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 it's, 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 and distribution. This report presents the combined version of MPI+OpenMP implementation of the problem.

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

2 Program Overview

2.1 Structures

We kept the structures used in the other implementation versions (Serial and OpenMP), adding some other auxiliary ones for handling and managing the buffering of the incoming and outgoing messages. These include an implementation of an array list (a.k.a. dynamic array).

2.2 Phases

1) **Initialization** - Includes the generation in one node (process with rank 0), and the distribution of the set of particles to the corresponding nodes. This distribution happens alongside the generation, as there are no guarantees that one node has enough memory to accommodate all the particles. Also, it is in this phase that the allocation, and initialization, of the local cells and other auxiliary structures is done.
2) **Center of Mass Calculation** - In each node, the local 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. After this, each node sends the results to its neighbourhood.
3) **New Position Calculation** - The local 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. According to the new position, the particle may or may not belong to the current node in the next iteration. If not, it is removed from the local set of particles and added to the buffer of the belonging process. In the end of this phase, the buffers are sent to its corresponding nodes and the node receives, from the other nodes, the particles which are added to the local set of particles.
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***. Each node calculates its local reduction and then a global reduction is made and finished by the process with rank 0.
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 by process 0. All the memory allocated in phase 1 for the particles, cells and auxiliary structures is freed. The *Finalize* routine is then called in all nodes.

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

3 MPI Version

3.1 The Approach to Parallelization

We approached parallelization by cell. Each process, at startup, is assigned a set of cells. We decided to go with this strategy because it scales better as the grid size increases. In the alternative, by particles, each process must hold the entire grid and do an *All to All* operation, and this becomes extremely expensive as the number of processes and grid size increases.

3.1.1 Phase 1 - Global

Phase 1 rises a problem since the particles' initialization cannot be parallelized due to the need of deterministic output from the *random* function. So, the initialization is made in one node (which is the node with rank 0), and each particle is sent to the corresponding process. This is made in batches, as the buffer for each possible process can accommodate a total of 10000 particles, and when this value is reached, process 0 sends the buffer, freeing it for more particles. In the end, all of the particles are in the corresponding processes.

3.1.2 Phase 2 - Local

In phase 2, in each process, the access to cells inside the particle iteration loop needs to be atomic to prevent race conditions. To solve this problem we use the directive of OpenMP atomic where the write to a memory position is atomic.

3.1.3 Phase 3 - Local

There is no possible way to parallelize Phase 3, the list containing the set of particles is modified as the iteration is made.

3.1.4 Phase 4 - Local

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 - Local

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.

3.2 Decomposition and Load Balancing

We decompose the cells matrix in a checker board pattern. This increases the scalability of the problem. Using approaches such as row wise or column wise, the number of processes is limited to the side of the grid, N, whilst using the checker board approach we can have up to N^2.

Given the number of processes, it decomposes the matrix in a way such as to maximize the use of the number of processes. Then, knowing the number of processors per line and the number of processes per column it is divided using the distributed approach of block decomposition where larger blocks are evenly distributed among the processes.

Load balancing is a concerned as there is a possible case where all the particles could end up in the same cell, overwhelming one node and not paralelizing at all. However, looking to the characteristics of this program in particular, the particles are well spread with high entropy balancing the load per cell. This is the result of the random function.

4 Optimizations

All the optimizations explained here are in also in serial and OpenMP versions of the code.

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 are 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 of 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 the RNL Cluster. We used the example with 200 million particles that was made public in the course's Fénix page, as the 300 millions could not run in one node (not enough memory). To get the times, we used *clock()* from the *time.h* lib.The obtained results are displayed in the Appendix. The speedup was made against the time of the serial version of the project which was 603.83 seconds.

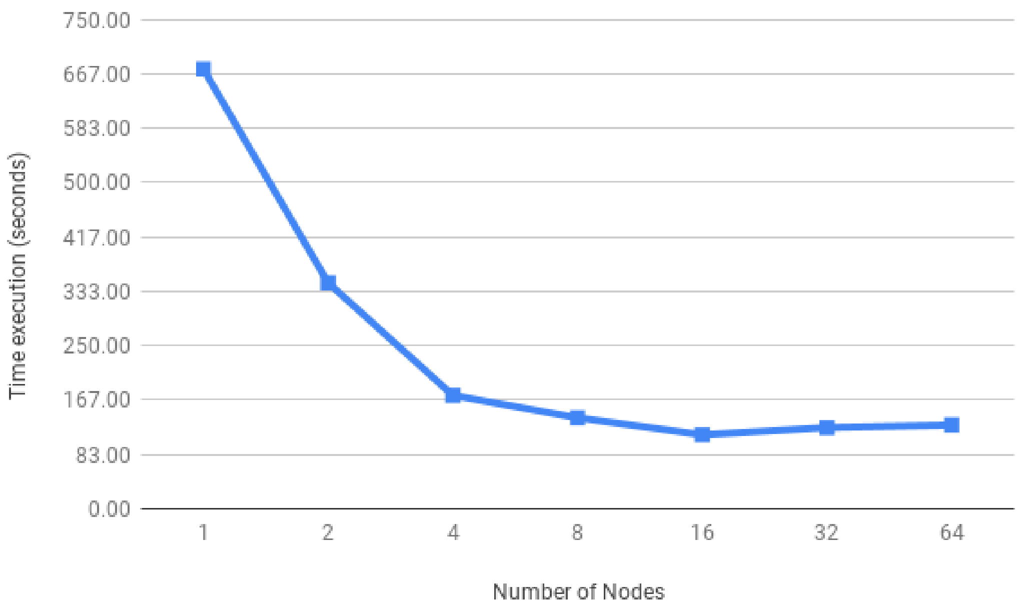
The performance results were a bit off from what we were expecting. We expected that the serial version would be faster than the MPI version with 1 node and the results show that. This happens because of the additional code of MPI routine and the extra calculations made. We expected a better performance when there are more processes, however, this does not happen. The main cause is the overhead in the communication. This makes it so that's only reasonable to use up to 8-16 processes. For 16 processes it improves in relation to 8, but not by a lot. For higher values, the communication starts to kick off and the times start to increase as the number as processes increase. This is not good at all, because it does not scale more.

The ideal speedup would be equal to the number of processes. It reaches its maximum at 16 threads (far from the ideal 16).

6 Conclusion

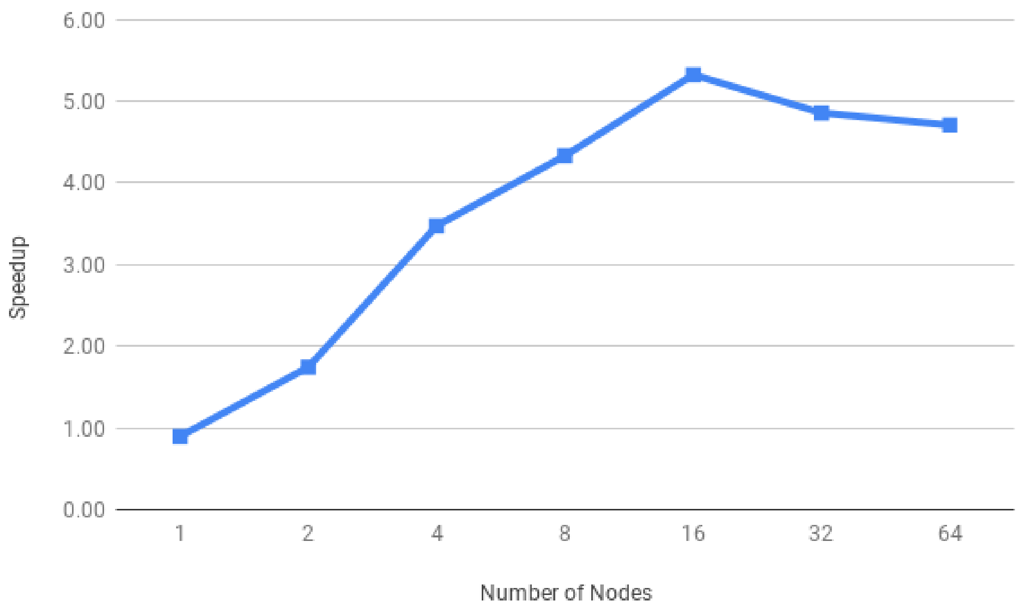
Despite the fact that the n-body problem is hard to parallelize, since all computations depend on the previous result of all other computations, we managed to create a solution that achieves good results in parallelization until a certain number of nodes, after that number, Amdahl's Law starts showing, and the overhead of the communication becomes the bottleneck of the system.

Appendix



#Nodes	1	2	4	8	16	32	64
Time (s)	673.85	346.40	173.85	139.39	113.43	124.38	128.28

Figure 1. Execution time.



#Nodes	1	2	4	8	16	32	64
Speedup (s)	0.8960	1.7432	3.4733	4.3319	5.3231	4.8549	4.7071

Figure 2. Speedup time.