Praktikum 2014

"Parallele Programmierung"

Universität Hamburg

Dept. Informatics / Scientific Computing

October 23, 2014

FluidSim

Paul Bienkowski
Author
2bienkow@informatik.uni-hamburg.de

Dr. Julian Kunkel
Supervisor
juliankunkel@googlemail.com

Abstract

This report describes my experiences of the development of a parallel cluster-enabled Simulation using MPI.
My project in particular simulates fluid particles in two-dimensional space by a simple repulsion formula and collision with polygon meshes. The goal was to be able to simulate air flow around a wing shape and determine a possible uplift that was generated only by the particles collisions.

Keywords: Simulation, Parallel, MPI, Fluid, Uplift, Particles, 2D

Contents

1 Introduction

1.1 Motivation

This is the project report for my practical course "Parallel Programming". The goal of this class is to develop any kind of parallel simulation, and implement it to be able to run on a student cluster using MPI.

For my project, I chose the simulation of particles, specifically fluid particles, in 2D space. This kind of simulation could be used in all kinds of research, mostly physics, where mainly experiments are used instead. An example is a classical wind tunnel to investigate how air flows around objects such as car bodies, aircraft wings and other objects that have to display certain aerodynamic properties.

To evaluate how realistic my model would be, I set my goal to simulate a wing-shaped mesh, and measure the force the particles apply to this mesh. I would consider the model sufficiently realistic if this force would be directed upwards.

1.2 Particle Model

In my model, each particle has three basic properties: position, velocity and the current force. Of these, only position and velocity need to be stored, force is recomputed every iteration.

Any two particles in the simulation domain repel each other, given they are in a specified radius of each other:

$$\text{force}_i := \sum_j \textbf{force}(|p_i - p_j|) \cdot norm(p_i - p_j) \tag{1}$$

This equation contains an important function **force**, which is explained in more detail further below. This function calculates the magnitude of the force based on the distance between the particles, and multiplies this with the direction vector between them. The sum of the forces caused by all other particles j is considered the force on particle i.

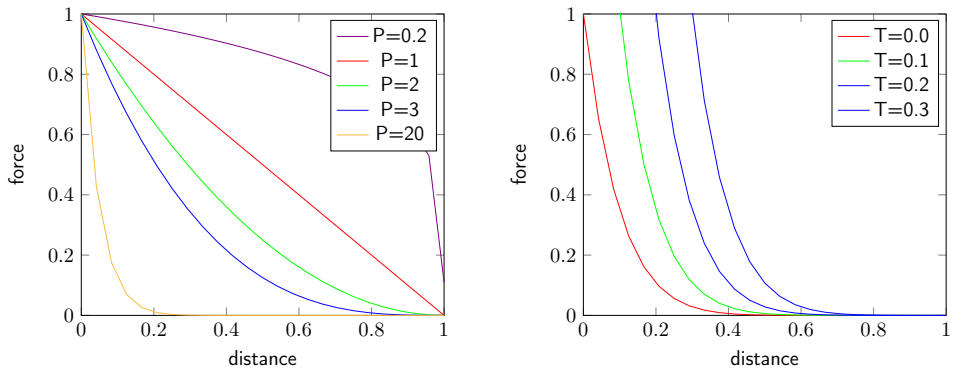
The force on a particle affects its velocity, and the velocity affects its position:

$$\text{velocity}_i := \text{velocity}_i + \text{force}_i \cdot dt \tag{2}$$

$$\text{position}_i := \text{position}_i + \text{velocity}_i \cdot dt \tag{3}$$

1.2.1 Magnitude of force

I chose a simple parametrized formula for the magnitude of force between two particles. There were two conditions this formula had to fulfill: a) it had to decrease to zero within a defined



(a) Changing the Force Power (P) with $F = 1$, $D = 1$, $T = 0$.

(b) Changing the Distance Threshold (T) with $F = 1$, $D = 1$, $P = 10$.

Figure 1: Changing Parameters in the Force Formula.

radius, such that particles further away than this distance would not interfere with each other, and b) it should configurable such that the formula could either approximate an elastic collision or a continous force without impact.

$$\textbf{force}(x) = \begin{cases} F \cdot \left(1 - \frac{x-T}{D}\right)^P & \text{for } 0 \leq x \leq D + T \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Where x is distance between the two particles, F a constant factor to apply to the force, D the minimum distance for influence between two particles, T the threshold that is subtracted from the distance, and P the power of the force.

The above formula with its four simulation constants (F, D, T and P) proved to be sufficient for the above mentioned conditions. Figure 1a displays the effect of the power P to the function. For higher values of P, the curve bends down, such that particles that are much closer to each other affect each other much more than those further away. In Figure 1b, one can see a change in the distance threshold T, that basically changes the radius of the particles. With $T > 0$ and $P > 10$, an elastic collision can be modeled, whereas other configurations result in "smooth" repulsion.

For later simulation, the following configuration proved successful.

D = 0.001 T = 0.06 P = 1 F = 20

1.3 Meshes

A mesh is a simple polygon, consisting of segments between its points.

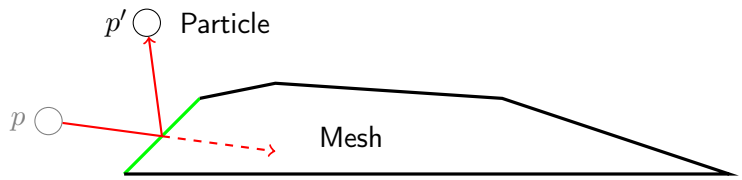


Figure 2: Particle colliding with a mesh fragment.

Before each position update of a particle (movement), collision with every line segment (fragment) of every mesh is checked. For that, the point of intersection between the line of movement of the particle i and the fragment f is calculated.

$$\text{position}_f + k_1 \cdot \text{direction}_f = \text{position}_i + k_2 \cdot \text{velocity}_i$$

This equation can be resolved for k_1 and k_2 as follows:

```
void intersect(const Vector& p1, const Vector& d1, const Vector& p2, const Vector& d2, double& k1, double& k2) {
    k1 = ((p2.x-p1.x) * d2.y - (p2.y-p1.y) * d2.x) / (d1.x * d2.y - d1.y * d2.x);
    k2 = ((p1.x-p2.x) * d1.y - (p1.y-p2.y) * d1.x) / (d2.x * d1.y - d2.y * d1.x);
}
```

If the point of intersection lies within the iteration's travel distance of the particle ($0 < k_1 \leq dt$) and on the mesh line ($0 < k_2 \leq 1$), the velocity is updated by reflection on the fragment normal and multiplication with a damping factor, and the particle is moved to the reflected point:

$$\text{position}_i := \text{position}_i + \text{velocity}_i \cdot (dt \cdot k_1) + \text{new\_velocity}_i \cdot (1 - dt \cdot k_1)$$

Furthermore, the mesh m receives an impulse in the direction of the fragment's normal:

$$\text{force}_m := \text{force}_m + \text{norm}_f \cdot (\text{velocity}_i \cdot \text{norm}_f)$$

2 The Simulator

The simulator, *FluidSim*, is the part of the project that applies above mentioned model to a set of particles and meshes inside a Domain. The simulator is an independent executable that can take the simulation parameters from the command line and outputs the generated particle data into a directory, one file per iteration. It is implemented in C++11 using OpenMP and OpenMPI.

2.1 Set implementation

In many parts of the program, sets of objects are used to work with. Since the order of these is usually not important, I implemented my own Set class (`QuickSet`), to be able to fine-tune for performance. The QuickSet supports common set operations (`insert`, `get`, `remove`, `clear`) as well as guarantees contiguous memory layout and pointer access. This is important to be able to send the data from a QuickSet via MPI.

The QuickSet internally allocates a fixed-sized buffer (whose size is configurable via command line parameter `-B`/`-buffer`). Opon insertion, the element is copied directly into this buffer using `memcpy`, for deletion only the element count is reduced and the last element is moved to the deleted element's index. This ensures contiguous memory layout without the need to move the remaining elements forward by one index.

2.2 Parallelization

There are two types of parallelization used in FluidSim. Since each particle has to be updated every iteration in a simple `for` loop, threading with OpenMP was a trivial implementation. Every process owns a distinct set of particles, and one `#pragma omp for` was nearly enough to parallelize this.

However, for cluster parallelization, a message passing implementation was required. In order to do so, the overall particle domain is divided into a grid with n cells, where n is the number of processes (see Figure 3a). Each process only updates its own particles using the positions of its own and its neighbours particles, and sends the updated positions to every neighbour.

Since only the neighbour cells are taken into account when calculating particles, two particles that are further away than one grid cell's size must not influence each other. Therefor particle/force model was designed such that such a maximal influence distance was given (see above).

When a particle moves into a different domain cell, upon receiving the new particle the target process inserts the particle into its own set, and the sending process removes it.

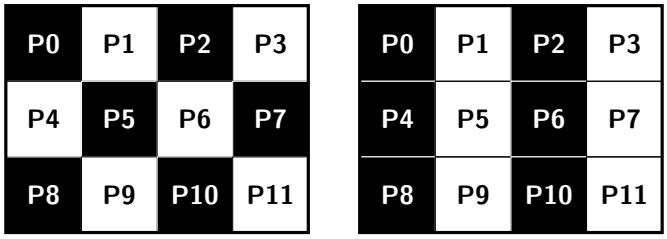
	Mode	Sending	Receiving	Directions
1.	Checkerboard	black	white	N, E, S, W
2.	Checkerboard	white	black	N, E, S, W
3.	Stripes	black	white	NE, SE, SW, NW
4.	Stripes	white	black	NE, SE, SW, NW

Table 1: Synchronization transactions

2.3 Synchronization

This section describes the scheme implemented that supports grid cells to synchronize with every neighbour. I designed this scheme such that any arbitrary sized grid could be synchronized in a fixed number of transactions (steps). The required transaction count turned out to be 16, two directions for eight neighbours. Every cell, except border cells, have to perform each of these transactions, as such, the scheme is very efficient.

To visualize the scheme, the grid may be "colored" in two different ways, see Figure 3. The coloring then determines in which step a process is either sending or receiving. Table 1 is a list of all transactions, in which the coloring mode is described, as well as which color describes which instruction.



(a) Checkerboard (b) Stripes

Figure 3: Modes of Domain coloring

2.4 Data output

Figure 4 describes the output format of the particle data. One file with this format is generated for every iteration. The first 32 bytes are the header, containing basic information about the iteration. Following this header, each process writes all of its particle positions in order, then all of its particles' position and velocity vectors. The offset of each process' data is calculated first, then propagated to the processes themselves, and then they write their chunk of data using MPI-IO.

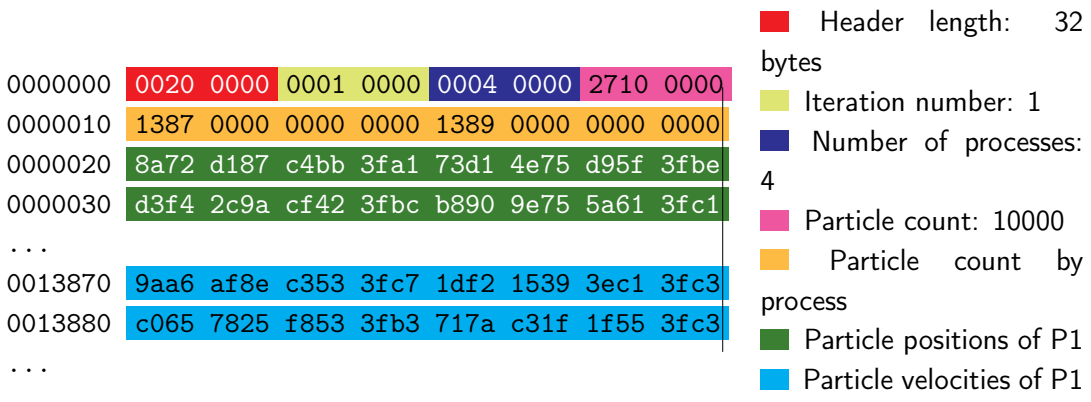


Figure 4: Iteration output file format

3 The Visualizer

3.1 Basic Technology

The visualizer (*FluidVis*) is the part of the project that takes the generated data from the simulator as input and displays them for analysis. Mostly, it is an OpenGL application, using SFML for simple rendering as well as window and input management.

This application is not very complicated or smart, just loading all of the data into memory. However, it has a few useful features, such as displaying the domain grid, coloring the particles by different features (process number, velocity, ...) and *Live* mode, where it reads the status from the master process and always loads the latest available iteration.

3.2 Data Transfer

While there might have been a number of possibilities to send live data from the simulator to the visualizer, such as using MPI itself, or some kind of networking or socket implementation, I chose a rather simple version, to keep the simulator clean.

The simulator's main process writes a `status` file at the end of each iteration, with some metadata, mostly the iteration count and grid size. The visualizer repeatedly reads this file and then opens the corresponding iteration file. For the simulation performed on a remote machine or cluster, I decided to use the same technique, just mounting the remote file system to a local path, so the implementation would be the same.

This worked out quite well, except that the internet connection to the student test cluster limited the speed at which I was able to inspect the generated data. Downloading it was actually much slower than the simulation, so I consider the simulation to be able to run in real-time.

4 Results

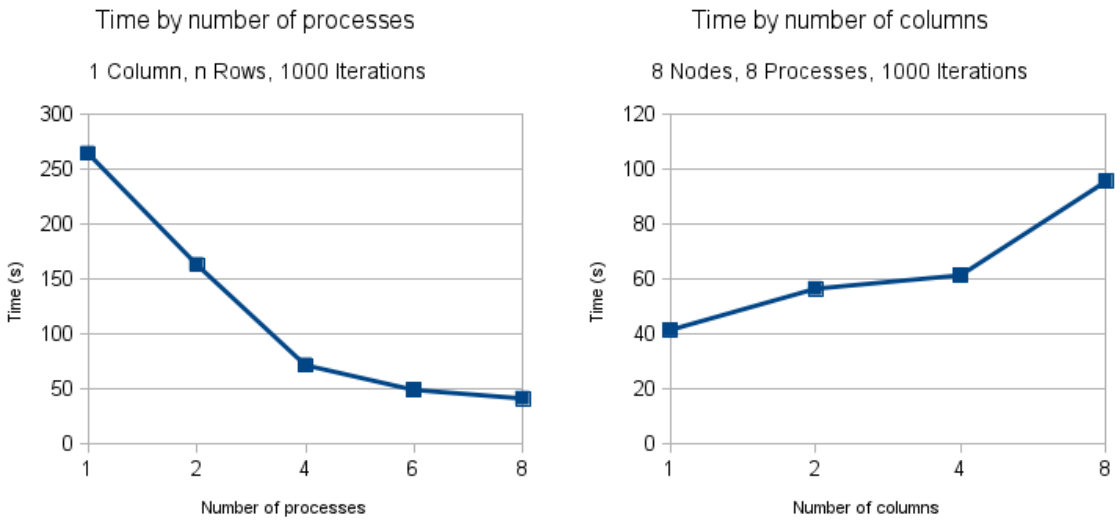
After implementing the simulation I took some time to play around with the

4.1 Performance

For performance measurement, I implemented a special simulator mode (`-M`/`-measure`) in which the processes do not write their data to the hard disk. The graphs in Figure 5 are created using this mode.

The speedup by increasing the number of processes is very much the expected result. There is of course some communications overhead added by increasing from one to two processes, which explains why two processes take noticably longer than half the time of one. However, from there on, the speedup is proportional to the number of processes used.

An interesting measurement was the time by grid size. Figure 5b shows some difference between a 1x8 grid and an 8x1 grid. Even though the communications scheme does not differentiate between the two axis, the 8x1 grid is much slower. This is due to the particles moving mainly in horizontal direction, and particles that cross the boundaries of grid cells have to be `removed` from one set and `inserted` into another, both of which takes more time than just updating its position.



(a) Time by number of processes

(b) Time by number of columns

Figure 5: Performance measurements on the student cluster

4.2 Difficulties

In general, there were not many problems creating and realising this project. However, there were a few pitfalls I encountered, both implementation and modelling difficulties.

It was somewhat difficult to get MPI-IO working, especially when I miscalculated some seek offsets and confused absolute with relative offsets. This led to some weird artifacts in the visualizer, since the data was offset and velocities and positions were mixed up. It took some time to realize the problem was in the output routines of the simulator, not the actual model or the visualizer.

I also happend to make some mistakes when initializing buffers, resulting in bad pointers or values. I could detect that this was my mistake when I printed some pointer values as hex and received `0xdeafbeed`. This is a variation on `deadbeef`, which happens to be the value uninitialized memory cells get on some systems. Until I figured this out, many assertion errors and segmentation faults were thrown.

On the mathematical part, I thought that 2D collisions of simple particles with polygons were going to be trivial, but it took quite a lot of time to get working, too. Linear algebra is often more complex than it looks.

4.3 Problems

There are still some problems with the software. Especially on the performance part, while the simulator is optimized for run-time (and this is further enhanced by the compiler). However, the memory usage of the simulator is way from optimal, there are lots of buffers that are not required or could be used more efficiently. This makes it impossible for me to test with more than 6 processes on my 8GB machine.

The set implementation I created for my particle collection uses `memcpy`. While this is the best possible way I found to manage dynamic collections, it is still a very slow process, especially when changing multiple items (inserting/deleting). There might be complex and specialized structures that could work better, but these would most probably not result in a coherent dataset, which would slow down transmission via MPI.

5 Conclusion

The most important fact to consider wrapping up is that I have reached my goal. The simulation works, I was able to implement the model in a sufficiently fast way, and simulate an aircraft wing with it. The simulation is even faster than the transmission to my visualizer, so I consider the simulation able to run in real-time.

While the software I wrote is not flawless and has some space for optimization left (such as load balancing and SIMD instructions), I learned a great deal about parallelization and how to structure a project of this kind.