Computação Paralela - WA1

Mestrado em Engenharia Informática

Afonso Marques

Departamento de Informática
Universidade do Minho
Mirandela, Portugal
pg53601

Renato Gomes

Departamento de Informática
Universidade do Minho
Porto, Portugal
pg54174

***Abstract*—The program to analyse and optimise is part of a simple molecular dynamics simulation code applied to atoms of argon gas (original code version in FoleyLab/MolecularDynamics: Simple Molecular Dynamics). The code follows a generic approach to simulate particle movements over time steps, using Newton laws of motion. In this report we shall refer to the program as MD.**

The code uses Lennard Jones potential to describe the interactions among two particles (force/potential energy). The Verlet integration method is used to calculate the particles trajectories over time. Boundary conditions (i.e., particles moving outside the simulation space) are managed by elastic walls.

***Index Terms*—algorithm, performance, single thread, optimization, metrics**

I. Introduction

The main goal of this project was to evaluate the benefits of optimization of code in a single thread, using C++, by making changes to certain algorithms present in the *MD.cpp* file.

We started off by removing complex calculations from the inside of *loops* plus replacing the use of the *pow* function from the *math.h* library with an uglier but better performing alternative. During this process, we took the care not to interfere with the calculations being made as to make sure the final output would not be changed in any way. Furthermore, we performed tests using the *perf* metrics command to see how well our version of the code was performing.

We also deconstructed smaller loops that had an average of three to five iterations, which in turn removed additional instructions and improved the performance, even if so slightly.

Lastly, the obtained results were analysed and compared with the expected results.

II. Implementation

The original version of the MD code consists of four main steps.

1) Put all the atoms in simple crystal lattice and give them random velocities that corresponds to the initial temperature we have specified, using the *initialize* function;

2) Based on the positions of the atoms, calculate the initial inter molecular forces. The accelerations of each particle will be defined from the forces and their mass, and this will allow to update their positions via Newton's law. The calculus is made in the *computeAccelerations* function.

3) Calculate various things like instantaneous mean velocity squared, temperature, pressure, potential and kinetic energy.

4) Print the final results to the terminal and additional results to the *cp_average.txt* and *cp_output.txt*. The *cp_output.txt* contains all the values of temperature (in Kelvin), pressure (in Pascal), kinetic, potential and total energy (in natural units n.u). The *cp_average.txt* contains all of the values that were printed on the terminal.

III. Algorithmic Complexity

1) *Potential()*: The potential function involves three nested loops. The outer two loops iterate over all particles in the system, resulting in N^2 iterations. Inside the innermost loop, there are several mathematical operations like addition, subtraction, multiplication, and power calculations. These operations are not nested within the loops and are generally considered to be constant time. So, the time complexity of the *Potential()* function is O(N^2).

2) *computeAcceletaions()*: This function also involves nested loops. The outer loop iterates over all particles, and the two inner loops iterate over the components for each pair of particles. The most computationally intensive part is the calculation of forces (f) and updating the accelerations. Inside the inner loops, there are constant time operations as well as calculations that depend on the number of dimensions (3 in this case). Overall, the time complexity of this function is also O(N^2).

IV. Optimizations and Metrics

Before looking ate the code, the group decided it was best to use the *gprof* tool to access what parts of the code used more time and computation power.

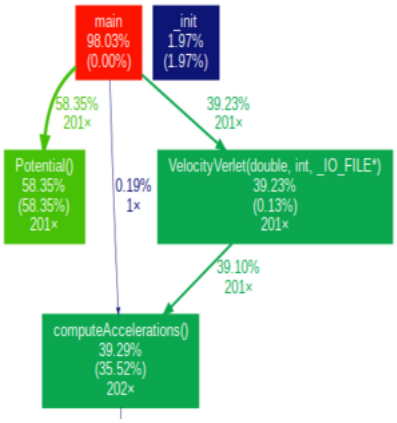

Fig. 1. *gprof* graph from the original code

Looking at the graph it's easy to see that the functions that take most of the time to execute are *Potencial()* and *computeAccelerations()*, making them the best candidates for optimization.

A. Optimizations

1) Potencial():

- We started by removing the *sqrt()* function replacing it with a multiplication. The *quot* variable became *sigma/r2*, which is the original *sigma/rnorm* squared.
- Next we removed the *pow()* functions, by replacing them with multiplications. As *quot* is now the original's *quot* squared, *term2* instead of multiplying *quot* 6 times, we can do it only 3 times. And *term1* is *term2* * *term2*.
- In the original code there's an *if (j!=i)* condition to ensure that a particle is not interacting with itself. By removing that condition and starting the loops from *i+1*, we can avoid self-interaction, and that way we have a big improvement in performance.

2) computeAccelarations():

- Similarly as in the *Potencial()* function, we started by trying to optimize the calculations, starting with the removal of the *pow* functions. The introduction of two new variables, *invrSqd* and *invrSqd3*, where *invrSqd* is defined as the reciprocal of the original *rSqd*, and *invrSqd3* is the cube of *invrSqd*. This change simplifies the code by avoiding repetitive division operations when dealing with the reciprocal. The calculation *2 * pow(rSqd, -7)* in the original is transformed into *2.0 * invrSqd3*. This change replaces the use of the pow function with direct multiplication. The second part of the calculation *- pow(rSqd, -4)* in the original is transformed into -1.0, because *invrSqd3 * invrSqd* is already used to represent the $1/r^4$ term.
- In the original *computeAccelerations()* function, the loops over k are nested within the loops over i and j. This means that for each pair of particles i and j, there's an inner loop over k (k=0, k=1, k=2). In the optimized version, the inner loop over k has been eliminated. Instead, each element is computed separately without an explicit loop. This eliminates the need for an inner loop and can lead to slightly improved performance and code readability.

B. Metrics

The code was tested for performance and reliability through 3 independent runs on the university's cluster (3 was the best we could do because otherwise the cluster would crash). To ensure robustness, the results of each run were carefully examined, and an average was computed to assess the code's behavior. This thorough testing approach helped verify the code's stability and performance across multiple executions.

TABLE I
TABLE WITH EXECUTION TIME FOR 3 ITERATIONS OF **ORIGINAL VERSION OF MD**

Iteration	Texe
1	236,95 sec
2	236.79 sec
3	236,92 sec

TABLE II
TABLE WITH EXECUTION TIME FOR 3 ITERATIONS OF **OPTIMIZED VERSION OF MD**

Iteration	Texe
1	6.82 sec
2	6.43 sec
3	6.65 sec

TABLE III
TABLE WITH METRICS FOR EACH VERSION OF MD

Version MD.cpp	#I	Average #CC	Average CPI	Average Texe
Original	1.243.580.424.482	780.836.157.356	0,6	236,8770 sec
Optimized	35.346.319.145	21.426.058.674	0,6	6,654 sec

References

[1] https://iopscience.iop.org/article/10.1088/1742-6596/1491/1/012022/pdf
[2] https://pt.wikipedia.org/wiki/Potencial_de_Lennard-Jones
[3] https://chem.libretexts.org/Bookshelves

V. Annexes

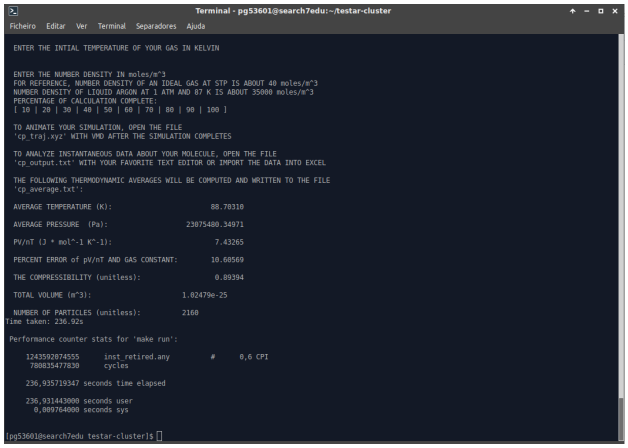

Fig. 2. Example of the original MD.cpp result in the Search cluster

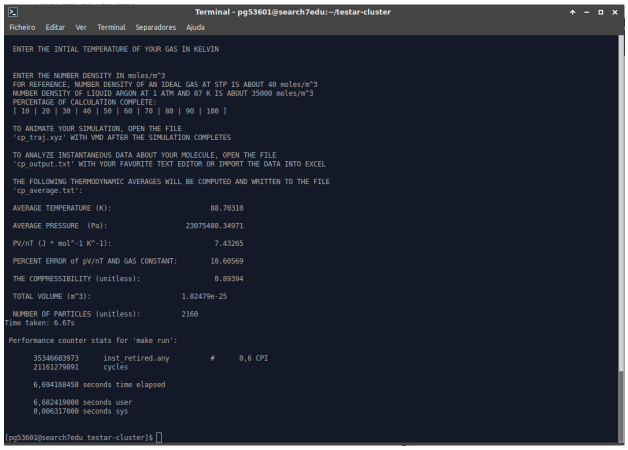

Fig. 3. Example of the optimized MD.cpp result in the Search cluster

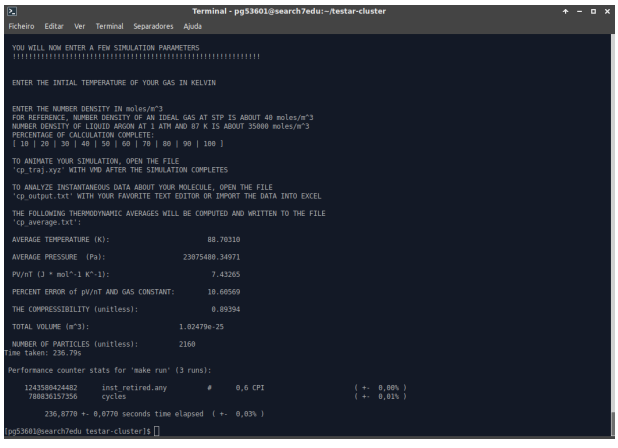

Fig. 4. Example of the result of 3 iterations of the original MD.cpp in the Search cluster

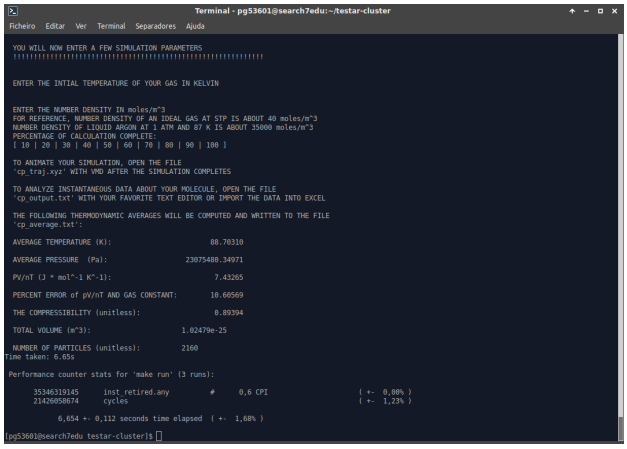

Fig. 5. Example of the result of 3 iterations of the optimized MD.cpp in the Search cluster

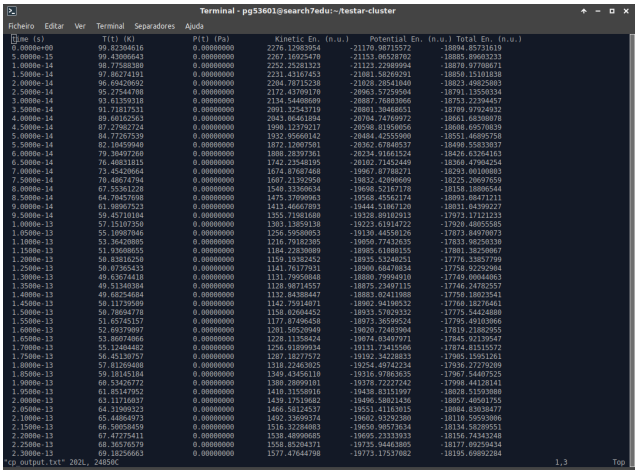

Fig. 6. cp_output.txt file obtained with optimized MD