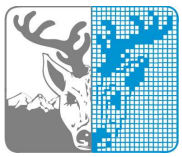
Parallel Simulation of Particle Fluids

Mattias Linde

December 13, 2007
Master's Thesis in Computing Science, 20 credits
Supervisor at CS-UmU: Kenneth Bodin
Examiner: Per Lindström



Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

Smoothed Particle Hydrodynamics (SPH) is a method for simulation of fluids such as water. The computational needs scale with the number of particles used. When a large number of particles are to be used, neither the CPU or RAM-memory is enough on a normal desktop computer for interactive results.

In this thesis a parallel SPH code in C++ has been develop for the use on clusters. The code can handle millions of particles given a sufficiently large number of CPUs are used.

The results reported in this thesis include examples with 10 million particles on 100 CPUs requiring less than 1 second per time step and 15 million particles being stepped at approximately 1.2s per time step.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This report describes the work involved in developing a parallel code for Smoothed Particle Hydrodynamics (SPH) simulations on large clusters.

The report is divided into seven chapters. First this introduction, thereafter *Theory*. The first parts of the theory chapter is about SPH and can be omitted when reading the report, but it is a good idea to at least give it a glance since it will help to understand the problems that have to be solved when making a parallel implementation. The last part of the theory chapter describes some basics for parallel performance metrics and is useful for those new to parallel computing to understand the results in chapter 5.

The second chapter is about *parallel particle algorithms*. In this chapter an overview about different parallel particle methods reported in scientific literature is given. This chapter is divided into two sections, one for short range forces and one for long range forces. Usually handling just short range forces is easier and requires less calculations which is the reason why that part is described first in the chapter. Some parts might be skipped of chapter 3 but it is important to at least read the part about *Spatial decomposition* which is used in the implementation.

Chapter 4 describes the *Implementation* and the changes required to a serial code to make it parallel. Thereafter comes *Results and discussions* which bring up the results and some of the problems that were experienced. The report ends with *Conclusions* followed by *Acknowledgements*.

1.1 Thesis description

Simulation and modeling of fluids and fluid like materials is an ongoing field of activity in scientific research, as well as in application areas such as engineering, computer games, motion picture visual effects and training simulators.

This thesis project is aimed at producing a parallel simulation code for fluid dynamics using the so called Smoothed Particle Hydrodynamics (SPH) method for solving Navier-Stoke's equations describing a fluid.

1.1.1 Problem Statement

The project should result in a simulation code that can run a parallelized version of SPH on large PC clusters and visualize the resulting fluid dynamics. To test the code some benchmark problems should be solved such as "dam break". The visualization should be done in real-time with e.g. OpenGL, as well as batch rendering using for example PovRay.

1.1.2 Goals

The collision detection used in the project should be based upon the code developed for "Collision Detection for Haptic Rendering"[14] (hereafter referred to as HOACollide) so the collision code from that project can be reused and further extended (e.g. add support for collision between new types of primitives).

To simplify simulations of particle systems, an API should be developed together with Andreas Grahn (who was also working on a master thesis at VRlab involving SPH simulations). This should result in that both projects model particle systems the same way and possibly prolong the lifetime of the code.

1.1.3 Methods

In order to reach the goals specified; an in depth study of parallelization of particle code should be performed. This involves problem decomposition and algorithmic aspects. Testing is another important aspect, both of collision code for particle neighbour search and communication so the correct particle neighbours are used and the correct data is exchanged.

1.2 Related Work

Much work have been done in the SPH-area since the introduction of the method in 1977 and SPH has been applied to many areas. There are also many particle simulation codes ranging from molecular dynamics (e.g. CHARMM) to stellar dynamics. Most of them are written in Fortran. There are some previous parallel SPH codes in C++ such as [11] and open source code SPH2000[3].

Chapter 2

Theory

2.1 Smoothed Particle Hydrodynamics

The Smoothed Particle Hydrodynamics (SPH) method was introduced by Gingold and Monaghan 1977 and independently by Lucy the same year. The method was developed for simulation of astrophysical problems, but it has been applied to several other areas where fluid simulation is one of them. Monaghan presented a good review [10] of the theory and applications of SPH in 2005 that for the interested reader contains more details about the method than what is covered here.

SPH is a meshless method with no fixed points, instead particles are used to carry the field quantities being simulated. In SPH, any field quantity can be approximated using a weighted sum,

$$A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.1}$$

where A is the scalar quantity being calculated at point $\mathbf{r}$. For all neighbour particles j within a certain distance, the mass m_j, density ρ_j, field value A_j and weight function W (usually denoted a smoothing kernel) are used to calculate the value. The distance between two points decides how much influence the neighbour particle gives, and if further away than a certain cut-off value, the smoothing length, h, the contribution is 0.

There are different kernel functions and it is possible to design own kernels for special cases. Some kernel functions are presented in [12]. A kernel should be normalized which is the same as

$$\int_r W(r, h) dr = 1 \tag{2.2}$$

Each particle represents a small part of the volume of the simulated fluid ($V_i = m_i/\rho_i$). The mass of a particle is constant during the simulation, but the densities need to be calculated at every time step. Equation 2.1 is used:

$$\rho(\mathbf{r}) = \sum_j m_j \frac{\rho_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) = \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.3}$$

Many simulations require the gradient and laplacian of field quantities and this only effects the kernel function in SPH. The gradient can be written as

$$\nabla A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.4}$$

and the laplacian

$$\nabla^2 A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r}_j, h). \tag{2.5}$$

2.2 Fluid simulations with SPH

The Navier-Stokes equations, first formulated in 1822, named after Claude-Louis Navier and George Gabriel Stokes, describe the motion of fluids such as liquid and gas. A common general form of the equation is

$$\rho \left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} \right) = -\nabla p + \nabla \cdot \mathbb{T} + \mathbf{f} \tag{2.6}$$

where p is pressure, T the shear stress and f is an external force. To make the equation applicable and useful it can be written as

$$\rho \left(\frac{D\mathbf{v}}{Dt} \right) = -\nabla p + \mu \nabla^2 \mathbf{v} + \mathbf{f} \tag{2.7}$$

where the term $\nabla \cdot \mathbb{T}$ representing shear stress has been replaced with a viscosity term. Mass is conserved in a fluid, and this can be expressed as

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{2.8}$$

In an incompressible fluid ($\partial \rho / \partial t = 0$) equation 2.7 can be simplified to

$$\nabla \cdot \mathbf{v} = 0 \tag{2.9}$$

A SPH formulation of the Navier-Stokes equations is used in this thesis for the parallel simulation. The acceleration of a particle is

$$\mathbf{a}_i = \frac{\mathbf{f}_i}{\rho_i} \tag{2.10}$$

where the force density consists of

$$\mathbf{f}_i = \mathbf{f}_i^{pressure} + \mathbf{f}_i^{viscosity} + \mathbf{f}_i^{external}. \tag{2.11}$$

The pressure and viscosity become

$$\mathbf{f}_i^{pressure} = -\nabla p(\mathbf{r}_i) = -\sum_j m_j \frac{p_j}{\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{2.12}$$

$$\mathbf{f}_i^{viscosity} = \mu \nabla^2 \mathbf{v} = \mu \sum_j m_j \frac{\mathbf{v}_j}{\rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{2.13}$$

The forces generated from equation 2.12 and 2.13 are not symmetric and solutions to this problem can be found in [10].

The Density is calculated from equation 2.3 and the pressure is calculated from

$$p = c^2(\rho - \rho_0) \tag{2.14}$$

where c is the speed of sound and ρ_0 a reference density of the fluid. From these equations all forces needed for the simulation can be calculated.

It should be noted that equation 2.14 results in pseudo incompressibility, i.e. volumetric variations and density fluctuations are penalized by the pressure force as expressed in 2.12. Alternatively one could use a constraint based formulation that strictly satisfies 2.9 at each time step, by solving the corresponding Poisson's equation. However, this goes beyond the scope of this thesis and is at time of writing not suited for interactive computing due to the increased level of computational complexity for solving Poisson's equation.

2.3 Performance

Here some basic notation for parallel programs and their performance will be presented. Some knowledge in this area is assumed in later chapters of this thesis.

2.3.1 Execution times

When measuring performance and comparing execution times some common metrics are used. The time for the serial program from start to finish is denoted T_s. In the same way, parallel execution time is called T_p, and is defined as the time from the start of the execution until all nodes have completed their task. The number of processors used for the problem is denoted as p and the problem size as n. The total CPU-time for a parallel instance therefor becomes

$$pT_p. \tag{2.15}$$

The parallel overhead (extra work not performed in serial execution) is:

$$T_o = pT_p - T_s. \tag{2.16}$$

Speedup is defined as the ratio between serial runtime and parallel runtime.

$$S = \frac{T_s}{T_p} \tag{2.17}$$

For speedup calculations the best serial implementation should be used but in reality the parallel code is often used and run on 1 processor. Efficiency of the parallel code can be written as

$$E = \frac{S}{p} = \frac{T_s}{pT_p} = \frac{1}{1 + \frac{T_o}{T_s}}. \tag{2.18}$$

2.3.2 Communication

In parallel code, communication is often required between the different processors. The metrics used when describing communication are t_s for startup time to send a message and t_w the time to send one word and m the message length. The time to send one message from one processor to another then becomes

$$t_s + t_w m. \tag{2.19}$$

The time for some common communication operations on a hypercube are presented in [4] and the expressions for the communication methods used are listed in Table 2.1.

Operation	Hypercube time
One-to-all broadcast, All-to-one reduction	$min((t_s + t_w m) log\ p, 2(t_s log\ p + t_w m))$
All-to-all broadcast, All-to-all reduction	$t_s log\ p + t_w m(p-1)$

Table 2.1: Communication costs

Chapter 3

Parallel particle algorithm overview

3.1 Introduction

The same ideas are used in different particle simulations, ranging from very small length-scale molecular dynamics to N-body astronomical simulations. These simulations can be divided into two groups, those with long range forces and those without. For astronomical simulations gravity between stellar bodies acts as a long range force, similar to Coloumb forces in molecular dynamics.

In simulations where only short range forces are handled, long range forces may still be present. Screening can occur where one nearby body blocks distant bodies and reduces the force addition from distant bodies so they can be omitted, or more commonly, accounted for using a mean field approximation, so that long range particle-particle forces are reduced to short range particle-particle forces and particle-field forces, where the field is represented on a grid. Truncation can also be used is some situations.

3.1.1 Short range algorithms

Here three different algorithms for short range interactions will be presented as described in [13] and [6].

Atom Decomposition

In atom decomposition (AD) each of the P processors is assigned N/P atoms. These atoms will belong to the processor during the entire simulation. A processor will compute forces and update positions and velocities only for its atoms.

When describing the computational work in the algorithm, a force matrix F with size N x N is often used. Since it is short range many positions in the matrix are zero and this sparse matrix is also skew-symmetric ($F_{ij} = -F_{ji}, i \neq j$) because of Newton's 3rd law. Two vectors of length N are used, x to store the positions of the atoms and f to store the forces.

Each processor is assigned a sub-block of N/P rows of the matrix F and corresponding parts of the vectors x and f. To calculate the force between two atoms a processor

needs to know the position of many other atoms which belong to other processors, therefor all-to-all communication is required so each processor has an updated x-vector.

Algorithm 1 Atom Decomposition

Require: Updated global position vector
1: Construct neighbour lists for all interactions in the sub block of F
2: **for all** Interactions **do**
3: Compute force between atoms and add to sub block of force vector
4: **end for**
5: Compute new positions for processors N/P atoms
6: Update global position-vector

This algorithm does not use Newtons 3rd law to reduce the amount of calculations. It can easily be modified to do so by making every processor having a force vector of length N instead of N/P and forces are accumulated for other processors atoms as well. This longer force vector needs to be communicated so every processor receives its part and can add all the forces for its atoms before the new positions can be computed.

The updated algorithm requires fewer computations but the amount of communication has doubled and is therefor hardly ever used since the first version usually performs better[13].

Each processor will have approximately the same amount of work to do if the amounts of zeros in the sub blocks of F are about the same. This does not have to be the case since density can vary over the simulated domain. To handle this problem the atom order can be randomly chosen at the start of the simulation.

The main problem with the algorithm is the global communication which scales with N. This puts a limit to how many processors that can be used efficiently.

Force Decomposition

Force decomposition (FD) is similar to AD but the force matrix F is blocked in a different way. Instead of each processor having blocks of N/P rows, they now have square blocks of size $(N/\sqrt{P})$ x $(N/\sqrt{P})$ similar to matrices in linear algebra. The idea for the algorithm comes from matrix multiplication[6].

Additionally, to the position vector x, there is also a vector x' which stores x in column order. To reference parts of the matrix subscript a and b will be used for rows and columns. a and b range from 0 to $\sqrt{P}-1$. To reference a certain processors part of a vector, z is used as index, range from 0 to $P-1$.

Algorithm 2 Force Decomposition

Require: Updated global position vector x_a and x'_b
1: Construct neighbour lists for all interactions in the sub block F_{ab}
2: **for all** Interactions **do**
3: Compute force between atoms and add to sub block of force vector f_a
4: **end for**
5: Fold f_a in row a (f_z)
6: Compute new positions in x_z with f_z
7: Expand x_z in row a (x_a)
8: Expand x_z in column b ($\mathrm{x'}_b$)

The amount of communication is reduced in this FD algorithm compared to the previous AD algorithm. The communication scales as $N/\sqrt{P}$ which performs much better than the previous algorithm when a large number of processors are used.

The presented algorithm above does not use Newtons 3rd law, but it can be modified by including a second force vector and some more communication and this normally results in a speedup[6].

The force decomposition algorithm is more sensitive to load imbalance than AD, but this can be solved with a random permutation of the atoms. With less memory usage and lower communication cost, FD is preferable over AD.

Spatial Decomposition

Spatial decomposition works by dividing the problem domain into boxes with particles, where each box is assigned to a processor. Each processor is responsible for the particles within the box, i.e. to update the positions and velocities. If a particle moves so it passes over from one box to another, it is assigned to the processor corresponding to the new box.

The communication is somewhat different compared to FD and AD. Since the boxes border each other all the particles needed for the computation is at max one box away (as long as the box size is larger than the force cutoff length).

The communication scheme works so that nearby boxes exchange information (positions) in one direction at a time. First east-west, then west-east. If the boxes are bigger than the cutoff length just parts of the particle positions needs to be exchanged. If, on the other hand, the boxes are smaller than the cutoff length, several steps in the same direction may be required so all particle positions are available for force calculations. The same approach is then used for north-south with the addition that newly received particles from other boxes may be included in the communication in the next direction. Finally communication up-down is performed. If the domain is decomposed in x-, y- and z-direction a box may be surrounded by up to 26 other boxes and all necessary information is gathered in six communication steps (as long as the boxes are larger than the force cutoff length).

Algorithm 3 Spatial Decomposition

Require: Updated positions of boundary particles of other boxes
1: Move particles to new boxes if they have moved out of current box
2: Make lists of particles needed for communication
3: Generate neighbour lists with interaction pairs
4: **for all** Interactions **do**
5: Compute force between particles and add to force vector
6: **end for**
7: Update positions from force vector
8: Exchange atom positions across box boundaries

3.1.2 Long range algorithms

Here some techniques will be presented for calculating long range forces.

Ewald summation

In molecular dynamics the molecules can be simulated in a solvent. To reduce surface effects the simulation box can be replicated infinitely in every direction which forms a lattice. This forms what is called a periodic boundary condition (PBC) [16].

The total Coulomb energy in a system with N particles when PBC is used can be expressed as: $U = \frac{1}{2}\sum_{n}^{i}\sum_{i=1}^{N}\sum_{j=1}^{N}\frac{q_i q_j}{r_{ij},n}$

Ewald summation was introduced in 1921 by Paul Peter Ewald as a way to rewrite the series that expressed the long range forces with PBC into two rapidly converging series and a constant term.

Three different parameters control the convergence rate of the Ewald series which affects the speed-accuracy ratio. Other techniques to improve performance includes truncation schemes and also even neglecting one of the two series if the simulation parameters have been chosen correctly.

Traditional Ewald summations is of O(N^2) but there are algorithms of order O($N^{3/2}$). There are also many other techniques and algorithms regarding Ewald sums. For further information see [16].

Barnes-Hut algorithm

When a group of particles are far away they can be seen as one new particle positioned at the center of mass of those particles and having their combined mass.

This can be realized with a tree structure, oct-tree in 3D and quad-tree in 2D.

The basic algorithm is:

Algorithm 4 Barnes Hut algorithm

1: Construct tree
2: Traverse tree from leaves to root and compute total mass and center of mass for parent nodes
3: **for all** Particles **do**
4: Traverse tree from the root and calculate forces
5: **end for**

If a leaf node is encountered while calculating the force, that one is used. If an interior node is encountered, depending on the distance, the approximated mass and mass center is used or all the child nodes are checked recursively.

There are different ways to do the algorithm in parallel and for more details see [5].

Another set of tree code algorithms (using oct-tree or quad-tree) are the Fast Multipole Methods which can perform long range calculations in down to O(n)[1].

Particle mesh methods

Several particle-mesh techniques are presented in [9], which can be referenced for more details.

Instead of calculating the force on each particle from particle-particle interactions, a mesh can be used for interpolation.

Poisson's equation

$$\nabla^2\phi = -\rho(x) \tag{3.1}$$

relates charge density to electric field potential. The field is

$$E(x) = -\nabla\phi \tag{3.2}$$

and the force on a particle i becomes

$$F_i = q_i E(x_i) \tag{3.3}$$

Algorithm 5 Particle Mesh algorithm

1: Calculate ρ at each mesh point
2: Solve Poisson's equation
3: Calculate E at all mesh points
4: Update positions and velocities

The algorithm does not only have to be used to calculate forces between electric charges, the idea can be applied to other areas e.g. gravitational long range forces.

When a particle is mapped to a grid point to calculate the density it seldom hits the point and there are different methods to handle this. Nearest grid point (NGP) is to assign the whole density contribution to the nearest grid point just as the name says. Another method is Cloud-In-Cell (CIC) which is that the density is smoothed over the nearest grid points.

There are other mesh methods and one commonly used is Particle-Particle Particle-Mesh (P^3M) which is a hybrid method combining particle-particle for short range forces and particle-mesh.

One example of a simulation using long-range force is "The Millennium Simulation" which needed to calculate gravitational forces when testing a model for the formation of the structure in the Universe. For the simulation, a combination of a hierarchical multipole expansion and Fourier transform particle-mesh methods were used. The number of particles used were $2160^3 \simeq 10^{10}$[15].

Chapter 4

Implementation

4.1 Overview

The implementation is aimed at being as modular as possible so that parts can be selected according to the user needs for the simulation. In the simulation there is a *Universe* which contains all objects being simulated.

Different types of *ParticleSystem*s can be added to the universe as long as they follow the base API for particle systems. A particle system contains particles which are stored in a *ParticleData* instance. This object can be replaced as well, depending on how the data should be stored in memory to achieve good performance. Also, more parameters can be added to the particle data if a needed parameter is not defined in the base class.

In the implementation used, the data is stored in one array for each property (position, velocity, ...). If needed, the data could be rearranged to an array with particle structs containing all the data but that would cause all the positions not to be consecutive in memory and reduce cache efficiency during neighbour search and make efficient communication harder.

The particle system can have emitters which can emit more particles according to some geometry, e.g. new particles are added from randomized positions on a line.

The particle system can also have operators that manipulate the particle data. These operators should be customized according to the implementation used for the particle data, so they can operate efficiently on the data directly instead of doing function calls for every particle property that should be read or written.

For a fluid simulation involving SPH calculations, operators are added to the particle system so the necessary calculations are performed.

Some operators act on pairs of particles (two particles being at maximum a certain distance apart). To find these pairs, a neighbour search algorithm is used (if one has been set in the particle system).

4.2 Simulation loop

The simulation is usually performed in a loop. This loop can consist of gathering statistics, doing rendering and a call to the update method in the universe object which will step the system.

What is being performed by the update method in the universe is the same no matter

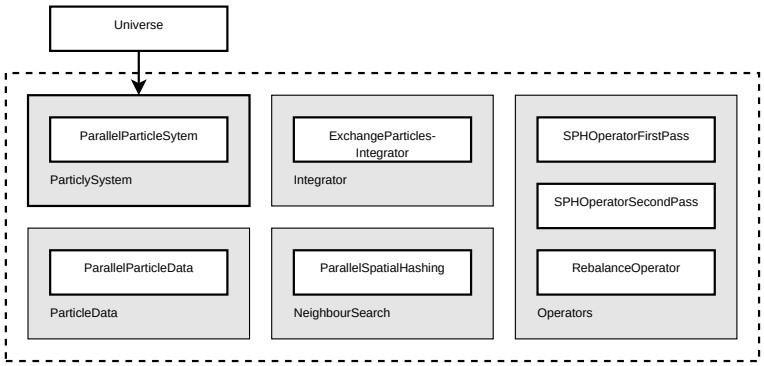


Figure 4.1: Example of classes used for a parallel dam break simulation

if the code is being run on a 1CPU machine or on a large cluster. The differences that do occur are because of different particle systems and different implementations of particle data being used.

Algorithm 6 Update in Universe object - universe.update(timestep)

```
1:  for all ParticleSystem p do
2:      p.update(timestep)
3:      for all p.emitters e do
4:          e.emit
5:      end for
6:      if p.NeighbourSearch.valid then
7:          p.NeighbourSearch.findNeighbours
8:      end if
9:  end for
10: for all ParticleSystem p do
11:     p.runOperators
12: end for
13: FindCollisions
14: for all ParticleSystem p do
15:     p.integrate
16: end for
```

An application can have more than one universe and perform several similar (or identical for that matter) simulations at the same time and compare results to see how one parameter affects the overall simulation.

4.3 Parallel implementation

In the implementation, spatial decomposition is used (see section 3.1.1), so that one-to-all and all-to-all communication can be avoided when performing the SPH calculations.

4.3.1 Emitters

In a serial implementation it is easy to know how many particles there are alive. On a parallel machine, where particles can be moved from one node to another, some new problems arise. Particles can die when their lifetime has exceeded a certain value and this makes it trickier. Particle lifetime and death is not used in the dam break simulations, but can be handled if needed.

In the serial case emitters are called if $num\_particles < max\_num\_particles$. If new particles should be emitted, each emitter is in turn called and they generate positions for the new particles. The new particles are inserted after the last particle alive so there will be no gap in memory.

In a parallel implementation each node knows how many particles it has, and an all-to-all broadcast is performed so all nodes knows about the particle distribution. Thereafter the global sum is calculated to decide if new particles should be emitted or not. When the simulated "world" is divided over the nodes and new positions should be generated, a problem that can occur is that different nodes generate different new positions (they are randomized). This could result in one particle being generated gets two different positions and two different nodes take ownership of said particle.

This problem could be handled by stating that the application using the simulation API should not randomize their own values while simulating since it could affect particle generation, but this is not an acceptable solution.

Instead, just the first node calls the emitter to generate new particles. The newly generated particles are then broadcasted from that node to all the other nodes. Every node then checks all the newly generated particles to see if there are any that reside in the nodes spatial domain, if so, those particles are copied from the message buffer to that nodes particle data.

4.3.2 Neighbour search

Finding the neighbours for particles is a requirement to be able to do the SPH calculations. For an overview on spatial hashing and efficient hash table construction see [7].

When the neighbours can be on other nodes such as in Figure 4.2, information about particles needs to be exchanged before the neighbour search algorithm can be used. To reduce communication bandwidth just the positions are transmitted.

When exchanging information, the particles a neighbour needs to know about are those max one search-radius away from the "border" dividing the two nodes. See Figure 4.3. A "Plimpton-scheme"[13] is used for the communication as illustrated with the following 2D-example:

Node 0 tells node 1 about the particles left of the border colored dark blue. Node 1 tells node 0 about some of its particles. Node 2 and 3 exchange information in the same way (those particles are marked light blue).

The next step is that node 0 and node 2 exchange information. Before they can do that, they have to check the newly received particles if some of them should be included. This is illustrated by the right part of Figure 4.3. Node 1 and 3 exchange data the same way but this is not included in the figure to make it more readable.

After the communication steps are performed, all the particle positions needed for the neighbour search are available on each node. The particles node 0 will know about is shown in Figure 4.4.

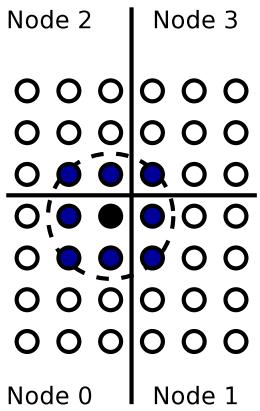


Figure 4.2: Neighbours divided over several nodes (2D-example)

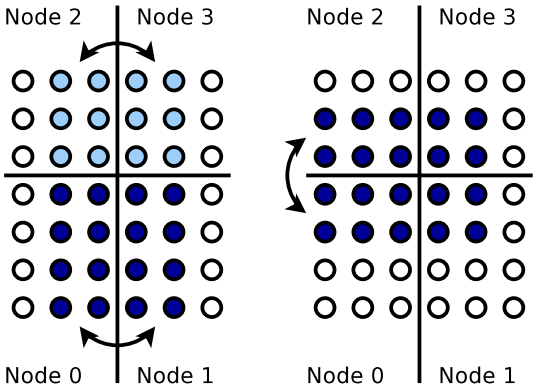


Figure 4.3: Communication for neighbour search, step 1 and 2 (2D example)

The particle positions received during the communication steps are stored after the positions of the particles on the receiving node. This simplifies the neighbour search which should be run on *num_p_on_node* + *num_received* particles instead of *num_p_on_node*.

For neighbour search, spatial hashing is used. The grid size used is the same as the length of the interaction-radius. This causes all neighbours to be max one grid-cell away. There are a total of 27 cells in 3D (3^3) that needs to be checked for a particle. That number is reduced to 13 plus the cell the particle resides in because of symmetry aspects. The code for neighbour search is based upon the work from [14, 7] and the method used for the hash table is described in [2].

To speed up further information exchange between nodes, data about which particles is close to node boundaries are cached.

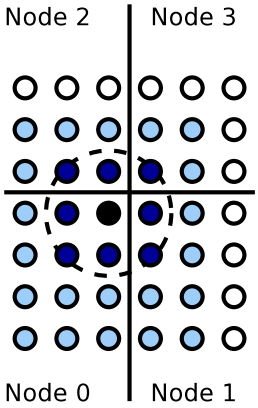


Figure 4.4: Neighbour search seen from "Node 0" and the particles the node knows about.

4.3.3 SPH calculations

The SPH calculations (being run as an two different operators on the particle system) are performed in two steps. The first step is to calculate densities from the particle positions and particle mass. This can be done without interacting with the other nodes. The positions of neighbour particles needed are still available locally from the neighbour search.

This approach depends on that all the particles have the same mass, otherwise some communication would be required.

The next step is to perform communication with the neighbour nodes in the same way as described in 4.3.2, first in X-, then Y- and last Z-direction so densities will be known for other nodes boundary particles.

From the densities, pressure is calculated for all the particles, both the nodes own particles and other nodes' boundary particles. This small work overhead of calculating the pressure for some particles on different nodes is used to reduce communication since both pressure and density are needed in the last step.

With positions, densities and pressure known, combined with the interaction lists from the neighbour search, the acceleration for each particle can be calculated according to the equations in section 2.1

4.3.4 Load balance

A box is used as a container for the liquid being simulated. The box is initially split into equally large regions for each node. When the dam break occurs after a predefined time, the box is made bigger. This causes the regions the different nodes have to become unequally large. As the particles moves towards the newly added volume of the box, the amount of work on the different nodes will change as well. To keep the workload about the same (and achieve better efficiency) the boundaries between the nodes are repositioned.

The boundaries can not be moved so all the nodes will have an equally big part of the container volume at the time of dam break. The particles will initially be in the

"old box" and flow to the new area so a new equilibrium is reached.

What is done is that the boundaries are moved in small steps (the length of the interaction radius) at certain time steps so the number of particles is kept about the same on every node when the fluid moves within the container.

The rebalance code is made into an operator and is the last operator being used before collision detection. Thereafter the system is integrated and the integrator will move entire particles between nodes if they are on the wrong side of some node boundary.

4.4 Memory usage

Here the amount of memory used for a simulation will be presented. The program needs memory for three different things, hold particle properties such as position and velocity, buffers for communication and buffers for efficient neighbour lookup. The amount used are presented in Table 4.2 where the symbols used are described in Table 4.1.

It is important that the memory used for velocities, particle positions and similar properties is stored in contiguous blocks for good cache performance. To prevent reallocation when a node increases its particle count (when the load is not evenly balanced) all nodes have allocated enough memory to be able to hold all particles.

The extra memory allocated is never touched unless the node actually receives more particles and this will in most cases only increase virtual memory used by the program. The memory usage that grows fastest with an increased number of particles is the memory used for neighbour search (unless the number of neighbours per particle is set to a very low number, typically lower than 8). The scheme of overallocation could cause some issues in rare cases if the amount of RAM memory + swap space is lower than the requested amount of memory in which the allocation will fail and program exit. Situations where this could occur would be simulations with hundreds of millions of particles.

Symbol	Description
N_{node}	Number of particles on node
N_{tot}	Total number of particles in simulation
N_+	N + number neighbour boundary particle node needs to know about
P	Size of particle data, depends on particle data implementation used (89 bytes with ParallelParticleData)
P_p	Size or particle data packed with MPI (depends on MPI implementation, P+overhead)
G	Maximum number of particles neighbour owns that node needs to know about +500
M	Maximum number of neighbours per particle
F	Size of float
I	Size of int

Table 4.1: Memory usage descriptions

In Table 4.3 the effects of allocation space for all particle data on all nodes can be seen. The most important thing is that the memory usage per node decreases a lot when a low number of nodes are used and a few more nodes are added. This makes it possible

Part of program	Reason	Memory amount
ParticleData		$N_{tot}P$
Neighboursearch	Hashtable	$3.6IN_+$
	Communication	$6GF$
	Interaction lists	$8MN_+ + 16N_+$
SPH Calculations	Communication	$16GF$
Integrator	Communication	$4GP_p$
Collider	Collision information	$7N_{node}F + N_{node}I$

Table 4.2: Memory usage

Particle count	Node count 1	10	25	50
100 000	37.3M	11.4M (113.7M)	9.64M (241.0M)	9.06M (453.2M)
500 000	186.6M	56.9M (568.6M)	48.2M (1205M)	45.3M (2266M)
1 000 000	373.2M	113.7M (1137M)	96.4M (2410M)	90.6M (4532M)
5 000 000	1866.3M	568.6M (5686M)	482.0M (12052M)	453.2M (22661M)
10 000 000	3732.7M	1137M (11371M)	964.1M (24103M)	906.4M (45322M)
15 000 000	5599.0M	1706M (17057M)	1446M (36155M)	1359M (67983M)

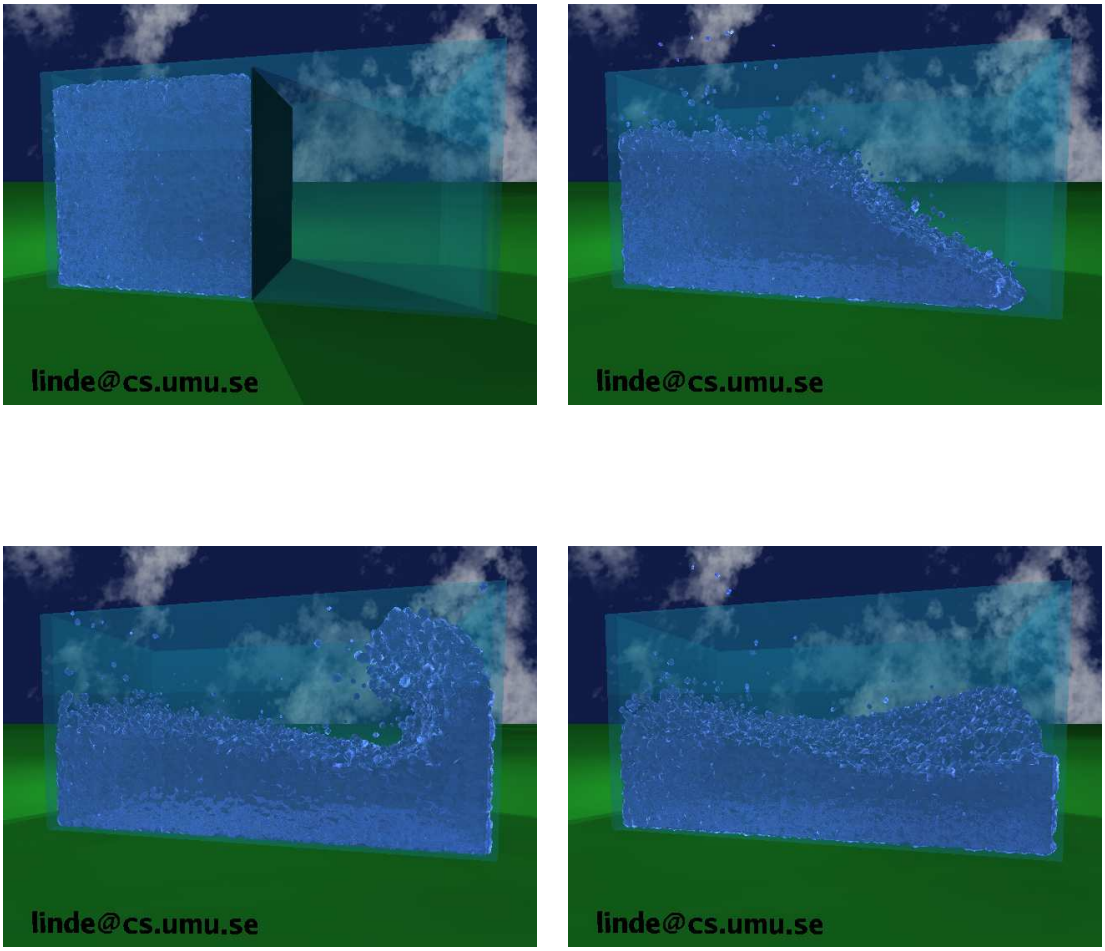
Table 4.3: Theoretical memory usage, per node and total, calculated with max 30 neighbours.

to run large problems on just a few nodes without memory setting the limit for what can be handled. 15 million particles could be run on 10 nodes if each node has 2GB of RAM and this setup could be tested without a cluster in a computer lab with software such as LAM[1] or OpenMPI[2].

[1]Local Area Multicomputer, http://www.lam-mpi.org/
[2]OpenMPI, http://www.openmpi.org/

Chapter 5

Results and discussion



In this chapter the results from some of the simulation tests will be described and the performance analyzed.

When developing the code both lab computers and the Sarek cluster have been used. All the test results are performed on Sarek. Verification of simulation results were performed by comparing the results from a serial implementation and parallel implementation. When the timings should be performed, saving particle positions to disk were turned off so it would not affect the timing.

The Sarek cluster has 192 nodes, 190 of those are computation nodes with dual 64-bit AMD 248 Opteron CPUs running at 2.2GHz and 8GB of RAM. The interconnection between the nodes are a Myrinet-2000 switched network with latency of a few microseconds and 250MB/s bandwidth[8]. The fact that the network is switched makes the implementation easier since the performance between any two nodes should be roughly the same and no special MPI calls needs to be performed to arrange the nodes in a logical layout to give less communication overhead.

5.1 Results

The test runs presented here were setup so the liquid being simulated should have the same physical properties as water and each particle should have has an average 20 neighbours. The actual number of neighbours do vary during the simulation, specially right after the dam break. The decomposition were done in 2D so each node would have to communicate in X- and Y-direction. The particle layout was set to resemble a cube, e.g. 100 000 particles were setup as 50x50x40.

Most test were performed at least two times to for improved statistics and to detect inconsistencies, and some even more to find the issues behind some strange time measurements observed (which will be described further shortly).

Results from 10 000 particle simulation	Number of CPUs				
	1	2	4	9	16
Runtime (s)	228.96	120.71	59.40	37.89	27.989
Runtime $\times$ CPU_{count}	228.96	241.42	237.62	341.01	447.97
Efficiency (%)	100	94.84	96.35	67.14	51.11
Framerate (Hz)	17.47	33.14	67.33	105.57	142.87

Table 5.1: Results from dam break simulations with 10 000 particles.

When the number of CPUs are increased the runtime gets lower, but with such a low number of particles as 10 000 the cost of communication becomes larger and the efficiency drops.

When a larger number of particles are used the efficiency does not drop as rapidly when more CPUs are added. It can be seen that the efficiency even becomes larger than 100% which is superscalar speedup when 4 CPUs are used and 50 000 particles.

This raises the question, with 4 CPUs extra communication overhead, some extra calculations not present when running on just one CPU, how can it take less time? This could happen if the memory access patterns were different and one case would have better cache hit ratio, but both programs use the same memory setup. The results seems to be unlikely, but the reason has to do with the computation nodes on Sarek.

Results from 50 000 particle simulation	Number of CPUs 1	2	4	9	16
Runtime (s)	1730.7	979.39	402.52	194.34	117.71
Runtime × CPU_{count}	1730.7	1958.8	1610.1	1749	1883.4
Efficiency (%)	100	88.36	107.49	98.95	91.89
Framerate (Hz)	2.3112	4.0842	9.9373	20.583	33.982

Table 5.2: Results from dam break simulations with 50 000 particles.

Each node has 2 CPUs and 8GB of RAM as previously mentioned. The hardware is designed so half of the RAM memory is faster to access from one CPU and the other half from the other CPU. The memory latency is 60-70ns and another 10ns is added if the RAM memory closer to the other CPU should be accessed. The runtime for the simulation is mostly bound by memory access times and this extra penalty at about 15% for memory access makes a big impact on total runtime.

Linux is the OS used on the cluster and the kernel used has no way of managing memory allocations so that the program requested memory will be allocated based on which CPU it runs on. Therefore no extra steps can be taken to help the program get better performance when submitting it to the batch system.

Each instance of the program makes several memory allocations and when several nodes are used it is next to impossible to get the optimal CPU-memory allocation and best runtime possible. These problems even made the test runs with 100 000 particles not to give any performance measurements worth using. The different memory access times could add as much as 5-10% extra total run time.

Results from 200 000 particle simulation	Number of CPUs 1	2	4	9	16	25
Runtime (s)	9762.5	5871.4	2714.9	1339.2	729.39	578.64
Runtime × CPU_{count}	9762.5	11743	10860	12053	11670	14466
Efficiency	100	83.14	89.90	81.00	83.65	67.49
Framerate (Hz)	0.41	0.68	1.47	2.99	5.48	6.91

Table 5.3: Results from dam break simulations with 200 000 particles.

Results from 500 000 particle simulation	Number of CPUs 1	2	4	9	16	25
Runtime (s)	33562	17852	8482.6	4003.6	2194.8	1342.4
Runtime × CPU_{count}	33562	35703	33930	36033	35117	33560
Efficiency	100	94.00	98.91	93.14	95.57	100
Framerate (Hz)	0.11918	0.22407	0.47155	0.99909	1.8225	2.9797

Table 5.4: Results from dam break simulations with 500 000 particles.

When the number of particles are increased to several hundred thousands or even millions and a larger number of CPUs are used, the difference between 1 and say 25

CPUs can result in better cache usage since the problem size on each node is smaller, but the memory timing issues makes it impossible to try and reach any conclusions.

5.2 Load balance

The most important thing to get good performance when running the simulation on the cluster is that the amount of work on the nodes are the same. Since the nodes interact with their neighbours the programs will be synchronized at some places in the simulation loop. If some of the nodes have a larger number of particles the other nodes will have to wait.

Part of simulation	Time in percent	
	100k particles, 1 CPU	100k particles, 9 CPUs
Neighbour search (calculation)	60.58%	43.40%
Neighbour search (communication)	0.05%	1.02%
SPH (calculation)	38.85%	17.87%
SPH (communication)	0.00%	25.84%
Step system	0.52%	0.54%
Exchange particles	0.00%	11.33%

Table 5.5: Time usage in different parts of the program in a dam break simulation

Table 5.5 shows the difference between where time is spent when the same simulation is run on 1 CPU and 9 CPUs. In an optimal case, the communication time should be low and calculation percentage just slightly lower in the 9 CPU case.

The results were that roughly 40% of the time was spent on communication, which is way too much. Most of the communication time was spent on "SPH-communication" which is exchanging pressure information. This communication step does not exchange much data, but the reason is that this is after the neighbour search and some of the SPH calculations and any imbalance in those two will show up in the communication afterwards were some nodes will have to wait.

The load balancing scheme used with moving boundaries between the nodes does not perform well enough in this case and needs to be improved. It is implemented as a general purpose way of rebalancing but in some special situations other solutions can give better performance. Manually configuring the values for the rebalancing as it is done currently could be replaced with an analysis of the number of particles on each node and wave propagation speed to better share the amount of work. Even if this analysis would add some overhead the end result could still be lower runtime and better efficiency.

Figure 5.1 shows how the work amount vary during the simulation. The reason for the initial increase in work amount, which is most obvious in the 1 CPU case, is due to the initial grid layout of the particles used when the simulation starts. When the particles find their equilibrium state the average number of neighbours will be higher than in the starting configuration.

If the load balance during the dam break was perfect, it would be almost like running a simulation up until the dam break and see the simulation time and amount of time spent in difference areas of the program to see how it behaves under good conditions.

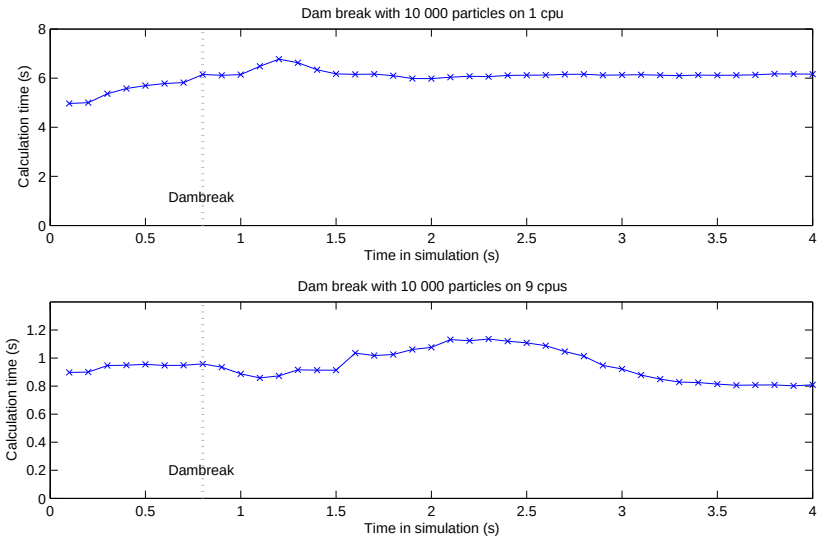


Figure 5.1: Frame times for 1 CPU and 9 CPUs with unevenly balanced workload

This was also tested and results are presented in Table 5.6 where 10 million particles were stepped at 0.827s per frame and 15 million particles at 1.223s per frame.

Part of simulation	Time in percent	
	15M particles, 100 CPUs	10M particles, 100 CPUs
Neighbour search (calculation)	51.90%	51.98%
Neighbour search (communication)	9.96%	9.88%
SPH (calculation)	29.50%	28.59%
SPH (communication)	3.87%	4.77%
Step system	0.90%	0.89%
Exchange particles	3.87%	3.88%

Table 5.6: Time usage in different parts of the program in large pre dam break simulations

Chapter 6

Conclusions

Here some final words about the implementation will be presented and some information about limitations and what was left out in the implementation.

6.1 Restrictions

The solution implemented consists of two parts, one user application which uses a few libraries, some developed during this thesis and some based on the work in HOACollide. Since the library routines are written so they should be able to handle a diverse set of node configurations, the code have been written thereafter and computations can not be overlapped with communications as much if the nodes always were placed in a grid with more than one CPU in each of x-, y- and z-direction.

The code does not currently support a particle interaction radius larger than the box size of each spatial domain assigned to each node (that is, the code does not support several "Plimpton communication steps" in the same direction).

6.2 Limitations

There is no way to get a real time view of the simulation. The cluster where the jobs were run did not support this, but nodes for interactive jobs are on the "work in progress"-list at HPC2N. The steps needed to be able to render the data in real time is to gather all positions on one node (which is really straight forward since sufficiently large memory buffers are already available and just one MPI-call is enough to do this). Thereafter the particle positions needs to be fed to the graphics card and preferably rendered with some shader.

Another limitation is that the user application using the particle system API and the parallel implementation needs to handle boundary conditions itself and that there is no built in support for different solutions.

This is no problem if spring-and-damper should be used to keep the particles reasonably within the container or if the velocity should be reflected (and possibly particles projected) when they hit the container. But if some extra layers with fixed SPH-particles were to be used on the boundary of the container to help the simulated particles to stay in the box there needs to be support from the library which now is not present.

It should not be the users responsibility to keep the load balance at a reasonable level, the library should handle this automatically since doing it with the BalanceOperator and calculating what becomes "educated guesses" is *not* good enough for long large scale simulations.

6.3 Future work

What is left as future work is to investigate the load balancing aspects further and combine this with more complex containers such as a L-shaped container or even a sphere which would cause bad performance with the current code.

Real time rendering from the cluster is another issue that would be a big addition to the implemented solution. Being able to see millions of particles being simulated, even if at about one frame per second, would be much better compared to the post processing which is needed now.

The plan is to use the code for scientific computations of e.g. blood flow in the heart. This will require further support for collision geometries based on triangle meshes or voxels and should fit well into the framework. Such simulations do require a resolution of 10-100M particles or more, and therefore parallelization is clearly required.

Chapter 7

Acknowledgements

I'd like to thank a number of people for the help and encouraging during my work on this master thesis:

Kenneth Bodin, my supervisor, for wide variety of things, from talks about particle systems to writing a project proposal to HPC2N so my jobs got higher priority.

Anders Backman for discussions regarding particle systems API:s and performance aspects.

Andreas Grahn for good cooperation when working on particle system API and code we both needed.

Nils Hjelte for providing me with marching cubes code to generate a mesh from a large number of particles.

The system engineers at HPC2N that helped me track down the reason for weird program run times which had to do with different memory access times.

This research was conducted using the resources of High Performance Computing Center North (HPC2N) and was done within Vinnova/SSF grant VINST #247.

References

[1] Rick Beatson and Leslie Greengard. A short course on fast multipole methods. http://www.math.nyu.edu/faculty/greengar/.

[2] C. Ericson. *Real Time Collision Detection*. Morgan Kaufmann, 2005.

[3] Sven Ganzenmüller, Michael Hipp, Stefan Kunze, Simon Pinkenburg, Marcus Ritt, Wolfgang Rosenstiel, Hanns Ruder, and Christoph Schäfer. *Efficient and object-oriented libraries for particle simulations*. Springer Verlag, 2003.

[4] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing, Second Edition*. Addison Wesley Pearson, 2003.

[5] Ananth Grama, Vipin Kumar, and Ahmed Sameh. Scalable parallel formulations of the Barnes–Hut method for n-body simulations. *Parallel Computing*, 24(5–6):797–822, 1998.

[6] Bruce Hendrickson and Steve Plimpton. Parallel many-body simulations without all-to-all communication. *Journal of Parallel and Distributed Computing*, 27(1):15–25, 1995.

[7] Nils Hjelte. Smoothed Particle Hydrodynamics on the Cell Broadband Engine. Master's thesis, 2006.

[8] HPC2N. Sarek - HPC2N Opteron Cluster. http://www.hpc2n.umu.se/.

[9] P. MacNeice. Particle-mesh techniques, 1995. http://ct.gsfc.nasa.gov/macneice/school/school.html.

[10] J. J. Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68:1703–1759, 2005.

[11] Matthias Müller. Molecular Dynamics with C++. An object oriented approach. *Proceedings of the Workshop on Parallel Object-Orented Scientific Computing*, Lissabon/Portugal, 1999.

[12] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. *Eurographics/SIGGRAPH Symposium on Computer Animation*, 2003.

[13] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117:1–19, 1995.

[14] Hans Sjöberg and Olov Ylinenpää. Collision Detection for Haptic Rendering. Master thesis project started autumn 2006.

[15] Volker Springel, Simon D. M. White, Adrian Jenkins, Carlos S. Frenk, Naoki Yoshida, Liang Gao, Julio Navarro, Robert Thacker, Darren Croton, John Helly, John A. Peacock, Shaun Cole, Peter Thomas, Hugh Couchman, August Evrard, Joerg Colberg, and Frazer Pearce. Simulating the joint evolution of quasars, galaxies and their large-scale distribution. *Nature*, 435:629, 2005.

[16] Abdulnour Y. Toukmaji and John A. Board Jr. Ewald summation techniques in perspective: a survey. *Computer Physics Communications*, 95:73–92(20), June 1996.

Chapter 8

User's Guide

8.1 Simulation data postprocessing - movie rendering

To run a simulation a config-file has to be created describing the simulation. Some sample configs are included with the source package which can be used as a template. The config describe various parameters such as number of particles, density, viscosity, particle container size and position.

When the simulation is being run it will produce datafiles with particle positions (if configured so in the config file). It is these files that needs to be processed to generate some visual results.

- Each node only saves the particle it has to a file (filenames are based upon node number and current frame). Depending on computer system/cluster used the files might be on different machines and local filesystem, if so they needs to be fetched and the first script (`01_getdata.sh`) provides a template for getting the data. This can not be generalized for arbitrary systems and has to be modified before usage.
- The next step is to merge all the individual files belonging to the same frame into larger files. The syntax is `02_merge.sh inputdir outputdir frame-step last-frame`, where the last two parameters comes from values set in the simulation config.
- The third step is to filter the data and remove particles which are slightly outside the container and not completely pushed back in. This depending on spring constant defined in simulation config. `03_filterdata.sh` is used to perform this.
- Thereafter the filtered data is converted from a large number of particles into triangle meshes. This process is in two steps, first create the mesh, then convert the mesh into a format suitable for povray. The syntax is similar to the merge-script: `04_createmeshes.sh indir meshdir povdir frame-step last-frame` where meshdir is a tempdir for the meshes and the final Povray meshes are stored in povdir. This process can generate *large* amounts of data. E.g. 1.8GB filtered data resulted in 5GB meshes and 12GB PovRay meshes.
- The PovRay meshes can then be used when raytracing and the last three scripts are to simplify that process. `05_createpovfiles.sh` acts like a template for generating a PovRay file for each frame using the corresponding mesh and calculating

individual translation for each mesh so it will be placed correctly inside the container. This approach is needed because each mesh is centered at origo and their width, height and depth are not the same.

– The sixth script, `06_raytrace.sh`, is used to run PovRay and generate image files.

– Finally the end result can be produced, a movie can be created from the image files with `07_makemovie.sh`.