DM818 Assignment 2

Parallelize Particle Simulation

Dan Sebastian Thrane
<dathr12@student.sdu.dk>
Lars Thomasen
<latho12@student.sdu.dk>

Winter 2015

Table of Contents

Chapter 1

Introduction

This reports documents the second mandatory assignment for the course DM818 Parallel Computing. In this assignment we were given a particle simulation which runs in $O(n^2)$ time, and were asked to code the following:

1. Change the implementation such that it runs in $O(n)$ time.
2. Parallelize the changed code using OpenMP/pThreads/MPI.

In order to measure proper performance gain, we need to optimise the serial algorithm first, and then parallelize it such that the comparison stays fair. It would be easy to show a massive performance gain if the algorithm for the parallel implementation is vastly superior to the one used for linear.

1.2 Work Load

The distribution of the workload has been equal, and pair programming in the imada terminal room have been the preferred method throughout this project.

Chapter 2

Making the implementation linear

2.1 The supplied algorithm

The given algorithm for the serial implementation (and the parallel versions) are as follows, clearly running in $O(n^2)$ runtime. This was unacceptable and thus a new algorithm has been developed.

```
for( int i = 0; i < n; i++ )
{
particles[i].ax = particles[i].ay = 0;
for (int j = 0; j < n; j++ )
{
     apply_force( particles[i], particles[j] );
    }
}
```

The code above shows the culprit, when applying force to all particles, for each particle, all particles are iterated again to do so. Since a particle can only be influenced by nearby particles, this is extreme expensive.

2.2 The new algorithm

The range for interaction for the particles was reduced to a smaller value, denoted as the "cutoff" value. This cutoff value was showcased as a grey border on a particle in the assignment.

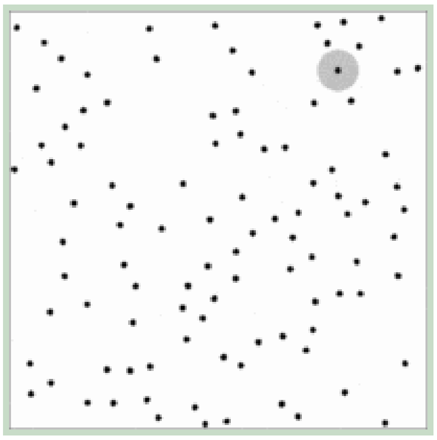


Figure 2.1: Snapshot of the GIF from the assignment showcasing the influence ring.

Having a reduced area of interaction, allows us to reduce the amount of particles influencing the amount of force needed to be applied to a given particle, thus we can reduce the second for-loop greatly.

In order to reduce the loop it is necessary to know which particles are within (or atleast close by) the particle we want to apply the force. To do this we needed to develop a data structure, to keep track of where the particles are positioned.

This data structure was not needed in the original solution since all particles were simply assumed to be within the cutoff range.

2.3 The data structure

In order to track the location of the particles, the coordinate system is divided into a grid, each coordinate holding all particles within a single cutoff range. It is then possible for a single particle, at a grid position, to find all particles in the surrounding grid positions, guaranteeing to find atleast all particles within its cutoff range.

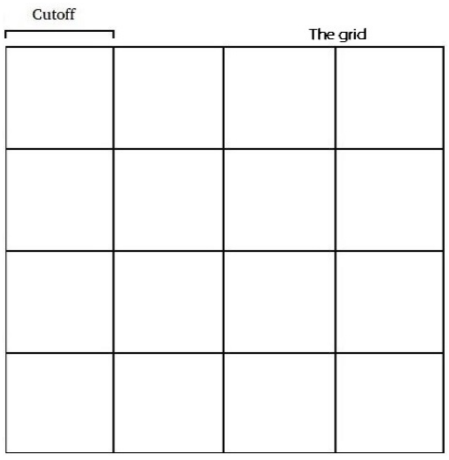


Figure 2.2: The grid structure, showcasing the size of a single position equals the cutoff size.

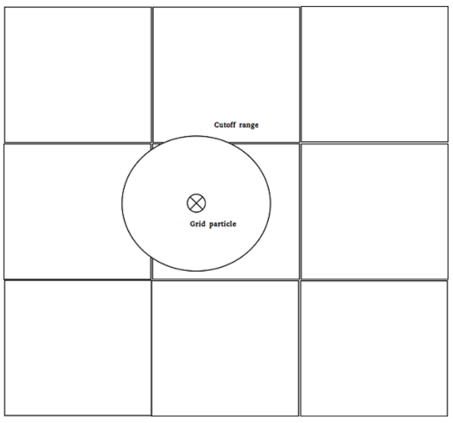


Figure 2.3: A particle located inside a grid positions, and its influence range.

Initially all particles are added to the data structure. It then has to be maintained whenever we move a particle, as the grid location could possibly change as the particle moves around.

2.4 The implementation

The implementation uses a separate file named `grid.cpp` and `grid.h`. These files holds the data structure and a series of helper methods that allows us to add, get and remove particles to the grid.

The grid itself is a vector of vectors with particles.

```
std::vector<std::vector<particle_t *> > grid;
```

The coordinate positions consists of doubles, while the grid positions are mapped to integers. To overcome this the coordinate positions are simply converted using the following formulae, which simply moves the decimal two places to the right and casting to an integer. Thereby giving us a consistent mapping of doubles to integers:

$$\frac{\text{double value}}{0.01}$$

Whenever we move a particle, we simply remove the particle from the grid completely, and then add it back in. This way the old position is removed, and the new position is calculated from the particles new position when added again.

This method assumes the particle always switches position in the grid, it could be calculated whether this actually is the case, but this was not done. Consequence to this method is that some unnecessary calculations might be performed whenever a particle do not move, since it would be removed and added back to the same grid position. The correctness of the algorithm is however not affected.

2.5 The graphs

Below is a graph plotted for different times (see appendix A) showcasing how the $O(n^2)$ and $O(n)$ algorithms do versus each other. It is easy to see that the $O(n)$ is indeed linear. Note that bigger sizes did not terminate in reasonable time for the $O(n^2)$ algorithm.

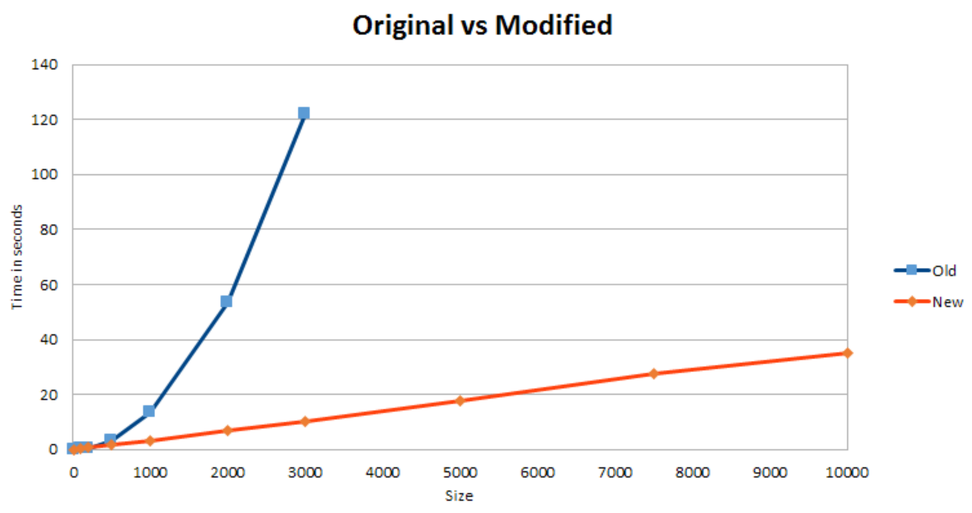


Figure 2.4: The original $O(n^2)$ algorithm (blue) versus the new $O(n)$ algorithm (red).

Chapter 3

Making the implementation parallel

Chapter 4

Discussion on pthreads, OpenMP, and MPI

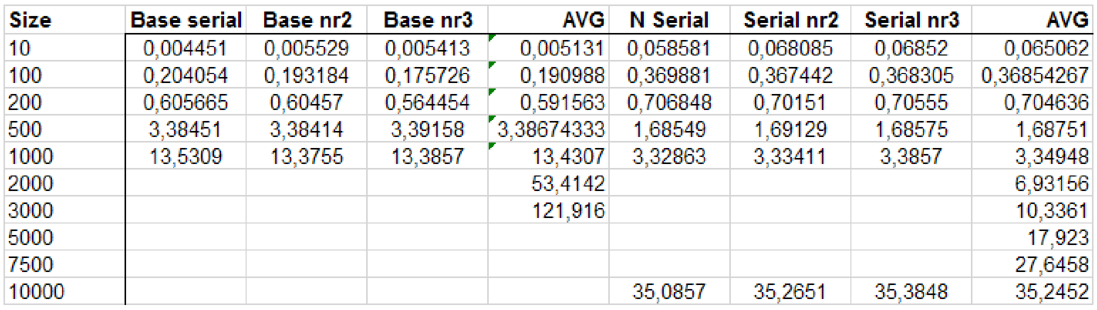
Chapter 5

Conclusion

Chapter 6

Appendix

6.1 Serial algorithm plotting data



Size	Base serial	Base nr2	Base nr3	AVG	N Serial	Serial nr2	Serial nr3	AVG
10	0,004451	0,005529	0,005413	0,005131	0,058581	0,068085	0,06852	0,065062
100	0,204054	0,193184	0,175726	0,190988	0,369881	0,367442	0,368305	0,36854267
200	0,605665	0,60457	0,564454	0,591563	0,706848	0,70151	0,70555	0,704636
500	3,38451	3,38414	3,39158	3,38674333	1,68549	1,69129	1,68575	1,68751
1000	13,5309	13,3755	13,3857	13,4307	3,32863	3,33411	3,3857	3,34948
2000				53,4142				6,93156
3000				121,916				10,3361
5000								17,923
7500								27,6458
10000					35,0857	35,2651	35,3848	35,2452

Figure 6.1: Plotting data for the linear runtimes.