INSTITUTO SUPERIOR TÉCNICO

MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

CPD

CPD OMP Report

Trabalho realizado por:	*Número:*
Duarte Correia	81225
José Nobre	84107
Rafael Forte	84174

Grupo: 10

**

05/04/2019

1 Parallelization

In this project we have three major phases. First it's necessary to calculate the center of mass of each cell, after that it's computed the force applied to each particle, and finally the program computes the new velocity and position for each particle. In the end it's is also computed the center of mass of the overall space.

These tree phase have to be done sequentially, since the second depends on data produced by the first, and the third depends on data coming from the second, this means we are gonna have to parallelize this zones separately, they cannot be computed at the same time (in parallel).

All of this zones can have huge amounts of data (depending on the inputs). We can divide this data per threads and apply the same function in order to parallelize and speed up the program, this is called data parallelism, and it's the way of parallelism we take advantage in this project.

This way each thread will have a portion of the data on its cache memory and we can program it to guarantee a good hit/miss ratio on the cache.

2 Decomposition

To parallelize our serial version, we have divided it into 4 different parallel zones. The first three are inside the serial for loop that corresponds to the number of timesteps to be accomplished by our simulation.

First we do a memset of the cells of our matrix, in order to reset all of them to 0 for the new calculations for the new timestep. After having the cells initialized, we do a parallel for loop which distributes the particles along the threads, each thread getting approximately the same amount of particles, and calculates the center of mass of each cell. Since the threads only compute a portion of all particles, and we don't know to which cell those particles belong before hand, all threads have to have a copy of all cells, in the end we use a reduction to sum together all those copies to get the final values of each cell.

After this is done it's necessary to divide the cells coordinates (x and y) for the total mass of the respective cell, this is only accessible after the reduction, so we do it in a parallel for just after the reduction is completed. This region does not benefits from the parallelism (since there are overheads involved in creating these regions) when using a low number of cells, but since it takes less than 1% of the total time with both versions, and if the number is increased there is gain in using the parallellization, it was decided to preserve it.

The next zone, still inside the timestep loop, is another parallel for, which is used to compute the gravitational force applied to each of the particles. As in the first parallel loop, it distributes the particles along the available threads. In this loop each thread has X number of particles, and only updates those X particles values (acceleration and position) so there is no need to create copies/do a reduction.

Finally, out of the timestep loop we do a parallel for with a reduction in order to calculate the total mass of the system of particles and the correspondent position of the center of mass.

3 Synchronization

As was already said in this report, our major parallel zones have data dependencies, this means some synchronization is necessary in order for the program to be accurate. Between each phase it's mandatory that all data has been computed, this means that at the end of each phase all the threads need to wait for the last one to finish before keeping executing the program. These barriers are placed at the end of each parallel for loop (automatically), ensuring that all the data needed in the next phase is ready to be used.

4 Load Balance

When programming in parallel we should take in to account the workload each thread receives along the program so that there aren't threads without work to do (wasting time and resources) while others are performing computations. In order to avoid this all threads should have an equal amount of work to perform.

In our project this load balancing is done at the parallel for loops. They divide the number of iterations of each for loop between the threads available, ensuring a good load balance (since all threads do the same operations the same number of times).

There are times where this division has some imbalances due to having a number of cycles that doesn't have modulo zero when divided by all the threads.

5 Performance Results

For the performance analysis of our program we ran it in a Intel Core i7-8700k with 6 cores and 12 threads. Some of the inconsistencies that we can observe on the charts between the 6 and 7 number of threads are due to hypertreading not being perfect, so the access to cache isn't the same has if it had 12 cores.

Looking at the charts below we can see that we have a nice speedup and efficiency with a low number of threads, that number goes down with a higher number of threads. This happens because we aren't working with real cores and the number of overheads to the number of threads also increases due to memory accesses, increasing the overall time of our simulation, since they share caches creating that displacement at the 7^{th} thread. The always decreasing values in efficiency are due to the parts that don't benefit from parallelism (and synchronization barrier) grow in relative time wasted compared to the parts that are parallelized following Amdahl's Law.

The values below are not from single runs of the program but averages of multiple tests (each test took ≈ 30 minutes for all the threads, same number for each thread).

To be noted that the loss of efficiency on the 12^{th} thread should be from being affected by other processes being run on the computer. Also for the test "./simpar_omp 2 5 10000 10" there is a gain in efficiency (≈ 1%) from 3 to 4 threads that should be due to a better cache hit/miss ratio gained by separating the data across the cores.

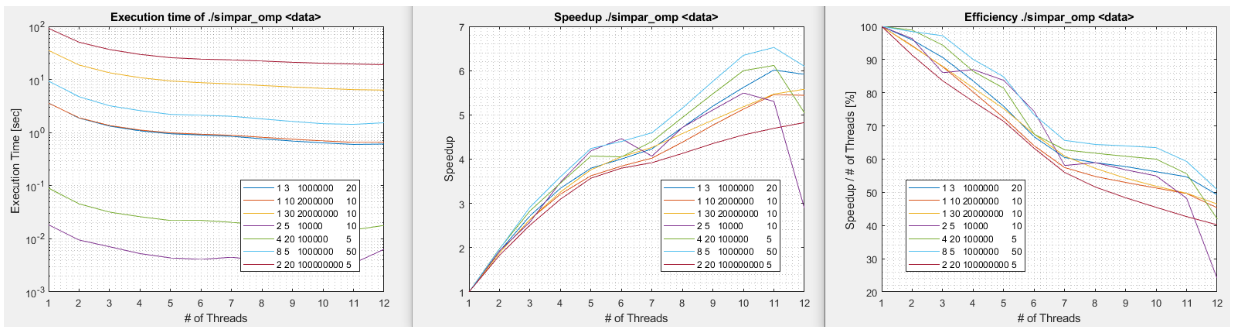

Figure 1: *Performance analysis of the parallelization made with a i7-8700k Intel processor (6 cores 2 threads each)*