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).

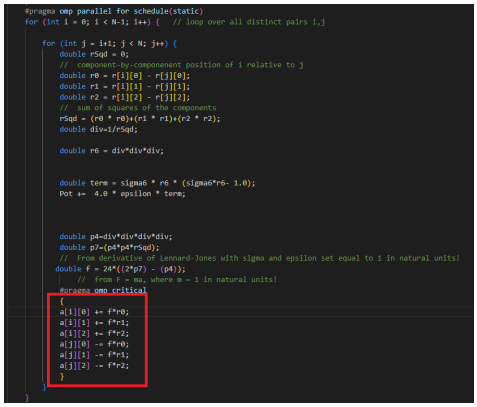
A. Parallelization analysis

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.

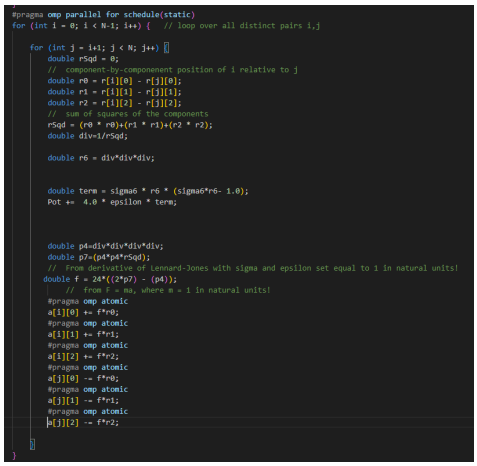
As we believed the issue was in the block where data races could occur, we decided to use the #pragma omp atomic



```
#pragma omp parallel for schedule(static)
for (int i = 0; i < N-1; i++) {   // loop over all distinct pairs i,j

    for (int j = i+1; j < N; j++) {
        double rSqd = 0;
        //  component-by-componenent position of i relative to j
        double r0 = r[i][0] - r[j][0];
        double r1 = r[i][1] - r[j][1];
        double r2 = r[i][2] - r[j][2];
        //  sum of squares of the components
        rSqd = (r0 * r0)+(r1 * r1)+(r2 * r2);
        double div=1/rSqd;

        double r6 = div*div*div;


        double term = sigma6 * r6 * (sigma6*r6- 1.0);
        Pot +=  4.0 * epsilon * term;



        double p4=div*div*div*div;
        double p7=(p4*p4*rSqd);
        //  From derivative of Lennard-Jones with sigma and epsilon set equal to 1 in natural units!
       double f = 24*((2*p7) - (p4));
            //  from F = ma, where m = 1 in natural units!
            #pragma omp critical
            {
            a[i][0] += f*r0;
            a[i][1] += f*r1;
            a[i][2] += f*r2;
            a[j][0] -= f*r0;
            a[j][1] -= f*r1;
            a[j][2] -= f*r2;
            }
    }
}
```

Fig. 3. Data race zone

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



```
#pragma omp parallel for schedule(static)
for (int i = 0; i < N-1; i++) {   // loop over all distinct pairs i,j

    for (int j = i+1; j < N; j++) {
        double rSqd = 0;
        //  component-by-componenent position of i relative to j
        double r0 = r[i][0] - r[j][0];
        double r1 = r[i][1] - r[j][1];
        double r2 = r[i][2] - r[j][2];
        //  sum of squares of the components
        rSqd = (r0 * r0)+(r1 * r1)+(r2 * r2);
        double div=1/rSqd;

        double r6 = div*div*div;


        double term = sigma6 * r6 * (sigma6*r6- 1.0);
        Pot +=  4.0 * epsilon * term;



        double p4=div*div*div*div;
        double p7=(p4*p4*rSqd);
        //  From derivative of Lennard-Jones with sigma and epsilon set equal to 1 in natural units!
       double f = 24*((2*p7) - (p4));
            //  from F = ma, where m = 1 in natural units!
            #pragma omp atomic
            a[i][0] += f*r0;
            #pragma omp atomic
            a[i][1] += f*r1;
            #pragma omp atomic
            a[i][2] += f*r2;
            #pragma omp atomic
            a[j][0] -= f*r0;
            #pragma omp atomic
            a[j][1] -= f*r1;
            #pragma omp atomic
            a[j][2] -= f*r2;

    }
}
```

Fig. 4. use pragma omp atomic

We tried other approaches to eliminate the data race from the code using temporary variables, The alternative we envisioned was a temporary array to store the value of 'a' and then update the array outside the loop.

Thus, the graph of speedup per threads was collected.

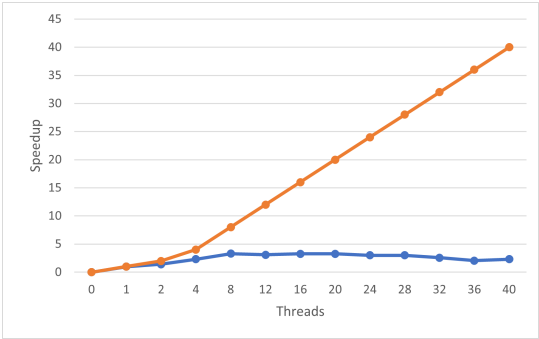


Fig. 5. speed-up x threads

We can observe that the speedup does not correspond to what is expected for the threads; this is partly because the C++ program has some of its functions not parallelized.

The behavior of the code parallelized by threads can be seen in Fig 6.

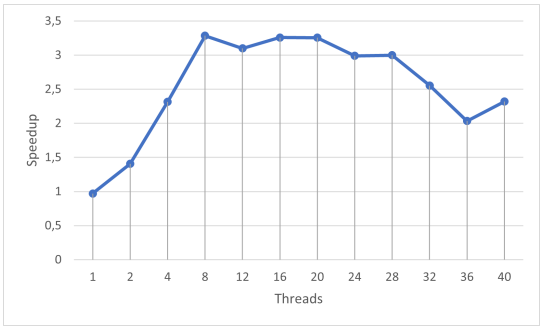


Fig. 6. speed-up x threads parallelization code

In Fig. 6, we can observe that the code does not exhibit a linear behavior, far from what is expected for parallelized code. This behavior can have various causes, including the fact that with more threads, there is increased competition among them and higher time spent by OpenMP to control them. Therefore, the program showed better performance with 8 threads.

IV. Conclusion

Analyzing the collected data, we noticed that the algorithm shows gains on the order of three times compared to the sequential version. However, the part of the code that was not parallelized, although it does not have an influence on the total time, may hinder a better utilization of additional threads.

A significant optimization of the code, such as the combination of the Potential and ComputerAccelerations functions, and removing the update of matrix 'a' from the main for loop, had a great impact on the total time gain and should also be considered for code optimization.

We emphasize that compiler flags indicating optimizations for mathematical operations are of fundamental relevance, providing a significant gain in the case of the studied code.

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 the runtime.