Comparing parallel matrix multiplication performance in different platforms using OpenMPI

Henrique Figueiredo Conte

January 2023

1 Introduction

In this project, we will implement the parallel matrix multiplication problem in C++ using the OpenMPI library. Then, we will compare the execution time with different amount of processes running locally, on Grid5000 and finally on SimGrid.

This will allow us to experiment with different parallel machines, learning how to use them and leveraging their pros and cons. Besides, we will learn how to use OpenMPI to create parallel programs.

2 The experiment

In order to compare the parallel matrix multiplication performance between different platforms, we will perform the following tests on my own machine, on Grid5000 and on SimGrid:

- Sequential execution with 1 process for matrices with size 50, 500 and 3000.
- Parallel execution with 2 processes for matrices with size 50, 500 and 3000.
- Parallel execution with 4 processes for matrices with size 50, 500 and 3000.
- On SimGrid and on Grid5000, parallel execution with 16 processes for matrices with size 50, 500 and 3000.

In that case, my machine is a MacBook Pro 2018 2,3 GHZ Quad-Core Intel Core i5, with 8GB of RAM.

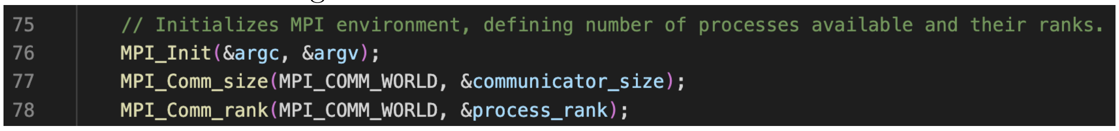
For all tests, we will get the execution time for them. This will show us if there is any improvement by changing the amount of processes executing the task, and how this improvement changes according to the size of the task.

3 Implementation

The parallel matrix multiplication problem was solved in C++, using the Open-MPI library. The full code is available on the folder along with this report, and it is also available on Github.

Let's see the most important parts of the code. First, we need to initialise the MPI environment:

Figure 1: MPI environment initialisation



```
// Initializes MPI environment, defining number of processes available and their ranks.
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &communicator_size);
MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);
```

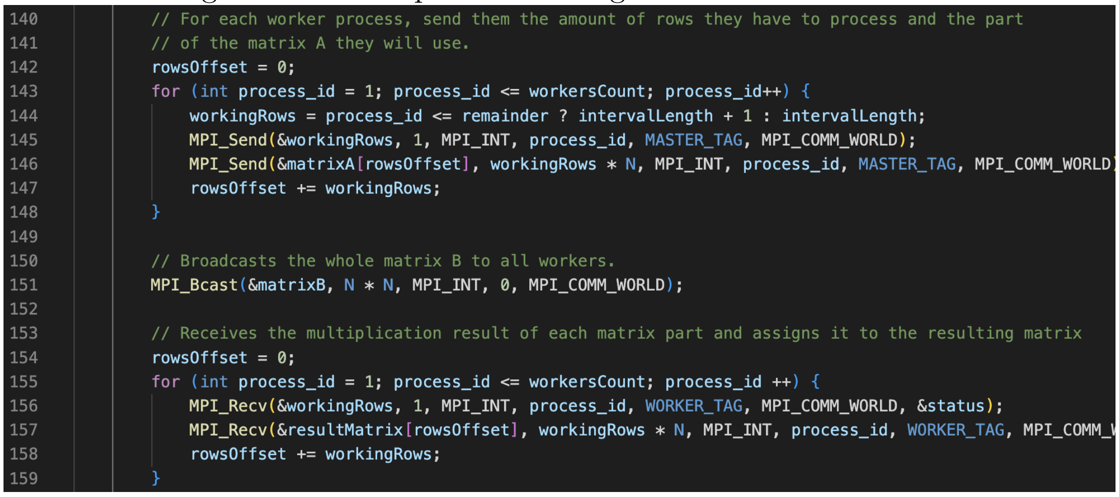
This will give us the amount of processes we will be working with, as well as their ranks.

Then, if we have at least two processes assigned to the execution, we will have one master processes and all the others will be called workers. On the master process, we will initialise a matrix A and a matrix B with random values from 0 to 10. Those matrix will be multiplied to get our final matrix C.

After having our matrices A and B, we will distribute the work among the worker processes. To do that, we need to know the amount of workers we have; the amount of matrix rows each worker will process; the amount of remainder rows in case we can't equally distribute the rows among the worker processes.

After having that information, we can run the synchronous communication between the processes. The code below belongs is executed by the master process:

Figure 2: Master process sending information to workers



```
// For each worker process, send them the amount of rows they have to process and the part
// of the matrix A they will use.
rowsOffset = 0;
for (int process_id = 1; process_id <= workersCount; process_id++) {
    workingRows = process_id <= remainder ? intervalLength + 1 : intervalLength;
    MPI_Send(&workingRows, 1, MPI_INT, process_id, MASTER_TAG, MPI_COMM_WORLD);
    MPI_Send(&matrixA[rowsOffset], workingRows * N, MPI_INT, process_id, MASTER_TAG, MPI_COMM_WORLD)
    rowsOffset += workingRows;
}

// Broadcasts the whole matrix B to all workers.
MPI_Bcast(&matrixB, N * N, MPI_INT, 0, MPI_COMM_WORLD);

// Receives the multiplication result of each matrix part and assigns it to the resulting matrix
rowsOffset = 0;
for (int process_id = 1; process_id <= workersCount; process_id ++) {
    MPI_Recv(&workingRows, 1, MPI_INT, process_id, WORKER_TAG, MPI_COMM_WORLD, &status);
    MPI_Recv(&resultMatrix[rowsOffset], workingRows * N, MPI_INT, process_id, WORKER_TAG, MPI_COMM_W
    rowsOffset += workingRows;
}
```

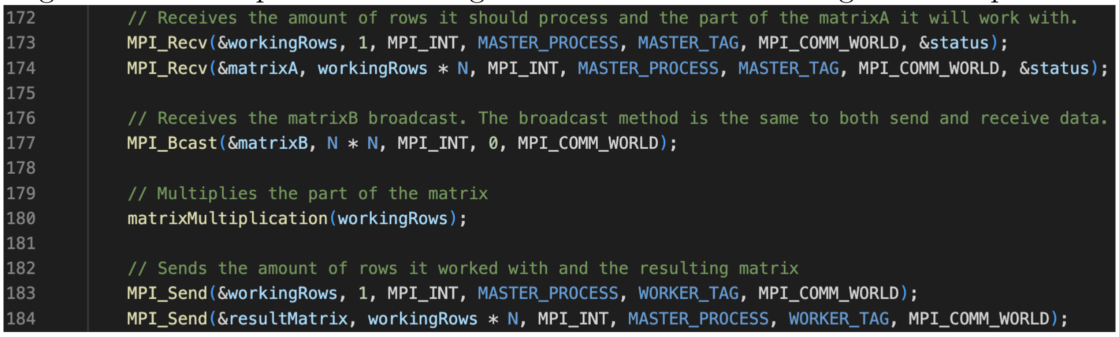
What the master process is doing is iterating among all workers and sending them their respective part of the matrix A they will be working with, along with the amount of rows they need to process. It does that by using the method **MPI_Send**.

Then, since all workers will use matrix B, we can broadcast it by using the method **MPI_Bcast**. This method has the same signature to both send and receive data.

Finally, after the workers have finished their job, we will receive their response by using the method **MPI_Recv**. In order to assign the result to the right portion of the matrix C, we use an offset that is incremented every iteration.

Now, on the worker side, we will to the mirrored send and receive operations, but we will also execute the matrix multiplication with the received matrix parts.

Figure 3: Worker process receiving information and executing the multiplication



```
    // Receives the amount of rows it should process and the part of the matrixA it will work with.
    MPI_Recv(&workingRows, 1, MPI_INT, MASTER_PROCESS, MASTER_TAG, MPI_COMM_WORLD, &status);
    MPI_Recv(&matrixA, workingRows * N, MPI_INT, MASTER_PROCESS, MASTER_TAG, MPI_COMM_WORLD, &status);

    // Receives the matrixB broadcast. The broadcast method is the same to both send and receive data.
    MPI_Bcast(&matrixB, N * N, MPI_INT, 0, MPI_COMM_WORLD);

    // Multiplies the part of the matrix
    matrixMultiplication(workingRows);

    // Sends the amount of rows it worked with and the resulting matrix
    MPI_Send(&workingRows, 1, MPI_INT, MASTER_PROCESS, WORKER_TAG, MPI_COMM_WORLD);
    MPI_Send(&resultMatrix, workingRows * N, MPI_INT, MASTER_PROCESS, WORKER_TAG, MPI_COMM_WORLD);
```

After all workers have finished their execution, we will have our final matrix and we can display it to the user, along with the execution time.

3.1 Grid5000

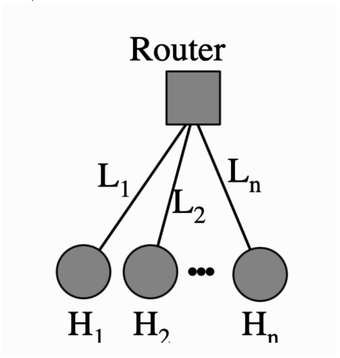
By using Grid5000, we can analyse the execution behaviour with multiple nodes. After creating an account, I had to **ssh** to connect with Grid5000 using my login. Then, to a Grid5000 site, like Grenoble, Nancy or Rennes. For my case, I chose Nancy because the others were in maintenance. Finally, I created a directory for MPI files, and I copied and pasted my parallel matrix multiplication code there.

In order to run the code, I had first to reserve 16 nodes using the command **oarsub -I -l nodes=16**. Then, to execute it, we can use the command **mpirun –mca pml ûcx -n 4 ./par_matrix_mult**, that will use 4 nodes and ignore warning and error messages from Grid5000.

3.2 SimGrid

For the parallel matrix multiplication problem, the main difference for us will be that we can decide the network topology when using SimGrid. The topology we will use is called Cluster with a Crossbar, provided by SimGrid. This topology has as many ports as hosts, allowing hosts to communicate at full speed, and the hosts are connected by a router.

Figure 4: Cluster with a Crossbar topology. Source: https://simgrid.org/doc/latest/Platform_examples.html



To run with SimGrid, we need to first compile the code using the SimGrid's compiler version of MPICXX. The command to do it is **smpicxx par_matrix_mult.cpp -o simgrid_par_mult**. Then, to run the code, we have to decide the topology it will use, defined on the flag **-platform**. This flag requires a XML file with the configuration wanteded. In our case, we used the example XML Crossbar topology offered by SimGrid.

Finally, to run the code, we execute the command **smpirun -n 4 -platform cluster_crossbar.xml ./simgrid_mult**, running in 4 different processes.

4 Performance analysis

First, as a common knowledge in Computer Science, the time complexity of the default matrix multiplication problem is $O(n^3)$. When paralleling the execution, ideally the performance would become $O(n^3)/P$, where P is the number of processes.

Nevertheless, when running code in parallel, we have to considerate the communication cost between the processes. If the processes share the memory or are executed in the same machine, the communication will be fast and it won't have a big impact on the performance. On the other hand, if we have to use network to handle the communication, the scenario might change.

Communicating via network is more costly than communicating inside a machine. With that being said, let's consider the variable *COMM_TIME*, as the communication time that will be added to the final complexity.

In the case of the parallel matrix multiplication problem and in the executed communication operations on this code, we have the following new complexities:

- Sending intervals of the matrix from the master to the workers: $O(n^2)$
- Receiving the computed results from the workers to the master: $O(n^2)$
- Broadcasting the second matrix from the master to the workers: $O(n^2)$

Then, since we have to execute those three operations, we would have a complexity increase of $O(COMM\_TIME * 3 * n^2)$. As the communication time gets higher, we will have a quadratic increase in the final execution time. In contrast, the closer the communication time gets to zero, the lower the impact it will have.

5 Results

To analyse the parallel matrix multiplication performance in different platforms, we ran the program with different amount of processes and multiple matrices sizes. For a matrix with size 50x50, we can see the results on the table below:

-	Local machine	Grid5000	SimGrid
1 processor	0.001901	0.002460	0.002811
2 processors	0.001388	0.043499	0.002966
4 processors	0.000825	0.098941	0.002845
16 processors	-	0.511650	0.003748

Table 1: Matrix with size 50x50

A matrix 50x50 can be considered a small one, and increasing the number of processes had an improvement when running on the local machine, but it became worse on Grid5000 and SimGrid. On Grid5000 and SimGrid, the processes are not necessarily executed on the same node, thus we can have latency delay on the communication between multiple processes.

Therefore, by having more processes, the latency has an overhead on Grid5000 and on SimGrid, making the performance worse. The local machine had the best overall performance.

When increasing the matrix size to 500x500, the execution time keeps below one second, but we start seeing the benefits from having multiple processes. On

-	Local machine	Grid5000	SimGrid
1 processor	0.450489	0.882841	0.443908
2 processors	0.381513	0.786401	0.381513
4 processors	0.285171	0.395519	0.325171
16 processors	-	0.610967	0.393449

Table 2: Matrix with size 500x500

the local machine, the execution only got faster, and on Grid5000 and SimGrid we saw an improvement up to 4 processes. By having 16 processes, the communication overhead was too big and it ended up taking more time than when executed with less processes.

-	Local machine	Grid5000	SimGrid
1 processor	146.959205	171.433433	169.921205
2 processors	136.723636	169.302913	145.413732
4 processors	56.783724	57.336790	122.958174
16 processors	-	14.989964	88.279217

Table 3: Matrix with size 3000x3000

Finally, we increased the matrix size to 3000x3000, which is large enough to see the difference among the different amount of processes but it doesn't take too long to execute. As shown in the table, there was a huge performance improvement related to the amount of processes.

The most impressive performance was on Grid5000, where we reduced the execution time from 171 to 14 seconds. Therefore, this shows how powerful it can be to execute tasks in parallel.

6 Conclusions

In conclusion, the matrix multiplication problem demonstrated to have significant benefits from parallelism, especially for bigger matrices. However, the performance didn't improve linearly with the number of processes due to the communication overhead and the fact the not all the lines of the code can run in parallel.

Furthermore, when comparing the execution on the local machine, Grid5000 and SimGrid, the local machine had the best performance for all executions up to 4 processors. However, the local machine reaches its processor limits with 4 of them, which becomes the performance bottleneck.

In that aspect, the platforms Grid5000 and SimGrid allow us to have a bigger parallelism. Besides, Grid5000 had a remarkable performance improvement when running with more processors.

In terms of ease to use, I personally found Grid5000 easier to understand and to use, but SimGrid allow us to easily simulate different network topology, which is a valuable asset. To sum up, both can be used to solve large scale problems with parallelism, and learning them is important to start exploring the parallel systems field.