# Work Assignment - Phase 1

1st João Brito
*Mestrado em Engenharia Informática*
*Universidade do Minho*
Braga, Portugal
pg53944@alunos.uminho.pt

2nd Paulo Oliveira
*Mestrado em Engenharia Informática*
*Universidade do Minho*
Braga, Portugal
pg54133@alunos.uminho.pt

***Abstract*—This report addresses the process of analysis and optimization of a molecular dynamics simulation program applied to argon atoms. The main purpose of this assignment is to enhance the program's execution performance. This report includes analysis of its code, implemented modifications, a discussion of the strategies employed and of the results obtained.**

***Index Terms*—optimization, performance, flags, code**

## I. Introduction

The program analyzed in this study is part of a simple molecular dynamics simulation code that models the movement of particles over time. The code employs a set of mathematical expressions to calculate the trajectories of the particles, to describe the interactions between them, and to compute their paths. The primary aim of this work is to decrease the program's execution time under specific conditions, such as the number of atoms, density, etc.., while also ensuring that the output of the simulation output remains consistent.

## II. Code Analysis

In the initial approach to the code, the group thought the best strategy would be to start by adding local variables to the various functions present in the program. However, it quickly became apparent that this approach was not the most effective as the performance of the analyzed program did not improve when executing the command "*srun –partition=cpar perf stat -e instructions,cycles ./MD.exe ¡ inputdata.txt*". Therefore, it was decided to restart this task of code analysis, resorting to the functionalities of gprof where it was observed that most of the time spent executing the program was seen in two main functions, *potential()* and *computeAccelerations()*. Consequently, it was decided that the focus to optimize the code would be centered in these two functions to enhance the overall performance.

## III. Optimization Strategies

The first strategy the group pursued was to analyze which flags would be used in the *makefile*. We opted to use the following flags:

- **O3**: Activates a high level of optimization during compilation, instructing the compiler to apply various optimizations, such as the elimination of redundant code.
- **Ftree-vectorize**: This flag enables vectorization optimization, which transforms loops into vectorized operations, improving the performance of operations that can be executed in parallel.
- **fno-omit-frame-pointer**: This flag retains the frame pointer register.
- **ffast-math**: This flag activates optimizations that can accelerate mathematical operations.
- *mavx*: This flag instructs the compiler to generate code that takes advantage of AVX (Advanced Vector Extensions) vectorization instructions, improving performance in intensive mathematical calculations.
- *march=native*: This flag optimizes the code for the architecture of the CPU on which the code is executed.
- *pg*: This flag is used to generate profile information during compilation, allowing the use of gprof functionalities.

Some of these flags were taken from the work assignments used in practical classes for the course, while others were obtained through various tests that the group conducted to determine which ones provided the most assistance in improving program performance.

During the use and testing of these flags, and through the analysis of the functions that the group decided to improve, we arrived at an initial solution that could benefit the performance of these two functions. We began by working on and simplifying the existing mathematical expressions in the functions. Seeing that they were far from ideal, we decided to replace the expressions containing **pow** and **sqrt**, functions from the **math.lib** library, as they are more time-consuming in terms of execution time than simple arithmetic operations like addition, subtraction, multiplication, and division. This substitution of **pow** and **sqrt** with equivalent mathematical expressions or simpler operations also made the code clearer and more readable. Focusing on expressions containing these functions, we made changed to calculate and simplify them along with other expressions that enclosed them, which were also not very optimized.

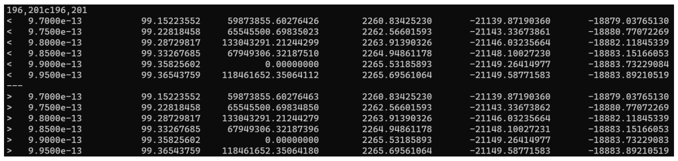
With that being said, and for better optimization, we decided to assign local variables to expressions with constant repetitions in order to perform them in smaller quantities. Another optimization we implemented was the locality of where the calculated values were inserted into the array *a[i]*, with their values stored in variables within the inner loop and only inserted into the array in the outer loop.

After concluding that there was nothing else to simplify in terms of mathematical expressions, we had the idea to apply a technique taught and explained during practical lessons called **loop unrolling**. The goal of this technique is by reducing the overhead of loop control code and increase the amount of work done in each iteration. This was achieved by executing multiple iterations of the loop code in a single pass. Using this proved to be very helpful as we encountered three different occasions where we had a for loop with a low number of iterations (0-3), significantly improving the execution of these functions.

Finally, after making minimal optimizations regarding mathematical expressions, such as avoiding or minimizing the use of multiplication or division operations in our equations, we had the idea to approach the problem from a different perspective and combine both functions to test the impact of this decision in our program. The merging of these two functions arose from the fact that they shared similar variables that could be handled in common for loops, reducing the overall number of cycles done. To maintain the output results unchanged, a global variable named **PEE** was created, which is responsible for storing the equivalent result that used to come from the **Potential()** function. This decision proved effective, as it allowed for a reduction in execution time and the total number of program instructions.

## IV. Results Analysis

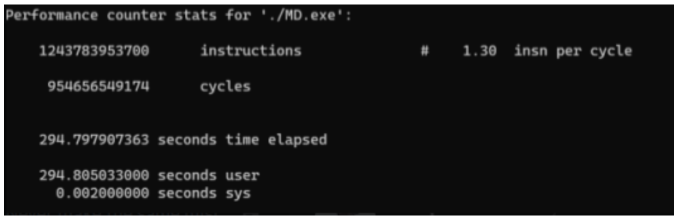
After all the implementations and techniques used to simplify and optimise the code, we needed to ensure that the obtained results were consistant with the initial results provided by the instructures. To do this, after running the command ”*diff cp-output.txt calculo1-output.txt*” we were able to compare and confirm the equality of the results, considering up to 12 numerical and decimal places of comparison:



```
196,201c196,201
<   9.7000e-13          99.15223552     59873855.60276426       2260.83425230       -21139.87190360       -18879.03765130
<   9.7500e-13          99.22818458     65545500.69835023       2262.56601593       -21143.33673861       -18880.77072269
<   9.8000e-13          99.28729817    133043291.21244299       2263.91390326       -21146.03235664       -18882.11845339
<   9.8500e-13          99.33267685     67949306.32187510       2264.94861178       -21148.10027230       -18883.15166053
<   9.9000e-13          99.35825602            0.00000000       2265.53185893       -21149.26414977       -18883.73229084
<   9.9500e-13          99.36543759    118461652.35064112       2265.69561064       -21149.58771583       -18883.89210519
---
>   9.7000e-13          99.15223552     59873855.60276463       2260.83425230       -21139.87190360       -18879.03765130
>   9.7500e-13          99.22818458     65545500.69834850       2262.56601593       -21143.33673862       -18880.77072269
>   9.8000e-13          99.28729817    133043291.21244279       2263.91390326       -21146.03235664       -18882.11845339
>   9.8500e-13          99.33267685     67949306.32187396       2264.94861178       -21148.10027231       -18883.15166053
>   9.9000e-13          99.35825602            0.00000000       2265.53185893       -21149.26414977       -18883.73229083
>   9.9500e-13          99.36543759    118461652.35064180       2265.69561064       -21149.58771583       -18883.89210519
```

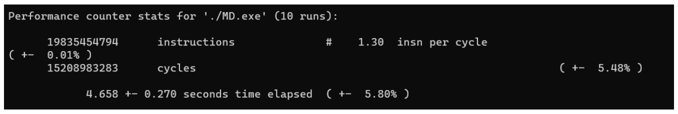
Fig. 1. Results comparison

We also observed a significant improvement in execution time and the initial number of instructions, with values decresing from 294 seconds and 124,378,395,700 number of instructions to an average of 4.65 seconds of program execution and 19,835,454,794 instructions.



```
Performance counter stats for './MD.exe':

    1243783953700      instructions              #    1.30  insn per cycle

     954656549174      cycles


    294.797907363 seconds time elapsed

    294.805033000 seconds user
      0.002000000 seconds sys
```

Fig. 2. Initial performance



```
Performance counter stats for './MD.exe' (10 runs):

      19835454794      instructions              #    1.30  insn per cycle
( +-  0.01% )
      15208983283      cycles                                                        ( +-  5.48% )

            4.658 +- 0.270 seconds time elapsed  ( +-  5.80% )
```

Fig. 3. Final performance

## V. Conclusion

This work allowed us to consolidate several topics covered in Parallel Computing classes, showing the importance of optimization techniques in improving the performance of a program. In short, although there are several possible optimizations for this code and, possibly, the optimization made by the group was not ideal, I think that the group managed to achieve the main objective of this phase, which was to significantly reduce the program’s execution time, while maintaining the readability of the code.

## References

[1] AJProença, Parallel Computing, MEI, UMinho, 2023/24

[2] Wolfram Alpha, ”Website for Computational and Numerical Analysis”, https://www.wolframalpha.com