Parallelizing the Simulation of Storms of High Energy Particles using OpenMP

Francisco Henriques, *Student, FCT-UNL,* Gabriel Gonçalves, *Student, FCT-UNL,*

Abstract—This paper explores different ways to parallelize a given sequential code that simulates the effects of high-energy particles bombardment on an exposed surface resorting to the OpenMP API for the C programming language. This study was developed within the scope of the course Concurrency and Parallelism at NOVA School of Science and Technology.

Index Terms—Parallel Architectures, Programming Techniques, Parallel Systems, LaTeX.

1 Introduction

With this paper we intend to analyse the performance impact of parallelization on a sequential code.

We will start from a basis of a sequential version of the code that simulates the effects of high-energy particles bombardment on an exposed surface.
From there we will study the multiple concurrency problems that there might exist on the serial version, remove those dependencies and analyse the efficiency of the given parallelization.
The source code was written in the C Language, with the purpose of evaluating the improvements of the performance using the OpenMP API.

The source code given consists in calculating the energy value for each cell within a layer and returning the maximum energy of the layer after each storm. This is done by sequentially iterating through every storm's high energy particle, retrieving its energy and position, using it to update the energy value for each cell in the layer. In the case of executing on multiple sequential storms, the energy relaxation between storms is also calculated.

2 OpenMP API and its uses

2.1 OpenMP

The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer. [1]

OpenMP is easy to use and consists of only two basic constructs: pragmas and runtime routines. The OpenMP pragmas typically direct the compiler to parallelize sections of code. All OpenMP pragmas begin with #pragma omp. As with any pragma, these directives are ignored by compilers that do not support the feature—in this case, OpenMP.
You add parallelism to an app with OpenMP by simply adding pragmas and, if needed, using a set of OpenMP function calls from the OpenMP runtime. These pragmas use the following form: #pragma omp ¡directive¿ [clause[[,] clause]...] The directives include the following: parallel, for, parallel for, section, sections, single, master, critical, flush, ordered, and atomic. These directives specify either work-sharing or synchronization constructs. [2]

3 Sequential Version

The sequential version of the code consists in the sequential iteration of each storm file and calculation of the maximum energy value and position on the layer for each storm.
The can be divided into 3 main computations:

1) The impact energy to each layer's cell
2) The cooldown of each layer's cell
3) The location of the maximum value in the layer and its position

3.1 Impact energy calculation

This part consists in iterating sequentially for each particle and computing the attenuated energy that it produces for each layer cell.

3.2 Cooldown of the layer

This part simulates the cooldown of the layer after each storm, by updating every cell's energy to the average of its own energy and the 2 adjacent cells.

3.3 Location of the maximum value and its position

This part consists in iterating through the layer cells and finding the one with the maximum amount of energy

4 Parallel Version

In this section it will be presented the final approach done on the improvements of the performance parallelizing the code.

To parallelize the aforementioned sequential version it was required to remove the concurrency problems so that each iteration of the loop would be independent from the other ones. Each piece of the sequential code presented a challenge for the parallelization.

4.1 Impact energy calculation and the function update

The main issue in this part of the code is that when parallelizing the cycle that iterates through each particle, different threads could be trying to update the value of the same position in the layer at the same time through the update function. This operation would cause an Read after Write, Write after Read and a Write after Write dependencies.
In order to remove those dependencies used a reduction on the layer array which creates a local copy of it in each thread removing the race conditions and after each thread has finished executing it reduces each thread's local copy into a single array by calculating the sum of the different arrays.

4.2 Cooldown of the layer and Location of the maximum

As both of the calculations presented in the previous section iterated over the layer cells, it was not necessary to execute the same loop twice, therefore it was combined into a single iteration through the cell array.

In order to parallelize this new formed loop, each thread computes the result of a portion of the layer array. It starts by calculating the first cell's cooldown value and then it loops through all the cells computing the next value and comparing the current node to its neighbours to find out if it is a local maximum and only then checking if it's larger than the thread's maximum finding the global maximum. We kept this preliminary check even though we didn't think it would improve performance because it was included in the original code and we wanted our results to be better through parallelization alone and not by code optimization. The global maximum is then found by choosing the maximum value returned by all threads. Note that in this loop we didn't use the higher level constructs provided by OpenMP to have better control of which thread's allocated work and better understanding of exactly where the frontiers lie.

One of the difficulties lies in these frontier values for each thread because the last value attributed to each thread depends on the first value of the next thread, this value is only computed after all the other threads have computed the non-frontier value, guaranteeing the correctness of the code. This assurance is provided by the use of the barrier keyword in OpenMP as that assures us that all threads have finished the work above it.

5 Comparison

In this section our different approaches to parallelizing the original program by minimizing runtime will be explored and each approach's performance tested in a node within a cluster. This node is composed of 4 AMD Opteron 6272 processors, having 32 cores and 64 threads and 64 GiB of DDR3 1600MHz RAM. [3]
All the future graphs shown were created by averaging the runtime results from 5 runs of the test number 7 from the test set given with 10^6 layer size and 64 threads.

5.1 Initial thesis

At first glance we assumed it would be better to parellelize the outer cycle of the Impact energy calculation explained in 4.1, also parallelizing the copy of the array in the section 4.2.1 and to parallelize both of the 4.2 cycles. This will be our base version for comparison (basis).

5.2 Impact energy calculation and the function update

As expected, parallelizing the outer loop is better than parallelizing the inner loop, as when only parallelizing the outer one, in each storm, each thread is assigned a set of particles to process and it does so until the end, while on the other hand, parallelizing the inner loop means that for every particle within a storm we will be dividing the layer in different parts for each thread to process. This is slower as it causes us to keep dividing small workloads between threads many times instead of initially dividing a bigger workload and letting the threads process things in parallel for longer.

We also analysed the possibility of parallelizing both cycles, however the results were worse than only parallelizing the outer cycle. This happens because of the same reason explained above, as the threads used for parallelizing the inner cycle would be better employed on the outer cycle.

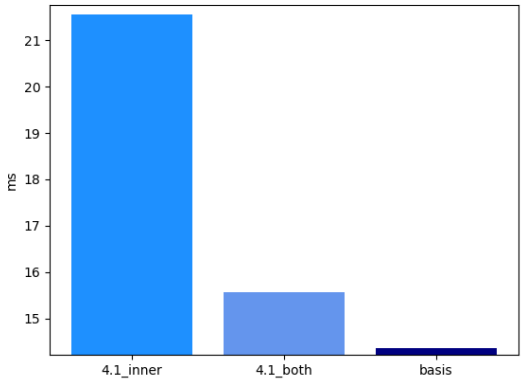

Fig. 1. Plot of the average runtime for inner loop, both loops and outer loop parallelization

As previously mentioned in the section 4.1, there was a race condition when updating the energy value of each cell position. Initially it was solved by making the increment of the cell value an atomic operation, however this was not

very efficient as it blocked the other threads for a small amount of time, therefore not being the best implementation. We then tried applying the reduction method previously explained and got a large performance improvement. However this implementation uses more memory as each thread has to maintain a local copy of the array.

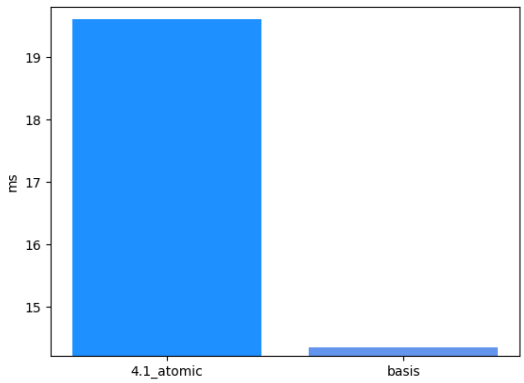

Fig. 2. Plot of the average runtime for atomic block and reduction method

5.3 Copy of the layer array

As our initial thesis claims, we thought that parallelizing the copy of the array should be better than doing it sequential as for every test we had a great amount of particles in the storm (over 10^6). However this did not come out to be true in our practical tests.

The time it takes to launch each thread to compute a segment of the array is greater than the time it takes to just copy that segment and therefore our initial thesis was wrong and we changed our final version to the one obtaining the best results.

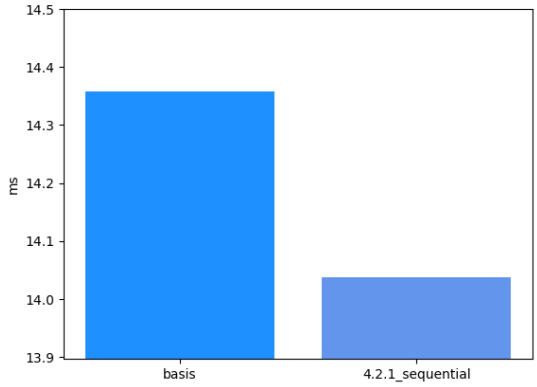

Fig. 3. Plot of the average runtime for parallelized and sequential copy of the array

5.4 Cooldown of the layer and Location of the maximum

As we stated in our thesis, the parallelization of the cooldown and location of the maximum of the layer is better because unlike in section 4.2.1, where the only operation was the copy of the array, in this section its worth to sacrifice the time it takes to launch the threads, as each thread computes the cooldown, updates the value of the cell and compares it to its neighbours, being a much more complex and time consuming task.

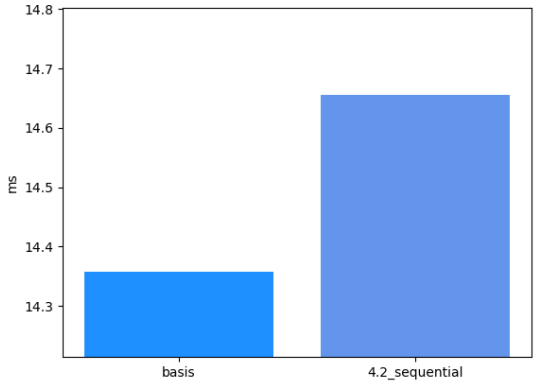

Fig. 4. Plot of the average runtime for parallelized and sequential calcule of the cooldown and location of the maximum

5.5 Final Implementation and the improvement of performance

As our thesis of what the final implementation would be had flaws, we developed a new version where we applied the newly gained knowledge and developed the final version where we did not parallelize the copy of the array (final).

This final version had a colossal performance improvement relative to the sequential code.

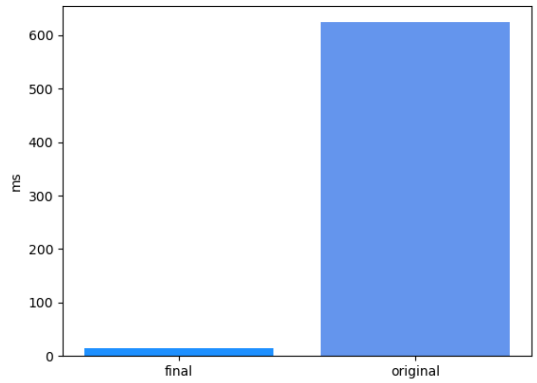

Fig. 5. Plot of the average runtime for parallelized and sequential code on 64 threads

We estimate an average total speedup for 64 threads of 44,46 which we considered to be very satisfactory and within theoretical limits.

5.6 Number of threads

After developing our final implementation, we studied what is the perfect amount of threads to launch when running our code. Having 64 available threads on the machine, we wondered what amount of threads our program should use to maximize performance. We initially thought that the right value should be 64 or slightly above that, since our code seems to use cpu resources pretty continuously, not waiting for I/O at any point.

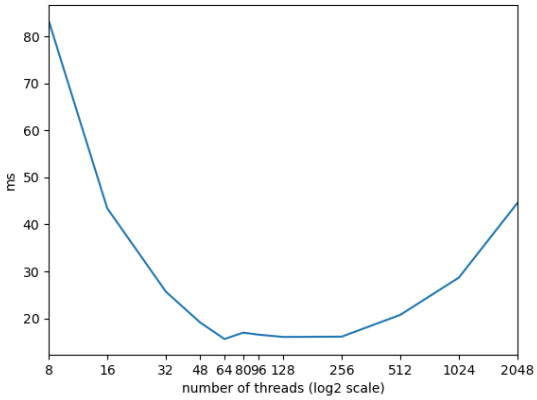

Fig. 6. Plot of the average runtime for an amount of threads launched

Obviously launching less than 64 threads leads to a higher runtime as it is not possible to evenly distribute the load over all the 64 logical cores. However our doubts were on the upper neighbours of 64, as there was a possibility our threads could be finishing at different times and maybe having small times of not fully using the cpu so having more threads with smaller workloads could possibly help balance the load and fill those gaps. After running the tests we noticed a small but clear improvement on average when launching 64 threads (same amount as logical cores) so we came to the conclusion that launching more processes than the amount of threads available is not worth it for this application.

6 Difficulties

6.1 Impact energy calculation and the function update

The first difficulty found was that the parallel implementation of the update function would return a maximum energy with a slight deviation from the sequential value (< 0.002).

At first it was thought that it was simply due to a bad parallel implementation, however after some researching, the real issue was revealed. The sum of floats that we thought was commutative, in fact was not. As the deviation from the sequential version was insignificant and it greatly improved the performance, it was decided that it was a better implementation than guaranteeing the exact same result since that result didn't even represent a more correct value, simply a different approximation due to the order of the addition of the floats.

6.2 Testing

Our other main difficulty was to test the code faithfully as our computers only have 4 threads and the with the small amount of access time to the aforementioned cluster we couldn't perform all the tests we first ambitioned.

7 Conclusion

In conclusion, small modifications in the implementation can greatly influence the performance or cause serious problems in the program. Being able to analyse the data, using profilers and removing dependencies that may alter the flow and output of a program in order to parallelize the program is crucial in the work of a computer engineer.

In this particular case, even though the sequential code was relatively simple, the parallelization had its difficulties when it came to removing some of the dependencies.

Having an API like OpenMP also facilitates the work a lot as we are not required to implement some of the tools that it provides that allow developers to focus on what is needed to optimize and not on the intricacies of launching and managing threads and thread pools.

References

[1] Web.archive.org. 2021. OpenMP.org - About the OpenMP ARB and OpenMP.org. [online] Available at: https://web.archive.org/web/20130809153922/http://openmp.org/wp/about-openmp/ [Accessed 31 May 2021]

[2] Su Gatlin, K. and Isensee, P., 2019. OpenMP and C++: Reap the Benefits of Multithreading without All the Work. [online] Docs.microsoft.com. Available at: https://docs.microsoft.com/en-us/archive/msdn-magazine/2005/october/openmp-and-c-reap-the-benefits-of-multithreading-without-all-the-work [Accessed 6 June 2021].

[3] DI-Cluster Wiki. n.d. Technical Description. [online] Available at: https://cluster.di.fct.unl.pt/docs/technical/ [Accessed 6 June 2021].

Acknowledgments

The authors wish to thank the teachers of the Concurrency and Parallelism Course and all the colleagues that answered and made questions in the piazza as it was a great help in the development of this report.

Comments

It could be interesting to allow the students a little bit more time in the cluster. Even though we understand this is a double edged knife because giving more time might mean that some students block the cluster for a long period of time.

Our suggestion is instead of a 2 hour overall maximum time, maybe some weekly or daily quota could be implemented so that students that start early are benefited.

INDIVIDUAL CONTRIBUTIONS

In order that both of us would have hands on every part of the work we divided the development as so:

1) Francisco Henriques: 40% code and testing, 60% report
2) Gabriel Gonçalves: 60% code and testing, 40% report

Overall we think this ratio was sustained and both students' workload was similar (50% 50%) We also used git extensively, using different branches for different parts of the assignment. One branch was also created for sharing code and run results from the cluster as we found this was the easier way to keep our versions synchronized between machines.

Francisco Henriques Student at NOVA School of Science and Technology, Computer Science Master's degree

Gabriel Gonçalves Student at NOVA School of Science and Technology, Computer Science Master's degree