Performance Optimization of a Molecular Dynamics Simulation

Catarina Felgueiras
Physics Engineering Student
Minho's University
Braga, Portugal
a100506@alunos.uminho.pt

Luís Silva
Physics Engineering Student
Minho's University
Braga, Portugal
a96534@alunos.uminho.pt

***Abstract*—With this work, we explored all the optimization techniques we knew up to that point with the aim of improving the code's runtime. At each step along our optimization strategy, we conducted a thorough performance evaluation of the developed implementation.**

Keywords—Optimization, performance, code, analysis, time, execution, OpenMP, MPI, single-thread , multithreading

I. Introduction

The program to be analysed and optimised in this project is a code that simulates the molecular dynamics of noble gases, with a focus on the Lennard-Jones potential used to describe interactions between two particles (force and potential energy). It should be noted that natural units are employed, so the values of various physical constants and parameters are set to 1. The code includes functions to calculate the kinetic and potential energy of the system, initialise the positions and velocities of the particles, and compute their accelerations based on intermolecular forces. The code initialises a particle system, updates their positions and velocities using the Velocity Verlet algorithm, and calculates various thermodynamic properties such as temperature and pressure.

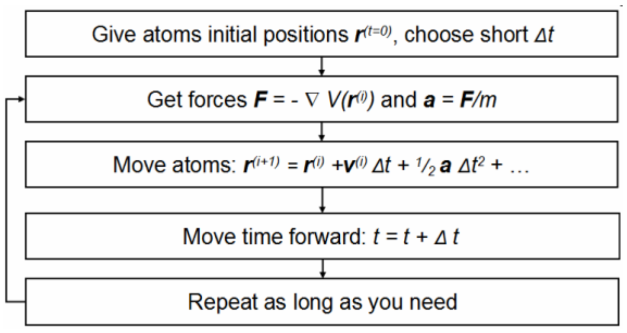
Give atoms initial positions $\boldsymbol{r}^{(t=0)}$, choose short Δt
Get forces $\boldsymbol{F} = -\nabla V(\boldsymbol{r}^{(i)})$ and $\boldsymbol{a} = \boldsymbol{F}/m$
Move atoms: $\boldsymbol{r}^{(i+1)} = \boldsymbol{r}^{(i)} + \boldsymbol{v}^{(i)}\Delta t + \frac{1}{2}\boldsymbol{a}\,\Delta t^2 + \ldots$
Move time forward: $t = t + \Delta t$
Repeat as long as you need

Fig. 1. Scheme with the algorithm implemented in code.

The code used as a reference will be the original code, in which we will correct and improve the optimizations implemented in the previous phases.

During this phase, several crucial steps were considered: execution to obtain the code profile and identify hotspots, analysis and modification of the code, compilation and execution, verification of code readability, and confirmation that the simulation outputs remain unchanged. After implementing all optimizations, a robust performance evaluation of the developed implementation is conducted.

A. Settings

All code builds in this stage were conducted using version 11.2.0 of the GNU Compiler Collection, better known as GCC. All measurements in this report were selected using the k-best 5 method, involving the selection of a set of 5 measurements with a precision of 5%. This means that, in addition to the presented measurement, there are four more measurements whose relative percentage deviation is less than 5% of the same value presented. All measurements were performed on SeARCH (Services and Advanced Research Computing with HTC/HPC clusters) on an INTEL x86-64 node, in the Ivy Bridge segment in the cpar partition, with two Intel Xeon E5-2670v2 processors (22 nm lithography). Each processor has 10 cores and 20 threads, so in this partition, 2x20 CPUs are available [1].

B. Original Code

Initially, in the single-thread optimizations, we have a thousand particles, but later on, the number of particles will be incremented to five thousand.

TABLE I. Execution Time and Other Information

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{\text{MISSES}}$** ($\times 10^6$)	**Time (s)**
1 012 051	0,65	653 735	2 453	261,5

Fig. 2. Table with the selected measurement of various parameters related to the original code.

Based on the execution profile of the original code, it was noticed that the *Potential* function consumes 68,52% of the execution time, followed by the *computeAccelerations* function, which occupies 31.55%.

II. Simplification of Complexity

Based on the code of the *Potential* function, the two loops *i* and *j*, ranging from 0 to N-1, contribute to a complexity of O(N^2). However, this complexity can be reduced by half when it is noticed that r[i][k] = r[j][k] for *i = j*, meaning that the initial loops can be reduced by more than half (considering the *k* loop as well). Consequently, there will be a decrease in the number of instructions and memory accesses, anticipating a reduction in the execution time.

TABLE II. Execution Time and Other Information

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{\text{MISSES}}$** ($\times 10^6$)	**Time (s)**
683 955	0,64	435 911	1 698	132,5

Fig. 3. Table with the selected measurement of various parameters related to the code after the first modification.

Reassessing the profile of the modified code, the *Potential* function now uses 51,4% of the execution time, followed by the *computeAccelerations* function, which occupies 48,8%. As a result of the reduced complexity, a decrease in all parameters was observed, as expected, including the execution time, which decreased by 49,3% compared to the previously measured time.

III. Optimization of Instructions within the Loops

As both functions now have approximately the same computational load, both functions are optimised by reducing the number of operations within the loops. Therefore, the main goal of this optimization is to simplify calculations in order to decrease the number of instructions.

A. Potential

Looking at the code of the *Potencial* function, it becomes evident that there is a possibility to avoid redundant calculations and remove function calls: the 'sqrt' function is unnecessary when working with squares, and using 'pow' for small exponents in this case is inefficient because the instructions for the pow and sqrt functions are located in a memory region that takes longer to access.

B. ComputeAccelerations

Again, since the exponents in the pow functions are small, calling the function becomes inefficient. However, multiplying several divisions is computationally expensive because division is more computationally intensive. An alternative is to perform only one division per iteration of j, reducing repetitive instructions.

TABLE III. Execution Time and Other Information

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{MISSES}$** ($\times 10^6$)	**Time (s)**
54 682	0,45	24 734	409	9,89

Fig. 4. Table with the selected measurement of various parameters related to the code after the second modification.

Reanalyzing the profile of the modified code, the *computeAccelerations* function now occupies 71,3% of the execution time, followed by the *Potential* function, which occupies 28,9%. We observe a decrease in execution time of 92,5% compared to the previously measured time.

IV. Vectorization + Unroll

During the first phase, due to a compilation option error, vectorization and unrolling were not applied (resulting in insignificant gains). In this final phase, we decided to correct and apply vectorization compilation options, where more values will be loaded into the cache when reading memory. Essentially, with vectorization, we can process more than one element simultaneously using vector registers that allow storing these elements. AVX vector registers have 256 bits, and since a double is 64 bits, we can store 4 doubles in an AVX vector register. Thus, we can process 4 doubles simultaneously.

Considering also that the functions with higher computational load, which calculate the distance between two particles along an axis, perform r[i][k] - r[j][k], we can take advantage of spatial locality by using r[k][i] - r[k][j]. In addition to the mentioned implementations, considering there is no data dependence between loops, we apply the compilation option to perform unrolling.

TABLE IV. Execution Time and Other Information

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{MISSES}$** ($\times 10^6$)	**Time (s)**
24 357	0,78	18 887	423	5,75

Fig. 5. Table with the selected measurement of various parameters related to the code after the third modification.

Reanalyzing the profile of the modified code, the *computeAccelerations* function now occupies 81,5% of the execution time, followed by the *Potential* function, which occupies 17,5%. We observe a decrease in execution time of 41,9% compared to the previously measured time.

By using optimizations at the single-thread level only, we achieved a 97.8% reduction in execution time compared to the original code.

V. Parallelization with OpenMP

OpenMP is a parallel programming API that enables the parallelization of sequential code on a single computing node (with shared memory). Moving into multithreaded parallelism, we have changed the number of particles to five thousand from the previous sequential version.

TABLE V. Execution Time and Other Information

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{MISSES}$** ($\times 10^6$)	**Time (s)**
130 555	0,78	101 611	2 748	40,65

Fig. 6. Table with the selected measurement of various parameters related to the sequential code.

In general, the key points of the *Potential* and *computeAccelerations* functions: the loops in both functions have high complexity with a large number of instructions, so the *pragma omp parallel for* directive is applied to ensure distribution of iterations among multiple threads. The variables used within the loops are defined within them to eliminate data races, a phenomenon where different threads access and modify the same variables simultaneously, leading to unpredictable results. By defining the variables locally, each thread operates on an independent copy of the variable in question, avoiding data races and ensuring consistency in the results.

A. Potential

As there is an accumulative variable within the loop, it cannot be defined inside the loop. To eliminate data races, the clause *reduction(+:Pot)* is applied. This command creates a copy of the accumulative variable for each thread, and at the end of all thread iterations, it sums the accumulated value to the accumulative variable defined at the beginning of the function. The proposed approach prevents issues when multiple threads (independent units of execution) are working with the same variable simultaneously.

B. *ComputeAccelerations*

The approach for parallelizing the *computeAccelerations* function was identical to the one explained earlier. In this case, the *reduction(+:a)* is applied, where the difference lies in creating a copy of the buffer for each thread, and this copy is accumulated at the end of all iterations.

TABLE VI. EXECUTION TIME AND OTHER INFORMATION

Instructions ($\times 10^6$)	CPI	Clock Cycles ($\times 10^6$)	L1$_{MISSES}$ ($\times 10^6$)	Time (s)
148 046	0,94	140 171	2 782	4,08

Fig. 7. Table with the selected measurement of various parameters related to the code parallelized with OpenMP (20 threads).

After applying parallelism to the two functions with the highest computational load, a 90.71% reduction in execution time was observed compared to the previously measured time. Reassessing the code profile, the *computeAccelerations* function now uses 70,36% of the execution time, a *gomp_team_barrier_wait_end* function occupies 16,06%, and the *Potential* function 9,72% is accompanied by the *gomp_barrier_wait_end* function at 0,66%. The main objective now is to reduce the computational load of the *gomp_barrier_wait_end* and *gomp_team_barrier_wait_end* functions.

C. *Load Distribution*

Considering the assignment of tasks to threads, the first thread will have much more work than the last one. Therefore, through the loop schedule, we can better control the distribution of the load among the threads. The simplest implementation is a *schedule (dynamic)* because iterations are assigned based on the thread's availability.

TABLE VII. EXECUTION TIME AND OTHER INFORMATION

Instructions ($\times 10^6$)	CPI	Clock Cycles ($\times 10^6$)	L1$_{MISSES}$ ($\times 10^6$)	Time (s)
135 990	0,88	120 094	2 765	2,41

Fig. 8. Table with the selected measurement of various parameters related to the parallelized code with OpenMP (20 threads).

As a result of the OpenMP parallelization, a decrease in the execution time 94,1% compared to the previously measured sequential time. To assess the significance of load distribution, a runtime graph was generated for both dynamic and static thread assignments:

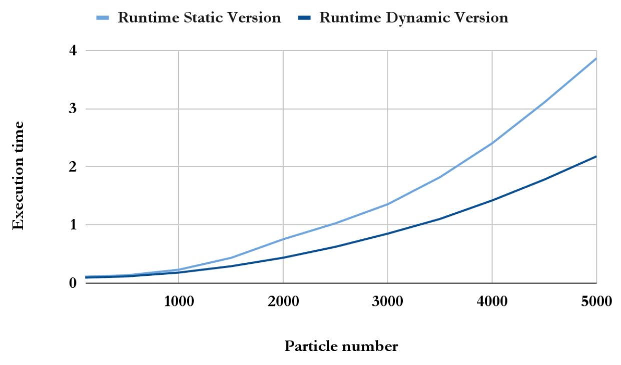

Fig. 9. "Execution time vs. Particle number, for schedule static and dynamic" Graph.

In the graph above, it's possible to observe a larger increase in static assignment. This is because as n increases, the execution time approaches that of the sequential version, thus increasing the overall execution time.

VI. OPENMP SCALABILITY ANALYSIS

Considering the multithreading optimization implemented, it is necessary to analyse the scalability of our code. Thus, with multiple measurements, the following graph of the gain as a function of the number of threads is obtained.

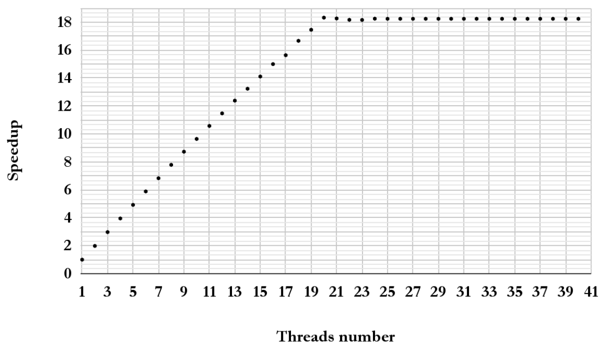

Fig. 10. "Speedup vs. Number of Threads" Graph

From the graph above and considering all the specifications of the cluster indicated in the introduction, i.e., in the used partition, we have 2 processors, and each of them has 10 physical threads, which, in turn, are composed of two logical threads. While the number of used threads is less than the number of physical threads, they will share the workload among them, and as expected, there is an approximately linear growth. This region of the graph is not perfectly linear due to the existence of sequential work in the code execution (*Amdahl's law*, strong scalability).

By increasing the number of threads beyond the number of available physical threads, logical threads are used, made possible by hyper-threading. However, the speedup remains approximately constant (Gustafson's law, weak scalability). This is because, although each thread is responsible for a smaller portion of the work, it implies a compromise of efficiency, as additional threads create synchronisation problems, competition for shared resources, and communication latencies. In this case, since we use dynamic scheduling, the use of hyper-threading is not advantageous because there will always be threads in waiting.

VII. PARALLELIZATION WITH MPI

MPI (Message Passing Interface) is a programming library that allows programmers to execute code in parallel across multiple computing nodes (with distributed memory).

In this third phase, the main goal is to implement MPI. Initially, an attempt was made to divide the particles into boxes/processes to work with local matrices and later with a global matrix to calculate interactions between particles from different boxes/processes. However, there were dependencies in the code that made it impossible to apply this approach, resulting in incorrect output. Some communication commands between processes were used during the optimization:

```
MPI_Barrier(MPI_Comm comm);
```

Fig. 11. *MPI_Barrier* syntax.

The *MPI Barrier* is a command that allows synchronising all processes to ensure that none of them proceeds until all have reached the point where the command is placed, with 'comm' being the communicator with the processes used for synchronisation.

```
MPI_Bcast(void *buffer, int count, MPI_Datatype
datatype, int root, MPI_Comm comm);
```

Fig. 12. MPI_Bcast Syntax.

The use of *MPI Bcast* is necessary to ensure that all processes have the same value in the count elements of the buffer. These elements are of type datatype and are transmitted from the root process to the MPI communicator 'comm'. Therefore, the data from the process with rank 0 is shared with all other processes.

```
Npp = N/size;
```

Fig. 13. Number of Particles per Process initialisation.

The number of particles per process is the total number of particles in the system, N, divided by the number of processes, size.

```
fst = Npp*rank;
```

Fig. 14. First index initialisation (depending on process).

```
lst = (rank == size - 1) ? N : (rank + 1 ) * Npp;
```

Fig. 15. Last index initialisation.

The index of the last particle assigned to each process, when this process is the last one (rank == size - 1), will be set to the total number of particles, N. Otherwise, it will be set to (rank + 1) * Npp, representing the index of the first particle of the next process.

In the *Potencial* function, we started by using the *MPI_Barrier* command to ensure that all processes had completed the local calculation before proceeding to the reduction. Next, we use *MPI_Reduce* to perform a local reduction (sum) of the potential energy from all processes. The result is stored in the potential energy variable of the process with rank 0. Finally, the function uses *MPI_Bcast* to broadcast the calculated potential energy to all other processes.

In the *computeAccelerations* function, the approach was identical to that used in the *Potencial* function. After calculating local accelerations for all particles in each process, *MPI_Reduce* was used to sum these accelerations from local interactions into a single acceleration vector in the process with rank 0. Finally, the function uses *MPI_Bcast* to broadcast the calculated acceleration vector to all other processes, ensuring smooth execution of *VelocityVerlet* and *Potencial*.

Despite OpenMP and MPI being quite different, the implementation idea in both is similar. While in OpenMP, we perform a reduction of variables that are modified within the parallelized loop, creating a copy in each thread and summing up at the end after all threads have finished their work. Analogously, in MPI, there is the creation of this local variable for each process, followed by the reduction of these variables into another variable.

TABLE VIII. EXECUTION TIME AND OTHER INFORMATION

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{\text{MISSES}}$** ($\times 10^6$)	**Time (s)**
152 665	0,76	201 852	2 904	4,18

Fig. 16. Table with the selected measurement of various parameters related to the parallelized code with MPI (20 threads).

There is an increase in the execution time of 70% compared to the OpenMP version, as the implemented approach is analogous to the OpenMP implementation without load distribution. However, by slightly modifying the code, it is possible to perform static load distribution, which in OpenMP corresponds to a schedule(static,1).

```
for (int i=rank; i<N-1; i=i+size)
    for (int j=i+1; j<N; j++)
```

Fig. 17. Cycles of the *Potencial* and *computeAccelerations* functions.

This happens because, as the rank varies from 0 to size - 1, and size is the number of processes, with the loops above, it is possible to distribute iterations among processes spaced by the number of processes.

TABLE IX. EXECUTION TIME AND OTHER INFORMATION

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{\text{MISSES}}$** ($\times 10^6$)	**Time (s)**
123 006	1,12	109 942	2 903	2,34

Fig. 18. Table with the selected measurement of various parameters related to the parallelized code with MPI (distribution and 20 threads).

As a result of the MPI parallelization, a decrease in the execution time 94,2% compared to the previously measured sequential time and 2,9% compared to OpenMP time.

VIII. MPI SCALABILITY ANALYSIS

Considering the multiprocesses optimization implemented, it is necessary to analyse the scalability of our code. Thus, with multiple measurements, the following graph of the gain as a function of the number of processes is obtained.

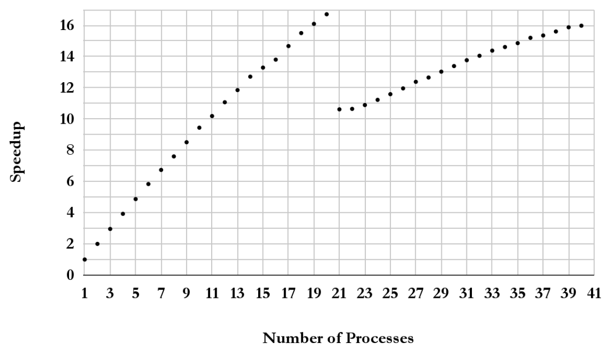

Fig. 19. "Speedup vs. Number of Processes" Graph

Again, considering the cluster specifications mentioned in the introduction, it is expected that up to 20 processes, the graph is approximately linear (Amdahl's Law, strong scalability). One significant difference compared to the OpenMP version is that after using 10 processes, strong scalability decreases (slope decreases). This is because, as processes need to communicate, the communication time between processors increases when communicating from one processor to another.

Another difference regarding OpenMP scalability is that when increasing the number of processes beyond the number of available physical cores, there is a significant decrease in speedup, and logical cores provided by hyper-threading are used. After the decrease in speedup, there is a linear increase with the increase in processes (Gustafson's Law, weak scalability). Although after 20 processes, each process is responsible for a smaller amount of work, as there are only 20 physical cores, one of the cores will become more overloaded than the others, leading to longer waits and hence a decrease in speedup. As the number of processes increases, this workload will be better distributed among processors, but in the end, it cannot achieve the speedup of 20 processes because, with more communications, there may be synchronisation problems and communication latencies.

IX. Particle Variation Analysis

Here is an analysis of the results obtained with the sequential code, the parallel version with OpenMP, and the parallel version with MPI. Four different parameters were analysed: execution time, the number of instructions, the number of clock cycles, and the number of level 1 cache misses.

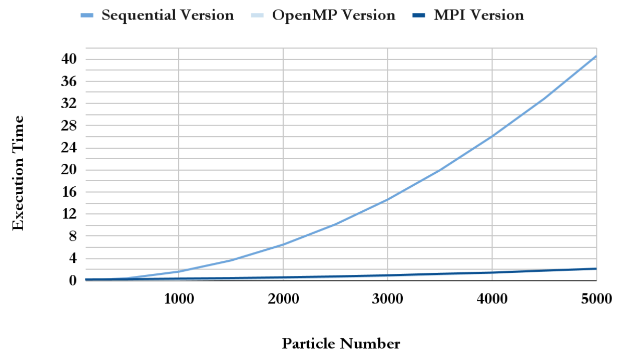

Fig. 20. "Execution Time vs. Particle Number" Graph.

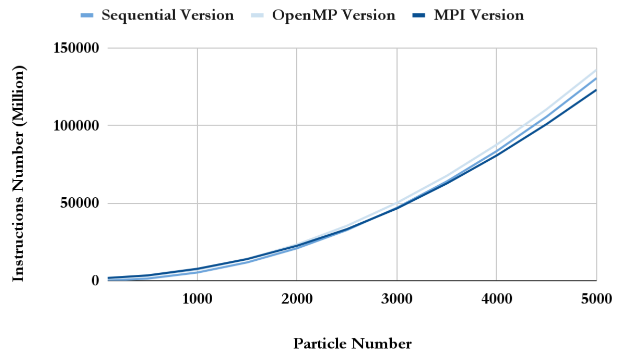

Fig. 21. "Instructions Number vs. Particle Number" Graph.

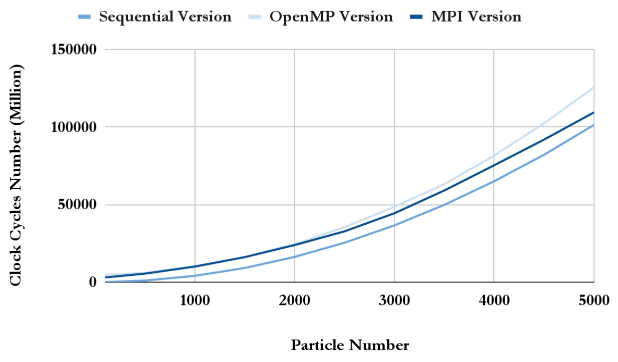

Fig. 22. "Clock Cycles Number vs. Particle Number" Graph.

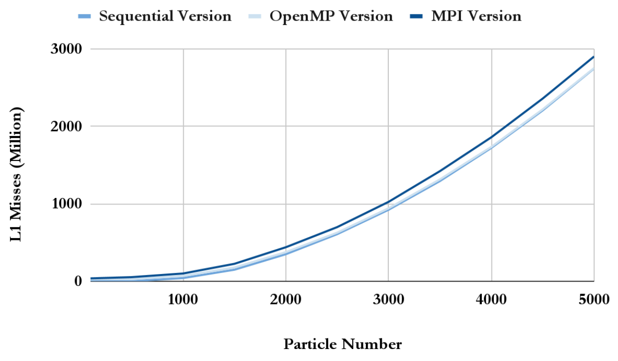

Fig. 23. "L1 Misses Number vs. Particle Number" Graph.

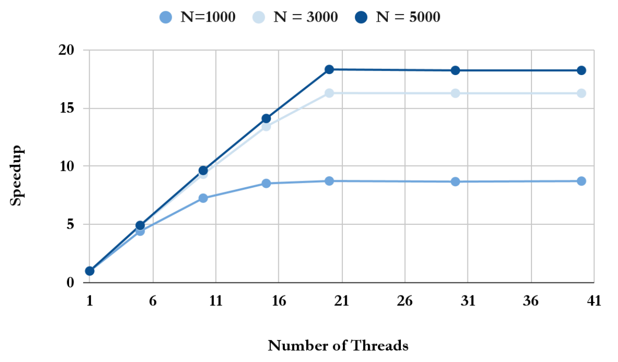

Fig. 24. “Speedup vs. Number of Threads" (OpenMP) Graph.

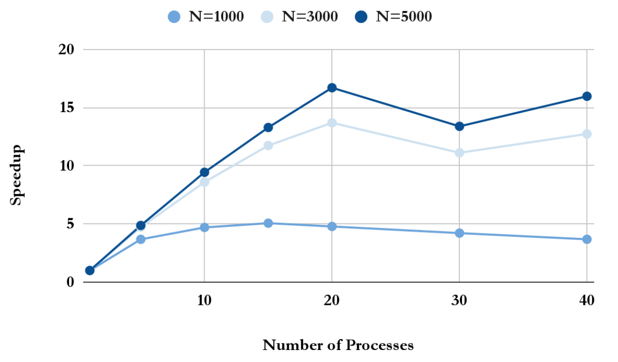

Fig. 25. “Speedup vs. Number of Processes" (MPI) Graph.

Observing the graphs above we can draw some conclusions. Firstly, both the parallel version with OpenMP and MPI have significantly reduced execution times when the number of particles increases. As for the remaining statistics, there are no significant changes.

The remaining statistics do not change much from version to version, this is because the only difference between the sequential version and the version with OpenMP and MPI is that in these the work is divided into several blocks, and thus this work is executed faster. This way, the number of instructions, the number of clock cycles and misses in the level 1 cache does not make sense to be different because it will only be distributed.

Observing the last two graphs, there is an increase in gain as the number of particles increases, both in the OpenMP version and in the MPI version. This means that with the increase in the number of particles we can take better advantage of parallelism.

Another conclusion that we can draw from the graphs above is that in all statistics, the growth with the number of particles is quadratic, this growth makes sense considering the complexity of the functions with greater weight.

X. Final Remarks

The analysis of Speedup depending on the number of colours used was useful to understand that in both MPI and OpenMP it is more favourable to use 20, as we have that number of physical colours. The best version we obtained with the MPI implementation was with load distribution.

TABLE X. Execution Time and Other Information

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**$L1_{MISSES}$** ($\times 10^6$)	**Time (s)**
123 006	1,12	109 942	2 903	2,34

Fig. 26. Table with the selected measurement of various parameters related to the code executed in MPI parallel with 20 processes.

Achieving a 94,2% reduction in execution time compared to the sequential code.

The analysis of particle number variation was very important to understand the impact of the program's efficiency with particle growth.

Thus, this work allowed us to understand the importance of knowledge of the computer structure and parallelization tools for the performance of the programs developed, considering the increasingly important energy factor.

XI. Future Work

A possible optimization, but which we were unable to implement, would be to calculate the closest interactions, considering that it will not affect the acceleration of particles that are very far away.

To implement this optimization we thought about dividing the box where the particles are located into sub boxes with the same dimensions, which depended on the number of processes. For example, with 3 processes we would have 27 sub blocks, 2 processes we would have 8 sub boxes.

The idea was to somehow iterate only nearby boxes and thus reduce the number of iterations. When a particle moves we could have to send it to another block and there would have to be communication between sub blocks.

Although the idea was designed for MPI, perhaps implementing it would be easier in CUDA. If we were able to implement this idea, the complexity of the problem would decrease to Nlog(N).

XII. References

[1] SeARCH | Services and Advanced Research Computing with HTC/HPC clusters. (n.d.). University of Minho. Retrieved from https://search6.di.uminho.pt/wordpress/ at 6th december of 2023