Particles Simulation

Leonardo Epifânio nº 83496
Pedro Lopes nº 83540

Abstract—We describe the developed solution for a parallel implementation of a simulator of particles moving in free space. This report outlines the approach used for parallelization and decomposition, the synchronization concerns, and how load balancing was addressed. We then present the performance results and discuss them.

1 Introduction

The objective of the program is to simulate the forces applied to a set of particles and the respective acceleration, velocity and position of each one. The problem, is known as the n-body problem, and is computationally very expensive for a large number of particles n. This report describes and discusses the implemented solution with special attention to the idiosyncrasies of the problem parallelization. In order to evaluate the parallel version and place a benchmark, a serial version was made and then parallelized with OpenMP.

In section 2 we start by describing the structures used in the program and the phases of the algorithm to achieve the final output. The section 3 presents the used approach and the idiosyncrasies rooted in the problem. Then we refer specific optimizations made to code both for the sequential version and for the parallel version. After that, in section 5, we measured some metrics and evaluated the parallel version against the serial to draw the efficiency of the implementation. We then conclude in section 6.

2 Program Overview

2.1 Structures

coordinate - Represents a coordinate or a vector in the 2 dimensional space. It has components x and y.
cell - Represents a cell. It has the ***total mass sum*** of the particles in the cell and the coordinates of the ***center of mass***.
particle - Represents a particle. It has the ***position***, ***velocity***, ***mass*** and the ***cell*** where it is located.
adjacent cells - It's an auxiliary structure in the form of a matrix that saves the ***adjacent cells*** of each cell, as it's constant throughout the program's life.

2.2 Phases

1) **Initialization** - Includes the allocation of memory and the initialization of the set of particles and the set of cells.
2) **Center of Mass Calculation** - The set of particles is iterated. In each iteration we access the ***cell*** to which the indexed ***particle*** belongs to, add into its ***total mass sum*** the ***mass*** of the particle, and also partially calculate the coordinates of the ***center of mass*** of this ***cell***. After iterating the set of particles, the set of cells is iterated and the definite coordinates of the ***center of mass*** are calculated.
3) **New Position Calculation** - The set of particles is iterated again. In each iteration we calculate the sum of the forces exerted by the corresponding ***cell*** and the adjacent cells. With the total force, we calculate the new acceleration, velocity and position of the particle.
4) **Overall Center of Mass Calculation** - After all steps are calculated, a new iteration through the particles is made in order to calculate the ***overall center of mass***.
5) **Print Results and Free Resources** - The position of the first particle and the ***center of mass*** calculated in the previous phase are printed to the console. All the memory allocated in phase 1 for the particles, cells and auxiliary structures is freed.

A Step consists in the execution of phase 2 and 3.

3 OpenMP Version

3.1 The Approach to Parallelization

In our OpenMP solution, we used data decomposition for the parallel regions in all phases. A simplistic representation can be seen in figure 1.

3.1.1 Phase 1

Phase 1 rises a problem since the particles' initialization cannot be parallelized due to the need of deterministic output from the *random* function, however we could allocate and initialize the cell structures, while this is done. To accomplish this, we could split these two functions into parallel sections. But as we analyze the time it takes for each of these two tasks to complete, we notice that the time to initialize the cell structures is extremely small (< 20 μs for 8 threads and 400000 cells), so we decided not to parallelize this region.

3.1.2 Phase 2

In phase 2, the access to cells inside the particle iteration loop needs to be atomic to prevent race conditions. In order to avoid the overhead caused by the synchronization of threads in this loop, we devised a new structure (also initialized in phase 1) called ***cells_threads***, which is a vector of cell grids, containing a cell grid for each thread, for the maximum number of threads (computed at the start of the program). With this structure, the loop is split up among the threads, and each thread computes their chunk of the loop using the cell grid allocated to that thread. This way, we remove the need for synchronization. At the end of this phase, another parallel loop is computed to aggregate the results for each cell. This process is a **reduction**, which, in this case, we needed to manually implement, since OpenMP cannot do that for us. In comparison to the approach of using synchronization, the overhead caused by our approach will be smaller for most cases, since the extra operations needed are simple and are done in a loop of size ***number of cells*** $*$ (***number of threads*** $-$ ***1***). The number of cells and threads will usually be relatively small, hence using reduction is the better approach.

3.1.3 Phase 3

In phase 3, we only parallelize the outer *for* loop, since the inner one has critical regions. In this parallel region there is a small imbalance overhead (representing 1.48% of all CPU time), since some threads will not execute part of the operations of the inner *for* loop. It skips to next iteration if the condition ***distance*** < ***EPSLON*** is not met. We tried to fix this problem with scheduling, but the overhead that came from using it overwhelmed the benefits.

3.1.4 Phase 4

Phase 4 presents a single parallel *for* loop, with a sum reduction applied to the shared variables, proving to be more efficient than using atomic operations, avoiding the synchronization overhead.

3.1.5 Phase 5

In phase 5, the freeing of the cell structures is not parallelized, since, by looking into ompp analysis results, we can see that it doesn't compensate the thread management overhead.

4 Optimizations

All the optimizations explained here are in both serial and OpenMP versions of the code. The OpenMP version is exactly the same code as the serial (except for the manual reduction explained in 3.1.2) with the additional ***pragma*** directives).

4.1 Cache and Memory Management

The current cache architecture takes advantage of the principle of locality, and so, since the access to a memory position fetches a whole memory block into the cache, a way of optimizing the hit rate is to maximize the usage of contiguous memory. In our program, the allocation is made with this in mind. All the matrices in the program allocate all of their cells' positions with a single call to *malloc()*. Afterwards, another array is allocated for each matrix, holding pointers to the first position of each of its rows.

Cache size is limited, so our structures only strictly hold the necessary information. This was one of the reasons we created the **adjacent cells** structure explained in section 2.1, instead of having these as an extra field in the cell structure. This structure is **readonly** throughout the program, which means there's no problems in sharing it between the cores' caches.

All the code was optimized to make the least memory accesses possible, since this is a time consuming task, and whenever this wasn't possible, we minimized the writes to avoid possible false sharing. False sharing was mitigated by performing writes in memory positions which are distant from each other.

Zeroing memory was another concern. To achieve this we used *calloc* (when allocating) and *memset*, this avoid loops and ends up being faster than the alternatives.

4.2 Instructions and Pipeline

Our goal was to minimize the number of instructions executed. Less instructions equals to less time of execution. Also, different instructions take different time to execute. For example, there's a clear difference when working with doubles and integers, so we just perform instructions with doubles when there's a need for it.

To decrease the number instructions, we avoid (almost completely) the normal indexation of matrices, by working with pointers. When there's the need to index the same position of a matrix several times, it's faster to get the pointer the first time, and just use it in the following statements, instead of calculating the position each time.

The square root function call, *sqrt()*, takes a considerable time to execute. In phase 3, we perform the operations with the squared distance, and only calculate the square root when there's a need to calculate the force.

For the pipeline, also regarding possible compilation optimizations that *gcc* might do, we tried to re-arrange the instructions to avoid data dependencies and maintain the pipeline full, without hazards.

5 Evaluation

The tests were performed in a machine running GNU/Linux with 16GB of RAM and Intel® Core™ i5-7500CPU @ 3.40GHz Quad Core (1 thread per core) with cache L1 (32Kb), L2 (256Kb) and L3 (6144Kb).

We used the examples that were made public in the project statement, plus the course's Fénix page, numbering them from 1 to 11 (starting with the ones presented in the statement). To study results, we chose tests 3, 4, 10 and 11 as they are the most time consuming. To get the times, in the OpenMP version, we used the *omp_get_wtime()* function, and, in the serial version, we used *clock()* from the *time.h* lib. Each test were run 4 times and we got the mean and the variation. The variation was negligible ($< 10^{-5}$).

We first evaluated the performance of the entire program, as shown in figures 2 and 3. We noticed that the serial version runs roughly 2-3 times faster, comparing with the results provided by the professor. The ideal speedup would've been equal to the number of threads, but achieving this is very difficult. However, the speedup we got is not far from the ideal. The problem here is, there's a part of the program that cannot be parallelized, as shown in section 3.1.1, taking up a considerable amount of the execution time (in test 11, it takes 23.4%). Because of this, we decided to also evaluate the program without the initialization and the finalization (freeing of resources) regions, and there was a huge improvement in speedup as we can see in figures 4 and 5. We noticed a peculiarity in tests 4 and 11 with OMP-2, where the speedup is greater than the number of threads, which might have been due to optimal conditions, regarding cache access.

With 8 threads, the times obtained are practically the same as with 4, which is what we expected, since the computer we ran the program on, only has 4 physical cores, meaning, it can run, at maximum, 4 threads in parallel.

By looking at the results, we inferred that there might be a correlation between the number of particles and the obtained speedup. To test this hypothesis, we evaluated the speedup as a function of the number of particles, testing the 4 thread OpenMP version against the serial version. The results are displayed in figure 6, where we can see that the speedup, in average (with a negligible variation of 0.0002), remains constant from 10M particles to 200M particles, its mean value being, approximately, 3.254.

6 Conclusion

Despite the fact that the n-body problem is hard to parallelize, since all computations depend on the previous result of all other computations, we managed to create a solution that achieves good results in parallelization, especially when we only consider the phases of the program that compose the problem itself (excluding the initialization and finalization), even achieving ideal speedups in two of the used test cases, and since the speedup remains constant as the number of particles goes up, we can also claim that the solution has good scalability.

APPENDIX

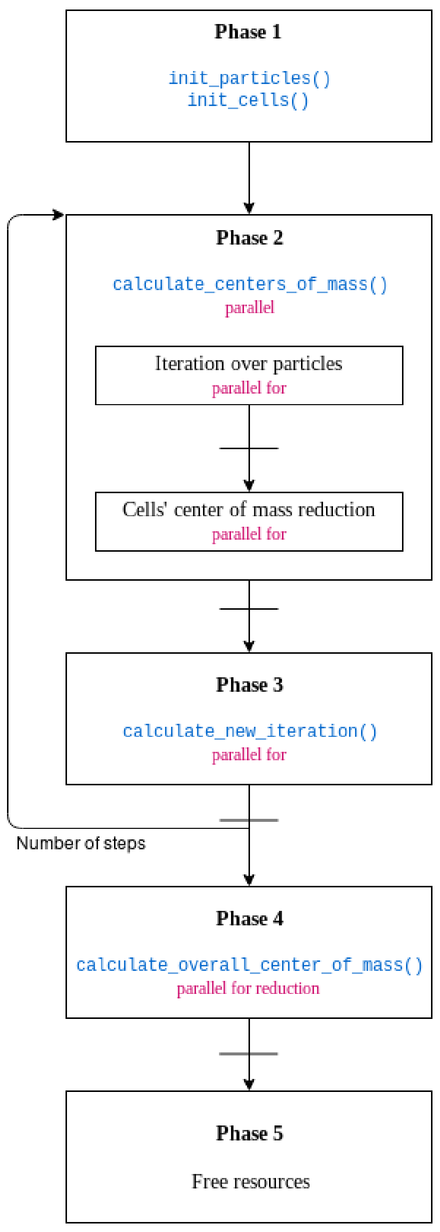
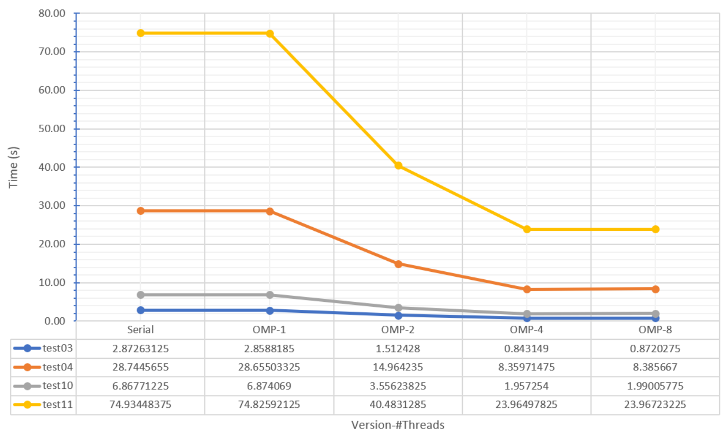
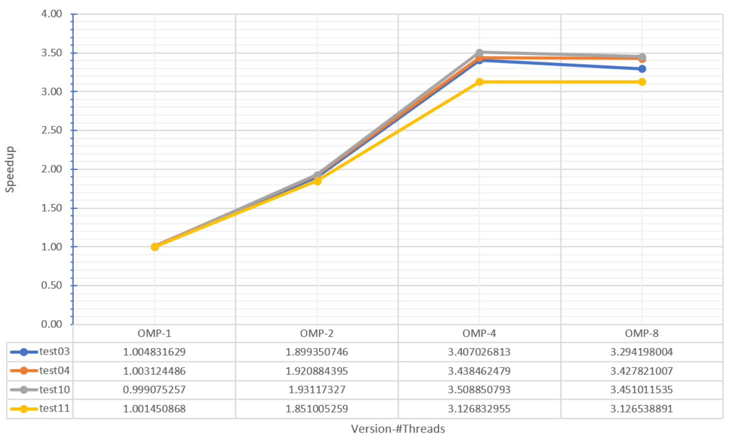


Figure 1. Program flow.



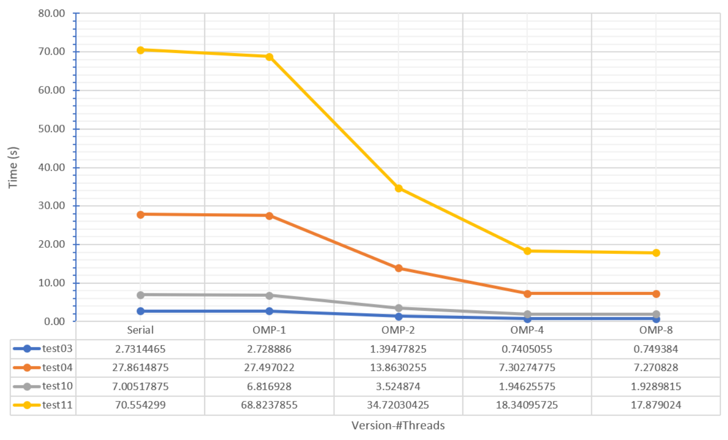
	Serial	OMP-1	OMP-2	OMP-4	OMP-8
test03	2.87263125	2.8588185	1.512428	0.843149	0.8720275
test04	28.7445655	28.65503325	14.964235	8.35971475	8.385667
test10	6.86771225	6.874069	3.55623825	1.957254	1.99005775
test11	74.93448375	74.82592125	40.4831285	23.96497825	23.96723225

Figure 2. Execution times including initialization.



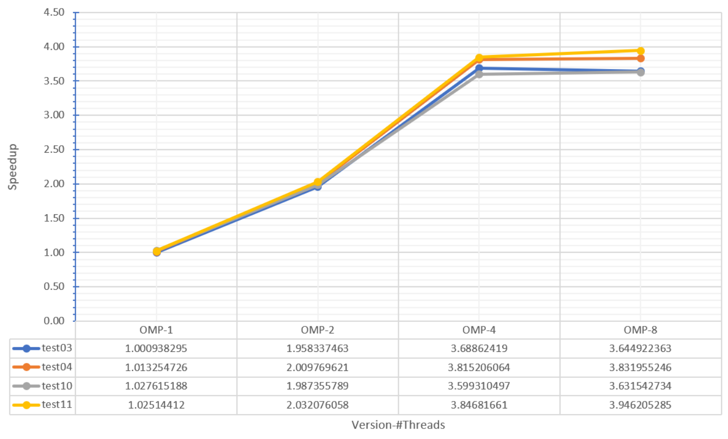
	OMP-1	OMP-2	OMP-4	OMP-8
test03	1.004831629	1.899350746	3.407026813	3.294198004
test04	1.003124486	1.920884395	3.438462479	3.427821007
test10	0.999075257	1.93117327	3.508850793	3.451011535
test11	1.001450868	1.851005259	3.126832955	3.126538891

Figure 3. Speedup including initialization.



	Serial	OMP-1	OMP-2	OMP-4	OMP-8
test03	2.7314465	2.728886	1.39477825	0.7405055	0.749384
test04	27.8614875	27.497022	13.8630255	7.30274775	7.270828
test10	7.00517875	6.816928	3.524874	1.94625575	1.9289815
test11	70.554299	68.8237855	34.72030425	18.34095725	17.879024

Figure 4. Execution times excluding initialization.



	OMP-1	OMP-2	OMP-4	OMP-8
test03	1.000938295	1.958337463	3.68862419	3.644922363
test04	1.013254726	2.009769621	3.815206064	3.831955246
test10	1.027615188	1.987355789	3.599310497	3.631542734
test11	1.02514412	2.032076058	3.84681661	3.946205285

Figure 5. Speedup excluding initialization.

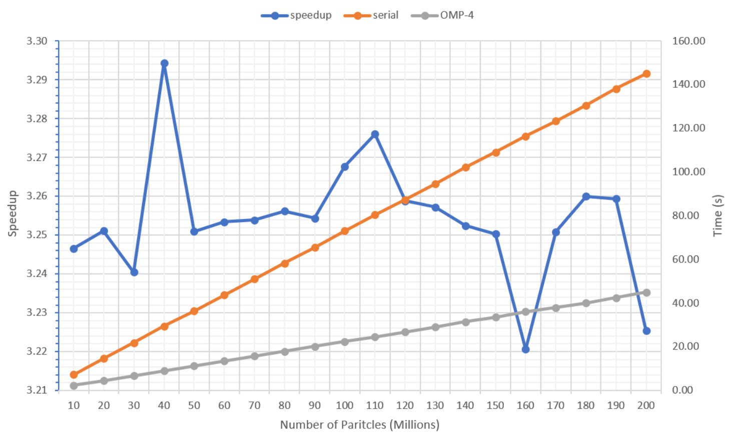


Figure 6. Speedup as a function of the number of particles.