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 MPI.

In order to measure proper performance gain, we need to optimise the serial algorithm first, and then parallelize that such that the comparison stays fair.

1.2 Work Load

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

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 showcasing the 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, each time all particles are iterated again to do so.

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.

Having a reduced area of interaction, allows us to reduced 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 an 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.

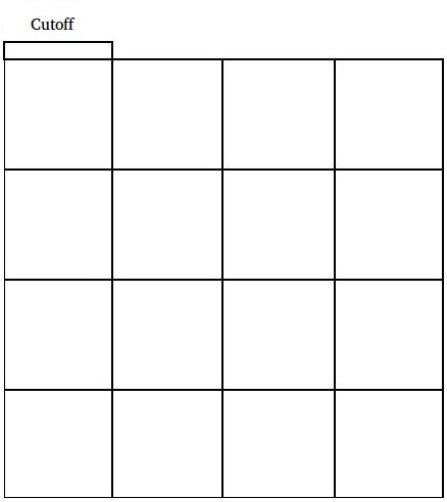

Figure 2.1: The grid structure, showcasing the size of a single position equals the cutoff size.

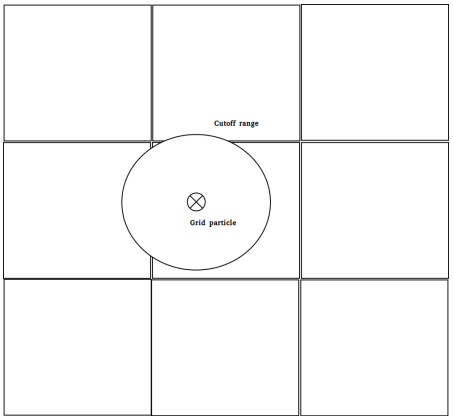

Figure 2.2: A particle located inside a grid positions, and its influence range.

This algortihm will include more particles than needed, but an upper bound for how many particles that can be within the 9 grid positions would still be far from n.

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 int:

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

Chapter 3

Making the implementation parallel

Chapter 4

Discussion on pthreads, OpenMP, and MPI

Chapter 5

Conclusion