SYDDANSK UNIVERSITET

DM818 ASSIGNMENT 2

Parallel 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 either OpenMP, PThreads, or MPI.

In order to measure proper performance gain, we need to optimize 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

A Linear Algorithm

2.1 The Base 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 problem with this algorithm is that it applies forces between all pairs of particles, while the rules for the particle simulation states that any particle may only be affected by nearby particles.

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.

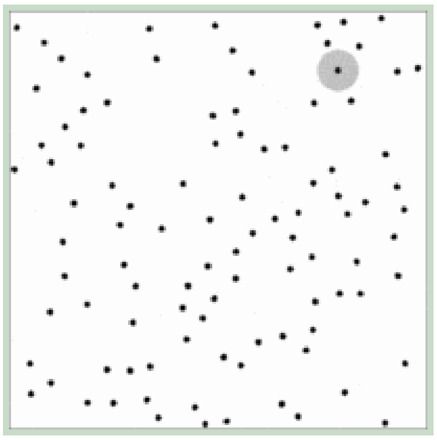


Figure 2.1: Snapshot of the GIF from the assignment showcasing the influence ring.

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 inner for-loop greatly.

In order to reduce the loop it is necessary to know which particles are within (or at least 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.

In order to track the location of the particles, the coordinate system is divided into a grid of cells. Each cell holds some number of particles within a sub-grid of the entire universe. The sizes of the cells are chosen to match the “cutoff” value, such that we only need to check the neighboring cells for collisions with a particle.

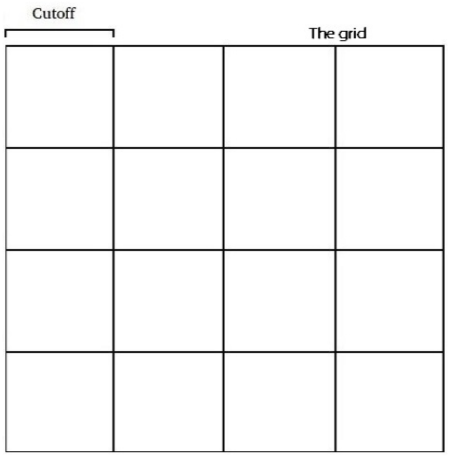


Figure 2.2: The grid structure, each square is the size of the cutoff distance.

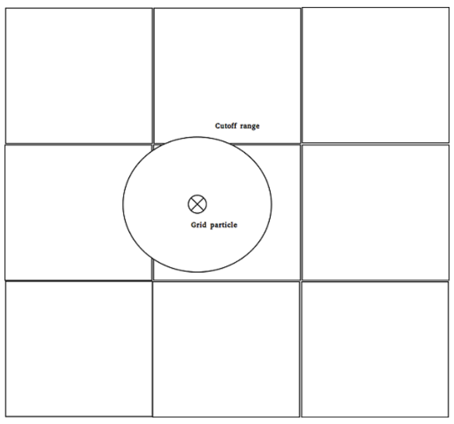


Figure 2.3: A particle located inside a grid position, and its influence range.

Initially all particles are added to the grid. When updates are performed on a particle, it will be moved to the appropriate cell as well.

2.3 Implementing the Grid Approach

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 2D array of particles.

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.

2.4 Performance of the New Algorithm

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.

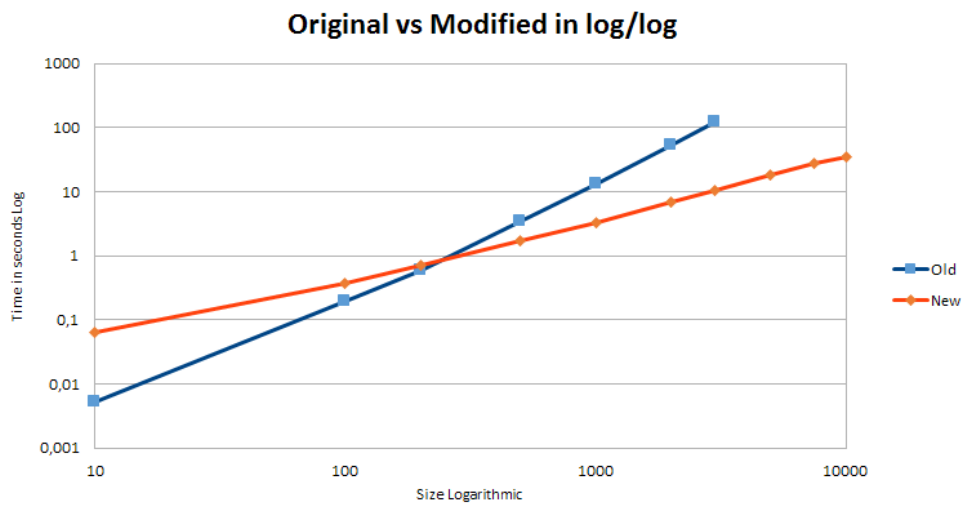


Figure 2.4: The original $O(n^2)$ algorithm (blue) versus the new $O(n)$ algorithm (red).

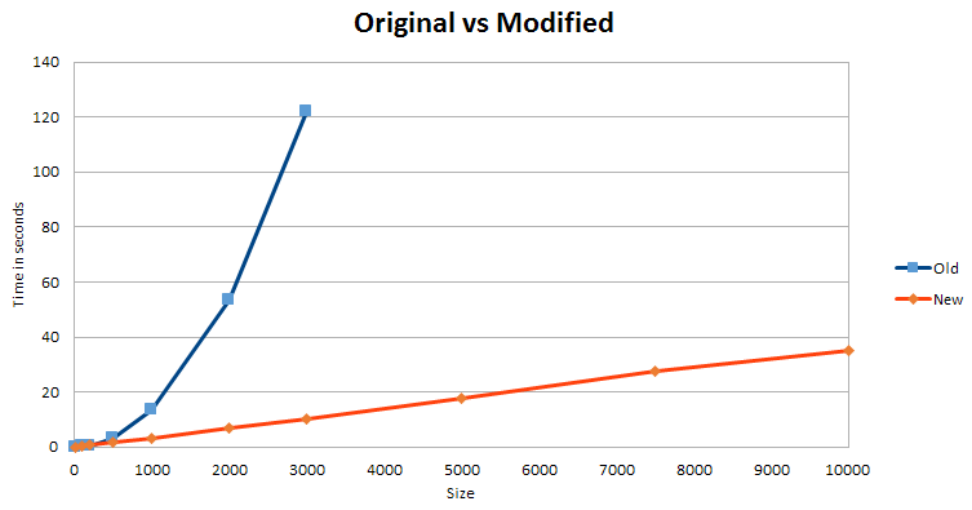


Figure 2.5: Same as above, but not using logarithmic axis.

2.5 Testing

Chapter 3

Parallelization using MPI

Throughout this chapter we will discuss how our serial implementation was parallelized. The parallelization was done using MPI, and we will be referring to concept found in this library whenever relevant.

3.1 Model for Communication

We would like to share the work onto multiple processors, to parallelize our serial implementation. Looking at our implementation it seems obvious to give each processor a set of cells to be responsible for, instead of giving each processor a set of particles to be responsible for. By letting each processor be responsible for a zone, it can perform most of the work completely locally, since each cell can only contain particles that will be affected by particles in neighboring zones. This means that only particles lying in a cell at the border of the zone, may be affected by a cell not owned by the local processor.

There are several ways in which the grid may be divided into zones for the processors. For the sake of simplicity each processor is given some number of rows in the grid, for which they are responsible. This means that any processor will have at most two neighboring processors, the one responsible for the rows above, and the one for the rows below.

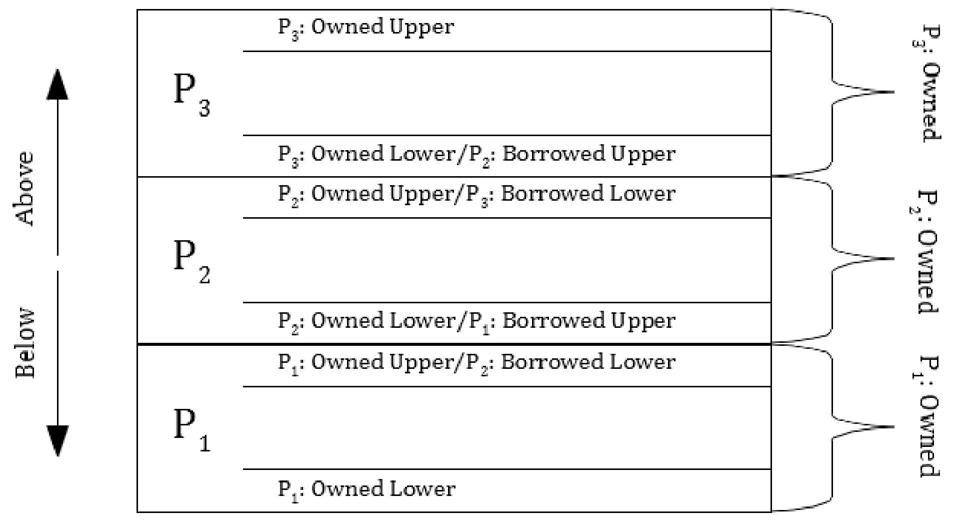

P_3: Owned Upper
P_3
P_3: Owned Lower/P_2: Borrowed Upper
P_2: Owned Upper/P_3: Borrowed Lower
P_2
P_2: Owned Lower/P_1: Borrowed Upper
P_1: Owned Upper/P_2: Borrowed Lower
P_1
P_1: Owned Lower
Above
Below
P_3: Owned
P_2: Owned
P_1: Owned

Figure 3.1: Splitting the grid into zones.

To deal with the problems of the zones lying at the border of a zone, we've introduced the concept of borrowed and owned zones as shown in figure 3.1. These zones are exactly correspond to exactly one row in the grid. Each processor will contain up-to-date "borrowed" copies of these zones. The figure how each processor identifies the different zones. Each processor will have to communicate with its neighbors to keep these zones in sync.

3.2 Initializing the System

The initial system is initialized at the root. Once all the particles have been created, a local grid is created, like in the serial version. This gives us the initial distribution, which is then distributed to all the processors in the system using `MPI_Scatterv`.

3.3 Synchronization of Nodes

As discussed previously, each processor will need to keep synchronized copies of zones that they borrow from each other. Since we in every iteration may make a change to these zones, we will need to perform this synchronization step in every single iteration as well. The synchronization process is two-fold, first we gather up-to- date information, and then we merge our information to get a correct picture of the system.

Step 1: Synchronize Processors

At the beginning of every iteration, we will need to synchronize all processors, such that they are all ready for the next iteration. Once they are ready for the next iteration, we may begin exchanging information. This synchronization can be done using the `MPI_Barrier` procedure.

Step 2: Prepare for Exchange with Neighbors

A processors' neighboring processes will expect a complete image of how the zone it is borrowing looks. This message needs to be prepared, such that we can transmit it, this process consists solely of packing all the particle into a single buffer.

Before beginning the exchange, we will also clear out any particles that reside inside of the borrowed zones. This is done since we will receive a completely new zone from our neighbors.

Step 3: Exchange Message with Neighbors

Once the messages are ready, we may exchange data with our neighbors. For efficiency reasons we use `MPI_SendRecv`, which allows for simultaneously sending and receiving data between two nodes. MPI requires that both parties involved in the transfer, call this function with each other as arguments. For this reason, the following pattern has been established: processors with an even rank communicate with the processor above it first, while processors with an odd rank will communicate with the processor below first. The exchange process between two processors go like this:

1. Exchange sizes (in particles) of zones
2. Exchange the particles, this exchange requires the knowledge of how many particles we will receive
3. Exchange sizes of local insertions into borrowed zones
4. Exchange particles involved in local insertions into borrowed zones

It should also be noted that this means that the synchronization process for the system as a whole, will only take the time of communicating with each neighbor, since these are all done in parallel.

Step 4: Update the World

We have already discussed that we share our owned zones with our neighbors. The reason for this fairly simple, we're the ones responsible for updating these zones, hence we are the ones capable of telling our neighbors about updates made in it.

However, it is also possible for particles to migrate from one processor to another. Hence we need to track whenever we perform an insertions of particles from our owned zone into a borrowed zone. These particles are all tracked in a separate buffer, and exchanged with the neighbor in step 3.

The insertions that we receive from our neighbors are then merged together with our own local view to give a completely up-to-date view of the system.

Step 5: Cleanup

Finally we clear out our local insertions from an owned zone into a borrowed zone. Such that we're ready for the next iteration.

3.4 Running the Simulation and Combining the Result

The simulation step itself is left mostly unchanged from the original serial implementation. It should however be noted that we only have to directly look at the particles we own ourselves. That is we do not need to loop directly over the particles in the borrowed zones, since if any of our owned particles collide with them, then this will be picked up when we look at the neighboring cells.

While the simulation is running, we write down where are particles are. To be able to merge the particles later, we have had to add an ID number to every particle, such that we may track any particle, even if it goes through several processors. Once the simulation is done, it is collected at the root node using `MPI_Gatherv`, sorted per iteration, and ID, and saved to the output file.

3.5 Limitations

The technique used for distributing the cells to each processor, puts some limitations to the number of processors one can effectively use with the system. The current implementation allows for at most as many processors as there are rows. And when hitting this case, we won't be very effective, since we will hold twice as many borrowed cells, that we hold owned cells. The system could allow for more processors by letting the zones be smaller squares, that do not need to have the same width as the entire grid. This does however come at the cost of having up to four neighbors instead of at most two and the increase complexity in communicating.

3.6 Performance

This section is used to document two aspects:

1. Overall performance
2. Breakdown of time used in a single run

Like previously, in order to confirm that the implementation is still linear, we can plot the time against the input size and visually see the result.

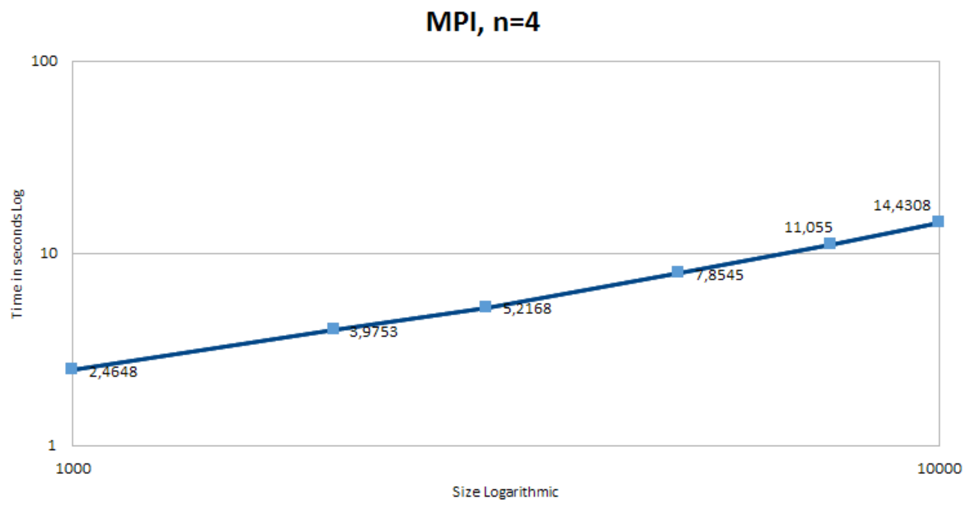


Figure 3.2: Runtimes for the MPI implementation using 4 processes.

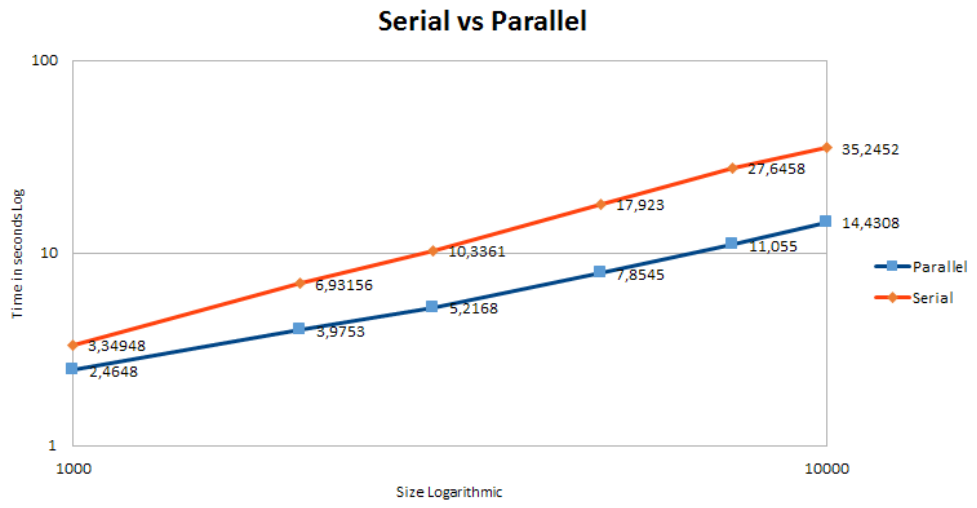
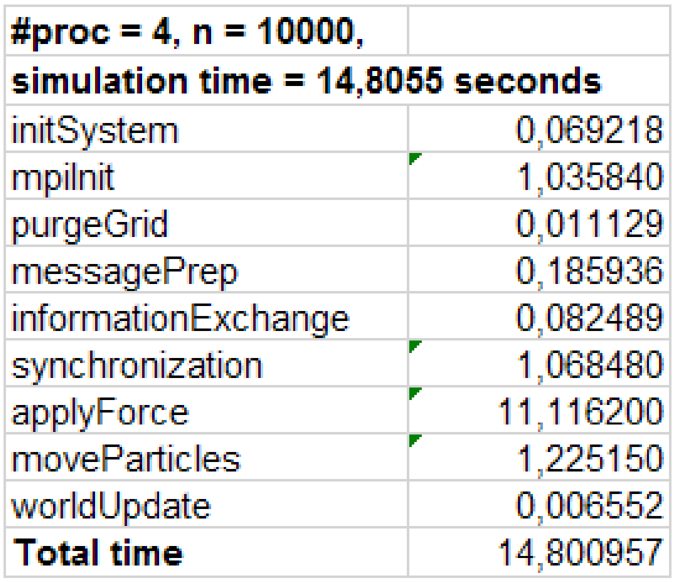


Figure 3.3: Serial runtime against the parallel runtime.

As seen in the second graph, the overhead from the parallel implementation almost negates the performance gain, but as the particle size increases, this constant overhead will have less of an impact. This makes it interesting looking at just how much overhead is present.

To find out how much overhead the parallel implementation has, we have inserted a series of "times zones" which accumulate the time the root process spends in a given section of the code. Using this we are able to account for exactly how much time has been spent calculating additional work.



#proc = 4, n = 10000,	
simulation time = 14,8055 seconds	
initSystem	0,069218
mpiInit	1,035840
purgeGrid	0,011129
messagePrep	0,185936
informationExchange	0,082489
synchronization	1,068480
applyForce	11,116200
moveParticles	1,225150
worldUpdate	0,006552
Total time	14,800957

Figure 3.4: Time spent calculating.

3.7 Testing

1. (Should also be added to serial)
2. Testing stuff

Chapter 4

Conclusion

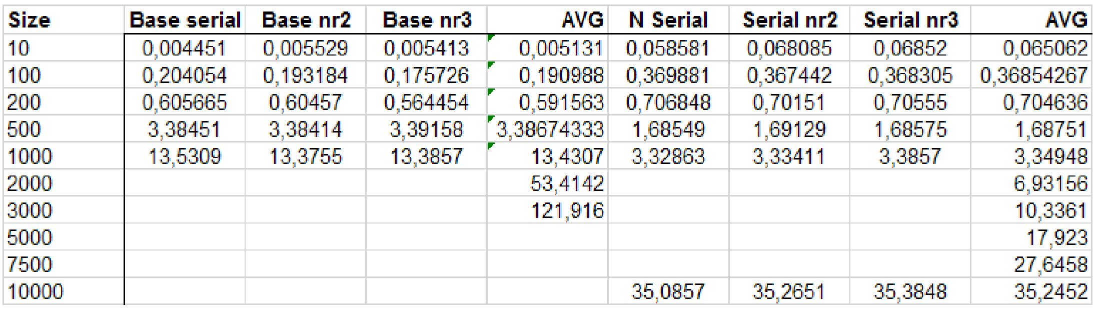
The serial algorithm for the particle simulation has been made linear by implementing a grid system, which is essential a data-structure used to look up particles at given coordinates. This modification allowed us to only apply force to a given particle with other particles nearby, thus not having to do this for every single existing particle.

The new and improved algorithm has been made parallel using MPI. This introduced some overhead, but this proved trivial as the input size (of particles) grew larger. Some problems persisted, such as it is not possible to use more processes than there will be rows in the grid. Overhead in the parallelization was accounted for by counting time spent is specific sections of the code.

Chapter 5

Appendix

5.1 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 5.1: Plotting data for the serial runtimes. Base being the handed out version, and N being the linear implementation.