Work Assignment CP - Phase II

1st Ivo Miguel Alves Ribeiro
Universidade do Minho
pg53886
V.N.Famalicão, Portugal
pg53886@alunos.uminho.pt

2nd Diogo Luís Almeida Costa
Universidade do Minho
pg53783
Trofa, Portugal
pg53783@alunos.uminho.pt

I. Introdução

In this report, we will analyse the development and results obtained in this second part of the practical work, which focuses on exploring shared memory parallelism, specifically using the OpenMP framework, with the overall goal of enhancing the overall runtime.

II. Identificação dos hot-spots

In this initial phase, we continued the work carried out in the first part of this project, focusing solely on the sequential version of our project. We identified code blocks with high computation time and made efforts to eliminate or reduce their computational intensity. By using gprof, we observed

```
index % time    self  children    called     name
                31.12    0.00     201/201         VelocityVerlet(double, int, _IO_FILE*) [2]
[1]    100.0   31.12    0.00     201         computeAccelerations_plus_potential() [1]
-----------------------------------------------
                                              <spontaneous>
[2]    100.0    0.00   31.12                 VelocityVerlet(double, int, _IO_FILE*) [2]
                31.12    0.00     201/201         computeAccelerations_plus_potential() [1]
-----------------------------------------------
```

Figure 1. Profiling initial sequencial version

that these blocks consisted of the functions **computeAccelerations_plus_potential()** and **VelocityVerlet()**.To reduce computational intensity, we made the following changes:

In the**computeAccelerations_plus_potential()** function, we removed the multiplication by 8, which was performed in each iteration, to be done only once outside the loops.
In the **VelocityVerlet()** function, we replaced the two blocks of two for loops present in the function body (one before and one after calling the **clearAmatrix()** and **computeAccelerations_plus_potential()** functions) with a single for loop that iterates through all the values of the arrays.
Additionally, we made changes in the for loop in the main function, eliminating the calculations of the variables **gc** and **Z**, and simplified the **MeanSquaredVelocity_and_Kinetic()** function by moving calculations outside the loop, performing them only once.

```
index % time    self  children    called     name
                29.91    0.00     201/201         VelocityVerlet(double) [2]
[1]    100.0   29.91    0.00     201         computeAccelerations_plus_potential() [1]
-----------------------------------------------
                                              <spontaneous>
[2]    100.0    0.00   29.91                 VelocityVerlet(double) [2]
                29.91    0.00     201/201         computeAccelerations_plus_potential() [1]
-----------------------------------------------
```

Figure 2. Profiling final sequencial version

III. Analyse the alternatives to explore parallelism

After the initial changes, we found that the points where we could explore and benefit from parallelism were the three functions within the for loop of the main() function, which correspond to higher computational intensity:

A. computeAccelerations_plus_potential()

This function, being the most computationally expensive in our program, led us to believe that introducing parallelism to it would result in a general improvement in program execution. We began by exploring parallelism using **#pragma omp parallel for** with synchronization using **#pragma omp atomic** for variables shared among different threads. While there was an improvement in runtime, we noticed that the model **#pragma omp parallel for schedule(runtime)** that we learned in classes resulted in better results. By employing the same approach and ensuring that variables accessed by multiple threads were consistently synchronized with **#pragma omp parallel for schedule(runtime) reduction(+:PE,a[:VECSIZE])**, we observed a substantial improvement. This is attributed to the fact that **schedule(runtime)** allows the scheduling type and chunk size to be determined at runtime, providing flexibility. The decision can be made dynamically based on the execution environment and input parameters.

B. VelocityVerlet()

As this function has two for loops,iterating over all positions of the arrays, we decided to test parallelism at these two points using **#pragma omp parallel for schedule(runtime)**.

C. MeanSquaredVelocity_and_Kinetic()

Finally, even though this function does not appear in the profiling-generated report, upon initial observation, we think that it would be beneficial to explore parallelism in it. This is because it is called, like the other two functions mentioned above, in each iteration of the **main() loop**, also taking advantage of the benefits of **#pragma omp parallel for schedule(runtime)**.

IV. selection and analyse the approach

As expected and as mentioned above, there was an improvement in the execution time by exploiting parallelism in the **computeAccelerations_plus_potential()** function. Given this

fact, and in an attempt to further enhance the execution time, we explored parallelism in other hotspots without giving up parallelism in this specific function. It's worth noting that the calculation of speedup, according to Amdahl's Law, was based on the execution time of the sequential model, with the value of 42.81 ms obtained as an average in tests of the sequential version without the optimizations mentioned above. The high speedup values, almost close to the expected ones, and all the values considered in the table with the execution time are the result of an average of 3 tests on the cluster.

Threads	5	10	15	20	25
Time(sec)	9,029	4,443	3,043	2,418	2,452
Speedup	4,741	9,635	14,065	17,702	17,459
Threads	30	32	35	38	40
Time(sec)	2,374	2,356	2,316	2,295	2,271
Speedup	18,030	18,165	18,484	18,648	18,850

Table I
RESULTS EXPLORING PARALLELISM IN THE FUNCTION COMPUTEACCELERATIONS_PLUS_POTENTIAL()

Threads	5	10	15	20	25
Time(sec)	9,265	4,491	3,417	2,492	2,453
Speedup	4,620	9,532	12,526	17,176	17,449
Threads	30	32	35	38	40
Time(sec)	2,379	2,365	2,318	2,619	2,263
Speedup	17,992	18,101	18,468	16,343	18,911

Table II
RESULTS EXPLORING PARALLELISM IN THE FUNCTIONS COMPUTEACCELERATIONS_PLUS_POTENTIAL() AND MEANSQUAREDVELOCITY_AND_KINETIC()

Threads	5	10	15	20	25
Time(sec)	9,602	4,905	3,556	2,875	2,891
Speedup	4,458	8,726	12,036	14,888	14,808
Threads	30	32	35	38	40
Time(sec)	2,748	2,704	2,731	2,602	2,539
Speedup	15,576	15,828	15,675	16,448	16,858

Table III
RESULTS EXPLORING PARALLELISM IN THE FUNCTIONS COMPUTEACCELERATIONS_PLUS_POTENTIAL() AND MEANSQUAREDVELOCITY_AND_KINETIC() AND VELOCITYVERLET()

Threads	5	10	15	20	25
Time(sec)	9,374	5,055	3,579	2,945	2,866
Speedup	4,566	8,467	11,959	14,534	14,937
Threads	30	32	35	38	40
Time(sec)	2,752	2,734	2,756	2,597	2,577
Speedup	15,555	15,658	15,533	16,480	16,610

Table IV
RESULTS EXPLORING PARALLELISM IN THE FUNCTIONS COMPUTEACCELERATIONS_PLUS_POTENTIAL() AND VELOCITYVERLET()

V. MEASURE AND DISCUSS THE PERFORMANCE OF THE PROPOSED SOLUTION

Analyzing the combinations described above in the tables, we verify that adding pragmas to all cycles in functions with higher computational power may not be useful, but we can observe a pattern. This pattern is that the lower the computational power within the cycle, the less need there is to explore parallelism. Considering this reflection,we identified two alternatives. One of them is to attempt to increase the computational power of each cycle by iterating in increments of 3, 6, or even 12 units per cycle and test the behavior of parallelism for these cases. The other option is to further reduce the computational power of these cycles and not explore parallelism in these areas. We have chosen to explore the first alternative by iterating in increments of 12 units in the function **MeanSquaredVelocity_and_Kinetic()** and iterating in increments of 6 units in the cycles of the fuction **VelocityVerlet()** However, the outcome of this test did not give us better results in terms of time so we have decided to cancel this initiative. Finally, we tried the second alternative, focusing on simplifying the calculations in the functions that are called recursively and attempting to maximize memory accesses to gain advantages in terms of memory hierarchy. After a new analysis of the code, we decided to simplify the **VelocityVerlet()** function. In this function, we calculate and store half the value of the **dt** variable. Additionally, we calculate and store the value of the multiplication, which is performed twice in each iteration, so we can avoid redundant calculations. Moreover, we utilize the first cycle of this function to set all positions in the array of accelerations to zero, eliminating the need to call the **clearAMatrix()** function that served this purpose.With the goal of decreasing the computational power of **MeanSquaredVelocity_and_Kinetic()**, we modified the loop in this function to only accumulate the squared values of each position in the velocity array. The updates to the global variables **KE** and **mvs** are now performed only at the end of the loop.

VI. FINAL DISCUSS

Threads	1	4	8	12	14	18
Time(sec)	39,093	9,752	4,927	3,335	2,860	2,284
Speedup		4,008	7,933	11,719	13,668	17,113
Threads	22	26	28	32	36	40
Time(sec)	2,135	2,116	2,115	2,131	2,115	2,094
Speedup	18,304	18,475	18,483	18,345	18,478	18,669

Table V
RESULTS EXPLORING PARALLELISM ONLY ON FUNCTIONS COMPUTEACCELERATIONS_PLUS_POTENTIAL() WITH OPTIMIZATIONS ON OTHER FUNCTIONS

The table described above reflects the outcome of the best version of the code achieved by the group, which, in summary, is a more optimized version than the sequential version presented in the previous phase e usufruindo do paralelismo na funçao **computeAccelerations_plus_potential()** segundo o openMP programming model que melhor se adequa ao nosso código.

In practical terms, improving from 42.81 seconds to 39.093 seconds in the sequential version of the code, with parallel optimization resulting in an execution time of 2.094 seconds, represents highly positive outcomes. However, in theory and according to Amdahl's Law, we recognize that despite being close to the ultimate goal with the use of parallelism, we haven't yet achieved the optimal gain expected. In an ideal model, this gain would follow a linear function y=x until the value of y equals 20. After that point, it would be a linear function y=20, where y is the number of threads and x is the value of speedup.