Computação Paralela - WA2

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, multiple threads, OpenMP, parallelization**

I. Introduction

The main goal of this second assignment was to evaluate the benefits of optimization of code with multiple threads by making use of the OpenMP tool to parallelize the code execution of the *MD* code.

We started off by analysing wich potential OpenMP instructions we could use to improve the parallelization based on previous experiments taken from the pratical classes of the course. Next we looked at the various *loops* contained within the *Potential* and *computeAccelerations* functions (wich were the ones that demanded more execution power in the WA1), and started testing various *pragma* instructions and analysing the performnce results with *perf*. We alternated the number of running threads and made use of a script file , *test.sh*, to test the code for each scenario.

One major change from the previous assignment was that the static number of particles, N, increased from 2160 to 5000. The obtained results were analysed and compared with the expected results.

II. 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).

III. OpenMP Optimizations and Results

In this section we will enumerate what OpenMP instructions we used in the *Potential* and *computeAccelerations* functions and where we used them, followed by a comparison between the results obtained from running the script file *test.sh*.

A. OpenMP Optimizations

1) Potential():

- *pragma omp parallel for reduction(+:Pot)*
  parallel for: This tells the compiler to parallelize the following for loop. Each iteration of the loop will be executed by a different thread.
  reduction(+:Pot): This is a clause that's used to handle a common issue in parallel programming. When multiple threads are updating the same variable, it can lead to race conditions, where the final value of the variable depends on the order in which the threads are executed. The reduction clause avoids this by creating a local copy of the variable for each thread, and then combining them in a specified way (in this case, addition) after all the

threads have finished.

- *pragma omp simd reduction(+:r2)*
  simd: This tells the compiler to vectorize the following loop. Vectorization is a process where multiple iterations of the loop are combined into a single iteration, and then executed simultaneously using SIMD instructions. This can significantly speed up the loop if the processor supports SIMD operations, wich is the case.
  reduction(+:r2): Equivalent to the *reduction(+:Pot)* from before, this clause avoids having multiple threads updating the same variable by way of creating a local copy of the variable for each thread, and then combining them in a specified way (in this case, addition) after all the threads have finished.

2) computeAccelerations():

- *pragma omp parallel for private(j, f, rSqd, rij)*
  parallel for: This tells the compiler to parallelize the following for loop. Each iteration of the loop will be executed by a different thread.
  private(j, f, rSqd, rij): This is a clause that's used to specify that the variables j, f, rSqd, and rij should be private to each thread. This means that each thread will have its own copy of these variables, and changes made to these variables in one thread will not affect their values in other threads. This can be useful to avoid race conditions, where the final value of a variable depends on the order in which the threads are executed.
  So, in summary, this directive will parallelize the following for loop, with each thread having its own private copy of the j, f, rSqd, and rij variables.

B. Results

In order to test how the performance variated for different numbers of running threads, we used a script file, *test.sh*, wich we executed multiple times to determine what the average execution time was for each scenario. We ran one thread, then two, four, eight, sixteen, thirty-two and finally forty.

It's worth mentioning that the output of the MD program is not altered in any way in both *MDseq* and *MDpar*.

TABLE I
TABLE WITH RESULTS OF *time.sh* FOR NUMBER OF RUNNING THREADS

#Threads	Real	User	Sys
1	0 m 47.206 s	0 m 47.195 s	0 m 0.002 s
2	0 m 35.577 s	0 m 48.813 s	0 m 0.013 s
4	0 m 21.742 s	0 m 53.458 s	0 m 0.017 s
8	0 m 12.925 s	1 m 4.306 s	0 m 0.030 s
16	0 m 8.030 s	1 m 27.156 s	0 m 0.050 s
32	0 m 6.577 s	2 m 29.346 s	0 m 0.084 s
40	0 m 6.898 s	3 m 26.431 s	0 m 0.103 s

The results from other tests are in the annexes.

Analysing the results, it's easy to determine that the best time came from running the parallelized version of MD with 32 threads, at an average of 6.52 seconds.

C. Scalability analisys

In parallel applications it is advised to make a graph of scalability analisys to analyse the speedup of the parallelized version of the code versus the single thread version. This way we can compare for each number of threads what is the ideal gain versus the real gain we obtained.

The best time obtained while running the sequencial version of MD was 50.37 seconds (see annexes). We will use this value to calculate the speedup for each number of threads using the following formula:

$$speedup = \frac{bestSeqTime}{parExecTime}$$

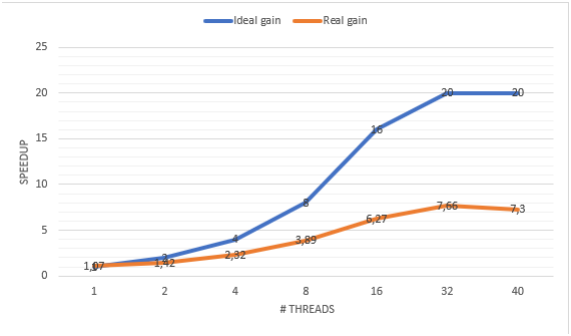

Fig. 1. Graph with speedup

TABLE II
TABLE WITH SPEEDUP RESULTS

#Threads	Speedup
1	1,07
2	1,42
4	2,32
8	3,89
16	6,27
32	7,66
40	7,30

IV. CONCLUSION

In conclusion, this assignment aimed to assess the advantages of optimizing code through multithreading using the OpenMP tool for parallelization in the context of the MD code execution. The process involved identifying potential OpenMP instructions, analyzing loops within key functions, and experimenting with pragma instructions while varying the number of threads. Notably, the assignment incorporated a significant increase in the static number of particles from 2160 to 5000 compared to the previous assignment. The results were thoroughly examined and compared against expectations, providing valuable insights into the performance implications of code optimization with multiple threads.

V. Annexes

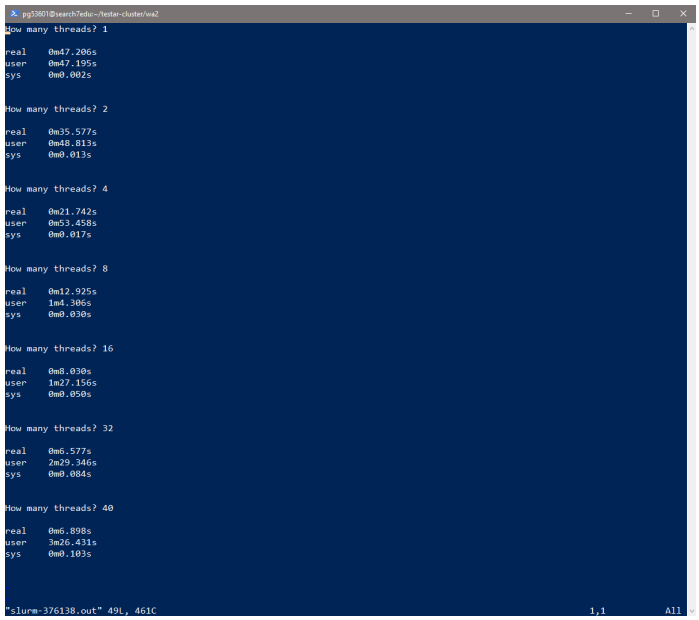

Fig. 2. Result 1 from *test.sh*

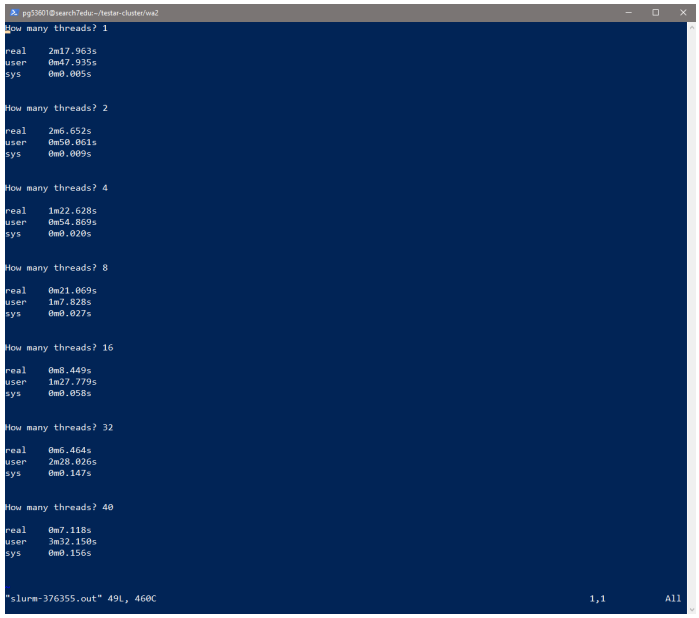

Fig. 3. Result 2 from *test.sh*

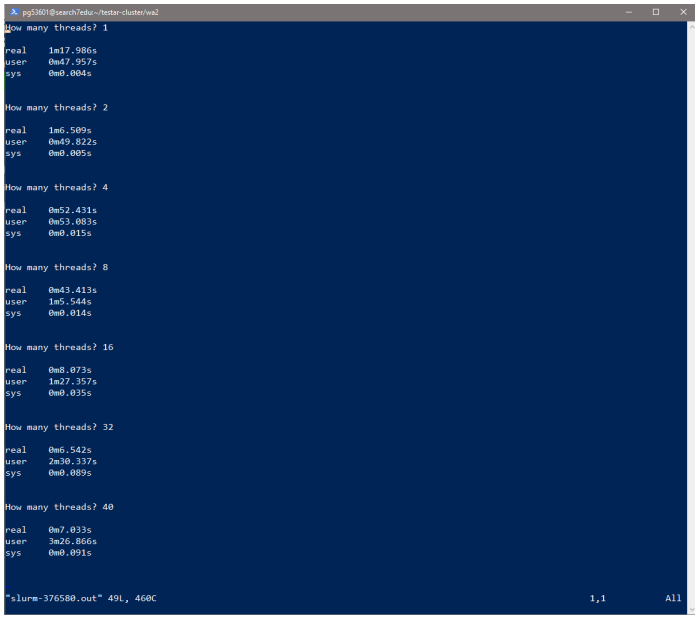

Fig. 4. Result 3 from *test.sh*

```
 AVERAGE TEMPERATURE (K):                      131.45506

 AVERAGE PRESSURE  (Pa):                  131001228.45076

 PV/nT (J * mol^-1 K^-1):                      28.47279

 PERCENT ERROR of pV/nT AND GAS CONSTANT:     242.44905

 THE COMPRESSIBILITY (unitless):               3.42449

 TOTAL VOLUME (m^3):                      2.37220e-25

 NUMBER OF PARTICLES (unitless):          5000
Time taken: 50.37s

 Performance counter stats for 'make runseq':

     189120066870      instructions:u            #    1,51  insn per cycle
     125457928568      cycles:u

    110,557909561 seconds time elapsed

     50,376665000 seconds user
      0,005077000 seconds sys
```

Fig. 5. Result from sequencial version of MD