Parallel Computing - WA2

1st José Carvalho
Departamento de Informática
Universidade do Minho
Braga, Portugal
PG53975@alunos.uminho.pt

2nd Miguel Silva
Departamento de Informática
Universidade do Minho
Braga, Portugal
PG54097@alunos.uminho.pt

***Abstract*—In this report we will explain the decisions we made for the second phase of the Parallel Computing group assignment. We will go in depth into the optimisations we managed to implement.**

***Index Terms*—Parallel Computing, C, Vectorization, SIMD, OpenMP**

I. Introduction

For the second stage of the assignment we had to improve the performance of our code through the usage of *openmp*. In our multiple attempts we ended up implementing *tasks*, but we also used normal threads in an alternative version. In this document we will go in detail into our implementation of the aforementioned features to increase performance, focusing on hot spots, parallelizable zones, and our results.

II. Hot Spots

The hot spots are the same as the ones we had on the previous phase. The functions *Potential* and *computeAccelerations* are the functions that take up the majority of our execution time.

This is due to the complexity of them, as *Potential* has a complexity of $O(n^2)$ and *computeAccelerations* has $O(n \log n)$, so our main goal is to reduce the impact that these functions have in the execution time.

III. Parallelizable zones

We found three zones that could be parallelized. The first two are on the functions referenced in II, that is because what ends up taking most of their execution time is the main outer cycle inside of them and we figured those cycles could be paralysed in both cases. The parallelization that can be applied here is one that splits the loop into blocks dividing the workload.

The third one is on the main function's loop cycle, that is because not all the operations inside of the loop are not dependent on each other. The cycle could be split into separate parallel workloads because the only part that requires sequential execution is the *VelocityVerlet* function, whilst the rest of the cycle's body is unnecessary for further iterations. Still, in this case we had to bear in mind that depending on how we implement the parallelization it can be more complex than on the other functions.

Of course, handling these situations can be a bit complex, since we must avoid data races and other issues. We explain in detail what we did to each of these parallel zones in section IV.

IV. Approach

Based on III we decided to use tasks instead of normal threads.

One reason behind it was that the parallel zones are inside functions, that will be called multiple times. If we did not use tasks, we would need to create all of the threads again every time we called those functions and that would have a huge impact on performance.

Another reason that made us use tasks, is related to the third parallel zone referenced at III, which is very difficult to paralyse without using tasks.

A. *computeAccelerations*

We will now explain what we did on *computeAccelerations*.

The main goal of this function is to update values of *Acceleration* array. These values are calculated based on the values inside of the Position array. The hot spot is the second loop, that appears inside this function. We can not simply use a *taskloop* at the outer loop or inner loop, because in both scenarios, we would cause a data race on the *Acceleration* array. With all this in mind, we thought of a solution that paralyses this hot spot cycle, without creating data races.

To avoid data races, we focused on replicating the *Acceleration* array. That meant that we created a matrix whose lines contain replications of the *Acceleration* array. The number of lines is associated with the number of threads created by *openmp* retrieved by the *omp_get_num_threads* function.

The first instructions from *computeAccelerations*, is to set all values from Acceleration array to 0, so we will need to set all the values on each position of the replica to 0.

So now we can divide the iterations of the main cycle. Each division works with a specific line of the matrix containing the replicas. Our only constrain is that we have to calculate the limits that each task will make. We tried to simplify that division, so all divisions will handle the same number of iterations, but we know that this is not the best division, since the complexity the loop complexity is $O(n \log n)$. This causes the first divisions to handle more work than the latter ones.

The next step is to sum all the values from each column of the replica matrix and associate it with the original Acceleration array.

Lastly, we have to free allocated memory for the replication matrix.

B. *Potential*

On the *Potential* function, it was easier to parallelize since we just had to resolve a simple data race caused by the *Pot* variable.

We decided to split the iteration of the main cycle into 2 different tasks. This division is done by creating 2 different variables and giving each division one of those independent variables. In the end we sum both variables and we get the final value of *Pot*, this emulates a reduce operation.

Like on *computeAccelerations*, all divisions will iterate the same number of iterations and since the complexity is $O(n^2)$ the work is split equality among the tasks.

No more tasks were created on this function, because we prioritised creating tasks on the *computeAccelerations* function and we do not want to overload the threads with tasks from *Potential*. We decided this because *computeAccelerations* takes longer than *Potential* and we will execute both at the same time. We explain how and why that is the case on IV-C.

C. *Main*

Lastly, lets explain our approach to third parallel zone referenced at III.

For this part of the code, we created a task that executes the independent parts of each loop iteration.

We worried about the preservation of the most recent values from the *Velocity* and *Positions* arrays and the variable *Press*. That was solved by using the primitive *first private*, that creates a copy of these arrays and variable with their most recent values.

Another concern is associated with data races for the variables *Tavg* and *Pavg*. These 2 variable are basically the sum of *Temp* and *Press* respectively. To solve these data races, we created an array that will store the values of each iteration of the cycle, and then we sum the values of the array, to have *Tavg*'s and *Pavg*'s final values correctly. This is similar to what we did on *computeAccelerations* but instead of a matrix we use an array emulating the behaviour of a reduce.

D. *Additional*

We did one more optimisation, but since it is not related with parallelization, we are just to going to mention it, but not explain it in detail.

We noticed that we could use the constructor *pragma omp simd*, to some parts of the code that can be vectorized. Those parts are the functions *PotentialDivision*, *PotentialMath* and *computeAccelerationsDivision*.

E. *Sum up*

On section VI, the figure 2 illustrates how we paralyse the code. Each function zone contains the global arrays / variables that are written / read inside them. All code not mentioned on the figure is sequential.

V. Results

To finalise this report we want to look at our results, analysing them and comparing them to theoretical values.

For starters, the time taken up by our sequential version of the code is 43.423 seconds, and as we increase the number of threads we get these results on our version with tasks:

TABLE I
EXECUTION TIMES AFTER PARALLELIZATION - TASK VERSION

#T	Time	#T	Time	#T	Time	#T	Time
5	7.027s	10	3.884s	15	2.821s	20	2.180s
25	2.748s	30	2.713s	35	2.563s	40	2.463s

In comparison, table II contains the times for a threaded version of our code, it is not as optimised and since it is not our main approach we will present less values.

TABLE II
EXECUTION TIMES AFTER PARALLELIZATION - THREAD VERSION

#T	Time	#T	Time	#T	Time	#T	Time
2	49.379s	10	37.580s	20	34.734s	40	33.324s

We created a graph using a python script that compares the values obtained for each case. The graph can be consulted on figure 1, and it is important to note that we could have also represented another line that showcased the gain obtained purely by focusing on the parallel sections which would be in between the theoretical and the task gain.

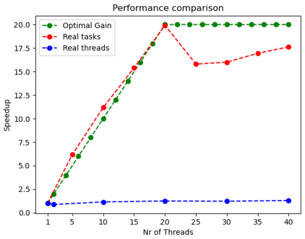

Fig. 1. Speedup obtained by using either tasks or threads

As the figure shows we attained a gain similar to the maximum theoretical gain for 20 threads with a speed up close to 20 (19.92). We even surpassed the theoretical gain for less threads, likely due to the usage of the *simd* constructor.

For more than twenty threads performance dropped, we believe that is caused by scalability not being perfect, task granularity increasing, and the effect of Amdahl's law that states that the maximum improvement in speed will always be determined by it's most significant bottleneck which limits the maximum gain at twenty trades.

VI. Attachments

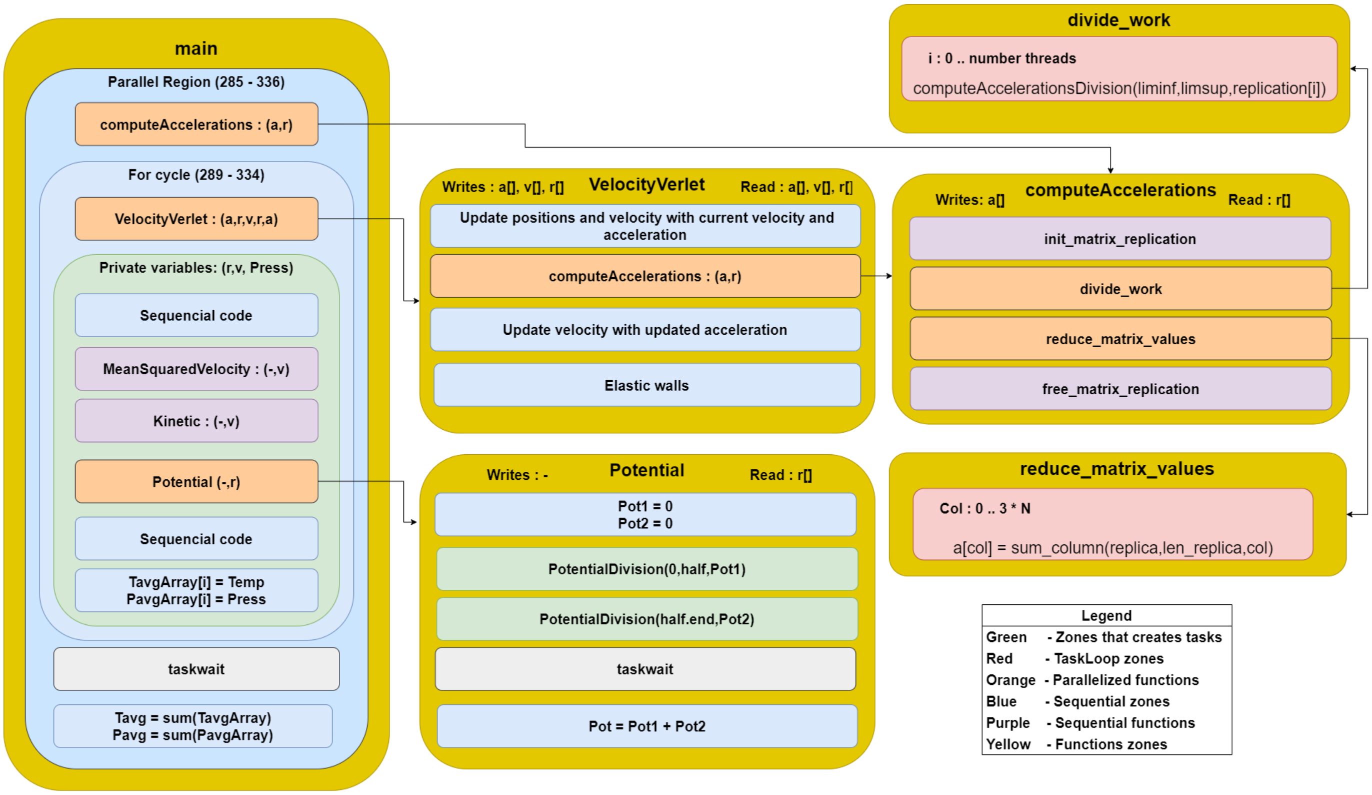

Fig. 2. Illustration of paralyse and sequential parts