Work Assigment 2 - Computação Paralela

1st Marcos Paulo P. de Cerqueira
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
PG52692

2nd Paulo H. P. de Cerqueira
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
PG52699

***Abstract*—This article presents a simple molecular dynamics (MD) program developed to simulate the properties of real gases using particles modeled by the Lennard-Jones potential. Our job was make the code run faster in an cluster environment provided by Universidade do Minho in Braga, using the OpenMP library, which enables the parallelization of the code, analyzing.**

***Index Terms*—Molecular Dynamics, Real Gases, Particle Simulation, Lennard-Jones Potential,OpenMP, Cluster.**

I. Introduction

A. Context As computer processors (CPUs) have advanced, there has been significant enhancement in the performance of individual cores through improvements in pipelining, branch prediction, and higher clock frequencies. Additionally, progress has been made by increasing the number of cores on CPUs and introducing features such as ”hyperthreading.” These advancements enable computers to concurrently manage multiple tasks or distribute a demanding task among numerous threads and cores.

B. The Problem and Strategy Adopted We face the challenge of optimizing the simulation of molecular dynamics in real gases, specifically using the Lennard-Jones potential and the OpenMP library. The computational intensity of this task arises from the complexity of molecular interactions, and the efficient parallelization of the code with OpenMP is crucial for improving performance on multicore systems. However, precisely integrating and optimizing the use of OpenMP in these simulations, especially in contexts involving real gases, remains a significant technical challenge that requires specialized approaches.

II. TESTS AND METRICS TO BE EVALUATED

In this project, the sole metric employed to assess the algorithm's performance will be its execution time. This metric is considered the most indicative measure of how effectively and efficiently the CPU resources are being managed.

The tests involved incorporating OpenMP directives into the provided code and measuring the performance gain compared to the sequential code. Various thread counts were utilized for this purpose.

```
AVERAGE TEMPERATURE (K):                  131.455064150292
AVERAGE PRESSURE  (Pa):                   131001228.450759902596
PV/nT (J * mol^-1 K^-1):                  28.472788520122
PERCENT ERROR of pV/nT AND GAS CONSTANT:  242.449049066562
THE COMPRESSIBILITY (unitless):            3.424490465357
TOTAL VOLUME (m^3):                       2.37220e-25
NUMBER OF PARTICLES (unitless):           5000
```

Fig. 1. Original code output.

Function	Module	CPU Time	% of CPU Time
computeAccelerations	MD.exe	69.566s	80.4%
Potential	MD.exe	16.665s	19.3%

Fig. 2. Original code time for functions.

III. Optimization

By using the Intel VTune Profiler, we observed that the two functions consuming the most resources are `computeAccelerations` and `Potential`. Due to a symmetry in the code structure, it became apparent that `Potential` essentially performs the same programming logic as `computeAccelerations`. Consequently, we were able to execute a code block only once, significantly reducing the overall execution time. (figure 2).

A. Parallelization analysis

Analyzing the combined code of the potential function and computerAccelerations, it was possible to determine that there were zones where ”data race” could occur, as shown in Figure 3.

Since it was a code block, the #pragma omp critical instruction from OpenMP was used, and in the for loop, we used the #pragma omp parallel instruction.

However, it was observed that the parallel execution times compared to the sequential ones were excessively large, which does not make sense for our goal. The execution time was reached with two threads, with the parallelized version reaching 527.104s and the sequential version 133.96s.

As we believed the issue was in the block where data races could occur, we decided to use the #pragma omp atomic

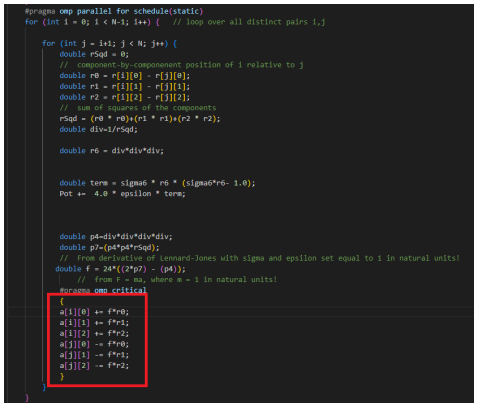

Fig. 3. Data race zone

instruction individually for the instructions of matrix 'a', as shown in Figure 4.

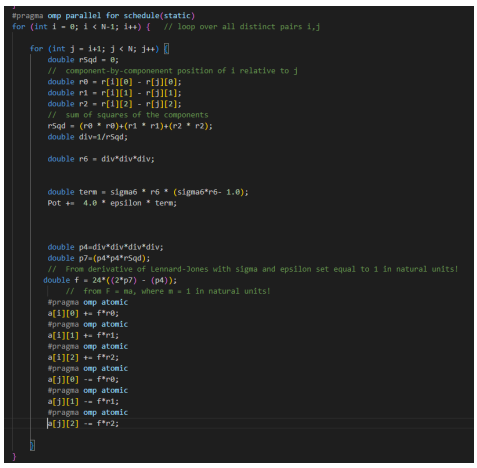

Fig. 4. use pragma omp atomic

We tried other approaches to eliminate the data race from the code using temporary variables, but we did not achieve a speedup gain. Among our results, the best we obtained was to put the for loop block inside the #pragma omp critical instruction, which, in our analysis, is not a good approach because critical is pessimistic, and while it executes, other threads are waiting.

Thus, the graph of speedup per threads was collected.

We can see that with 24 threads, we obtained a speedup of 4, far from ideal. It is worth noting that we ran the same code, and it exhibited different behaviors, varying the speedup considerably. At another time, we obtained maximum speed-ups of 1.22

We believe that the problem presented is not well-suited for parallelization with OpenMP because the limiting code block inside the main for loop prevents significant improvements in the presented time. This is due to both the #pragma omp critical and #pragma omp atomic directives delaying the execution of threads. Additionally, the speedup less than 1 occurs

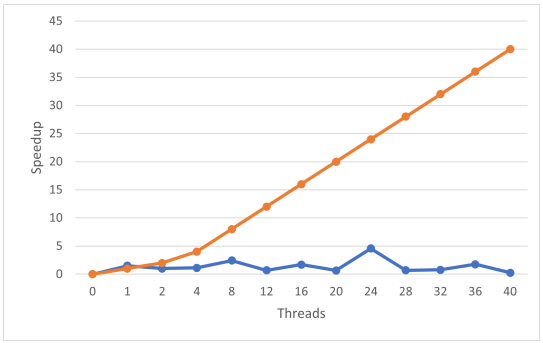

Fig. 5. speed-up x threads

because OpenMP spends time managing thread execution, which does not provide an advantage in this example.

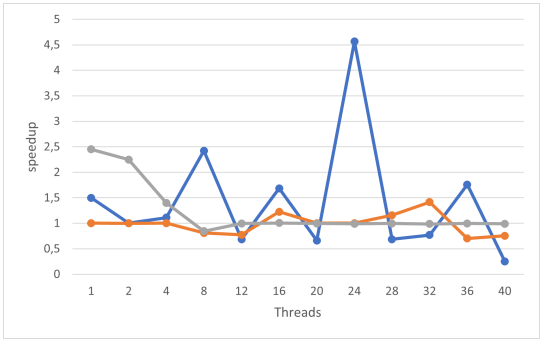

Fig. 6. speed-up x threads parallelization code

In Fig 6, we show the same parallelized code and the achieved speedups. We observe a significant variation in execution time and anomalous behavior across different threads.

IV. Conclusion

Analyzing the collected data, we noticed that in the case of the presented algorithm, parallelization with OpenMP does not yield significant gains, as was the objective of this work.

Gains through code optimization, such as the case of combining the Potential and ComputerAccelerations functions, hold more relevance. Considering that the execution time is relatively low and OpenMP incurs a time cost in managing threads, the parallel version becomes dispensable in this scenario.

We observed a significant variation in measurement times associated with the cluster, something beyond our control.

However, for didactic purposes, it was important to examine how OpenMP directives behave differently and have a significant impact on execution time.