.txt`. Data is sorted by frequency and then alphabetically before writing.

Final Phase

This phase notifies nodes that the process is complete and they can delete generated files to start a new session. After receiving acknowledgment from all nodes, the master concludes its process and writes performance metrics to an optional CSV file.

4.3 Timing of Each Phase

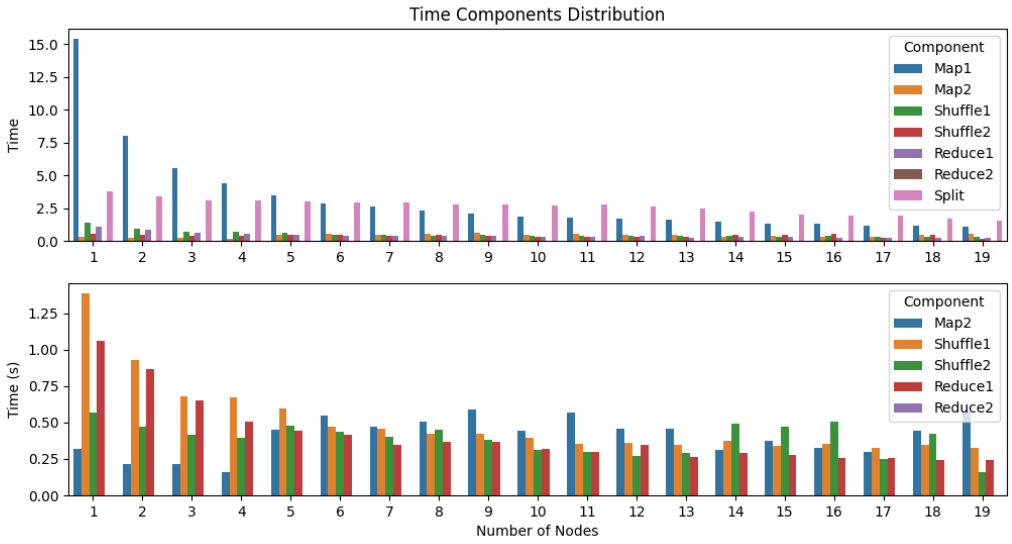

Figure 1

In Figure 1, it's possible to observe the time spent in each phase of the process. It's clear that phases Map1 and Split consume the most time. However, they also show the most significant change when increasing the number of nodes in the system. In the lower part of the graph, it's possible to analyze the other phases in more detail, noting a significant reduction in Shuffle1 and Reduce1 as the number of nodes increases.

5 Experimental Results

5.1 Different Number of Computers

The tests were conducted with a fixed data size of 264MB while varying the number of machines.

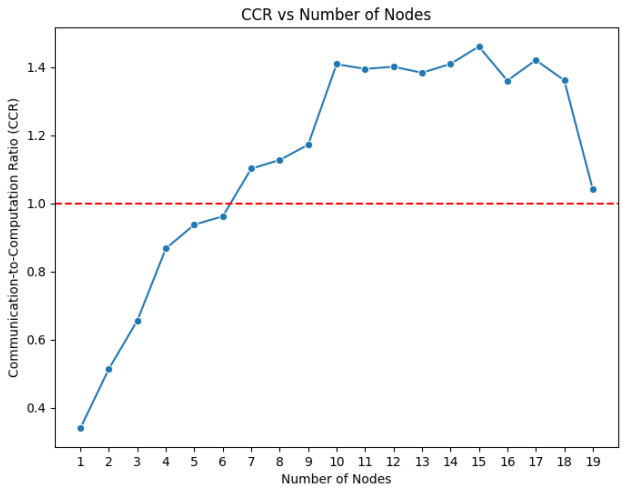

Figure 2: Performance metric vs. number of nodes.

In Figure 2, it's evident that with 7 nodes, the processes start to spend more time on communication than on computation itself. The final drop in the metric appears to be just noise.

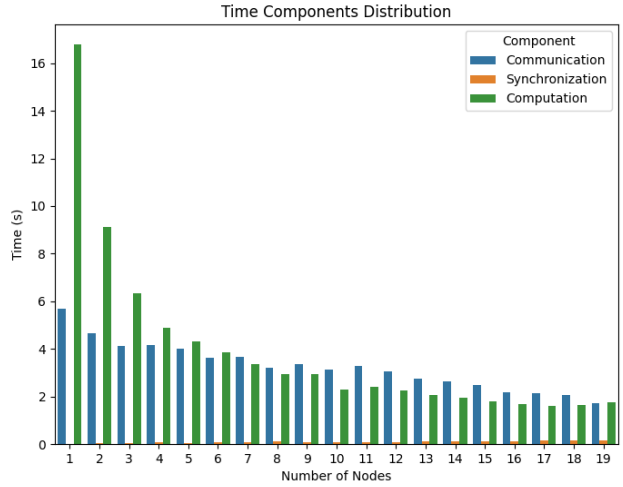

Figure 3: Time distribution across different components.

In Figure 3, it is noticeable that the majority of time is spent on computation when the number of nodes is low, but this time quickly decreases as the number of nodes

increases. Synchronization takes an insignificant amount of time but seems to grow with the number of nodes.

From 9 nodes onwards, communication time surpasses computation time, yet it tends to decrease as the number of nodes increases.

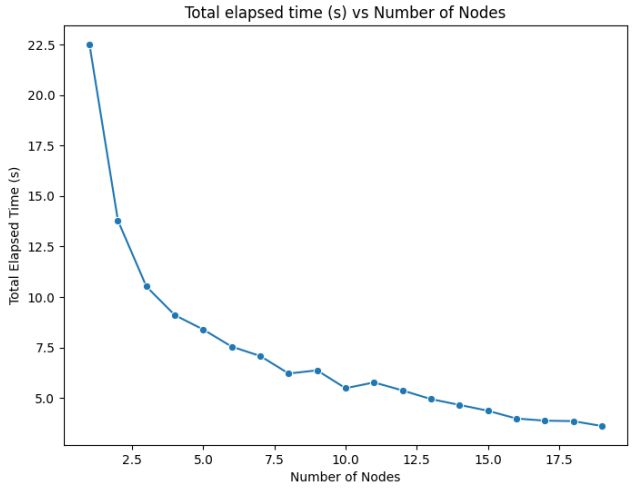

Figure 4: Total elapsed time vs. number of nodes.

As expected, Figure 4 shows that the total time significantly reduces with fewer nodes. However, as the number of computers increases, the time continues to decrease but at a much slower rate.

5.2 Different Dataset Sizes

The tests were conducted with 6 nodes while varying the dataset size.

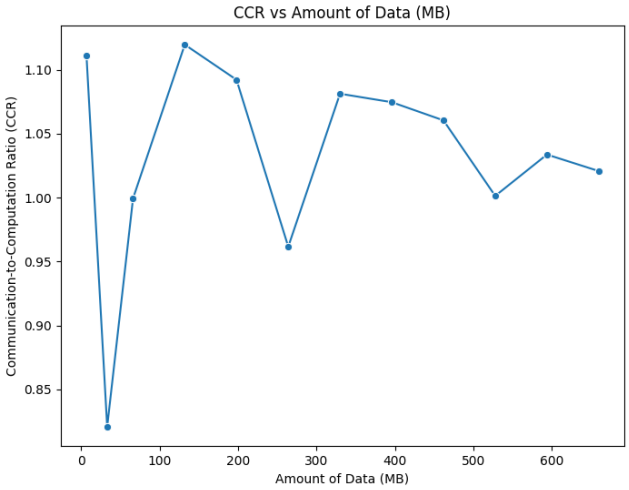

Figure 5: Performance metric vs. amount of data.

In Figure 5, no correlation is observed between the CCR metric and the increase in data size.

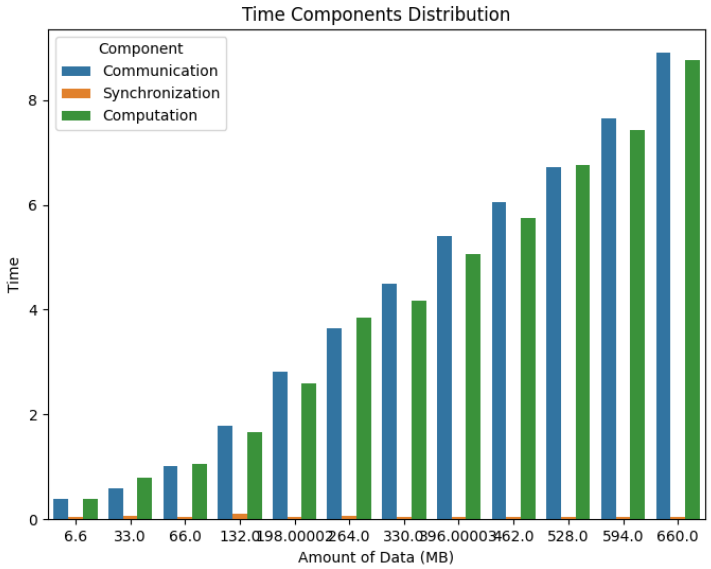

Figure 6: Time distribution across different components.

In Figure 6, it is evident that both communication and computation times increase linearly with the amount of data processed.

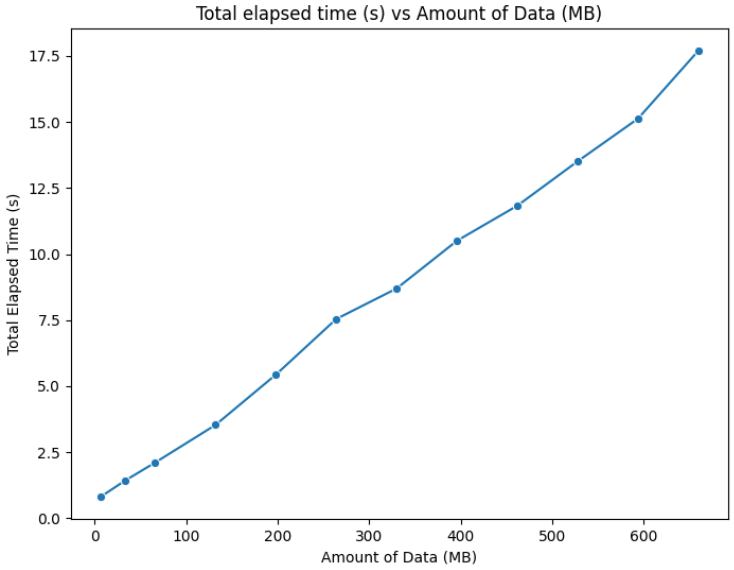

Figure 7: Total elapsed time vs. amount of data.

In Figure 7, it's also evident that the total computation time increases linearly with the amount of data processed.

6 Validation of Amdahl's Law

6.1 Reference for Speedup

In this experiment, the speedup was calculated using a reference execution of the same MapReduce-based algorithm. This reference execution was performed using a single node as shown in Table 1.

Number of nodes	**Amount of data (MB)**	**Total elapsed time (s)**
1	247.5	5.380117999
1	99.0	2.493672948
1	49.5	1.319524045
1	9.9	0.236494411

Table 1: Reference values for Speedup

6.2 Speedup Graph Analysis

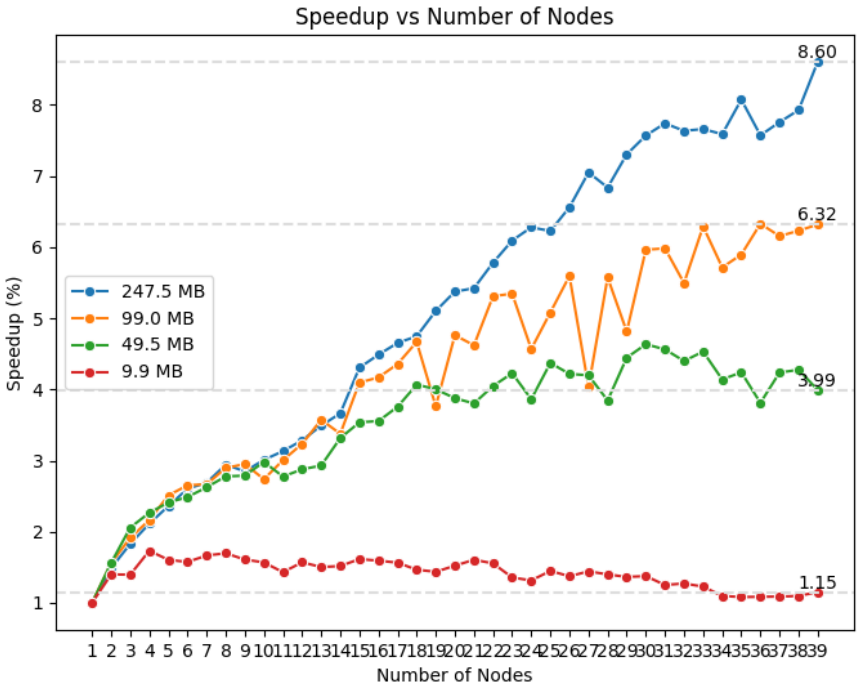

Figure 8: Speedup versus Number of Nodes

In Figure 8, it is noticeable that the curve varies according to the amount of data sent over the network. This variation occurs because as we increase the data volume, the code becomes more parallelizable, yielding greater advantages by parallelizing tasks.

The plot shows that with 247.5 MB of data, we did not reach the "limit" of the curve with just 40 machines. This amount was insufficient to replicate the expected behavior of Amdahl's Law.

At 99.0 MB, the curve begins to stabilize around 6.32.

The 49.5 MB curve closely follows the expected curve, stabilizing at a speedup of 4.

However, at 9.9 MB, the speedup stabilizes at 1.15. Furthermore, increasing the number of machines worsens the performance, indicating that the overhead of parallel processing outweighs the time saved in computation.

These observations highlight the importance of balancing data volume and parallel processing efficiency in optimizing distributed computing systems.

6.3 Global Parallel Portion

In distributed computing systems, understanding the scalability and parallelizability of tasks is crucial. One approach to gauge the parallelizability is by analyzing the

speedup achieved with increasing computational resources, typically represented by the number of nodes.

A metric often used for this purpose is the *global parallel portion*, which indicates the portion of the workload that benefits from parallel execution across multiple nodes. This can be inferred from empirical data, such as the performance metrics observed during experiments.

To quantify the global parallel portion more precisely, we can use the formula:

$$P = \frac{S_p - 1}{S_p - \frac{1}{n}}$$

where S_p represents the speedup achieved with n nodes. This formula provides a numerical measure of the scalability of the system, helping to optimize performance and resource allocation in distributed computing environments.

To deduce the global parallel portion, we analyze the relationship between the amount of data processed and the corresponding speedup achieved. From the experimental results, we have gathered the following data points:

Amount of Data (MB)	**Global Parallel Portion**
247.5	0.907013
99.0	0.863840
49.5	0.769256
9.9	0.136352

Table 2: Data showing the relationship between amount of data processed and global parallel portion.

From Table 2, it is evident that as the amount of data increases, the global parallel portion tends to decrease slightly. This suggests that a larger dataset reduces the efficiency gain from parallelism due to increased communication overhead or other factors affecting scalability.

The global parallel portion values indicate the proportion of the workload that can be effectively distributed and executed in parallel across multiple nodes. Higher values indicate better parallelizability, where a larger portion of the workload benefits from increased parallel resources.

7 Optimizations

7.1 Memory vs. Hard Disk

Programs are executed in the /tmp directory on Linux; however, files transferred via FTP are stored in the /dev/shm/ directory. This makes reading them faster, as they are already loaded into the computer's memory.

7.2 Communication Methods

The system employs both FTP and sockets for communication. FTP is used for file transfers, while sockets are used for sending messages. FTP connections are opened only when necessary to retrieve or send files, preventing issues related to reaching the maximum connection limit.

7.3 Other Optimization Approaches

Other optimizations involve the use of threads by both the node and the master to accelerate certain processes, which will be discussed in the next section.

8 Thread Usage

8.1 Current Thread Usage

Threads are constantly used in the code. The master creates a thread for each client socket. Asynchronous processes are managed using the CompletableFuture class, which allows for readable code and great flexibility in defining sequences and synchronization.

Threads are also used when the node or master handles large amounts of files. For splitting files, the master uses parallel streams.

Additionally, threads are utilized when nodes generate group files during the group phase, creating multiple threads to expedite file division.

8.2 Potential Thread Usage

When dealing with large amounts of data, it is always beneficial to explore the use of threads. Future optimizations should aim to apply threads in all phases of each node to divide tasks and further accelerate processes.

9 System Resilience

The system's resilience to node crashes is not explicitly addressed. If a node crashes, it could potentially halt processing or result in incomplete data aggregation. To mitigate this issue, the algorithm could be modified to include task replication across multiple nodes for fault tolerance or to implement a consensus protocol for state synchronization. These strategies would enhance the system's robustness and ensure the operation even in the event of individual node failures.

10 Conclusion

This project successfully implemented a distributed computing system capable of processing large datasets using a MapReduce-like framework. The system effectively handled up to 660 MB of data, demonstrating significant performance improvements through various optimizations such as memory-based file storage and multi-threading. The integration of FTP for file transfers and sockets for message communication ensured reliable data exchange between nodes.

A notable achievement was the ability to establish stable connections with up to 40 machines by optimizing connection handling. However, the program exhibited instability beyond this number, with the cause of this instability remaining unidentified.

Despite these successes, the project faced challenges in managing very large files due to the high memory overhead of storing all file content in lists. Addressing this issue would require further development to reduce memory usage and enhance scalability.

Future work should focus on improving the system's resilience, potentially incorporating fault tolerance mechanisms to handle node failures more gracefully. Additionally, exploring further optimizations, particularly in the use of threads, could enhance the system's efficiency and scalability. Overall, the project provides a solid foundation for distributed computing, with ample opportunities for refinement and extension.