Tong Wu, Zhaolin Fang, Leonardo Kuffo

Group 6: Assignment 4

MPI & OpenMP

1. Introduction

As part of this assignment related to Parallel Programming, we performed two experiments regarding the computation of the Matrix-Vector multiplication algorithm using two frameworks for parallel programming: MPI & OpenMP. Matrix-vector multiplication has a $O(n^2)$ complexity and due to the way the results are computed, it can be parallelized. Both experiments made use of the Lisa computing cluster at SURF for computing power.

2. Experiments

The idea of both experiments is to parallelize the computation of a Matrix-Vector multiplication. In both experiments, we randomly generated elements of type `double` for the Matrix and the Vector. The goal of both experiments is to measure the speedup of the operation when changing: a) the number of nodes/threads used, b) the size of the Matrix and Vector **(N)** and c) The number of times to perform the multiplication **(R)**. For c), each successive iteration uses the result of the previous multiplication as the new Vector for the next multiplication. It is important to say that the reported execution times do not include the time taken to generate the random vector and matrix.

2.1. Message Passing Interface (MPI)

For this experiment, we set up 1 master and up to 15 workers. The Matrix and the initial Vector are generated on the master process. The master process is going to distribute the Vector to the workers as a broadcast message. The Matrix is going to be distributed among the workers using a **Row-Block** approach. In order to do this, the master process computes the number of rows blocks in which the matrix has to be divided in such a way that each worker receives roughly the same amount of rows. Finally, the master sends to each worker the number of rows they are going to receive and the block of rows of the matrix itself. Logic is implemented in both master and worker to only send/receive the block of the matrix on the first iteration R. Once the worker has received the block of rows, they perform the matrix-vector multiplication.

2.1.1. Data Distribution: As we mentioned, we used a **Row-Block** approach to distribute the matrix among the workers. The latter means to only communicate once with the worker to send all the data that the worker must process. We want to highlight that we first implemented a *Row-Cyclic* approach. However, we discovered that with big values of N the process was unable to finish. This is most likely due to the huge communication overhead that was created by sending the matrix in a cyclic fashion to remote processors (i.e. one row per message, leading to N communications between masters and workers). In a cyclic fashion, even the smallest milliseconds of latency in communication can add up to hours of the execution of sending and receiving all the rows on a big matrix. Hence, we preferred the Row-Block approach. Using the latter, workers will only communicate 1 time with the master to receive the matrix block they have to process. Even though we are sending more information, the latency of the communication only happens once. The workers will be idle before starting their first processing, however, they will not be idle between communications.

2.1.2. Results and Discussion: Figure 2 shows each execution time in seconds with respect to the number of threads used for computation for a fixed N of 31K and R of 500. Using 2 threads the execution time was 684.32 seconds. On the other hand, using 16 threads the execution time was 60.56 seconds. This means that we achieved a speedup of roughly 11.4. We can clearly see that the execution time reduces in a logarithmic fashion until we reach 16 processors. In addition to this, the

more number of procesors the smaller the speedup. This could be in part due to the communication and latency overhead that the master suffers when communicating with more workers.

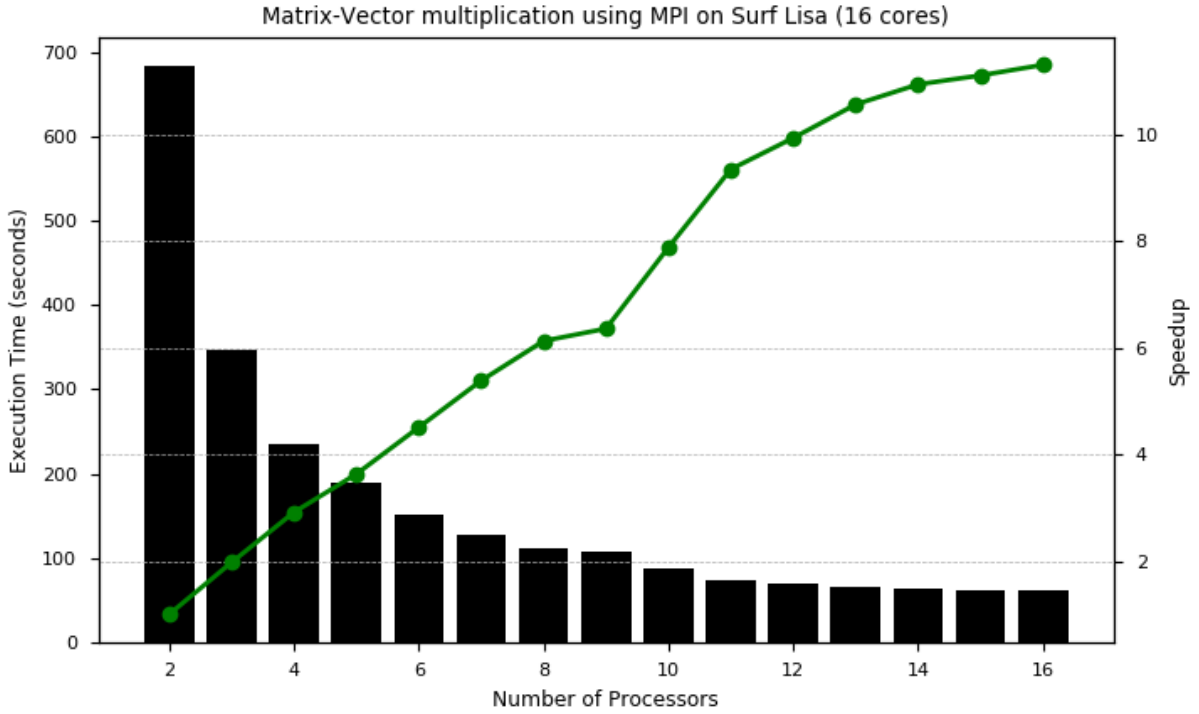

Figure 2. Matrix-Vector multiplication using MPI on Surf Lisa (16 cores) incrementing the number of processors available for MPI.

Figure 3 shows each execution time in seconds with respect to the size of the Matrix and Vector with a fixed R of 100 using 2, 8 and 16 threads. This figure depicts the rapidly increasing execution time of the computation when we increase the problem size, regardless of the number of processors. This is no surprise since we already knew the complexity of the algorithm. This proves us that the parallelization will only make our solution viable up until a certain size of matrix.

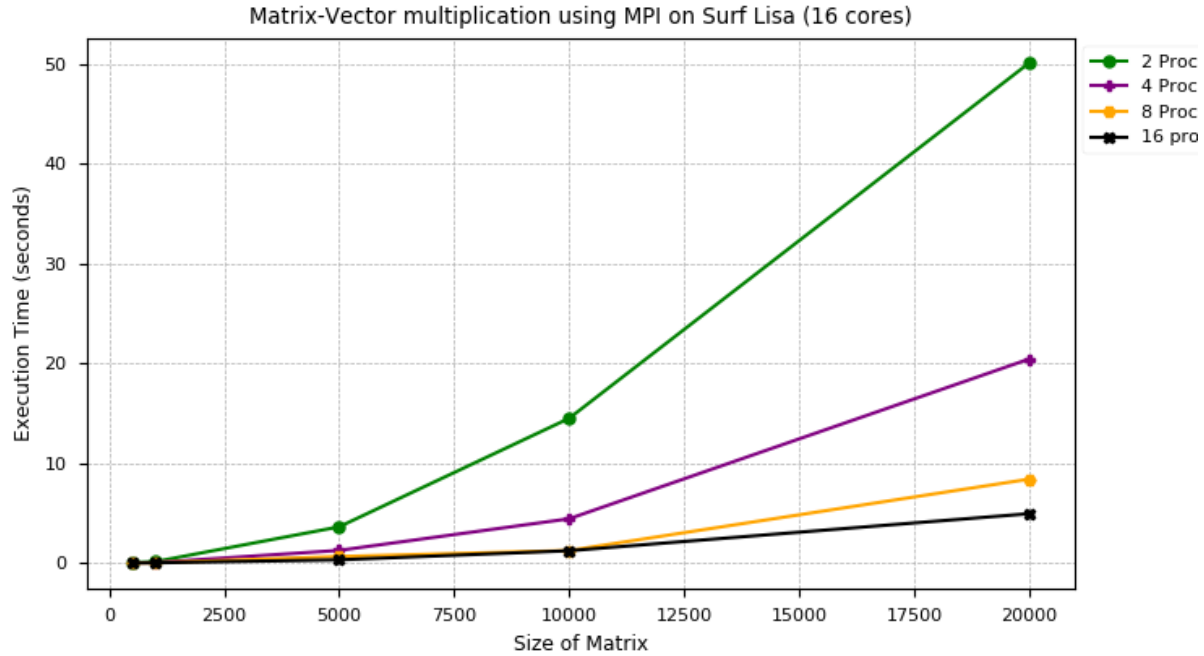

Figure 3. Matrix-Vector multiplication using MPI on Surf Lisa (16 cores) incrementing the problem size.

Figure 4 shows each execution time in seconds with respect to the number of iterations with a fixed N of 20K using 2, 4, 8 and 16 threads. In here we can see that the number of iterations scale with the execution time linearly regardless of the number of threads. This makes sense since adding iterations is just adding repetitions to the same computation (R * $O(n^2)$).

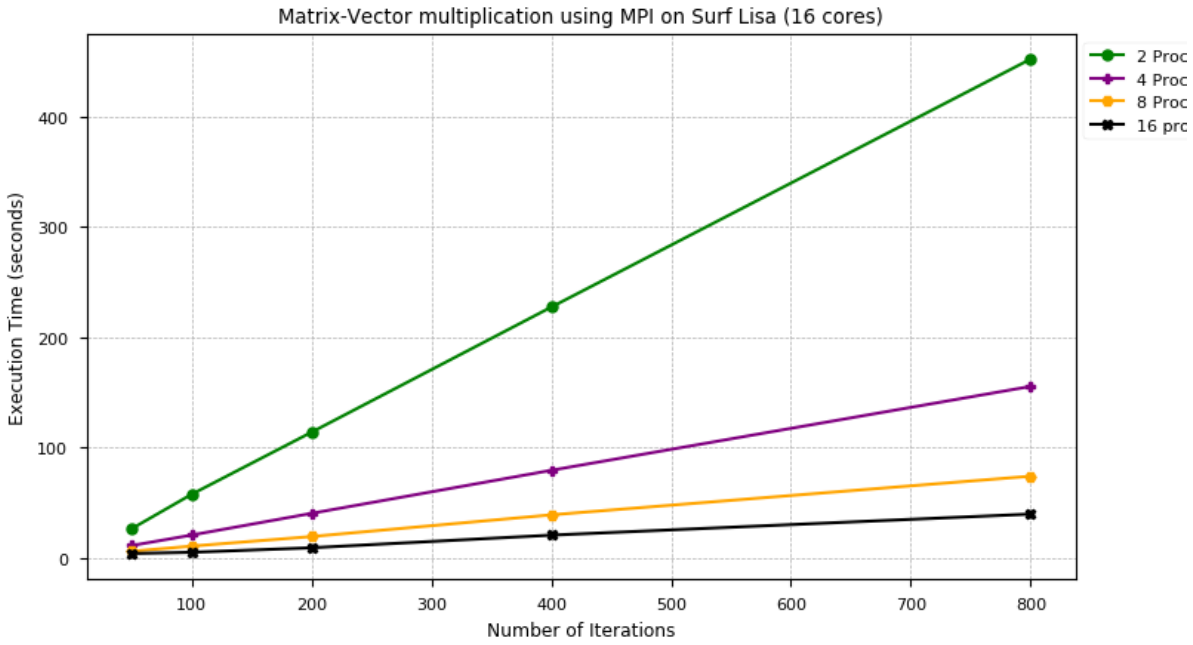

Figure 4. Matrix-Vector multiplication using MPI on Surf Lisa (16 cores) incrementing the number of iterations available for MPI.

Finally, using MPI we were able to scale up the computation to exactly **95K rows** in 59.57 seconds using 16 processors and no iterations.

2.2. OpenMP

For this experiment, we added a pragma to a C code in the reduction of the summary of each operation by rows. Hence, the code is parallelized in N threads dictated by the `OMP_NUM_THREADS` variable. For each time we run the code, we used a different number of threads starting from 1 until 16.

2.2.1. Data Distribution: The multiplication is performed in a **Row-Cyclic** fashion. Each thread will be in charge of performing a multiplication of each row. Since the matrix is accessible to all threads, we do not have such a big problem with communication latency as in MPI.

2.2.2. Results and Discussion: Figure 5 shows each execution time in seconds with respect to the number of threads used for computation for a fixed N of 32K and R of 500. Using 1 thread the execution time was 741.73 seconds. On the other hand using 16 threads the execution time was 59.49 seconds. This means that we achieved a speedup of roughly 12.47. We can clearly see that the execution time reduces in a logarithmic fashion until we reach 16 threads. In addition to this, the more number of threads the smaller the speedup. This could be due to that created thread must perform threads operations such as context switching and scheduling which causes overhead with the more amount of threads.

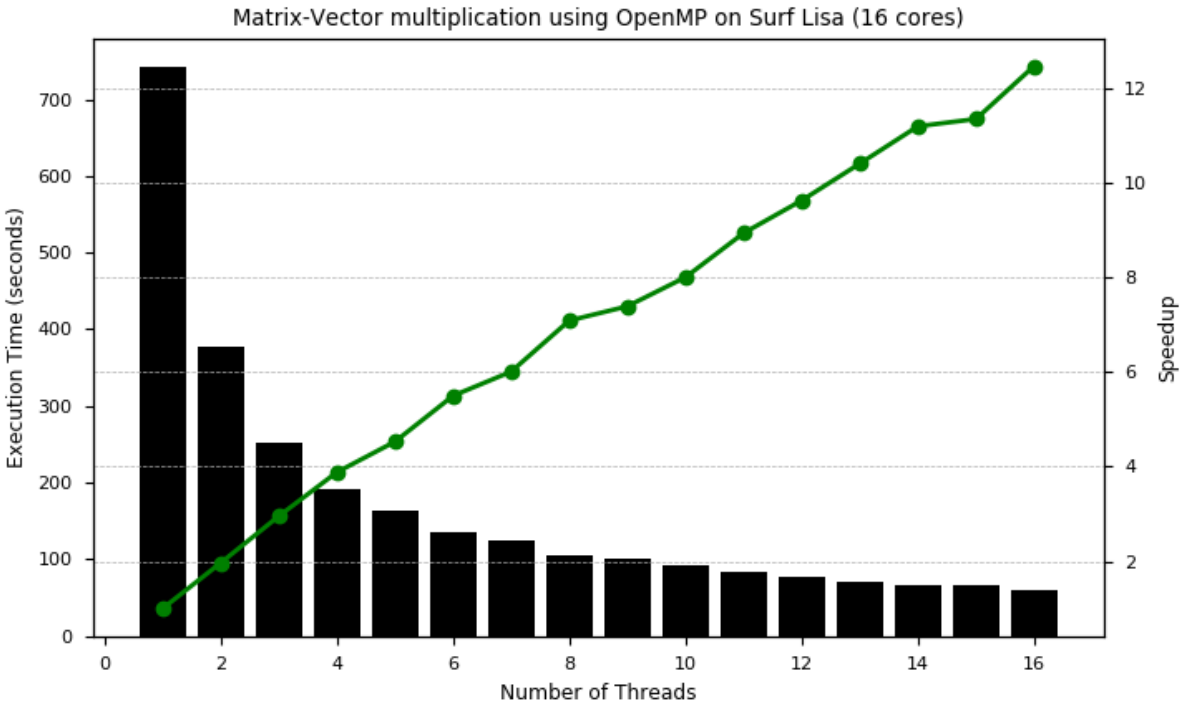

Figure 5. Matrix-Vector multiplication using OpenMP on Surf Lisa (16 cores) incrementing the number of threads available for OpenMP.

Figure 6 shows each execution time in seconds with respect to the size of the Matrix and Vector with a fixed R of 1000 using 1, 2, 4, 8 and 16 threads. From 15K elements to 25K elements, in each case (2, 4, 8 and 16 threads) the execution time was always roughly 3 times slower. The latter shows us how despite parallelization, the execution time will increase equally with the size of the problem *regardless* of the number of threads we are using.

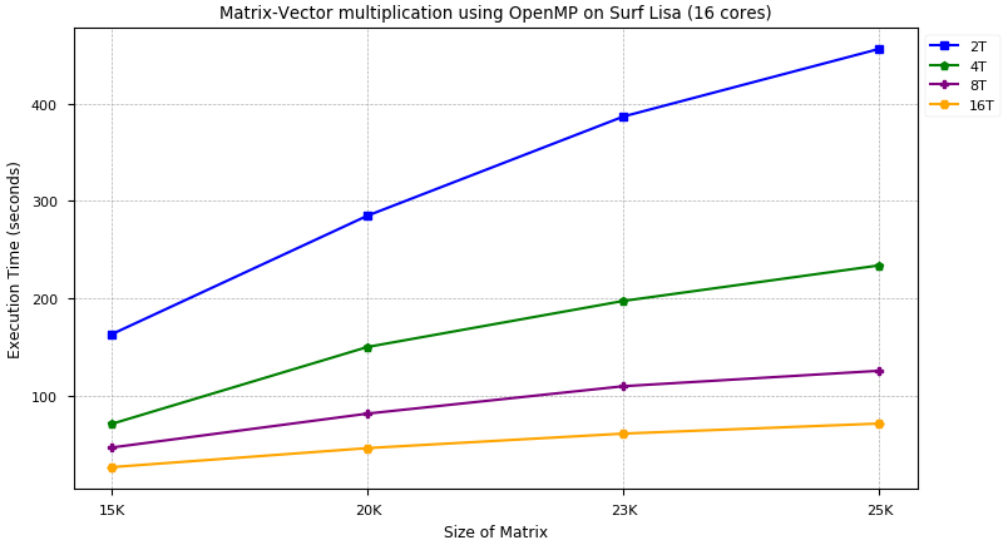

Figure 6. Matrix-Vector multiplication using OpenMP with a different number of threads threads on Surf Lisa (16 cores) incrementing the size of the Matrix.

3. Conclusion

We have parallelized an algorithm using OpenMP and MPI programming interfaces. In doing so, we have proved that data distribution layouts can make a great difference in the efficiency of parallelization. Up to the point that it can make a parallel program even worse than a sequential version. Furthermore, we have proven that more speedups are harder to achieve with a higher number of cores due to communication overhead and data distribution mechanisms. In addition to this, we proved that iterating over this same algorithm scales linearly the complexity of the algorithm regardless of the parallelization.

4. Implementation

The implementation code can be found attached to the submission. In addition to this, we link on this report the implementations in our Github repository:

- MPI
- OpenMP

Both codes can be compiled by executing the following command: `make mat_vec_mul`
N and R parameters are changed inside the mat_vec_mul.c file.