Simulation of Storms of High Energy Particles in MPI

Bruno Carmo, *57418*, Sahil Kumar, *57449*

Abstract—In this report, we investigated the behavior of a parallel program when optimizing a simulation problem. This problem revolves around the simulation of high-energy particle bombardment on an exposed surface such as the surface of a space vessel in outer space. We approached the problem by using a methodology consisting of identifying potential bottlenecks and hotspots and resolving them using different parallel program patterns and the Message Passing Interface (MPI) standard. After obtaining the optimized parallel program we benchmarked it against the given sequential program and analyzed the results of our experiments. We concluded that in a real-world environment, the optimized parallel version would bring clear benefits in the usage of the simulation program.

Index Terms—High-performance computing, Message Passing Interface, parallel programs.

1 Introduction

Parallel computing is often referred to as an execution model that leverages parallel computers to execute multiple operations or execution flows simultaneously. High-performance computing consists of using this kind of process used on highly parallel (and computing powerful) hardware to process massive multi-dimensional datasets and solve complex problems at high speeds.

In this report, we present our solution for the project assignment of the course of High-Performance Computing. This project consists of writing a MPI program that implements a simulation of high-energy particle bombardment on an exposed surface, such as the surface of a space vessel in outer space. This program is based on a provided sequential code that simulates the effects of high-energy particle impacts on a cross-section of the surface, represented by an array of control points. The program computes the energy accumulated on each point after the impact of several waves of particles and, after each wave, it identifies what control point received the highest energy, that presents the higher risk of being damaged.

Our objective in this project is to is to parallelize the simulation using MPI, in order to optimize its performance. We also analyze the results and compare them with the sequential version to make sure of its accuracy.

2 Methodology

In order to understand and analyze the performance of our program we used the *Gprof* profiling Tool. These kinds of tools are usually used to identify most of the code's major performance bottlenecks, hotspots, and where most of the resources are being used to run the program to understand where and how time is being spent.

Using this tool we decided to implement a cumulative methodology to optimize our code, this is, progressively optimizing the portions of the code that used the most CPU time. Using this we discovered that the *update()* function, which is called to update the energy value of a single point in the impact layer, is where the program spends most of its time with 82.30%. Afterward, the next functions that used the most CPU time according to *Gprof*, were the layer update with 13.63% and the initialization of the layer with 2.18%. The remaining functions do not spend much time in the CPU and are displayed in descending order: Copying the layer, locating the maximum, and updating the layer using the ancillary copy. Further analysis of the program's code helped us understand which parts of the code were good candidates for parallel acceleration using MPI.

2.1 Code Optimization

Our implementation of the MPI code involves dividing the layer array between all processes and having every process compute their portion of the layer and then gathering the individual results.

One major problem we needed to address was that the layer size may not be dividable by the number of processes, so we decided that the first process (rank 0) had to compute the remaining control points. To achieve this we used the *MPI_Scatterv* [2] function to scatter the layer in sub-layers using specific layer displacements and sub-layer sizes for each process.

2.1.1 Initialize layer to zero

Instead of the main process initializing the layer to 0 then scattering the layer, each process initializes their own sub-layer to 0, dividing the workload.

2.1.2 Update the energy value for each cell in the layer

Each process computes the energy update of the control points contained, upon each storm, in their respective sub-layer.

2.1.3 Copy values to the ancillary array

Using the implementation of *MPI_Scatterv* we are able to send to each process an overlapped sub-layer, this is, sub-layers containing a neighbourhood 1 of their respective adjacent sub-layers. Using this we skip the initialization and

CPU	Cores	Memory	Clock
Intel Core i5-8265U	4	8 GB DDR4	3.90 GHz (Turbo)
Intel Xeon E5-2609 v4	32	32 GB DDR4	1.70 GHz

TABLE 1
Hardware specification of computational environments used

the copying of the layer, saving memory and computational resources.

2.1.4 Update layer using the ancillary values

In the relaxation process, each control point is updated with the average value of three points: the previous one, the point itself, and the next one.

To implement this we first use *MPI_Gatterv* [2] function to have a complete layer (having process 0 as the root) then we call *MPI_Scatterv* to divide this array in the previously mentioned overlapped sub-layers as each process accesses 3 positions of the sub-layer.

2.1.5 Locate the maximum value in the layer, and its position

Lastly, we identified that we could use the overlapped sub-layers again for computing the local maximum in the layer, as it also accesses 3 positions of the sub-layer. Again we start by the *MPI_Gatterv* function to have a complete layer (having process 0 as the root) then we call *MPI_Scatterv* to send each process an overlapped sub-layer.

Each process proceeds to compute the local maximum and its position (in the original layer) in their respective overlapped sub-layer, then we call the *MPI_Reduce* [2] function using *MPI_MAXLOC* [2] to get the global maximum in all the processes,

3 Experimental Analysis

After parallelizing the simulation using MPI, we needed to check if our solution did indeed optimize its performance and maintain the correctness of the program. For that, we compared the performance of our solution against the performance of the most optimal sequential version (compiled with the *-O3* flag of *gcc*) of the simulation. In order to compare performances, we averaged the runtimes of both versions over 10 runs of the programs in the DI-Cluster. We checked the correctness of our in two different computation environments.

3.1 Computational Environments

We designed our solution and checked its correctness using the hardware in our local computational environment, our development machines (Intel Core i5). Afterward, as suggested, we ran our solution using two nodes (*bulbasaur-2* and *bulbasaur-3*) of the *bulbasaur* cluster inside the DI-Cluster (Intel Xeon). To compare the two computational environments, we present their CPU specifications in table 1 after consulting [1], [5] and [6].

Observing the table, we can notice that the *bulbasaur* cluster has better resources than our local environment. Nonetheless, when comparing the runtimes of the MPI version in our local environment against the *bulbasaur* cluster, we noticed that our solution run faster in our local environment. This occurs probably because of the difference in clock speed of both environments. We clarify that our local environment runs our solution using turbo frequency because of the Intel® Turbo Boost technology [7].

3.2 Benchmarks

Considering that the format of the wave test file was given we could make our test files. Unfortunately, because of a lack of time, we benchmarked our solution against the sequential program using only the test files given.

We executed our solution and verified its performance with different combinations of wave files, however for the sake of simplicity and understandability, we used the given test cases that are described as follows:

- **Test Case 1**: a general debugging test used to check the correctness of the solution. The test has 4 waves and each wave has different numbers of particles ranging from 5 particles to 8 particles.
- **Test Case 2**: a small workload test with 6 waves where each wave has $20\,000$ particles.
- **Test Case 3 to 6**: race condition tests. The tests check for proper communication in the borders of partitions and are executed one file at a time. Each test has only 1 wave with 4 particles.
- **Test Case 7**: an optimization test to check the performance of the solution. The test has 4 waves and each wave has $5\,000$ particles.
- **Test Case 8**: a reduction efficiency test to check the effectiveness of the solution. The test has 3 waves and each wave has only 1 particle, but it should be executed with 100 million control points.
- **Test Case 9**: a behavioral test used to check that the behavior of the sequential program in the array extremes has not changed even after the optimizations done in our solution. The test has only 1 wave with 3 particles and was executed with 16 control points.
- **Test Case 10**: the same test as test case 9 but with 17 control points.

Our solution passes all the tests except test case 9. For that test case, our solutions just return values that do not equal those of the sequential version meaning that our solution does not preserve the behavior of the sequential program in the array extremes.

In the following subsections, we present the results of three different experiments: the performance gain of successive cumulative optimizations, the speedup of the final MPI version against the sequential version, and the comparison of the speedup gained using CUDA against MPI.

3.2.1 Performance Gain of Cumulative Optimizations

Considering the results returned by *Gprof*, one interesting experiment we thought of, was the comparison of the speedups gained using our cumulative optimization approach. Taking into account the experience from the previous project, we knew that any parallelization approach (CUDA or MPI) would gain a higher speedup when feeding heavy workloads to the program. Therefore, the results of

Optimization	Average Runtime	Step Speedup	Cumulative Speedup
Sequential	150.157964	0	equal
Parallelized update function	13.073421	11.485743	11.485743
Parallelized copy function	13.065460	1.000609	11.492742
Parallelized max function	13.122950	0.995619	11.442393
Parallelized update with ancillary function	13.069945	1.004055	11.488798

TABLE 2
Performance gain of cumulative optimizations

Testcase	Sequential	MPI	Speedup	Result
1	0.000015	0.056916	0.000243	equal
2	27.041956	2.447438	11.186854	equal
3	0.000005	0.061205	0.000084	equal
4	0.000005	0.058159	0.000087	equal
5	0.000005	0.058675	0.000085	equal
6	0.000004	0.061939	0.000068	equal
7	150.157964	13.069945	11.488798	equal
8	3.481492	3.613154	0.982885	equal
9	0.000004	0.059561	0.000068	not equal
10	0.000003	0.059345	0.000045	equal

TABLE 3
Comparison of runtimes of sequential version and MPI version in the cluster

this experiment, as shown in table 2, are obtained by using test case 7.

As expected, the most speedup gain occurs when we optimize the update function to execute in parallel. For the remaining functions, there is not a large speedup gain when optimizing them, as we can confirm through their step speedup. This goes line with the profiling report returned by *Gprof* where the sequential program spent most of its runtime in the *update* function and less in the other functions.

Observing the table, we notice that the step speedup goes below 1 when optimizing the max function. This most likely occurrs because of our implementation, however, it could also be because of the overhead (communication and synchronization) that happens in a message passing approach.

3.2.2 Speedup of MPI version against sequential version

Taking a real-world program optimization approach, we compared the runtime of the sequential version with the runtime of the MPI version, to see if the MPI version was worth it. We executed both versions in the DI-Cluster, as demonstrated in table 3.

Looking at the table, as expected, we can perceive that in the test cases where the objective is to feed a big workload to the program to check its optimization performance, like test cases 2 and 7, the MPI version has a faster runtime than the sequential version which in consequence gains a bigger speedup. However, for small workloads, the runtime of the MPI version is penalized by the overhead that occurs with message passing across processor cores. This same behaviour happened in the previous project regarding the optimization with NVIDIA's CUDA API. Therefore, taking into account this behaviour, parallel optimizations should only be done when there is a big workload. If not, the cost of parallel optimization (communication, synchronization, etc) is bigger than the its benefit.

Testcase	CUDA	MPI	Result
1	0.03	0.000243	equal
2	31.14	11.186854	equal
3	0.01	0.000084	equal
4	0.01	0.000087	equal
5	0.01	0.000085	equal
6	0.01	0.000068	equal
7	52.54	11.440100	equal
8	1.38	0.982885	equal
10	0.01	0.000045	equal

TABLE 4
Comparison of speedups of CUDA version and MPI version in the cluster

Considering the case of big workloads and the correctness of our solution we can say that our solution is indeed worth it for real-world program optimization.

3.2.3 CUDA vs MPI

Another interesting experiment to do was to compare the speedup gained when using an approach based on NVIDIA's CUDA API against an approach based on MPI. The results are shown in table 4.

Observing the table, as anticipated, the speedup gained when using CUDA is always higher than the speedup gained when using MPU. This occurs mostly because of the overhead that happens in an approach based on message passing but also because of the specialization of the GPU for this kind of tasks and the huge difference in cores between CPUs and GPUs.

Regardless of the optimization approach, by observing the table, we note that for small workloads, a sequential approach is always better, taking into account, that the speedup for those workloads is always below 1. This means that the sequential version has always a faster runtime than the CUDA and MPI version for this types of workloads.

4 Conclusion

This report presented the methodology we used to complete the assignment given and the experimental analysis we made to verify our solution's performance, correctness, and behavior.

We managed to optimize all bottlenecks identified maintaining the correctness of the program for most cases. Our solution also passed all our test cases in the local environment and only failed test case 9 in the DI-Cluster.

We concluded based on our experimental analysis that if this problem assignment was taken to a real-world application then the parallel optimization of the sequential program using MPI would surely bring benefit to the users of the program taking into account that it is faster and still maintains correctness. However, a solution based on GPUs, if available, (such as the previous project) would be more efficient because it does not have the overhead of messages passing between machines or cores.

REFERENCES

[1] DI-Cluster, http://cluster.di.fct.unl.pt/docs/technical/
[2] MPI Forum, https://www.mpi-forum.org
[3] MPI Tutorial, https://mpitutorial.com
[4] MPI Documentation, RookieHPC, https://rookiehpc.org/mpi/docs/index.html
[5] Intel Core i58265U, Intel, https://ark.intel.com/content/www/us/en/ark/products/149088/intel-core-i5-8265u-processor-6m-cache-up-to-3-90-ghz.html
[6] Intel Core E52609v4, Intel, https://ark.intel.com/content/www/us/en/ark/products/92990/intel-xeon-processor-e5-2609-v4-20m-cache-1-70-ghz.html
[7] Intel Turbo Boost Technology, Intel, https://www.intel.com/content/www/us/en/support/articles/000007359/processors/intel-core-processors.html

ACKNOWLEDGMENTS

We want to thank Hugo Pereira (58010) for helping us launch the solution in the DI-Cluster and for telling us some specific characteristics and behaviour of the DI-Cluster.

INDIVIDUAL CONTRIBUTIONS

We distributed the work using the same method as the previous project. As such, we divided the work taking into account the number of code blocks we had to optimize. Considering that there were 5 code blocks to optimize, Bruno Carmo optimized 3 code blocks and Sahil Kumar optimized 2 code blocks. Sahil Kumar additionally created the script that checks the correctness of the solution and the speedup gain to balance the workload distributed. Therefore each member contributed with 50%.