Parallel Computing - Work Assignment (Phase 1)

José Moreira
Software Engineering
University of Minho
Braga, Portugal
pg53963@alunos.uminho.pt

Santiago Domingues
Software Engineering
University of Minho
Braga, Portugal
pg54225@alunos.uminho.pt

***Abstract*—Parallel Computing is an architecture in which the main goal is to improve the execution of a given program by breaking down larger problems into smaller, in a way that those smaller and often similar parts can be executed simultaneously by multiple processors. One of the main concerns of Parallel Computing is how good a software can be in terms of performance (response time, resources management, etc.).**

***Index Terms*—Instruction Level Parallelism, memory hierarchy, data structures organization, vectorization**

I. Introduction

This document serves as the report for the initial phase of the **Parallel Computing** work assignment, that consists in exploring different optimization techniques in a single threaded program, using several tools for code analysis/profiling, improvement and performance evaluation (based in execution time, number of instructions and number of clock cycles). The program, which simulates the behaviour of argon gas atoms, follows fundamental principles of physics, including Newton's laws and the Lennard Jones potential model. This way, the key steps of this assignment are: code analysing and reduction of its execution time, while a lot of performance tests are made and the code legibility is maintained.

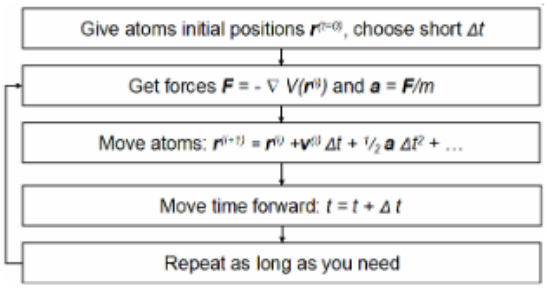

Fig. 1. Particle movements simulation over time steps, using the Newton law

II. Code Analysis

First of all, the initial challenge was to understand the code and all the dependencies among its functions. Although being a quite simple program, it required a somehow high number of functions to execute correctly, which didn't have the same weight in terms of performance and, clearly, contributed in different percentages for the final execution time. We reached this goal by analyzing the program through the **gprof**. With this tool, we could observe the most heavy functions and their performance characteristics: **Potential()** and **computeAccelerations()**. This important step has revealed itself as the starting point for all our assignment.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 61.62     22.64    22.64      201   112.66   112.66  Potential()
 38.46     36.78    14.13      202    69.98    69.98  computeAccelerations()
  0.03     36.79     0.01      201     0.05    70.02  VelocityVerlet(double, int, _IO_FILE*)
  0.00     36.79     0.00     6480     0.00     0.00  gaussdist()
  0.00     36.79     0.00      201     0.00     0.00  MeanSquaredVelocity()
  0.00     36.79     0.00      201     0.00     0.00  Kinetic()
  0.00     36.79     0.00        1     0.00     0.00  initialize()
  0.00     36.79     0.00        1     0.00     0.00  initializeVelocities()
```

Fig. 2. Initial code profiling

After this major step, the group made some tests, running **perf**, that could show the numeric attributes of the code performance: number of **instructions**, number of **clock cycles** and **execution time**.

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

    1256799036804      instructions              #    1,57  insn per cycle
     799615780704      cycles

    247,127541888 seconds time elapsed

    242,935223000 seconds user
      0,003000000 seconds sys
```

Fig. 3. Initial code performance

Once we didn't found it relevant, we got the following results with only one performance run: **247.128s**, with nearly $\mathbf{1.257 * 10^{12}}$ instructions and nearly $\mathbf{7.996 * 10^{11}}$ clock cycles.

III. Modifications

Since the main goal was to reduce the execution time for the given set of conditions while maintaining all simulation outputs, we started by simplifying the mathematical calculations of the **Potential()** function. For example, we removed the calls for the function **pow()** because they were too costful in terms of time and, in an initial phase, replaced it with the equivalent multiplication expressions. Further into the work, we took a look at the *Lennard-Jones potential* equation and saw that we could simplify even more, removing the **sqrt()** calls with decreasing the respective power values. After this

important step, we removed some repeated calculations by saving those values in local variables, accessing the **r** matrix in consecutive steps, so the vectorization could do its job.

Following these significant changes, we had to make a deeper analysis of the code we had in our hands. After some discussions, we could realize that the **Potential()** function was doing some unnecessary job. Since its main objective was calculating the potential energy of the system, considering the interactions between particles with one another. But the main problem is that the function was considering the interaction, for example, $a \rightarrow b$ and $b \rightarrow a$. Once it uses the positions of the particles to do those calculations, the energy released by the interaction $a \rightarrow b$ is the same amount of energy released by the interaction $b \rightarrow a$. So, we changed our code to calculate only one of those interactions (for all pairs), making the function to return the final result *** 2**.

Posterior to the previous modifications, we went to optimize the second heaviest function, **computeAccelerations()**. Similar to the steps we made with **Potential()**, we started by simplifying the calculations and removing the repeated ones, by saving their values in local variables. These small steps made our code faster and improved its legibility. One of the biggest optimization phases of our assignment came next, when we understood that the two most heavy functions of the code were doing identical calculations. They were accessing the same matrix and their middle maths were exactly the same, where the only differences were the final applications of those calculations (**Pot+= vs f=**). So, we created a new function, **computeAccelerationsPotential()**, responsible for returning the value of the potential energy of the system, while updating the acceleration matrix. We were able to do all these changes, because the last call of the **computeAccelerations()** function was being made inside **VelocityVerlet()**, after it updated the **r** matrix values for the last time. To take advantage from this, the return type of this last function was updated, from **double** to **double***, so it could return its original value together with the **computeAccelerationsPotential()** result. To finish this big step, we ended by accessing the **main()** of the program, declaring a local variable (**double ***) with the responsibility of saving both values mentioned before.

At the end, we just had to use one last optimization technique, the **loop unrolling**, that basically consists in removing loop cycles when they can easily be substituted (or decreased), manually, by another instructions with the same objective. All these steps, together with adding flags in the **Makefile** (**-g -fno-omit-frame-pointer -pg -O3 -march=native -mavx -ftree-vectorize -msse4**), made our assignment objectives possible.

IV. Results

On this phase, we can prove that our effort was successful. All the techniques used (**code profiling**, **vectorization**, **loop unrolling**, etc.) produced some amazing results. This time, we ran perf using a repetition flag (**-r 10**) that allowed us to observe the next average values: the execution time of the final code is, approximately, **5.075s**, with nearly **2.080 * 10**^{10} instructions and nearly **1.619 * 10**^{10} clock cycles. With this information, we can relate that our actual program executes about **48.70x** faster, with **1.236 * 10**^{12} instructions and **7.834 * 10**^{11} cycles less.

```
                     Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.19% of 5.40 seconds

index % time    self  children    called     name
                5.40    0.00     201/201         VelocityVerlet(double, int, _IO_FILE*) [2]
[1]    100.0    5.40    0.00     201         computeAccelerationsPotential() [1]
-----------------------------------------------
                                               <spontaneous>
[2]    100.0    0.00    5.40                 VelocityVerlet(double, int, _IO_FILE*) [2]
                5.40    0.00     201/201         computeAccelerationsPotential() [1]
-----------------------------------------------
```

Fig. 4. Final code profiling

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

     20795081399      instructions              #    1,28  insn per cycle                                ( +-  0,01% )
     16191822418      cycles                                                                           ( +-  4,47% )

          5,075 +- 0,256 seconds time elapsed  ( +-  5,04% )
```

Fig. 5. Final code performance

V. Conclusions

This assignment gave us a really good perspective about software performance and different ways to improve the parameters that consume the most time and CPU during the execution of a program. Through the utilization of several tools and techniques, we could comprehend the main challenges of **Parallel Computing** and also the big impact that it has in nowadays programs. Although this was a relatively simple program, it was capable of showing how powerful simple operations such as simplifying a mathematical calculus or using a loop unroll technique can be in the final execution time. The synergy of all of those factors, combined with the deployment of bigger and more high-powered machines, can really make an impact in more sophisticated projects and initiatives of those big technological enterprises that are trying to change the technological panorama as we know. All the process involved in the realization of this task made us understand, in a very practical and interesting way, some of the main concerns and goals of the current software systems and allowed us to increase our knowledge in this area. Concluding, we can express our satisfaction with the final result of our assignment and, this way, we finish, here, the present report of the **Work Assignment (Phase 1)**, relative to the curricular unit of **Parallel Computing**.

References

[1] Sobral J., "Molecular Dynamics simulations: A short overview," Univ. Minho, Accessed on October 13, 2023.

[2] Navarro Cristobal A, Hitschfeld Nancy, Mateu Luis, "Molecular dynamics simulations: advances and applications," Dovepress, Accessed on October 13, 2023.

[3] Liang Xuejun, Humos Ali A., Pei Tzusheng, "Vectorization and Parallelization of Loops in C/C++ Code," Jackson State University, Accessed on October 15, 2023.