Computação Paralela

Trabalho Prático 3

Luís Araújo
PG54004
Universidade do Minho
Mestrado em Engenharia Informática

Mateus Pereira
PG54089
Universidade do Minho
Mestrado em Engenharia Informática

***Abstract*—This document explores optimizing a molecular dynamics simulation program using parallel computing techniques on the SeARCH cluster. In this last phase of the project we implemented various MPI while keeping scalability in mind by making several performance tests, using the number of samples, the number of clusters and the number of processes as metrics and interpreting these results.**

I. Introduction

This report, as part of the *Computação Paralela* course's third practical work, focuses on optimizing a molecular dynamics simulation program by transitioning from the parallel MDpar.cpp made (and modified by us now) in the second assignment to the parallel versions MDmpi.cpp, which uses various MPI approachs to parallelize the program, and the MDparmpi.cpp, which uses a combined approach with the previous OpenMP. For each of our strategies we'll discuss their implementation, the problems we had while implementing them and the conclusions we took from this experience.

II. Simplifying and Optimizing Key Functions and Data Structures

In the first assignment, we focused on optimizing key functions and data structures of a molecular dynamics simulation program. Our primary goal was to reduce computational overhead by simplifying the *Potential()* and *computeAccelerations()* functions. Through profiling via perf and gprof (and gprof2dot), we identified these as the most expensive functions in terms of computational resources. We achieved significant performance gains by replacing the *pow()* function with a series of multiplications and streamlining the loop structures. Additionally, we optimized the Makefile and implemented vectorization, which further accelerated the execution time from around 70 seconds to approximately 6-7 seconds.

III. Implementation with OpenMP

The transition to the parallel version of the molecular dynamics program, MDpar.cpp, marked a significant leap from the sequential implementation in MDseq.cpp. This evolution was a primary focus of our second assignment. The implementation with OpenMP involved identifying computational hotspots through performance profiling and subsequently applying parallel computing techniques.

We again utilized the *perf* tool for profiling, which revealed that the *computeAccelerationsAndPot* function was the major computational bottleneck, accounting for 98.87% of the execution time. Our approach was to parallelize this function using OpenMP directives, significantly improving the program's performance. The successful parallelization of the *computeAccelerations* function, despite its smaller overhead, also contributed to the overall performance enhancement.

One of the notable challenges was ensuring numerical precision during the parallel execution. Slight discrepancies were observed in the 12th digit of the *cp_output* file, raising concerns about the accuracy of the parallel implementation. Rigorous testing and fine-tuning of the parallelization strategy were employed to address these issues.

IV. Advanced Parallelization with MPI

With the foundation laid in the previous assignments, our third assignment embarked on a more advanced parallelization using MPI. The primary goal was to further enhance the efficiency of the molecular dynamics simulation by implementing MPI-based parallelism.

A. Reduce approach

We experimented with different MPI methods, starting with the Reduce approach. This method proved effective in consolidating data from multiple processes to a single process, which is critical in simulations where global properties need to be calculated from local data. The implementation of *MPI_Reduce* significantly optimized the data aggregation steps in our program.

In hindsight, *MPI_Reduce*'s implementation wasn't that "complex", although we spent way too many hours going over it. It works similarly to OpenMP's reduce statement - each process does its own work on the loop - and, after computing their share of the Pot and acceleration variables, they sum the result on their respective global variables using *MPI_Reduce*, to save them on the main ranked process and *MPI_AllReduce* to distribute the reduction result back to all processes.

B. Gather approach

The Gather approach, implemented via *MPI_Gather*, was another pivotal strategy in our MPI parallelization efforts. This method allowed us to collect data from all processes and

assemble it in a predefined order on the root process. Similar to the Reduce approach, we also explored *MPI_Allgather* for distributing the gathered data to all processes, but our primary focus remained on the standard Gather implementation for detailed performance analysis.

We again implemented it on the function that consumed the most resources, the before-mentioned *computeAccelerationsAndPot*. We created local acceleration vectors and a local Potential variable so that each process could calculate its own part of the work. For both of them we used Gather to "join" these local variables into single "intermediary" ones. After that we copied it into the respective global variable that's shared between all processes using *MPI_Allgather*, ending the process.

C. Joint Reduce-OpenMP approach

Lastly, after we successfully implemented all the MPI methods we thought it would be interesting to analyse the behaviour and performance of our program if we joined both the MPI and the OpenMP approaches. Binding them together was pretty simple, we just imported the OpenMP library and copied the line in which we used the operators, the same way as implemented in the last phase(WA2). We then compared both results, as explained later in the report.

V. Drawbacks

While MPI and OpenMP offer significant performance improvements, they also come with their own set of drawbacks. Implementing these technologies required a thorough understanding of parallel computing concepts and careful code restructuring. Debugging and testing also became more complex, especially in identifying and resolving data race conditions and ensuring consistent numerical precision across different parallel execution paths.

When testing with MPI we realized there was a slight deviation of the output results from the default ones in a couple lines of code. This is because, just like in the previous assignments, each process computes their part of the overall work on a different time period and, when the results are summed together in the various MPI functions used in our program, the results tend to be different in some places, with the 12th digit shifting in up to 5 lines of code.

A. Pipeline Approach

In our exploration of MPI techniques, we experimented with a pipeline approach using the Recv and Send operators, as initially introduced in our first MPI class. Despite its theoretical promise, this approach consistently yielded incorrect output values. We deduced that the underlying issue was inherent in the *computeAccelerationsAndPot* function's dependency on the values from the previous iteration to commence the subsequent one. This sequential dependency likely led to the observed inaccuracies in output.

Moreover, we encountered potential deadlocks when implementing certain pipeline configurations. Given that each process in our setup required data from other processes to proceed, any delay or misalignment in the pipeline could result in a deadlock situation. Specifically, processes could become indefinitely stalled at the Recv operator, awaiting data from a preceding process that, due to the deadlock, was never dispatched. This realization led us to reconsider and eventually abandon this pipeline strategy in favor of more reliable parallelization methods.

VI. Performance tests

Our performance evaluation focused on various aspects of the parallel implementations. We conducted tests to assess the impact of different numbers of processes and threads on the overall performance. The following subsections will present detailed analyses and accompanying data tables.

As requested, we coded a new Makefile that runs the sbatch commands automatically so that possible automatic tests run by the professors run smoothly.

TABLE I
Execution Time and Speedup Analysis

#Processes/#Threads	MPI (s)	OpenMP (s)	Joint Approach (s)*
1	28.436	22.115	2.107
2	22.928	11.175	2.053
4	13.440	5.665	1.720
8	7.820	2.888	1.548
10	6.496	2.376	1.564
12	5.528	1.982	1.783
16	4.322	1.533	2.114
20	3.657	1.265	2.641
24	3.149	1.225	2.794
28	4.017	1.187	26.665
32	3.917	1.144	31.325
36	3.627	1.118	35.710
40	3.785	1.089	42.850

* In this approach the number of threads is 40, but the number of processes is incrementing so that we can check what is the best number, concerning the number of processes.

The final part of our report details the results obtained from the implementations and tests conducted. We have included attachments in the form of images and supplementary material that illustrate the performance metrics and the impact of our optimization strategies.

TABLE II
Execution Times for MPI and OpenMP with 40 Processes/Threads, respectively

#Particles (N)	MPI (s)	OpenMP (s)
5000	3.785	1.089
2500	1.214	1.087
1000	0.519	1.021

A table with a speed comparison between the newly implemented MPI and the previous OpenMP. MPI is running with 40 processes and OpenMP with 40 threads.

TABLE III
INSTRUCTIONS AND CYCLES ANALYSIS

Processes	Instructions (Billions)	Cycles (Billions)
1	149	89
2	164	134
4	174	162
8	181	172
20	201	215
40	227	356
40 (OpenMP)	96	119

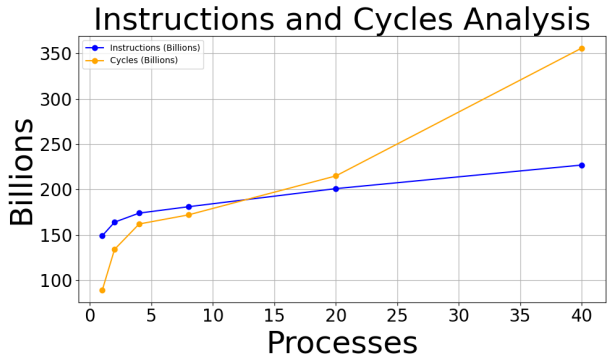

Fig. 1. Graph depicting the relationship between the number of processes and the corresponding billions of instructions and cycles.

As both the table and image show, the memory utilization in our OpenMP and MPI implementations exhibits distinct characteristics, largely influenced by their respective parallelization methodologies. With OpenMP, it allows threads to access and modify shared data concurrently. Such an arrangement typically leads to reduced resource consumption, particularly in terms of cache usage. Consequently, this can diminish cache misses and minimize the frequency of memory accesses, thereby enhancing the overall efficiency of the program.

Conversely, our MPI-based version employs a process-based approach to parallelization. Each process in MPI operates independently with its own dedicated memory space. While this independence can be advantageous for certain computational tasks, it also implies a higher resource demand. Specifically, it results in increased memory and cache usage since each process maintains its own separate data. Unlike the shared memory context of OpenMP, the distributed memory model of MPI can lead to duplicated data across processes. This duplication potentially occupies valuable cache and memory resources with redundant information, which might not be efficiently utilized due to the isolated nature of MPI processes.

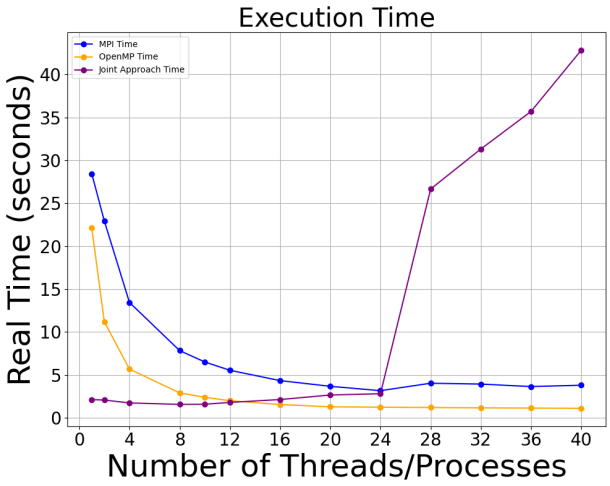

Fig. 2. Graph depicting the execution time in function of the number of threads/processes for each of the different approaches we took.

Finally, in this image we can analyse that the time with MPI will never surpass (be better) than with OpenMP. This is, for example, due to the overhead of Inter-Process Communication (IPC), which can be more time-consuming compared to the thread-level communication in OpenMP. The cost of sending and receiving data between processes in MPI can offset the benefits gained from parallel execution, especially if the communication is frequent or involves large data transfers. Another conclusion we can take from this graph is that while using MPI, the best number of processes we can use is 40 (or 20 depending on the point-of-view, they are very similar). Finally, we can also see that in the distribution of processes by 40 threads, the best optimal number of processes is 8.

VII. CONCLUSION

Our journey through the assignments revealed the intricacies and challenges of parallel computing. The transition from OpenMP to MPI, and eventually to a hybrid model, demonstrated that while parallelization can significantly boost performance, it requires careful consideration of data management, process synchronization, and computational overhead. The experience gained from these assignments provided valuable insights into the practical applications and optimizations in high-performance computing environments.

Throughout our project, we extensively optimized the molecular dynamics simulation program using parallel computing techniques. Initial efforts focused on enhancing key functions, leading to significant performance improvements. Transitioning to parallel processing, we employed OpenMP and later advanced to MPI, successfully integrating these methods to improve simulation efficiency. Comprehensive performance tests revealed insights into memory utilization and execution efficiency, with OpenMP demonstrating efficiency in resource consumption, while MPI faced challenges in scenarios with extensive data transfers. Overall, the project refined our skills in parallel computing and deepened our understanding of its applications in high-performance computing,

providing valuable insights into data management complexities and process synchronization.

The project not only optimized key functions but also delved into advanced parallelization with OpenMP and MPI. While OpenMP showcased efficiency in resource consumption, MPI's process-based parallelism faced challenges, especially with frequent data transfers. The experience enhanced our technical skills and provided valuable insights into managing computational overhead in parallel environments.