4
Computational Framework

In this chapter we will discuss the main issues involving computational efficiency and implementation. The results of performance were obtained in a machine with the following specification:

- Intel® Core™ i7 CPU X990 @ 3.47GHz 2.79GHz
- Installed memory: 24.0GB
- Windows 7 64 bits

4.1
Implementation

This work was developed in C++ (26, 24) using many libraries with C and C++ API. API stands for *Application Programming Interface* and is a set of rules – functions, classes, methods – to enable software components to communicate with each other. In our case, the software is communicating with the libraries – or libs. The libs are used basically to add to the software an interface, a linear system solver, sparse matrix manipulation and 3D drawings.

4.1.1
User Interface

The interface was developed with IUP (22) and OpenGL (25). OpenGL was chosen to enable the drawing of 3D objects – the domain, particles and so on.

4.1.2
File Formats

The software reads and writes binary and text files for different purposes. The text files are to set the data of the simulation. The parameters of the fluid, such as ρ and μ, the boundary conditions – moving wall or no tension – and the particles parameters. These files are saved in Lua (20) format to enable easy additions of parameters on the file and to store data in tables. The user can modify these files outside the software, as it is straightforward to understand.

The simulation results are stored in binary files not to lose numeric precision. Besides, it takes less time to write and read and also avoids the user to modify it. Only the software can read or write these files.

4.2 System Solving

The Jacobian matrix that arises in the Newton Method is a sparse matrix and it is assembled with the triplets $(i, j, value)$, where i is the row index, j the column index and $value$ the value itself. The triplets can be duplicated, what speeds up the matrix assembly. We implemented a C++ class to hold the sparse matrix and deal with the solver libraries. It is commonly called a wrapper, because the rest of the code does not need to worry about the solver. It just tells the class to solve the system and it does the job internally.

The matrix has the dimension $N_{DOF} \times N_{DOF}$, where N_{DOF} is the total number of degrees of freedom.

$$N_{DOF} = 6N_{nodes} + 4N_{elems} + 6N_{parts} \tag{4.1}$$

N_{nodes} is the total number of nodes (with 6 DOFs: 3 of $\overrightarrow{V}$ and 3 of $\overrightarrow{\lambda}$), N_{elems} the total number of elements and N_{parts} the total number of particles (with 6 DOFs: 3 of $\overrightarrow{V}_{P_k}$ and 3 of $\overrightarrow{\omega}_{P_k}$).

As the mesh used is regular, the number of nodes and elements is the product of the number of nodes and elements in each direction.

$$N_{nodes} = N^x_{nodes} N^y_{nodes} N^z_{nodes} \tag{4.2}$$

$$N_{elems} = N^x_{elems} N^y_{elems} N^z_{elems} \tag{4.3}$$

The number of nodes in x, y and z directions are, respectively, N^x_{nodes}, N^y_{nodes} and N^z_{nodes}. They depend on the number of elements in each direction, N^x_{elems}, N^y_{elems} and N^z_{elems}.

$$N^\star_{nodes} = 2N^\star_{elems} + 1 \tag{4.4}$$

where $\star$ denotes the directions x, y or z.

The dimension of the Jacobian matrix may be large enough to make it important to find an efficient solver. Our first approach was to try the iterative solver library IML++ (1). The same one used by Lage (21). We tried all the solvers provided, however we did not manage to make it converge to a solution. Even for non-particulate flows.

Then we tried a direct solver library: UMFPACK (9, 8, 11, 10). It works fine for non-particulate flows and for particulate flows with coarse meshes – up

to $(6\times6\times6)$ elements. For more refined meshes we compiled the code in 64 bits, because it required more than 2GB to be allocated. It works, fine for meshes up to $(11\times11\times11)$, when the system solving takes more than 2 hours, see fig 4.2. The band of the Jacobian matrix explodes as the last $DOFs$ are the $DOFs$ of the particles. So we have the last $6\times N_{parts}$ rows and columns with nonzero values at any column or row respectively. This is due to the movement of the particles through the mesh. We did not use reordering algorithms because it would have to be used at every time step. So, we focused on solving this bad shaped Jacobian.

However we use a direct solver, an iterative one would be better. It is recommended as future work to look for an iterative solver that deals with this Jacobian.

4.3 Jacobian Sparsity and DOF numbering

The sparsity of the Jacobian matrix was studied in order to find a way to speed up the system solver. The non-zero values of the global matrix –for a mesh $5\times5\times5$ and one particle– are depicted in figure 4.1. Observe the this matrix is banded except for the last columns and lines. In these columns and lines, there are the $DOFs$ of the particles ($\overrightarrow{V_{P_k}}$ and $\overrightarrow{\omega_{P_k}}$).

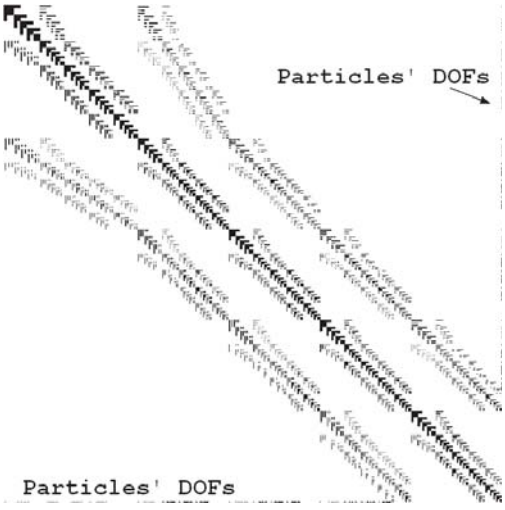

Figure 4.1: Global Jacobian matrix for a mesh 5x5x5 with one particle. Notice the bottom lines and the right hand side columns.

4.3.1 System Partitioning

Taking the sparsity of the Jacobian into account, we partitioned it into four sub matrices to try to speedup the solver. We decomposed the Jacobian J into

the matrices J^{11}, J^{21}, J^{12} and J^{22}.

$$J = \left[\begin{array}{c|c} J^{11} & J^{12} \\ \hline J^{21} & J^{22} \end{array}\right] \tag{4.5}$$

Consider a 5×5 matrix J for simplicity. The system to solve is $Jx = b$

$$\left[\begin{array}{ccc|cc} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} \\ J_{21} & J_{22} & J_{23} & J_{24} & J_{25} \\ J_{31} & J_{32} & J_{33} & J_{34} & J_{35} \\ \hline J_{41} & J_{42} & J_{43} & J_{44} & J_{45} \\ J_{51} & J_{52} & J_{53} & J_{54} & J_{55} \end{array}\right] \left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ \hline x_4 \\ x_5 \end{array}\right] = \left[\begin{array}{c} b_1 \\ b_2 \\ b_3 \\ \hline b_4 \\ b_5 \end{array}\right] \tag{4.6}$$

or

$$\left[\begin{array}{c|c} J^{11} & J^{12} \\ \hline J^{21} & J^{22} \end{array}\right] \left[\begin{array}{c} x^1 \\ \hline x^2 \end{array}\right] = \left[\begin{array}{c} b^1 \\ \hline b^2 \end{array}\right] \tag{4.7}$$

With the previous partitioning we have the following relations:

$$J^{11}x^1 + J^{12}x^2 = b^1 \tag{4.8}$$

$$J^{21}x^1 + J^{22}x^2 = b^2 \tag{4.9}$$

$$\tag{4.10}$$

We can solve for x^1 and obtain x^2 by matrix computations. One important characteristic of our Jacobioan matrix is that the last $6N_{parts}$ rows and columns represent the DOFs of the particles. They are clustered in the sub matrices J^{21}, J^{12} and J^{22}. The sub matrix J^{22} is diagonal – see Appendix B.4 – then it is easily inverted. Solving for x^1 we have:

$$[J^{11} - J^{12}J^{22^{-1}}J^{21}]x^1 = b^1 - J^{12}J^{22^{-1}}b^2 \tag{4.11}$$

and to compute x^2 we have:

$$x^2 = J^{22^{-1}}b^2 - J^{22^{-1}}J^{21}x^1 \tag{4.12}$$

4.3.2 Performance Results

We ran the problem with one particle and different meshes to compare the performance between the solver with the partitioning and without it. The

meshes range was from $3 \times 3 \times 3$ to $11 \times 11 \times 11$, always keeping the same number of elements in each direction.

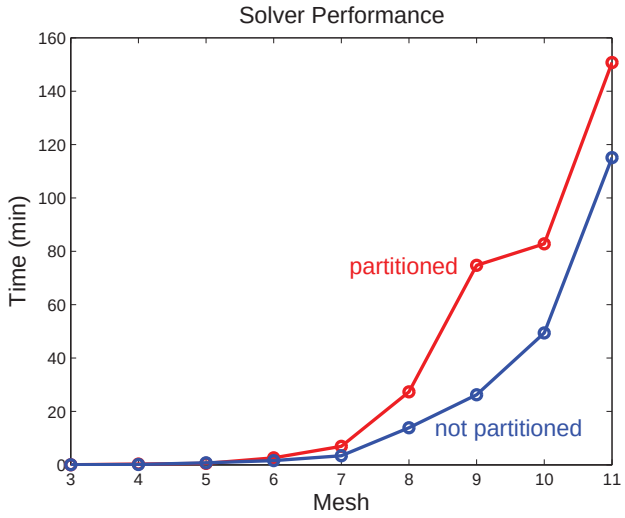

Figure 4.2: Elapsed time in the solver with partitioned Jacobian matrix and not partitioned

After running the problem with both approaches, we found out that partitioning was not really speeding up the process. So we remained using the non partitioned Jacobian. Even the best result that we had was not good enough. The time is still very high for the system solving, what makes it unfeasible. We should take into account that for each time step, the system must be solved for each iteration in the Newton Method. So, the system is solved at least once for each time step.

4.4 Regular Mesh Generation

In this work we use regular meshes on the whole domain. The domain is always a 3D box with its 6 faces. The mesh generation follows successive interpolations as in the next algorithm.

```
1  Point3i I(0);                    /*3D integer iterator          */
2  Point3d p00z, p10z, p11z, p01z;/*bilinear surface vertices   */
3  Point3d p0yz, p1yz;              /*straight line vertices       */
4  Point3d pxyz;                    /*3D point                     */
5  Point3d d00z, d10z, d11z, d01z;/*steps for the surfaces       */
6  Point3d d0yz, d1yz;              /*steps for the straight lines*/
7  Point3d dxyz;                    /*step for the point           */
8
9  /*Initialize the first bilinear surface*/
10 p00z = p000; p10z = p100; p11z = p110; p01z = p010;
11
```

```
12  /*steps in each direction*/
13  Point3d d = Point3d (1.0/static_cast<double>(nnodes.x-1),
14                       1.0/static_cast<double>(nnodes.y-1),
15                       1.0/static_cast<double>(nnodes.z-1));
16  d00z = (p001-p000)*d.z;
17  d10z = (p101-p100)*d.z;
18  d11z = (p111-p110)*d.z;
19  d01z = (p011-p010)*d.z;
20  int global_id = 0; /*global id of the node*/
21  for (I.z=0;I.z<nnodes.z;++I.z)
22  {
23   p0yz = p00z;
24   p1yz = p10z;
25   d0yz = (p01z-p00z)*d.y;
26   d1yz = (p11z-p10z)*d.y;
27
28   for (I.y=0;I.y<nnodes.y;++I.y)
29   {
30    pxyz = p0yz;
31    dxyz = (p1yz-p0yz)*d.x;
32    for (I.x=0;I.x<nnodes.x;++I.x)
33    {
34     nodes.push_back(new Node(pxyz,global_id++));
35     pxyz += dxyz;
36    }
37    p0yz += d0yz;
38    p1yz += d1yz;
39   }
40   p00z += d00z;
41   p10z += d10z;
42   p11z += d11z;
43   p01z += d01z;
44  }
```

From the 3D box, we generate bilinear surfaces, as in figure 4.3. To define this surface, we only need 4 points ($p00z$, $p10z$, $p11z$ and $p01z$). Then, straight lines are generated ($p0yz$,$p1yz$) in the surfaces and, finally, points are interpolated in all lines. This algorithm can take any set of six points enclosing a concave volume. The faces of the box do not need to be parallel, neither plane. They can be bilinear surfaces. However, in this work, we only use plane faces and parallel to their corresponding pair.

The reader may have noticed the classes Point3d, Point3i and Node. Point3i is basically a holder for 3 integers (x, y, z). Point3d is a holder for 3 real numbers (x, y, z) and provides methods to manipulate it such as a

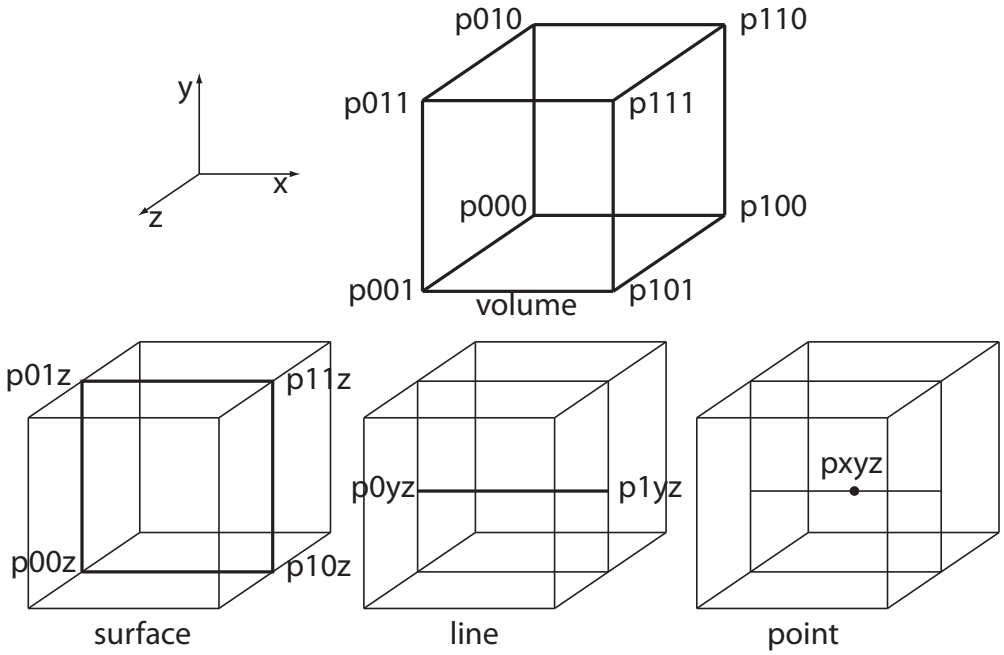

Figure 4.3: The steps of mesh generation. From the volume, we generate surfaces, then lines and points.

3D vector. Node derives from Point3d and have also other attributes, such as *global_id*, that identifies it among other nodes. It also has informations about boundary conditions and particles.

The variables nnodes.x, nnodes.y and nnodes.z correspond to N^x_{nodes}, N^y_{nodes} and N^z_{nodes}. They are given as well as the points that define the volume ($p000$, $p100$, $p110$, $p010$, $p001$, $p101$, $p111$ and $p011$).

4.5 Parallelization

Parts of the code were parallelized with *OpenMP* (7). We used the *pragma* directive of compilation for a *for* loop:

```
#pragma omp parallel for
for (int i(0);i<n;++i)
{
 /*for loop body...*/
}
```

This gave us some speedups, however the bottleneck was not parallelized: the Jacobian assembly. It could be parallelized, but to do so, some changes should be done. Our Jacobian Matrix should be partitioned into as many blocks as threads, but we didn't focus on that. We tried to parallelize the assembly

without taking proper care of this aspect, but no substantial speedup was gained. Besides, we considered it out of the scope, so we left this for future works.