Domain Decomposition for Colloid Clusters

Pedro Fernando Gómez Fernández

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2004

Authorship declaration

I, Pedro Fernando Gómez Fernández, confirm that this dissertation and the work presented in it are my own achievement.

1. Where I have consulted the published work of others this is always clearly attributed;
2. Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;
3. I have acknowledged all main sources of help;
4. If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;
5. I have read and understand the penalties associated with plagiarism.

Signed:

Date:

Matriculation no:

Abstract

This project involves the creation of a 'toy' model that will simulate particles moving in a liquid using the 'lattice-Boltzmann' method. The model will have to implement domain decomposition and distributed data using the MPI library for a two dimensional problem.

Particles experience forces from the liquid that is represented by a lattice and from interaction with other particles.

Problems arise when particles have to check for interaction forces with particles in surrounding subdomains and when they have to move from one subdomain to a contiguous one.

All the algorithms needed to create a 'toy' model for this problem will be described.

At the end, an implementation of the problem in C language is created and tested checking serial result against parallel results.

Table of Contents

Illustration Index

1. Introduction

1.1 Solid-Fluid Mixtures

The physical problem that will be considered in this project is that of solid (colloidal) particles suspended in a fluid, or mixture of fluids. The solid particles are usually larger than the molecules that make up the fluid; typically their size is about few μm. Real systems will be made up of many particles.

There are a lot of important applications of this model, for example: paints, cosmetics and food. So a software model to simulate this physical problem if of great interest. As this kind of problem will require high computation, a software model that can be parallelized among several processors will be required.

1.2 The Lattice Boltzmann Method

The method that will be used to model this physical problem in a computer will be the Lattice Boltzmann Method[1]. The aim of this method is to solve the Navier-Stokes equations for fluid dynamics. This method is based on a regular discrete lattice at which the fluid properties are computed. This lattice could be a 3 dimensional lattice as the one shown in figure 1. But in this project, a 2 dimensional one will be used for simplicity as the one shown in figure 2.

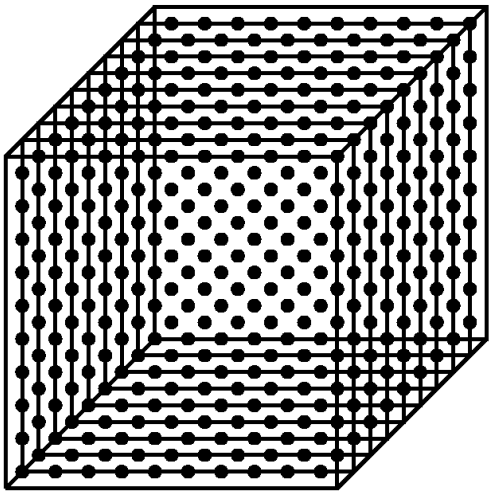

Figure 1: 3D Lattice.

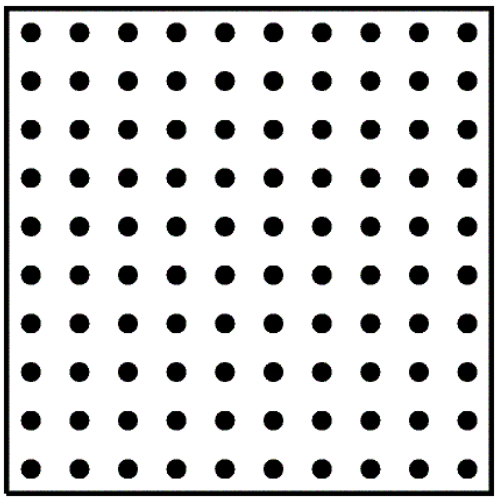

Figure 2: 2D Lattice

In this method, the lattice spacing between points is a fixed Δx, and in time fluid properties are computed at discrete time steps Δt.

The Lattice Boltzmann method has a lot of convenient features. It has the ability to represent moving objects that will be useful to represent the solid particles suspended in the fluid as will be see in the next section. All the operations are local so it is good for parallelization and in addition, only halo swaps between neighbouring sites are required for domain decomposition.

1.3 Solid Particles in Lattice Boltzmann

The solid particles suspended in the fluid [5, 6] can be represented as solid spheres (or circles in the 2 dimensional problem) which have a radius of some Δx (lattice spacings) and a position and a velocity that will allow them to move smoothly across the lattice. This means that the centre of the solid particle can be positioned anywhere in the lattice and not only in the lattice points. A single particle is shown in figure 3. The lattice points inside the particle are considered solid points while the lattice points outside the lattice are considered liquid points.

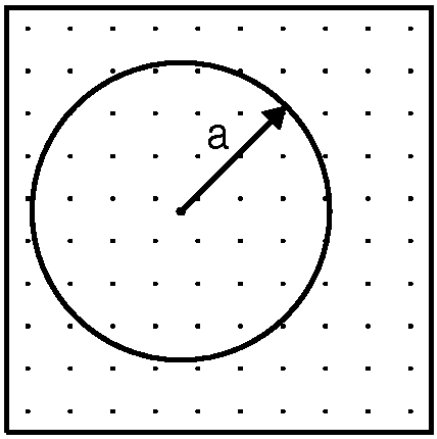

Figure 3: A single particle of radius a. The centre of the particle moves continuously across the lattice.

For each particle, there is a solid-fluid boundary condition that is included by identifying 'links' which connect solid to fluid lattice points. From that, a discrete shape for each particle is obtained; this will change as the particle moves in the lattice. The force between the particle and the fluid will be calculated by adding contributions from all the links around the particle. The new velocity and position of the particle for that time iteration can be calculated from the resulting forces.

The long range hydrodynamic interactions between the particles are represented by the Lattice Boltzmann fluid, but when the particles are very close together, additional forces may have to be added explicitly. This will give rise to a molecular dynamics-like problem which involves the particles only.

For example, representation of lubrication forces between the particles [4] requires an additional force at close range which depends on the gap 'h' between the particles and the difference in particle velocity. As the size of the force scales 1/h, it can become very high as the particles approach to touching. This fact will require a special treatment of the velocity update of the particles when they are very close together. This situation for two particles is shown in figure 4.

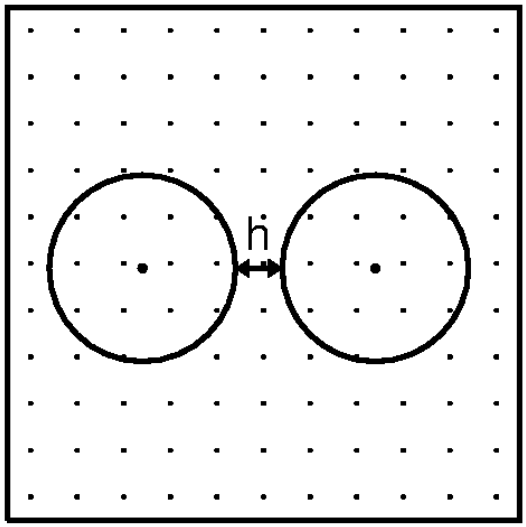

Figure 4: 2 particles close together where the gap 'h' between them is very small.

This last situation can involve a large number of particles, this will be referred as the cluster problem. It is shown in figure 5. Here, the special treatment of the velocity update will require that all particles in a cluster will be treated together [4].

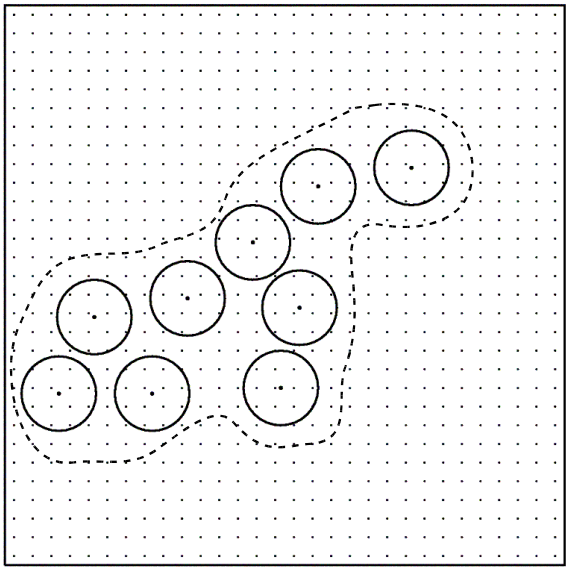

Figure 5: Cluster problem: several particles near each other forms a cluster.

1.4 Project Aim

The aim of this project is to produce a working version of the particle problem described using domain decomposition and message passing via MPI [7]. As the full Lattice Boltzmann problem with hydrodynamic interactions is complicated[3], this

project will concentrate only on a 'toy' model which involves only a basic representation of the fluid lattice and the particles. However, all the relevant communications will be considered.

The constraint is that the fluid lattice will be subject to regular domain decomposition because this is the decomposition for the fluid lattice.

There are a number of main problems for particles:

- Data distribution and decomposition.
- Updating the data as particles move across the lattice.
- Computing the sum of force contributions around a particle (which may involve up to 8 processors in 3 dimensions and up to 4 processors in 2 dimensions).

The analysis can consider 2 dimensions and 3 dimensions; but code will be 2 dimensions as there are no significant new issues in moving it to 3 dimensions.

The aim is to identify an elegant way to do the problem. Absolute efficiency cannot be assessed as full fluid problem is not involved.

2. Analysis of the Problem

This project consists in the creation of a 'toy' model that will simulate solid particles in a liquid. The program will distribute the work involved with both fluid and particles by domain decomposition using the MPI library.

This problem involves the simulation of the movement of particles inside a liquid. For simplicity, the problem is reduced to 2 dimensions. The liquid is represented by a lattice which has regularly distributed points. This is the Lattice Boltzmann model. This lattice will have periodic boundaries. That means that if a particle arrives to a boundary of the lattice, the solid particle will appear in the opposite boundary.

Before going on to discuss the details of the solution, this section provides an analysis of the different problems involved.

2.1 Fluid only problem

For the fluid only problem, all the lattice points will have to be updated (see figure 6 which is the whole lattice to be computed). It seems that domain decomposition will work perfectly for this problem because each lattice point has the same computing load. The region is rectangular and the computation of each lattice point only depends on the local point and the adjacent lattice points.

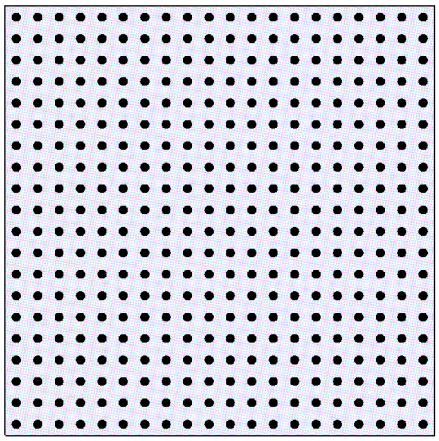

Figure 6: Lattice without particles

It is very easy to divide the whole lattice in smaller lattices for each subdomain (see figure 7 for a 2X2 domain decomposition where the whole lattice has been divide in 4 subdomains). This will be a good decomposition as every subdomain will have the same load, so it will be a load balanced decomposition.

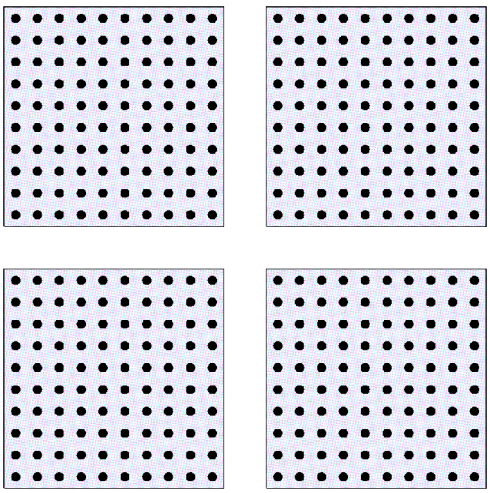

Figure 7: Domain decomposition of the lattice

As MPI will be used, the most direct way to deal with the domain decomposition is the use of halos in each subdomain. The halos will contain the information about the adjacent lattice points. They will be swapped at every time iteration by means of MPI messages in every dimension of the lattice. (See figure 8, where subdomains with the halos and halo communication are shown). The program will swap the halos, compute an iteration and repeat this process again until all the iterations required are performed[1].

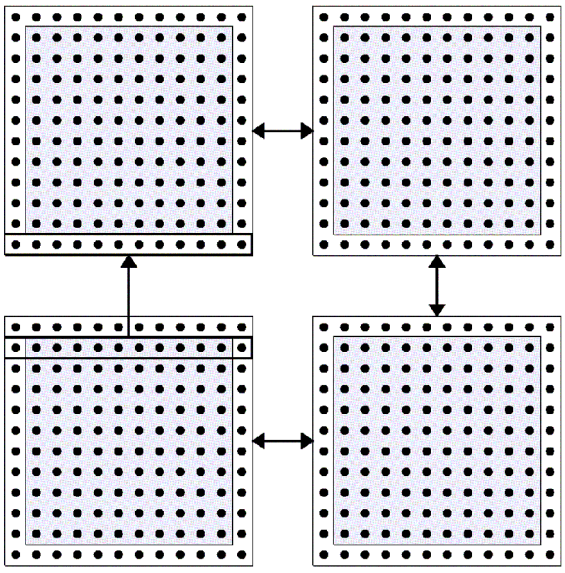

Figure 8: Halo communication of four subdomains in two dimensions.

1 It is possible to use bigger halos and then more than one iteration between communications will be possible with a more complex code, but as will be seen later, when particles will be added to the problem, a communication will be required after each iteration.

It is important to notice that the lattice will have periodic boundaries, so the halos at a boundary will be swaped by the subdomain in the opposite side of the lattice.

Another important thing is that the computation at a lattice point will depend on the lattice points around it, even in the diagonal directions, so it seems that a diagonal communication will be needed. However, that is not true, as the two dimensional communication will also carry this information. This can be see in detail in figure 9: the upper right subdomain will need the upper right lattice point of the bottom left subdomain which is shadowed in the picture. In the vertical communication, represented by the arrow labeled with '1', the shaded point is copied to the upper left subdomain halo. Then in the horizontal communication represented by the arrow labeled with '2', the shadowed point is copied to the correct position in the halo of the upper right subdomain. So after the vertical and horizontal communications the needed lattice points at the corners have also been copied.

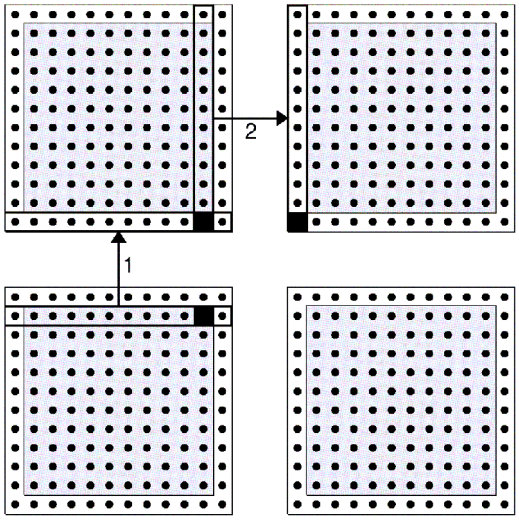

Figure 9: Corner communication. The black square is swapped first to the up subdomain and then to the upper right subdomain avoiding diagonal communication.

Notice the scaling of computation and communication as the size of the problem increases. In the two dimensional case, if a lattice of size LxL is used, the computation scales as L^2 but communications scale as L. In the three dimensional case, computation will scale as L^3 and communications as L^2. That means that the scaling of communications will be always smaller than the scaling of the computation of the problem.

2.2 The Particle only Problem

Inside this fluid lattice, some solid particles are simulated. The particles are spheres, or in this 2 dimensional simplification, circles. They can move freely in a region.

A particle is moving accordingly to the forces applied to it. This forces will appear if two solid particles are near enough, then an explicit repulsion force may be involved.

If only particles were involved (no fluid), the problem would look like molecular dynamics.

2.3 Fluid with Particles

When the fluid problem and the particle problem are added together, the program must take into account the coupling between solid and fluid. This is because in the real program, the particles will require data from the fluid lattice to compute a force. Because the fluid is not represented in the 'toy' model, the force calculation is not done. Instead, the program will calculate the number of lattice points the particle is over, it will only calculate the ones it has access to (even when it is easy to calculate this points only knowing the coordinates and radius of the solid particle). As this is a simulation of the calculation of the force from fluid, the update of the position of particles will be done after calculating these points.

There are different solutions that can be used to resolve the parallelization problem using MPI. Some possible solutions are explained in the following sections before describing the actual solution used.

2.3.1 Functional Decomposition

As this problem will be high computationally demanding for large systems with many particles, a MPI implementation to deal with the solid particles will be implemented. If the only requirement were to deal with the particles, a domain decomposition may not be the preferred solution. This is because this will not be a load balanced problem as a regular distribution of solid particles in the lattice is not guarantied. Instead of that, a better solution will be to distribute similar amounts of particles to each processor. The problem is that this program, or at least its concepts, will be integrated in a bigger program that also deals with the liquid problem. That program has a regular distribution of process as the liquid points are regularly distributed in the lattice. That is why the bigger program is load balanced and a domain decomposition using MPI is very efficient. For the combined problem, an efficient approach might have some processors for the liquid points domain decomposition problem and some others to compute the solid particles in a distributed way (see figure 10).

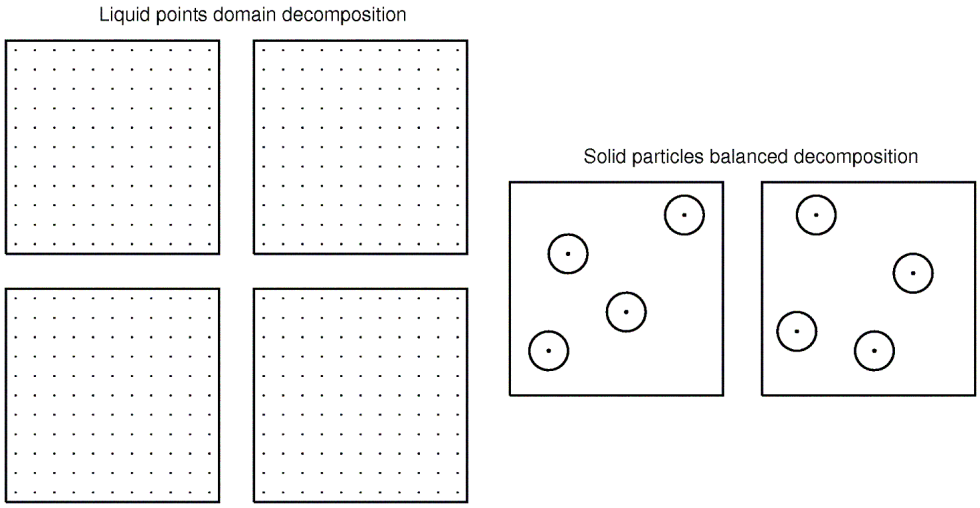

Figure 10: Functional decomposition for 6 processors. 4 processors are used for the domain decomposition of the liquid problem and 2 are used for the decomposition of the solid particles problem.

There is a need for communication between the liquid points and the solid particles, so if a solid particle 'S1' is over a liquid point 'L1', the liquid point 'L1' must know about that situation. That will require that every processor owning a subdomain of the lattice with the liquid points will communicate with each processor that is processing solid particles. So, if for example we have a subdomain decomposition of 2x2 for the liquid points and 2 processors for the solid particles, the following communications will be required:

- Among the subdomain decomposition, each subdomain will communicate with adjacent subdomains, so if the decomposition is 2x2, 2x2 (horizontal communications) + 2x2 (vertical communications) = 8 communications will be required.
- Between the sub-domain decomposition and the solid particles, 2x2x2 = 8 communications will be required.

The total communications have been doubled compared to the case when there are halo communications only.

In general, if the subdomain decomposition will take x processors and np is the total number of processors, the solid particles will take np – x = y processors. In a subdomain two dimensional decomposition of axb, 2xaxb communications will be required.

An axa subdomain decomposition will be assumed. So a = $\sqrt{x}$, and communications required will be: $Communications_{subdomain} = 2 \times \sqrt{x} \times \sqrt{x} = 2 \times x$.

Communications between the sub-domains and the solid particles process will be: $Communications_{subdomain-solidParticles} = x \times y$.

So the total communications will be:

$$Communications_{total} = 2 \times x + x \times y = x(2+y) \quad ,$$

and: $$\frac{Communications_{subdomain-solidParticles}}{Communications_{subdomain}} = \frac{y}{2} \quad .$$

From these results it is possible to see that if the number of processors is high and the ones used for particles is not too low, the communications will be considerably increased. Also, if the processors used for particles is high, the communications are as well increased independently of the number of processors. For example: for np = 512 and y = 50, communications are increased 25 times compared to the only the subdomain communications.

These results are only indicative, as it must be taken in account the amount of computing between communications, so the proportion between communications and process; but anyway these results seems to indicate that this approach is only valid when the np and the yp are both low; and in this case, a balanced decomposition for the solid particles processing is not so important.

2.3.2 Domain Decomposition

A second approach is to make a purely domain decomposition. In the domain decomposition, each subdomain will be in charge of part of the lattice and also of the solid particles in that region of the lattice (See figure 11).

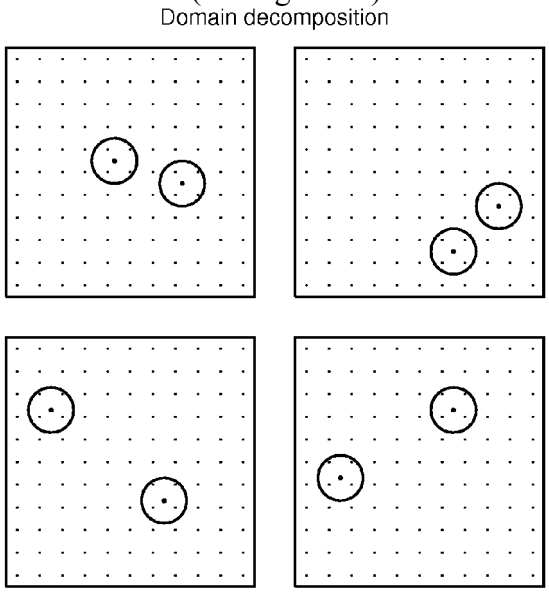

Figure 11: Domain decomposition for the whole problem.

It must be noticed that from the results in the previous section, doing a domain decomposition for the particles may not be completely load balanced. However it saves a great number of communications between processes and will be more efficient.

Some problems arise with this solution:

- At certain times, some particles will move from one subdomain to a contiguous one.
- Some times, a particle close to the boundary can appear in more than one subdomain.

Two possible solutions to handle these problems with domain decomposition are explained in the next two sections.

2.3.2.1 Replicated Data

Each solid particle has its centre in a single subdomain, but it is possible that the solid particle will cover more than one subdomain. In fact, a solid particle can cover between 1 and 4 subdomains in the 2 dimensional problem (when it is in the corner of a subdomain) at the same time. This is a problem as the liquid points will have to know about the solid particle, so each of those subdomains will have to know about the particle.

A possible solution for this problem is to use 'replicated data'. That means that each subdomain can know about all the solid particles in the lattice (See figure 12).

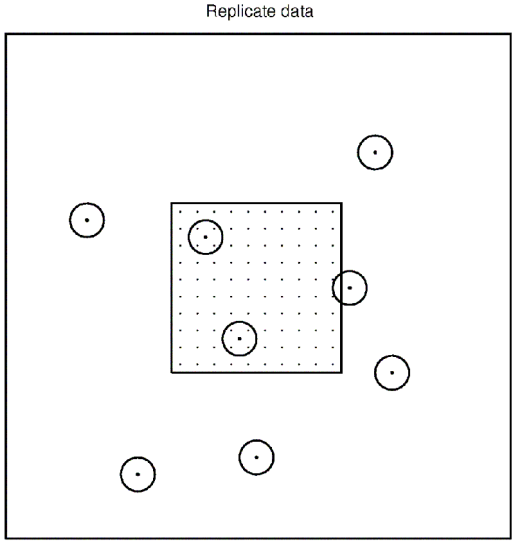

Figure 12: Replicated data. Each subdomain contains the information about a part of the lattice and all the solid particles.

This solution has the advantage that is very easy to implement as the only communication required between subdomains for the particles is a global reduction communication. This reduction communication will allow that if a solid particle obtain differ-

ent forces from different subdomains, the forces will be added before moving the particle. This communication will also allow to add all the liquid forces the solid particle has in the case that the particle is over more than one different subdomains.

But this solution has the disadvantage that each subdomain has a lot of data that does not need. All the particles in a subdomain that are not over its area will have to perform update calculations that will be replicated over all the subdomains. Also the global communications will not scale with the number of process; whatever number of processes used, the global communications will scales as N^2 where N is the number of particles.

2.3.2.2 Distributed Data

A more complex solution is 'distributed data'. This means that each subdomain only know about the particles in the subdomain and the ones that are very near to it (See figure 13).

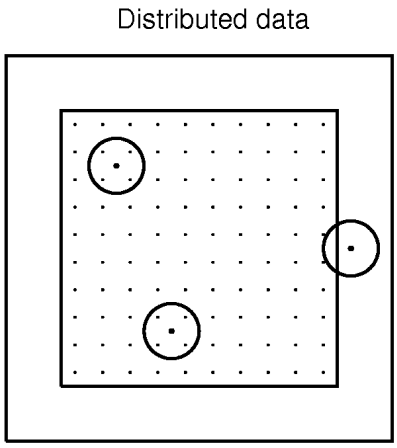

Figure 13: Distributed data. Each subdomain contains the information about a region of the lattice, the particles in that region and the ones that are very near to the region.

This will allow the calculation of the forces of each particle to be done totally or partially in each subdomain. If it can only be performed partially, a reduction communication among the subdomains that hold a copy of that solid particle is performed. The same happens in the calculation of the liquid forces the solid particle has.

This solution has the advantage that each subdomain only has the necessary information for it, but it is more difficult to implement. Also, as no global communications are required, the communications will scale according to the number of processes; for more processes, less communication required for each one.

2.3.3 Decomposition of choice

The 'distributed data' solution will be implemented in this project as the 'replicated data' solution does not scale. This will require communication between subdomains at each iteration to send the new particles that a subdomain will have at the boundaries. This communication will be mandatory as a subdomain will not be able to know if it will receive a new solid particle at the boundary or not, so it will always try to receive a message at each iteration[2].

2.4 Particle interactions

When there are two solid particles close together, they may be repulsed by an additional force added to prevent them overlapping. This repulsion force will only be applied when the two solid particles are nearer than a fixed critical gap. This critical gap will be usually about half the size of separation between lattice points. Inside a lattice, there are two solutions to detect if two solid particles are nearer than the critical gap or not; they will be discussed in the next sections.

It is important to notice that these issues are involved in the serial solution of the problem.

2.4.1 Direct method

It is possible to check each solid particle with the rest of the solid particles to determine if they are nearer than the critical gap.

There is also a problem associated with the domain decomposition and the interaction detection. This is when a particle is in a subdomain and the other is in another subdomain but they are nearer than the critical gap. So each subdomain will have to check particles in the other subdomains as well.

The problem of the direct method is the complexity of the calculation, as each solid particle will have to check with all the other particles. If P is the number of solid particles for each solid particle a check must be performed for all the other particles, that means, to check P*(P-1) solid particles. With this, the complexity of the problem is $O(P^2)$.

The good thing about this solution is that it is easy to implement.

2.4.2 Cell Lists

The solution is the use of 'cell lists'. This is a standard molecular dynamics technique [2]. In the lattice region, cells of a certain dimension will be created and solid particles will be in the corresponding cell (the one that correspond to the centre of the solid particle). The dimension of the cell will be chosen in order that for a particle in a

2 Could do this via single-sided communication, but decide not to because of uncertainty about portability and performance.

cell, that cell and the cells around it will contain all the particles that may be closer than the critical gap. This size will be two times the radius of particles plus the critical gap (See figure 14).

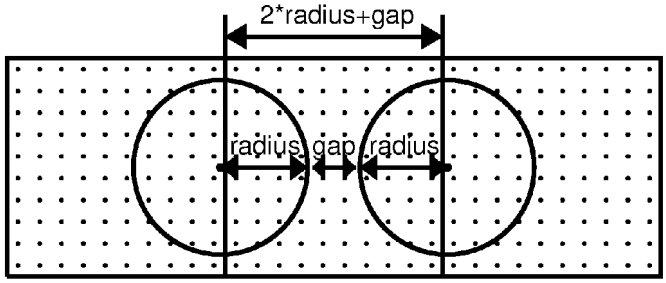

*Figure 14: Size of the cells. If size is 2*radius + criticalgap, then if two particles are not in contiguous cells, they will be further than the critical gap.*

Then each solid particle will be only checked with solid particles on the same cell and the adjacent cells to this one in order to know if they are nearer than the critical gap.

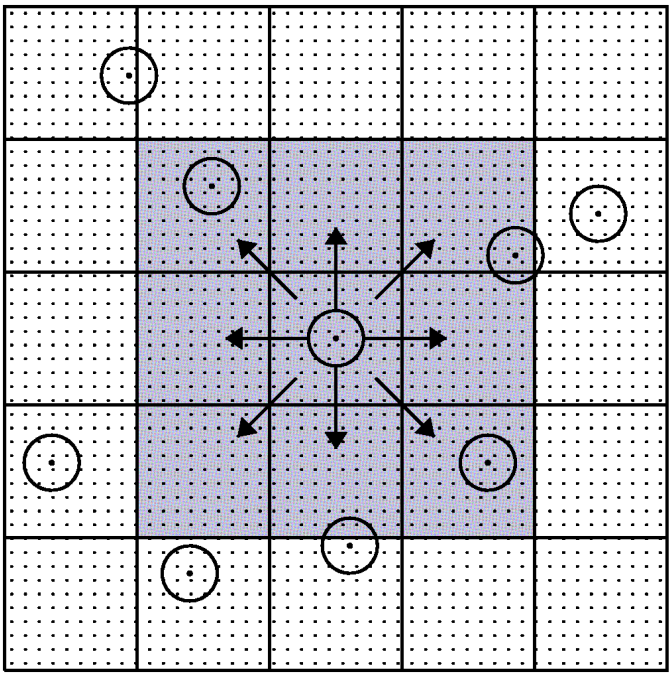

Figure 15: A particle will only have to check for interaction with other particles in the cells around its cell.

For a fixed solid particle density, the number of particles inside a cell will be the same in average, so each solid particle it will have to check for particle interaction against a constant number of neighbors. If P is the total number of solid particles, the problem has a complexity of $O(P)$. The constant number of neighbors depends on size of cells and the density of the particles.

This solution is more difficult to implement, but increases greatly performance for high number of particles.

Now, in parallel, with the cell lists method there is an easy solution for checking for interaction with particles in around subdomains. That is that each subdomain will have the information of solid particles in the same subdomain and also the solid particles that are in the adjacent cells to the subdomain from around subdomains.

Because all this the cell lists method will be used for this program.

2.4.3 Clusters of solid particles

When several particles are close to each other, the calculation will require that all the solid particles will know about all the other solid particles in the cluster (see figure 5). That will be a problem if the different solid particles are owned by different subdomains. In that case, the subdomains that have information about any of the solid particles of the cluster will have also to know about all the solid particles of the cluster.

This case will not be developed in this project. However, some analysis and design will be performed in order to study the possible implementation of this case in future. A simple algorithm for this will be explained here and will be detailed in the next section 3.5. First, in each subdomain, a search for clusters will be performed. If a cluster is found, all the particles will be saved as an object that will allow to calculate forces for the clusters. Second, before calculating the forces for the cluster, the cluster will be checked to know if it extends outside the subdomain. If so, communications will be performed in order to get the information for the whole cluster. Different ways of doing this communication are possible, they will be discussed in the next chapter. Third, with all the information, each subdomain will perform the calculation of the forces of the solid particles of the cluster.

2.5 Communications

For this program, several things will have to be communicated among the subdomains for a correct functionality. The information to be communicated is the following:

- Particles moving to another subdomain: when a particle leaves a subdomain to arrive in another one, this information will have to be communicated between the two subdomains.
- Liquid forces and interaction forces on a particle will have to be reduced among subdomains if the particle is over more than one subdomain at the same time.

- Particles outside the subdomain: a subdomain will need the information of the solid particles in the cell lists around the subdomain in order to be able to calculate all the possible interactions among the particles. These particles can be considered as the ones at the halos of the subdomain, so a halo containing the solid particles will have to be swapped among the subdomains.

In a direct approach, three different communications will be required every time step. It will be possible to reduce this to two, but to understand better this, a description of the three communications will be described before. For the correct functionality of the program the following steps will have to be followed:

1. All subdomains will swap their halos containing the solid particles in the borders of the subdomains. With this all the information needed to calculate the possible interactions among the solid particles in the subdomain is available.
2. All the solid particles not in the halos will calculate the forces due to solid particle interactions, and those due to interactions with the fluid.
3. A reduction communication will be performed in order that solid particles over more than one subdomain will have the total liquid forces of the particle.
4. The new position of the solid particles not in the halo will be calculated.
5. If a solid particle change its position to other subdomain, a communication will be performed to communicate subdomains about this fact.

In order to reduce the number of communications, the following steps can be followed instead without loss of information:

1. As before, all subdomains will swap their halos containing the solid particles in the borders of the subdomains. With this all the information needed to calculate the possible interactions among the solid particles in the subdomain is available.
2. Also as before, all the solid particles will calculate the forces due to solid particles interactions and the liquid forces of a solid particle. But if a solid particle is over more than one subdomain, it will only calculate the forces from interaction with solid particles in the same subdomain. This happens even if the particle is in the halo.
3. A reduction communication will be performed to add all the forces and number of points of the solid particles over more than one subdomain.
4. The new position of the solid particles are calculated even if they are in the halo but are partially over the lattice. If the particle leaves the subdomain, no communication will be performed as in the receiving subdomain, the same particle will be in the halo with the sames forces and it will move inside the lattice of that subdomain.

It can be seen that only two different communications will be needed: halo communication and forces communication. These two types will be described in more detail in the next sections.

2.5.1 Halo Communications

At the beginning every subdomain will contain a region of the lattice and the solid particles whose centres are in that region. In figure 16 this situation can be observed. While the solid particle 'B' can know about all the solid particles in the cell lists around it, particles 'A' and 'C' will have to know about the solid particles that would be in the halo. It can be seen that the solid particle 'C' is over two different subdomains and so a communication will have to be performed to add all the force contributions. The solid particle 'A' is completely over this subdomain and so it will have to calculate all the forces without communication.

So it is possible to conclude that a halo communication will be required always before calculating the forces of the solid particles. Even if the subdomain knew about the previous state of the halo, the communication will be needed because particles an enter and exit the halos by the external borders. Particles that are only over the halo and not partially over the lattice can also change position.

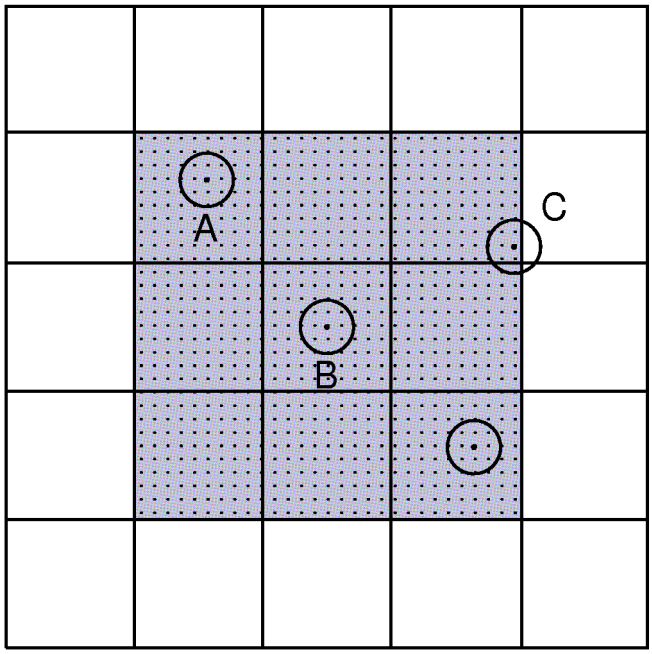

Figure 16: Subdomain before the halo swap. In the centre (the shaded region), a region of the lattice and four solid particles are stored. Around, the halo is still empty.

Once the halo communication have been performed, the situation will be the one shown in figure 17.

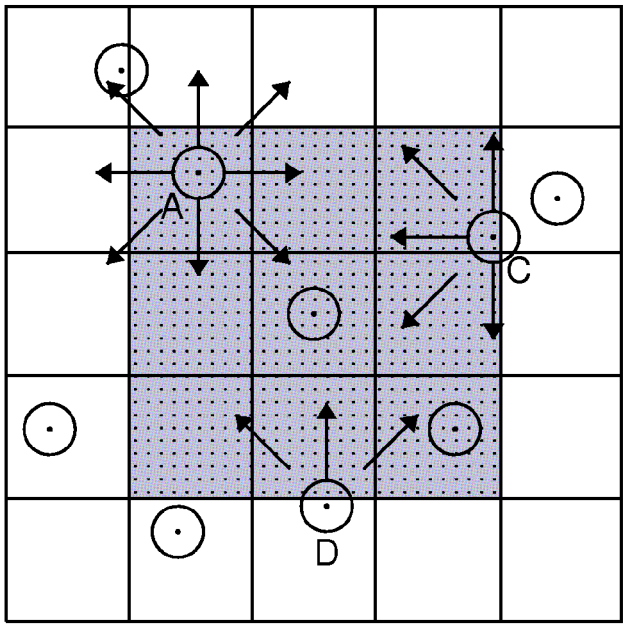

Figure 17: Subdomain after the halo swap.

In this situation, the solid particles can be classified into three groups before beginning to calculate the forces.

The first group will be the one of solid particles that are completely over the subdomain region of lattice. Then, like the solid particle 'A' they will calculate all the forces caused by solid particle interactions and will calculate the liquid forces without further communication.

The second group will be the one of solid particles that are completely over the halo of the subdomain. No forces and no liquid forces will be calculated for them. This particles will not required further communication neither.

The third group will be the one of solid particles that are partially over the subdomain lattice and partially over the halo, independent of if the centre is in the subdomain lattice or in the halo. The solid particles 'C' and 'D' belongs to this group. Particles of this group will only calculate the forces that result from interaction with solid particles that are in the lattice subdomain and not in the halo[3]. Solid particle 'D' will not calculate interaction forces with the solid particles in its own cell list as it is in the halo. These particles will calculate the partial liquid forces in the subdomain.

3 This is not absolutely true as it will see in the design chapter in more detail. Actually, a solid particle will calculate interaction forces with all the solid particles except with the ones that are in cell lists that owns to a halo that will be swappe in the same direction as that solid particle will be communicated to accumulate interaction forces and liquid forces. In practice it is almost the same as stated before but for some exceptions.

2.5.2 Force Communications

The trivial case is when a solid particle is completely within one subdomain, then the liquid points inside the solid particle will be determined by only one processor and stored in the particle. The problem arises when the particle is over more than one subdomain. When a solid particle is over more than one subdomain, it will calculate partial liquid forces in each subdomain and then communicate in order that the solid particle will gather the liquid forces of each subdomain and add this information to know the total forces on it.

It is also needed to communicate the partial forces because of solid particle interactions for the particles that are in the boundaries of the subdomains.

There are also two ways to perform this communication:

- It is possible to perform this communication at each iteration. But that means that with the previous communication, two communications will be performed at each iteration.
- As each subdomain knows about particles that are in the boundaries, the communication can only be performed when necessary as the receiving and sending subdomain will know a priori if the communication will be necessary or not. This will reduce the number of communications when no particles are at the boundaries.

The second method will be used in this project.

So now taking as a starting point the final situation explained in the previous section, there are three groups of solid particles.

First, consider particles that are completely inside the lattice region of the subdomain. No action is required for this solid particles. After forces communication for other particles is performed, their position will be updated.

Second, consider solid particles that are completely over the halo region of the subdomain. Again, no action is required for these solid particles. After this communication is performed, they will not require any position update as they will remain in the halo for the next iteration. It is supposed that if a solid particle is completely in the halo, its centre will not move to the lattice region in the next iteration; particles are supposed to move smoothly and if not, results will be invalid.

Third, consider solid particles that are partially over the lattice region of the subdomain and partially over the halo. These solid particles only have partial results for the interaction forces and for the liquid forces of them. So these particles will perform a communication to accumulate the forces. This communication can be in one or in two dimensions. If the solid particle is in a side of the halo, then a one dimension communication will be performed, but if it is in a corner of the halo of the subdomain, a two dimension communication will be performed in order that all the subdomains containing that solid particle will add all the forces. This will be described in more detail in the design chapter. It is possible to see the situation before the communication for one dimension in figure 17.

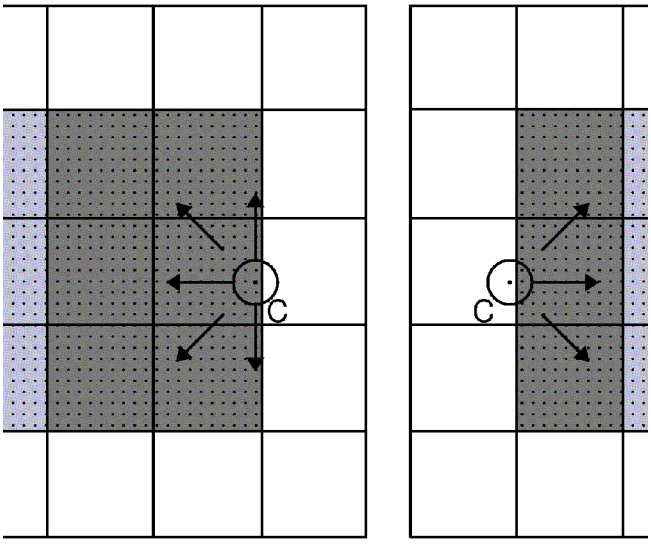

Figure 18: Two copies of the same solid particle in two different subdomains. Each copy has different partial forces before communication. When the forces for the two copies are added, then each copy will have the interaction forces from each of the nine cell lists.

After every solid particle has all the partial forces, their position and velocity can be updated. It may happen that a particle leaves one subdomain and enters another. If this happens in one subdomain, the particle will move and leave the lattice region of the subdomain, so it will be deleted of that subdomain. In the other subdomain, as all the copies of the solid particle have the same forces, its position will be updated in the same way, but now it will leaves the halo to enter the lattice region of the subdomain. So no explicit communication will be needed to notify when particles leaves or enter a new subdomain as it will be automatically updated. It is possible to see this in figure 19 when the forces are added and both copies of the particle have the same force addition. In figure 20 after the update of the position, a copy of the solid particle leaves the lattice region of the subdomain and the other copy enters in the lattice region of the other subdomain.

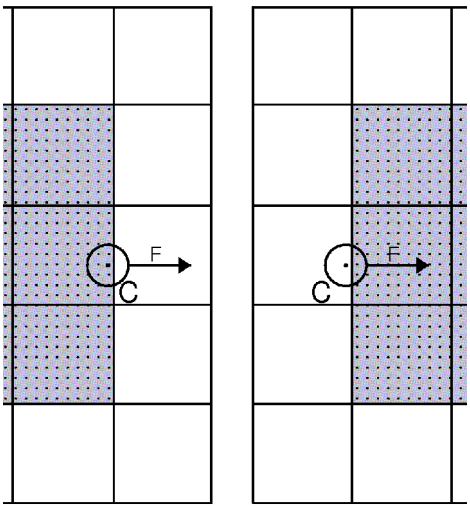

Figure 19: The two copies of a solid particle in different subdomains when all the forces are added.

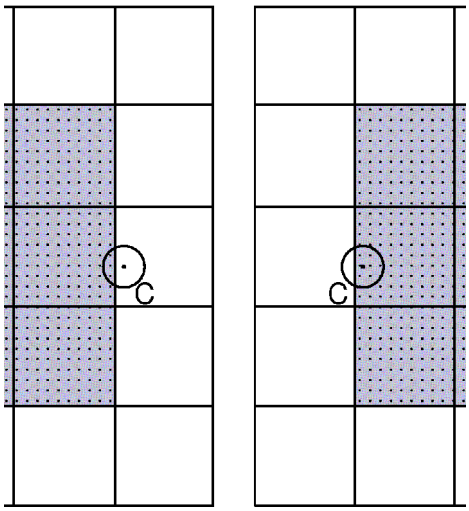

Figure 20: The two copies of a solid particle in different subdomains after the update of the position. The one on the left subdomain should be deleted.

3. Design

In this chapter, it will be explained how the final program will be designed from the analysis done in the previous chapter. Important parts of the final code will be shown as examples of how the design is implemented.

3.1 Program structure

The program is composed of five modular units:

- Constants. A header file with the most important constants that will be used in the program.
- Particles. A header and a .c file containing structures and functions to work with solid particles.
- Cell lists. A header and a .c file containing structures and functions to work with cell lists.
- MPI. A header and a .c file containing structures and functions to work with MPI communications, but without calling explicitly MPI functions in the program.
- Main program. A .c file that coordinates all the program.

The constants part will be explained in the next section. The particle, cell list and MPI parts will be explained in following sections. Here an explanation of the main program part is given.

In the main program, first, the MPI is initialized using the structures and functions of the MPI part.

```
int main(int argc, char *argv[]){
  process *p; // Struct that will contain process specific
              //  information
  MPIWrapperInit(argc, argv);  // Initialize MPI
  p = MPIWrapperInitProcess(); // Initialize the structure of the
                               //  process and all the necessary
                               //  things for MPI
```

Then, the solid particles and the cell list in each subdomain is created and initialized.

```
  // Creation of the particles
  a = particleCreate(id,xPos,yPos,a,xVel,yVel);
  ...
  // Creation of a particle list
  list = particleListCreate();
  // Add the particles to the particle list
  addParticle(list,a);
  ...
  // Create the cell list and add the particles of the particle list to it.
  cell = cellListCreate((XSIZE/((float)p->xDim)), (YSIZE/((float)p->yDim)),
                        (XSIZE/((float)p->xDim))*p->x,
                        (YSIZE/((float)p->yDim))*p->y, list);
```

After that, the main loop is performed. In this loop, the following steps are followed: swap the halos, calculate the forces of the solid particles and the liquid forces of them, accumulate forces of the solid particles and liquid forces of solid particles that are over more than one subdomain, and iterate the solid particles (that means calculate their new positions).

```
// Main loop of the program
for(i = 0; i < iterNum; i++){
  cellListSwapHalos(cell, p);
  cellListCalculateForces(cell);
  cellListAcumulateForces(cell, p);
  cellListIteration(cell, INCT);
}
```

Then, results are printed.

```
// Print contents of the cell list
printCellList(cell);
```

At the end, the MPI is finalized.

```
MPIWrapperEnd();  // Finalize MPI
```

With this general vision of the structure of the program, everything will be described in the following in detail.

3.2 Main Constants

For this program, some global constants are required and they will be stored in a include file "constants.h" in order that they can be changed easily. These constants are:

- `XSIZE`: is the size of the lattice in the x dimension.
- `YSIZE`: is the size of the lattice in the y dimension.
- `DELTAX`: is the separation between lattice points.
- `CRITICALGAP`: is the maximum distance between two solid particles when they will present a interaction between them.
- `INCT`: is the increment of time between iterations of the program.
- `RADIUSMAX`: is the maximum radius that a particle will have.

The constants `CRITICALGAP` and `RADIUSMAX` will be used to determine the size of the cells of the cell lists, as show in figure 14 in the previous chapter. The size of the cells of the cell lists are not exactly the size stated in section 2.4.2 but at least that size. The real size will depend on the `XSIZE`, `YSIZE` and the number of processors. This will be explained in more detail in the cell lists section.

`XSIZE`, `YSIZE` and `INCT` are only used in the main program. `RADIUSMAX` is only used in the creation of a cellList to calculate the size of the cells. `CRITICALGAP` is used in the creation of a cellList to calculate the size of the cells and to determine if a force will have to be calculated in the `particleCalculateForcesSame` and

`particleCalculateForcesTo` functions. `DELTAX` is only used in the `particleCalculatePoints` function to calculate the number of lattice points inside a solid particle.

The constant `HALO` which refers to the number of cells in the halo has the value 1, is used to make more clear the code. The value of `HALO` should not be changed.

3.3 Data Structures

For this program, five structures will be used, two for particles, one for particles lists, one for cell lists and one for an MPI process. They are described in detail in the next subsections.

3.3.1 Solid Particles

A solid particle will be composed of the following data:

An identifier: a value that identifies uniquely the solid particle. The centre of the particle is a pair (for the 2 dimensional problem) of floating point numbers that represent the position of the solid particle on each coordinate. The radius is also a floating point number that is the radius of the solid particle. The velocity consists of two floating point numbers that represent the current velocity of the solid particle in both coordinates of the lattice. The force is two floating point numbers that represent the current forces applied to the solid particle for both coordinates of the lattice. The force will determine the new velocity. The total lattice points inside the solid particle is an integer that represents the force on the particle from the fluid.

More data will be necessary for parallelization purposes because of the need to calculate partial and total values of forces.

Also, as it will be seen in the next subsection 3.3.2, a pointer for the next particle of the particle list will be required. With all this the structure for the particle will be as following:

```
// Struct of particles
struct _particle{
  int id;                     // Identifier of the particle
  float x, y;                 // X and Y coordinates
  float a;                    // Radius of the particle
  float u, v;                 // Velocities of the particle
                              //  in the x and y axes
  float partialfx, partialfy; // Partial forces
  float totalfx, totalfy;     // Total forces
  int totalPoints;            // Total number of points
  int partialPoints;          // Partial number of points
  struct _particle* next;     // Pointer to next particle
};
typedef struct _particle particle;
```

As in MPI it is not possible to send linked lists as messages, another structure will be provided for particles that will be used when they will be sent in MPI messages. This structure will not have the value of the pointer to the next particle but it will not also have the values for the total forces and lattice points inside the particle as only the partial values of those data will be sent. This structure is as follows:

```
// Struct of a particle used to be sent
struct _particleItem{
  int id;                    // Identifier of the particle
  float x, y;                // X and Y coordinates
  float a;                   // Radius of the particle
  float u, v;                // Velocities of the particle
                             //  in the x and y axes
  float partialfx, partialfy; // Partial forces
  int partialPoints;         // Partial number of points
};
typedef struct _particleItem particleItem;
```

A method for creating particles is provided with the following prototype:

```
particle* particleCreate(int id, float x, float y, float a, float u, float v);
```

It will be used to create particles with an identifier, two coordinates, a radius and a initial velocity.

3.3.2 Particle Lists

For storing the particles in a region[4] there are two possible options: to store them in an array or in a linked list. Each option has advantages and disadvantages that are explained here.

The most direct way would be to use an array to store all the particles in a certain region. This will have the advantage that an array is the format that particles will have to to be sent in an MPI message. The problem is that the particles will change from a particle list to another particle list during the execution of the program, so if an array will be used, it will have to have the ability to resize. In addition, if a particle leaves the region (and so the list) it is time-consuming to shift all the other particles in the array to eliminate the empty position in the array.

If a linked list is used, the advantage is that the problems of resizing and holes will not appear. The problem is that before sending particles, they will have to be saved in an array in order that it will be possible to send them in a MPI message.

It must be noticed that this does not mean that with the array solution there is no need to save the particles before being sent to another array. The list of particles in a region and the particles that are needed to be sent may not be the same.

With this, the linked list solution is chosen, but it will be possible to implement the array solution only making modifications in the particle part of the program.

4 This region will be a cell of the cell list for this program.

In this program, a particle list will contain the particles that are inside a cell of a cell list. This means that if a particle moves outside the region of the cell, it will have to change to the corresponding cell. Different solutions to do this will be explained in the cell lists subsection, but the chosen solution requires that each cell will have information about the surrounding cells. This information will be a pointer to that cell. So the structure of the particle lists will be as follows.

```
// Struct of a particle list used as cells in the cell lists
struct _particleList{
  int size;                // Number of particles
  struct _particleList* N;  // Cell at the North of this cell
  struct _particleList* NE; // Cell at the NE of this cell
  struct _particleList* E;  // Cell at the East of this cell
  struct _particleList* SE; // Cell at the SE of this cell
  struct _particleList* S;  // Cell at the South of this cell
  struct _particleList* SW; // Cell at the SW of this cell
  struct _particleList* W;  // Cell at the West of this cell
  struct _particleList* NW; // Cell at the NW of this cell
  struct _particle* list;   // Pointer to the first particle
};
typedef struct _particleList particleList;
```

A method for creating lists of particles is provided with the following prototype:

```
particleList* particleListCreate();
```

It will be used to create empty lists of particles. To add particles to the lists, the following method will be used:

```
int addParticle(particleList* l, particle* p);
```

That will add the particle `p` to the particle list `l`, if `l` or `p` are `NULL`, no particle is added.

Several methods are created to deal with particle lists which can be seen in Appendix A in the particle.h section. The most important ones will be described in this chapter and can be classified as: constructors, add and remove particles, linked list to array and array to linked list, forces and points related, movement of particles, and printing methods.

3.3.3 Cell Lists

The cell list structure has to meet some requirements. First of all, if a particle moves and it changes its position from one cell to another, the particle will have to change to the other cell. Also, for calculating interacting forces of a particle, it should be easy to find the cells around the particle to check for interaction forces with all the particles in those cells.

Two possible solutions have been found for this (but more can exist) and are explained here.

The first one is to make the cell list a two dimensional array. Then for moving particles, iterate the particles updating their position. After that, extract all the particles that are not in the cell they should be, and add the particles to the cell list again in its

correct position. It should be done in a high level (in the cell list methods) as the cell will not know about the cells around. For calculating interaction forces, it also should be done from a high level as the cell does not know about the around cells that are needed to calculate the interaction forces. This way can present some problems as not always all the surrounding cell are required.

The second method is to create links in every cell that will point to the surrounding cells. With this method, after updating the position of the particles, they can move to the corresponding cell depending on the link values. For calculating interaction forces, the calculation can be done at a lower level as every cell will know about all the surrounding cells. For cases when not all the surrounding cells are needed, the links can be changed to deal with those circumstances.

The second approach has been taken for this program. As stated in the previous subsection, every cell (that is a particle list) has the ability to save information about the surrounding cells.

Before explaing in more detail this solution, some conventions used are stated. In the whole program, the x and y coordinates are present; the x coordinate will extends horizontally from the left to the right. So 'left', 'West' and 'W' will refer to the x coordinate towards the negative values; and 'right', 'East' and 'E' will refer to the same towards the positive values. The y coordinates will extent vertically from down to up. So 'up', 'North' and 'N' will refer y coordinates towards the positive values and 'down', 'S' and 'South' will refer the same towards the negative values. Also, in all the program when a `i` variable is used as the index in an array, it will be used for an x coordinate and the `j` variable will be used for a y coordinate.

With this information and knowing from last chapter that also a halo will be created for each cell list, it is easy to describe the cell list structure. Here is the code:

```
// Struct of cell lists
struct _cellList{
  float xsize;         // Size in the x dimension
  float ysize;         // Size in the y dimension
  float x0, y0;        // Lower coordinates
  float x1, y1;        // Higher coordinates (lower coordinate + size)
  int xDim, yDim;      // Number of cell in each dimension
  particleList** list; // Two dimensional array of cells
};
typedef struct _cellList cellList;
```

A method for creating cell lists is provided with the following prototype:

```
cellList* cellListCreate(float xsize, float ysize, float x0, float y0,
                         particleList* list);
```

This creates cell lists with size `xsize` and `ysize` with origin coordinates `x0` and `y0` and including the particles in `list`. If some of these particles are not in the region of the cell list, they will not be included in the cell list. The cell list created will also have a halo of cells.

The links of the cell will be created in the constructor, and will be modified by the following methods:

```
int cellListSetLinks(cellList* c);
int cellListSetLinksForForce(cellList* c);
int cellListSetLinksForMove(cellList* c);
```

The first one is the one used in the constructor, the second one sets the links for being able to calculate forces and the third one sets the links to allow movement of the particles in the cell lists. Now this settings will be explained.

First, the links for moving particles will have to allow particles not in the halo to change to other cells not in the halo. This is because particles moving to cells in the halo will be deleted (See figures 19 and 20 for explanation). These links are shown in figure 21.

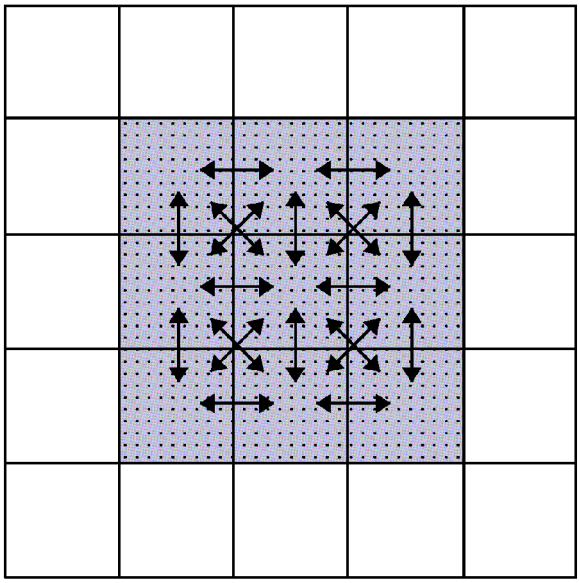

Figure 21: Links required by the cells not in the halo for movement.

Three different types of cells inside the halo can be found:

The first will be called MNA (for: Movement Not in halo A) and is when the cell is not at the border of the lattice. These cells will link to all the cells around them, which can be seen in figure 22.

The second will be called MNB and is when the cell is in the border of the lattice but not in a corner. These cells will link to all cells around them except in the direction of the halo (shown in figure 23).

The third will be called MNC and is when the cell is in the corner of the lattice. These cells will link only in directions where there is no halo (shown in figure 24).

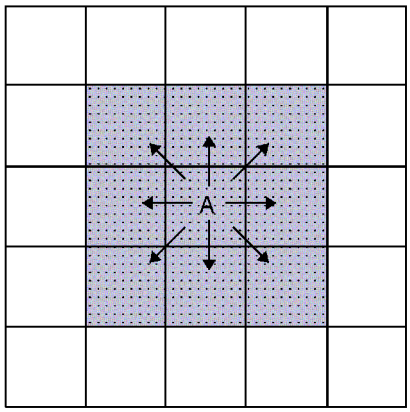

Figure 22: Links needed for movement by a cell not in the borders of the lattice.

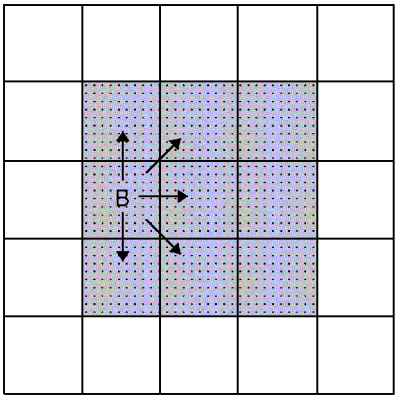

Figure 23: Links needed for movement by a cell in the borders of the lattice.

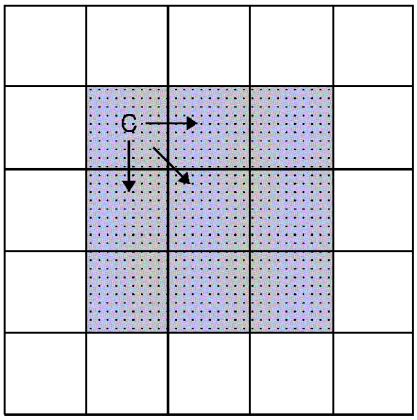

Figure 24: Links needed for movement by a cell in the corners of the lattice.

Second, the links for moving will have to allow particles in the halo to change to other cells that are not in the halo, but not to cells in the halo. This is because as before, particles that move to positions in the halo will be deleted. These links are shown in figure 25.

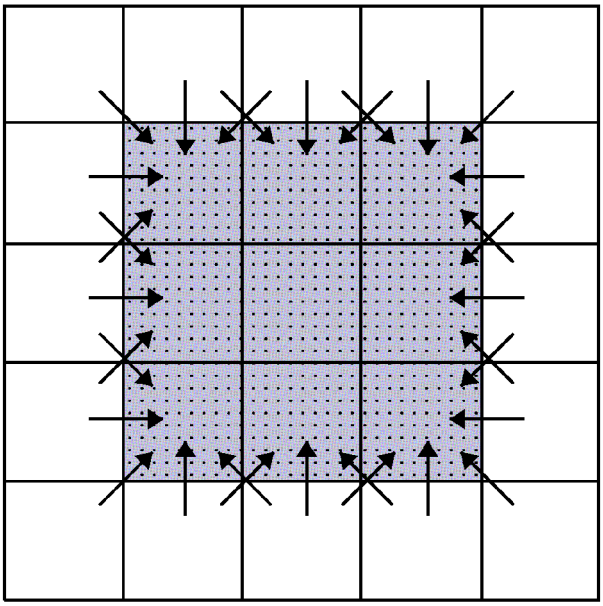

Figure 25: Links required by the cells in the halo for movement.

Again, three different types of cell in the halo can be found:

The first will be called MHA (for: Movement in Halo A) and is when the cell is not in the two cells nearest to the corner. This can be seen in figure 26.

The second will be called MHB and is when the cell is in the second cell nearest to the corner (shown in figure 27).

The third will be called MHC and is when the cell is in the corner (shown in figure 28).

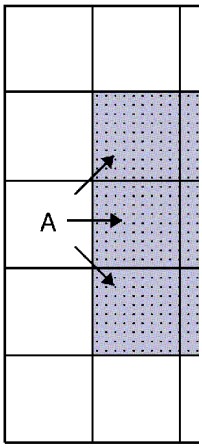

Figure 26: Links needed for movement by a cell that is not in the two cells nearest to the corner.

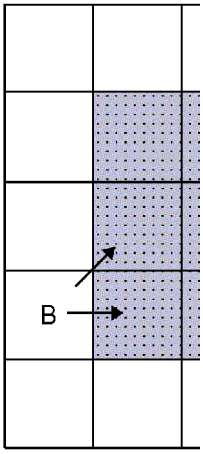

Figure 27: Links needed for movement by a cell that is in the second cell nearest to the corner.

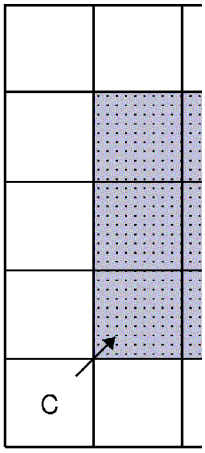

Figure 28: Links needed for movement by a cell that is in the corner.

So, before any iterations to update the position of the particles, all the links of the cells will have to be in the shown states.

For calculating interaction forces, this is required:

First, cells not in the halo will have to know about all the cells around them. These links can be seen in figure 29.

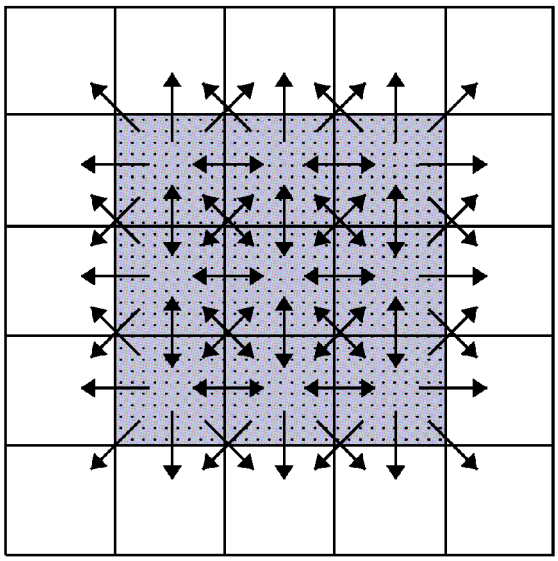

Figure 29: Links required by the cells not in the halo for calculation of interaction forces.

Here only one type of cell are and will be called FNA (for: Forces Not in halo A), it is the same as the one show in figure 22.

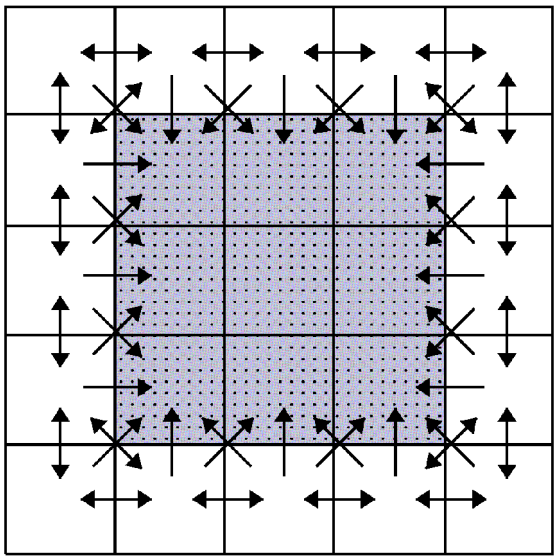

Figure 30: Links required by the cells in the halo for calculation of interaction forces.

Second, for cell in the halo, again, they will have to know about all the cells around them, but taking in consideration that they will not know anything about cells outside the cell list. So this links are showed in figure 30.

Here two types of cells are found:

The first will be called FHA (for: Force in Halo A) and is when the cell is not in the corner. It can be seen in figure 31.

The second will be called FHB and is when the cell is in the corner. It is show in figure 32.

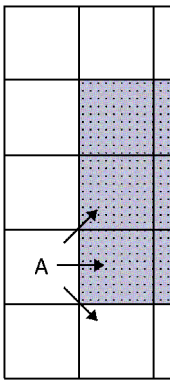

Figure 31: Links needed for forces by a cell that in a side.

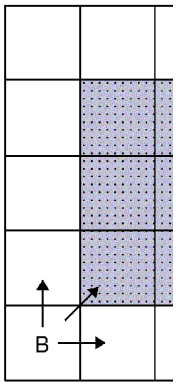

Figure 32: Links needed for forces by a cell that in a corner.

So, before calculating the interaction forces of the particles, all the particles will have to be in the shown states.

This change will have to be performed each iteration, and to change every cell will be inefficient, but taking care it is possible to change only the following: for particles not in the halo, only cells of type MNB and MNC have to be changed to be like cells FNA, that means to change the links of less cells than the number of halo cells. For cells in the halo, cell of type MHB will have to change to cells of type FHA, and cells of type MHC will have to change to cells of type FHB; that means to change the links of 12 cells in total for halos (3 in each corner).

The implementation of all the links settings can be tedious, but it will help greatly following work.

Several methods are created to deal with cell lists, they can be seen in Appendix A in the cellList.h section. The most important ones will be described in this chapter and can be classified as: link setters, add and remove particles, get and set halos, forces and points related, main and printing methods.

3.3.4 MPI Process

A structure will be created that will contain all the information that is specific to each process and also all the MPI specific data needed in the program. The methods that use this structure are the only methods that will use real MPI functions in the program. This allows the rest of the program to do not have to deal with real MPI functions.

This structure is shown here:

```
struct _process{
  int myid;                 // Identifier of the process
  int numprocs;             // Total number of processes
  int x, y;                 // Coordinates of the process
                            //   in a 2 dimensional decomposition
  int xDim, yDim;           // X and Y dimensions of the
                            //   decomposition
  MPI_Comm topology;        // The 2 dimensional decomposition
                            //   topology
  MPI_Datatype MPIParticle; // The datatype definition of a
                            //   particleItem needed to send MPI
                            //   messages containing this type
                            //   of data
};
typedef struct _process process;
```

As seen, it contains information to identify the process and its coordinates, the total number of processes and the dimension of the decomposition, and at last MPI data necessary for sending MPI messages in this decomposition.

To initialize an MPI program, two methods are used that have the following prototypes:

```
void MPIWrapperInit(int argc, char *argv[]);
process* MPIWrapperInitProcess();
```

The first will initialize the MPI program and the second will initialize the process structure.

To finish the MPI program, the following method will be used:

```
void MPIWrapperEnd();
```

For sending data among process, four methods are created, e.g.:

```
particleItem* SendUpReceiveDown(process* p, particleItem* send, int size,
                                int* receiveSize);
```

This method will send up an array of `particleItems` of size `size` and receive in the opposite direction an array of `particleItems` that will be returned and the `reciveSize` value will be set to its size.

The other three methods are analogous.

```
particleItem* SendDownReceiveUp(process* p, particleItem* send, int size,
                                int* receiveSize);
particleItem* SendRightReceiveLeft(process* p, particleItem* send, int size,
                                   int* receiveSize);
particleItem* SendLeftReceiveRight(process* p, particleItem* send, int size,
                                   int* receiveSize);
```

No more methods are declared for this structure.

3.4 Communication

Two different communications will be performed in the program, one communication for halo swaps and another one to accumulate interaction forces and number of lattice points of solid particles that are over more than one subdomain. This means that the halo swap communication will be always performed, but the accumulation communication will be only performed when necessary. In this section, both communications will be explained in detail. Other sections of the program must also be considered in order that these communications succeed.

3.4.1 MPI Communications

Data that will have to be communicated by MPI in this program are lists of particles. There is an inherent problem associated with the solution of using linked lists for the lists of particles, that is, MPI cannot send linked lists but only arrays. To solve this, before each communication, particles that will have to be sent will be copied to an array. If the solution of using arrays instead of linked lists was taken (section 3.3.2), there will be also the need to copy the needed particles to other array, so this is not a backdraw of the solution taken.

For help in transforming linked lists to arrays, and arrays to linked lists, two methods are provided:

```
int particleListToArray(particleList* l, particleItem* a, float x0, float y0);
particleList* particleArrayToList(particleItem* a, int size, float x0, float y0);
```

These methods will transform a linked list `l` to an array `a` and an array `a` to a linked list `l`. There is also another problem because of the periodic boundaries of the lattice. This is that if a particle will be communicated from a boundary of the lattice to the opposite one, the coordinates of the particle will not match the ones at the other side. Absolute coordinates of particles will not be sent but coordinates relative to the current border of the subdomain. That is why the `x0` and `y0` arguments are provided to these methods. For example, if the lattice is of size 12x12, a particle in position (11,5) will have to be sent to the halo of the opposite subdomain, the value of `x0` will be the one at the boundaries of the region, 12 in this case. The sent particle will have a x coordinate of 11-12 = -1. At the other subdomain, the value of `x0` will be the one at the boundaries of the region that is 0, so when the particle is received, its new x coordinate will be -1+0= -1 that will be the correct coordinate because periodic boundaries. This can be seen in figure 33.

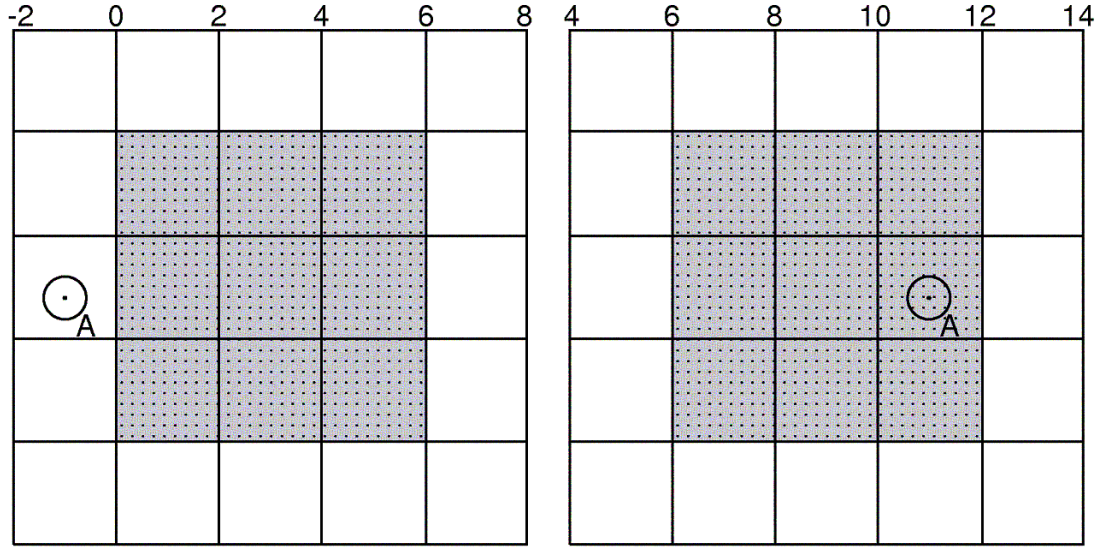

Figure 33: The particle A in the right with x coordinates 11, will be sent because of periodic boundaries to the halo in the left subdomain, where the x coordinate will have to be -1. This is resolved sending relative coordinates to the border of the lattice region.

These arrays of particles will be sent using the methods explained in the previous section. Their internals are the same except for the coordinate and direction of the communication. It is as follows:

```
// Calculate source and destiny subdomains
MPI_Cart_shift(p->topology, 1, 1, &sour, &dest);

// Send message to the destiny subdomain
MPI_Issend(send, size, p->MPIParticle,dest,0,p->topology,&r1);
// Check for receiving message and find its size
MPI_Probe(sour, 0, p->topology, &s1);
MPI_Get_count(&s1, p->MPIParticle, receiveSize);

// Allocate memory for receiving message
receive = (particleItem*)malloc(*receiveSize*sizeof(particleItem));
```

```
  // Receive message from source
  MPI_Recv(receive, *receiveSize, p->MPIParticle, sour, 0, p->topology, &s2);

  // Wait for sending completion
  MPI_Wait(&r1, &s3);
```

So, after finding the source and destination subdomains for the message, the message is sent by a non blocking send. If then checks for received message and find the size of the receiving message, enough memory for it is allocated and the message is received. At the end, message send completion is waited for .

3.4.2 Halo Regions

At the beginning of each iteration, halos are swapped in two dimensions as shown in figure 8 (communications due to periodic boundaries are not shown in the figure). The information of the corners of the halos is swapped by this two dimension communication as shown in figure 9.

For swapping halos, the following method is provided:

```
int cellListSwapHalos(cellList* c, process* p);
```

This method will perform the following steps:

First, it will clear the halos of particles from previous iterations. Then it will take all the cell that will be used to send a halo, put all the particles in an array and send them to the corresponding subdomain. After that, it will receive an array of particles from the opposite side and put them in the corresponding halo cells. This will be repeated for the four sides.

After this communication is done, all the halos contain the corresponding particles and it is possible to calculate interaction forces and the number of lattice points inside particles.

3.4.3 Adding forces

The total force from fluid and interactions on each particle is required. When calculating this for particles entirely in one subdomain no communication is needed. Some extra considerations will have to be taken in account for those particles not entirely in one subdomain that will require communication for calculating the total forces.

After swapping the halos, the following method will be called:

```
int cellListCalculateForces(cellList* c);
```

This method will do the following:

```
int cellListCalculateForces(cellList* c){
  // Set links of the cell list to calculate forces
  cellListSetLinksForForce(c);
  // Set forces of the particles to 0
  cellListClearForces(c);
```

```
  // Calculate interaction forces in the same cell
  cellListCalculateForcesSameCell(c);
  // Calculate interaction forces with around cells
  cellListCalculateForcesAroundCells(c);
  // Calculate number of points inside particle
  cellListCalculatePoints(c);
}
```

First, it will set the links of the cell list to calculate forces. Then it will set the forces of all particles to 0. After that it will calculate the interaction forces with particles in the same cell. Then it will calculate interaction forces with particles in around cells. At last it will calculate the number of lattice points that are inside each particle.

Every particle calculate the number of lattice points inside the particle including the ones in the halo, but all of them should only consider the lattice points that are in the lattice region that corresponds to the current subdomain. That is why for calculating the lattice points for a particle, using the following method the lower and highest coordinates of the lattice region are provided:

```
int particleCalculatePoints(particle* p, float x0, float y0, float x1, float y1);
```

Calculating interaction forces will require more considerations. First, is possible to classify according each particle in different positions. For each coordinate, the following positions exist:

- Particles whose centres are in the halo.
    - A. Particles whose centres are further to the lattice region than their radius.
    - B. Particles whose centres are nearer to the lattice region than their radius.
- Particles which centres are not in the halo.
    - C. Particles whose centres are further to the lattice region than their radius.
    - D. Particles whose centres are nearer to the lattice region than their radius.

With combinations for both dimensions, 16 possible cases exist (AA, AB, AC...). If it is thought that these positions are symmetric, then, 7 positions exist for every coordinate A, B, C, D, E (symmetric of C at the opposite side), F (symmetric of B at the opposite side) and E (symmetric of A at the opposite side). And 47 possibles cases exit combining both coordinates. From them, there will be 26 cases that will require different cells for calculating interaction forces. That is because those 47 cases comprises 22 different cases where the particle is completely in the halo (those cases containing an A or E case in any coordinate: AA, AC, DE...), and no calculation will be performed for those particles, so no difference among those cases is appreciated.

In figure 34 can be seen all those cases (except the ones of a particle completely in the halo) the figure represents 4 subdomains and 9 particles on them, but because halo swap, they are replicated in different subdomains.

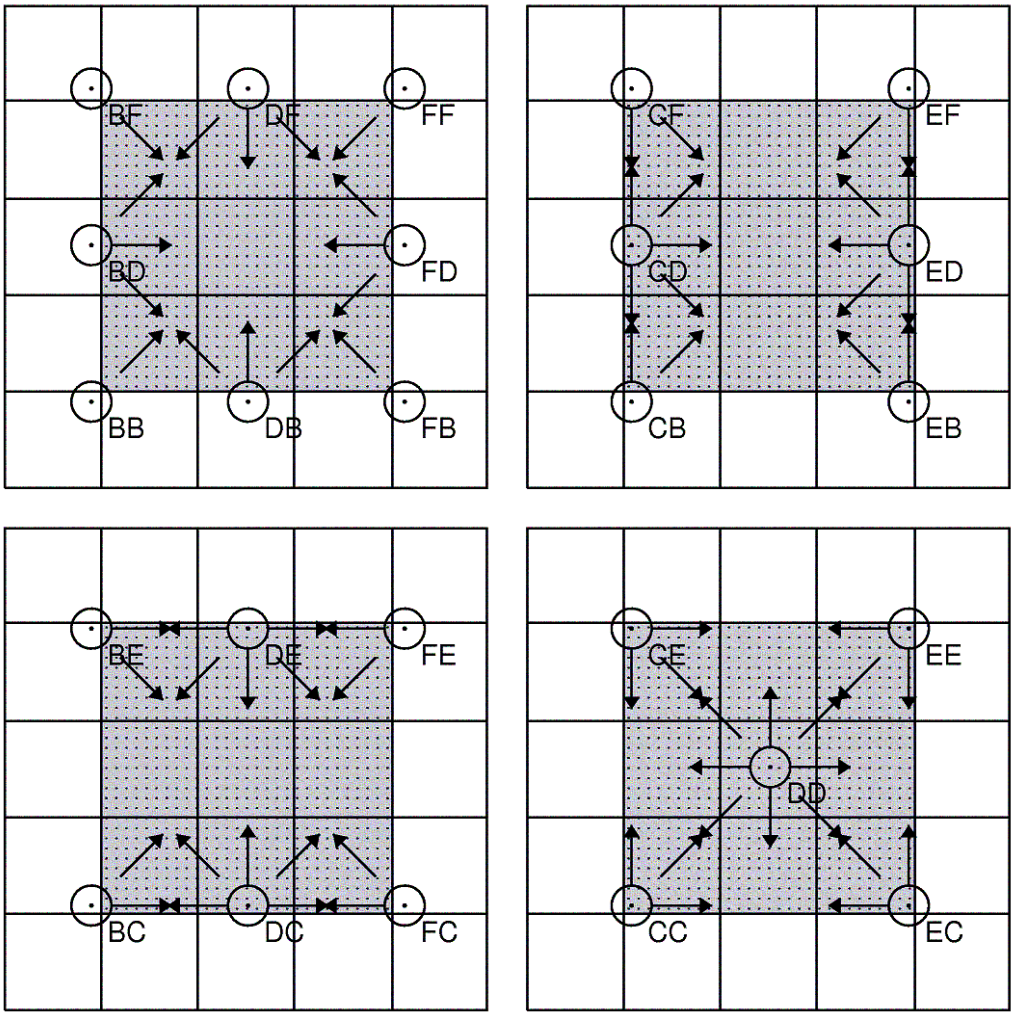

Figure 34: Possible particle position cases and directions to where interaction forces will be looked for in a 4 subdomain representation with 9 particles and their copies because of halo swap.

A particle for each case is shown with arrows representing the direction of the cells around the particle that should be used to calculate interactions. It is easy to check for correctness of the arrows chosen because it can be seen that, for example, particle FD is a copy of the particle CD but in a different subdomain due to swapping the halos. This is applicable to all the particles in the figure. In fact, of the 25 particles in the figure, there are only 9, the rest are duplicates in the halos. Then, for particle CD, 5 cells around it are checked for interaction forces, and for its unique duplicate FD 3 cells around are checked for interaction forces, all this add the 8 necessary cells around to look for interaction forces. A particle will only have to check for interaction forces in its own cell if that cell is not in the halo; if not, those forces will be duplicated. A more difficult example is particle CE; it has three copies that are CB, FB and FE. CE looks for 3 cells, CB for 2, FB for 1 and FE for 2, that again are in total the 8 necessary surrounding cells.

After following, two notices must be made.

From figure 34, it is possible to see that there exists the possibility that a particle like FB is not over any region of the lattice region of the subdomain. Even in that case, this particle will look for forces in the stated directions. That is because even if the particle is not over any lattice region, if it is of the type FB, that implies that three of types CB, CE and FE will exists in other subdomains and will require add interaction forces.

Also, from the picture, it seems, that no halo cells are considered for looking for interaction forces. That is not actually true as it can be seen in picture 35.

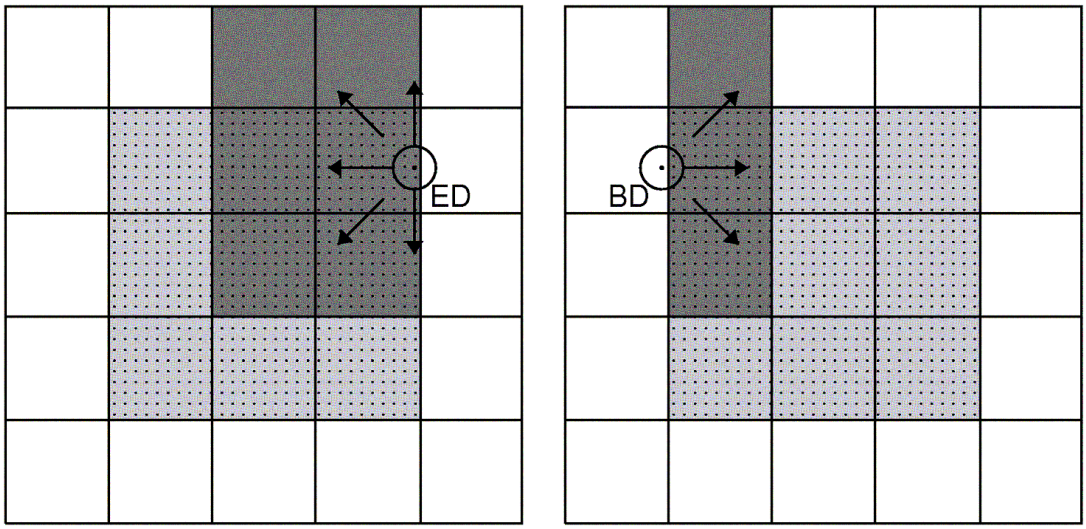

Figure 35: Two copies of the same particle may look for interaction forces in the halo cells.

In the figure 35, a particle of type ED and its copy of type BD will look for interaction forces in halo cells as no vertical communication will be required.

From all this the following conclusions can be extracted:

- Only particles that are in a cell not in the halo will calculate interaction forces in their cells. So the method `cellListCalculateForcesSameCell(c)`, will be applied only to cells not in the halo.
- A particle not of the type DD and not completely in the halo will have to perform a communication, it can be one or two dimensional communication, depending of the type of particle.
- A particle, when looking for interaction forces in the around cell, will look for all cells that are not in a halo in the direction it will have to communicate for adding forces.

With these conclusions it will be possible to implement a function to calculate interaction forces without considering every type of particle position. Here this way will be summarized.

This algorithm is implemented in the method:

```
int particleListCalculateForcesAround(particleList* l, float x0, float y0,
                                      float x1, float y1);
```

First, some variables are required:

```
  int inRLHalo;  // Particle in the right or left halo
  int inTBHalo;  // Particle in the top or bottom halo
  int upComm;    // Up communication will be performed
  int downComm;  // Down communication will be performed
  int rightComm; // Right communication will be performed
  int leftComm;  // Left communication will be performed
```

And they will be initialized to the correct values; 0 is for false and 1 is for true.

After that, each direction will be used for calculating interaction forces if it accomplish the stated conditions:

```
      if((!upComm) && (!inRLHalo)){
        particleCalculateForcesTo(tmp, l->N);
      }
      if((!downComm) && (!inRLHalo)){
        particleCalculateForcesTo(tmp, l->S);
      }
      if((!rightComm) && (!inTBHalo)){
        particleCalculateForcesTo(tmp, l->E);
      }
      if((!leftComm) && (!inTBHalo)){
        particleCalculateForcesTo(tmp, l->W);
      }
      if((!rightComm) && (!upComm)){
        particleCalculateForcesTo(tmp, l->NE);
      }
      if((!rightComm) && (!downComm)){
        particleCalculateForcesTo(tmp, l->SE);
      }
      if((!leftComm) && (!upComm)){
        particleCalculateForcesTo(tmp, l->NW);
      }
      if((!leftComm) && (!downComm)){
        particleCalculateForcesTo(tmp, l->SW);
      }
```

Now all the partial interaction forces and lattice points are calculated, so they have to be communicated.

For this, the following method will be used:

```
int cellListAcumulateForces(cellList* c, process* p);
```

It is very similar to the method used for swap the halos. This performs the following steps:

First, it will take all the cell that will be used to send a halo, extract all the particles that will need to accumulate forces in that direction, put them in an array and send them to the corresponding subdomain. After that, it will receive an array of particles from the opposite side and instead of add the particles to the cell list, they will be accumulated, that means that particles with the same identifier will add its partial

forces and number of points and save the addition in the total forces and total points values (that is why particles required an identifier). This will be repeated for opposite direction. After this, total forces and total points will be stored also as partial results. Then two more communication perpendicular to the previous ones will be performed.

The need of changing the total results to the partial ones between horizontal and vertical communications is because it is needed that partial results between diagonal subdomains has also to be added and this is done in the way shown in the figure 9.

After this, all particles has the necessary forces and the number of points that was required for update its position. This is done by the following method:

```
int cellListIteration(cellList* c, float inct);
```

First, links of the cell list is changed for movement with the method:

```
int cellListSetLinksForMove(cellList* c);
```

Then each particle update its velocity according to the previous velocity and the new forces; and after that each particle will check in case they will have to move to other cell in the cell list.

3.4.4 Memory Management

The memory management of this program can be tricky as memory is allocated in every interaction. So all the unused memory should be freed before the next iteration.

When particle halos are swapped halos are first cleared and particles in the halos should be freed. Then, the arrays used for sending and receiving messages should be freed after using them as they only store temporary data. Also, when an array of particles is received, this is converted to a temporary particle list and then added to the cell list, so these temporary lists should also be freed.

When particles calculate interaction forces and lattice points, no new memory is allocated, so nothing needs to be freed.

When communication to accumulate forces is performed, as when particle halos were swapped, the arrays used for communications should be freed. Also, temporary particle lists are used than should be freed. It is important to notice that for accumulating forces, more than one copy of particles are used and after adding the corresponding values, the copies obtained from communications should also be freed.

At last, when particles are updated no more memory is allocated, but if a particle leaves the lattice region, it should be also freed.

3.5 Clusters

In the analysis chapter (section 2.4.3), an algorithm to deal with cluster of particles was introduced. Here the details of how the communication of the cluster is performed are given.

Two methods can be used.

3.5.1 Reduction Method

For this method, two communications will be performed.

After each subdomain knows if it has a cluster inside and if the cluster extends outside the subdomain in any direction, this information will be broadcast to all the subdomains. After that, each subdomain will be able to determine which subdomains have parts of the cluster.

Then, all the particles of the cluster will be broadcast to the subdomains that have a part of the cluster. So after that, every subdomain involved will have the information of all particles of the cluster. Ultimately, a cluster might involve the whole system; in that case the situation will be the same as replicated data.

3.5.2 Communications Method

For this method, each subdomain that contains a part of the cluster will send the particles of the cluster to the subdomain that will continue the cluster around it. The subdomain will receive messages with cluster particles from surrounding subdomains, add its particles of the cluster and send the message again. The subdomain receives message from other subdomains until it receives the message it sent originally.

There are two possible classes of subdomains. The ones that only have a side that will extend the cluster and the ones that will have more than one side where the cluster will be extended.

The first class will send a message to one side, and will receive messages also from that side. If this is not the message the subdomain sent at the beginning , it will add its cluster particles and send back to the same side the message that was received. If the subdomain receives its original message again it has all the particles in the cluster. It is possible to see this in figure 36.

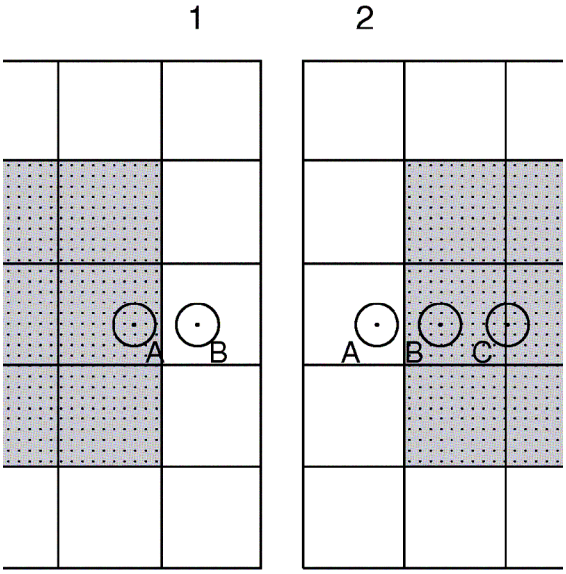

Figure 36: A cluster in two subdomains

In figure 36, the subdomain 1 will send a message with the particle A to subdomain 2. The subdomain 2 will do the same and will send a message with particles B and C to the subdomain 1. After this, subdomain 1 receives a message from 2, it was not the original message the subdomain sent, so it will add to the message particle A and send back to subdomain 2 the message, now with particles A, B and C. Subdomain 2 will receive a message from subdomain 1 that is no the original message it sent, so particles B and C will be added to the message and sent back to subdomain 1. At the end, subdomain 1 receives the first message it sent with particles A, B and C, and subdomain 2 receives the original message it sent with particles A, B and C. The communication is finished and both subdomains have the information of all the particles in the cluster.

The second class will send a message to one of the sides that has continuity for the cluster. Then until the original message is returned, if a message is received that is not the original message, it will add its particles to the message and send the message to another side of the subdomain where the cluster extends. When the message is returned, the subdomain will send the message to another side until the message has been sent to all the sides. After this, the message is sent back to the side that first received the message. When the original message the subdomain sent at the beginning is returned, it is sent to another side to repeat the same process until it has been sent to all sides, then the subdomain has all the particles of the cluster.

Each subdomain receiving a message will copy the particles that are in the messages with care to not replicate them. At the end, those copies will have all the particles in the cluster.

Special attention should be taken when clusters have special shapes like rings (including those created because periodic boundaries), etc.

3.6 Testing Strategy

This project consists in the creation of a 'toy' model that will simulate solid particles in a liquid. The program will distribute the work involved with both fluid and particles by domain decomposition using the MPI library. In order to achieve the final program, the following steps will be followed:

- creation of a serial program that will deal with one particle,
- creation of an MPI program that will deal with one particle,
- creation of an MPI program that will deal with more than one particle.

Each of these steps will add more requirements to the previous one.

Tests will be performed in order to check the correct results from the created programs each time one of these steps are reached.

The program will be tested on one processor to check different things like particle changing cells, periodic boundaries, particle interactions, correct movement of particles. When this work with one processor, more processors will be used and check the results with the one processor results.

Also tests to check for memory leaks have been performed, guarantying that the final program will not increase its size in memory over its execution.

The program uses standard C, standard C libraries and the MPI library, so no portability problems should appear.

4. Discussion and Summary

4.1 Discussion

If the concepts of this program are used in other programs that involve domain decomposition and data that moves from one subdomain to another, some more general requirements may be needed to fit these concepts to the other program. Here some possible modifications are commented on.

4.1.1 Badly Distributed Problems

This may be the most interesting one as it can also be applied to this project.

Two cases are possible:

It is possible that most of the particles in the lattice are concentrated in a few subdomains forming a cluster. In this case, few domains are performing all the calculation required for particles while most do not perform calculation for particles.

It may seem, that using no domain decomposition may help, but without domain decomposition, the cell list method cannot be used (unless replicated data is also used) and that increases the complexity of the problem as seen in section 2.4.

This can be solved using the previously described method with the cluster solution of the reduction method modified in the following way:

After each subdomain knows if it has a cluster inside and if the cluster extends outside the subdomain in any direction, this information will be broadcast to all the subdomains. After that, each subdomain will be able to determine which subdomains have parts of the cluster it has in part. But this calculation will be performed by all subdomains, so at the end, every subdomain will know about distribution of every cluster.

Then, all the particles of the cluster will be broadcast to all subdomains. So after that, every subdomain will have the information of all particles in all clusters. Then , calculation of forces of particles in clusters can be done in a distributed way and every subdomain will be in charge of a same number of particles.

After calculations are performed, results are sent to the subdomains that need the results to update the particles.

It is also possible that most of particles are concentrated in some subdomains but not forming clusters. Then two approaches are possible:

It is possible to use the functional decomposition explained in section 2.3.1 where the process of the particle will be well distributed among the processors used for particles.

It is also possible to make that regions of the lattice for subdomains will be assigned dynamically in order that each subdomain will have similar amount of computing. This possibility may be the hardest to implement.

4.1.2 Critical Gap Bigger than Subdomain Size

If it is needed that one particle can interact with other particles further that one subdomain away, the following changes can be made:

- Make the cell list size to be the whole subdomain.
- All the particles will participate in the communication for accumulating forces.
- The message to communicate forces will be send to one side but to the number of consecutive subdomains required. It then will receive the same number of messages from the opposite subdomains.

4.2 Summary

In this project a problem involving data structures that move from one subdomain to another one and interact with an underlying lattice is implemented.

On the way, some different solutions and algorithms has been developed —usually more than one for the same problem. The most convenient one was then choosen.

At the end an efficient 'toy' model that fits the initial requirements for calculate particles moving in a liquid is implemented.

Test have also been realized in order to verify the correctness of the implementation.

Some ways to expand this model are described in order that this project can be also for other purposes.

Appendix A: Header Files

constants.h

```
1 ////////////////////////////////////////////////////////////////////
2 //
3 // Project: Domain Decomposition for Colloid Clusters
4 // File:    constants.h
5 // Author:  Pedro Fernando Gomez Fernandez
6 // Version: 1.5
7 // Date:    5-9-2004
8 //
9 ////////////////////////////////////////////////////////////////////
10 //
11 // Description: Definition of the main constants of the program
12 //
13 ////////////////////////////////////////////////////////////////////
14
15 #ifndef _CONSTANTS_H
16 #define _CONSTANTS_H
17
18 #define XSIZE 100         // Size of the lattice in the x dimension
19 #define YSIZE 100         // Size of the lattice in the y dimension
20 #define DELTAX 1.0        // The separation between lattice points
21 #define CRITICALGAP 10.0  // The maximum distance between two solid
22                           //   particles when they will present a
23                           //   interaction between them.
24 #define INCT 0.01         // The increment of time between
25                           //   iterations of the program
26 #define RADIUSMAX 4.0     // The maximum radius that a particle
27                           //   will have
28
29 #endif
```

particle.h

```
1. ////////////////////////////////////////////////////////////////
2. //
3. // Project: Domain Decomposition for Colloid Clusters
4. // File:    particle.h
5. // Author:  Pedro Fernando Gomez Fernandez
6. // Version: 1.5
7. // Date:    5-9-2004
8. //
9. ////////////////////////////////////////////////////////////////
10. //
11. // Description: In this file, structures and prototypes related
12. //   to particles and lists of particles are described.
13. //
14. ////////////////////////////////////////////////////////////////
15.
16. #ifndef _PARTICLE_H
17. #define _PARTICLE_H
18.
19. #include <stdio.h>  // Needed for FILE definition
20.
```

```
21.//////////////////////////////////////////////////////////////////
22.// STRUCTURES
23.//////////////////////////////////////////////////////////////////
24.
25.// Struct of particles
26.struct _particle{
27.  int id;                    // Identifier of the particle
28.  float x, y;                // X and Y coordinates
29.  float a;                   // Radius of the particle
30.  float u, v;                // Velocities of the particle
31.                             //  in the x and y axes
32.  float partialfx, partialfy; // Partial forces
33.  float totalfx, totalfy;    // Total forces
34.  int totalPoints;           // Total number of points
35.  int partialPoints;         // Partial number of points
36.  struct _particle* next;    // Pointer to next particle
37.};
38.
39.// Struct of a particle used to be sent
40.struct _particleItem{
41.  int id;                    // Identifier of the particle
42.  float x, y;                // X and Y coordinates
43.  float a;                   // Radius of the particle
44.  float u, v;                // Velocities of the particle
45.                             //  in the x and y axes
46.  float partialfx, partialfy; // Partial forces
47.  int partialPoints;         // Partial number of points
48.};
49.
50.// Struct of a particle list used as cells in the cell lists
51.struct _particleList{
52.  int size;                 // Number of particles
53.  struct _particleList* N;  // Cell at the Noth of this cell
54.  struct _particleList* NE; // Cell at the NE of this cell
55.  struct _particleList* E;  // Cell at the East of this cell
56.  struct _particleList* SE; // Cell at the SE of this cell
57.  struct _particleList* S;  // Cell at the South of this cell
58.  struct _particleList* SW; // Cell at the SW of this cell
59.  struct _particleList* W;  // Cell at the Western of this cell
60.  struct _particleList* NW; // Cell at the NW of this cell
61.  struct _particle* list;   // Pointer to the first particle
62.};
63.
64.typedef struct _particle particle;
65.typedef struct _particleItem particleItem;
66.typedef struct _particleList particleList;
67.
68.
69.//////////////////////////////////////////////////////////////////
70.// CONSTRUCTORS
71.//////////////////////////////////////////////////////////////////
72.
73.// Constructor for particles: this create particles with an identifier 'id',
74.//   two coordinates 'x' and 'y', a radius 'a'
75.//   and a initial velocity 'u' and 'v'
76.particle* particleCreate(int id, float x, float y, float a, float u, float v);
77.
78.// Constructor for particleList: creates an empty particleList
```

```
79.particleList* particleListCreate();
80.
81.
82.////////////////////////////////////////////////////////////////
83.// ADD AND REMOVE PARTICLES
84.////////////////////////////////////////////////////////////////
85.// Add a particle 'p' to the particle list 'l'. Return the pos
86.//  where 'p' is added; 0 if not added.
87.int addParticle(particleList* l, particle* p);
88.
89.// Accumulate forces and points of particle 'p' to a particle
90.//   in the particle list 'l' with the same identifier.
91.//   Return 0 if not found and 1 if success
92.int acumulateParticle(particleList* l, particle* p);
93.
94.// Remove the particle with the identifier 'id' from the
95.//   particle list 'l' and return that particle
96.particle* removeParticle(particleList* l, int id);
97.
98.// Remove the first particle of the particle list 'l'
99.//   and return that particle
100.particle* removeFirstParticle(particleList* l);
101.
102.// Clear the particle list of particles and free all memory
103.//   used by those particles
104.int particleListClear(particleList* l);
105.
106.
107.////////////////////////////////////////////////////////////////
108.// LINKED LIST TO ARRAY AND ARRAY TO LINKED LIST
109.////////////////////////////////////////////////////////////////
110.
111.// Transform the partcle list 'l' in an array or particles 'a'. The
112.//   memory of the array should be allocated before calling this
113.//   method. Position of particles are saved relative to the
114.//   coordinates 'x0' and 'y0'. Return the number of particles.
115.int particleListToArray(particleList* l, particleItem* a,
116.                        float x0, float y0);
117.
118.// Transform the array or particles 'a' of size 'size' in a
119.//   partcle list 'l'. Position of particles are saved added
120.//   to the coordinates 'x0' and 'y0'. Return the list.
121.particleList* particleArrayToList(particleItem* a, int size,
122.                                   float x0, float y0);
123.
124.
125.////////////////////////////////////////////////////////////////
126.// FORCES AND POINTS RELATED
127.////////////////////////////////////////////////////////////////
128.
129.// Clear a list of particles freeing the memory of each particle
130.int particleListClearForces(particleList* l);
131.
132.// Calculate interaction forces of each particle in the list
133.//   with all the other particles in the same list
134.int particleListCalculateForces(particleList* l);
135.
136.// Calculate interaction forces of each particle in the list
```

```
137.//   with particles in cells around this. x0, y0, x1, y1 are
138.//   the coordinates of the lattice region  of the subdomain
139.//   it belongs to.
140.int particleListCalculateForcesAround(particleList* l, float x0, float y0,
141.                                       float x1, float y1);
142.
143.// Calculate interaction forces of particle 'p'
144.//   with all the particles from 'next' to the end of the list
145.//   if they are in the same list
146.int particleCalculateForcesSame(particle* p, particle* next);
147.
148.//Calculate interaction forces of particle 'p'
149.//   with all the particles from 'next' to the end of the list
150.//   if they are in different cells
151.int particleCalculateForcesTo(particle* p, particleList* next);
152.
153.// Calculate number of lattice points inside a particle for
154.//   each particle of the list. x0, y0, x1 and y1 are the coordinates
155.//   of the lattice region of the subdomain it belongs to.
156.int particleListCalculatePoints(particleList* l, float x0, float y0,
157.                                  float x1, float y1);
158.
159.// Calculate number of lattice points inside a particle for
160.//   particle 'p'. x0, y0, x1 and y1 are the coordinates
161.//   of the lattice region of the subdomain it belongs to.
162.//   Returns the number of points inside.
163.int particleCalculatePoints(particle* p, float x0, float y0,
164.                             float x1, float y1);
165.
166.// Get the number of particle in the particle list 'l' that will
167.//   perform a vertical communication for adding forces. 'y' is the
168.//   y coordinate of the border of the lattice region where the
169.//   communication will be performed. 'x0' and 'x1' are the
170.//   x coordintes of the lattice region in that cell.
171.//   Returns the number of particles that will perform communication
172.int particleListGetVerticalForcesNum(particleList* l, float y,
173.                                      float x0, float x1);
174.
175.// Get the number of particle in the particle list 'l' that will
176.//   perform an horizontal communication for adding forces. 'x' is the
177.//   x coordinate of the border of the lattice region where the
178.//   communication will be performed. 'y0' and 'y1' are the
179.//   y coordintes of the lattice region in that cell
180.//   Returns the number of particles that will perform communication
181.int particleListGetHorizontalForcesNum(particleList* l, float x,
182.                                        float y0, float y1);
183.
184.// Get the particles in the particle list 'l' that will perform
185.//   a vertical communication for adding forces and save them in
186.//   the particle array 'a'. 'y' is the y coordinate of the
187.//   border of the lattice region where the communication
188.//   will be performed. 'x0' and 'x1' are the x coordintes
189.//   of the lattice region in that cell
190.//   Return the number of particles added to the array
191.int particleListGetVerticalForces(particleList* l, particleItem* a, float y,
192.                                   float x0, float x1);
193.
194.// Get the particles in the particle list 'l' that will perform
```

```
195.//   an horizontal communication for adding forces and save them in
196.//   the particle array 'a'. 'x' is the x coordinate of the
197.//   border of the lattice region where the communication
198.//   will be performed. 'y0' and 'y1' are the y coordintes
199.//   of the lattice region in that cell
200.//   Return the number of particles added to the array
201.int particleListGetHorizontalForces(particleList* l, particleItem* a,
202.                                    float x, float y0, float y1);
203.
204.// Save the total results of each particle in the list as the partial ones.
205.int particleListTotalToPartial(particleList* l);
206.
207.// Calculate the interaction force between two particles separated
208.//   'dist' in the direction of the vector defined by 'xnorm' and
209.//   'ynorm'. The result is stored in 'xf' and 'fy'
210.int force(float dist, float xnorm, float ynorm, float* xf, float* yf);
211.
212.
213./////////////////////////////////////////////////////////////////
214.// MOVEMENT OF PARTICLES
215./////////////////////////////////////////////////////////////////
216.
217.// Update the position of each particle of the particle list 'l'
218.//   with a time increment 'inct'. x0, y0, x1 and y1 are the
219.//   coordinates of the lattice region of the subdomain
220.//   it belongs to.
221.int particleListIteration(particleList* l, float inct, float x0, float y0,
222.                          float x1, float y1);
223.
224.// Relocate each particle of the particle list 'l' if they
225.//   have change of cell after updating their position.
226.//   x0, y0, x1 and y1 are the coordinates of the lattice
227.//   region of the subdomain it belongs to.
228.int particleListRelocation(particleList* l, float x0, float y0,
229.                           float x1, float y1);
230.
231.
232./////////////////////////////////////////////////////////////////
233.// PRINTING FUNCTIONS
234./////////////////////////////////////////////////////////////////
235.
236.// Print the contents of a list of particle in the screen
237.int printList(particleList* l);
238.
239.// Print the contents of a list of particle in a file
240.int printListToFile(FILE* fd, particleList* l);
241.
242.#endif
```

cellList.h

```
1. ////////////////////////////////////////////////////////////
2. //
3. // Project: Domain Decomposition for Colloid Clusters
4. // File:    cellList.h
5. // Author:  Pedro Fernando Gomez Fernandez
6. // Version: 1.5
7. // Date:    5-9-2004
8. //
9. ////////////////////////////////////////////////////////////
10. //
11. // Description: In this file, structures and prototypes related
12. //   to cell lists are described.
13. //
14. ////////////////////////////////////////////////////////////
15.
16. #ifndef _CELLLIST_H
17. #define _CELLLIST_H
18. #include "particle.h"
19. #include "MPIWrapper.h"
20. #include <stdio.h>
21.
22. ////////////////////////////////////////////////////////////////
23. // STRUCTURES
24. ////////////////////////////////////////////////////////////////
25.
26. // Struct of cell lists
27. struct _cellList{
28.   float xsize;         // Size in the x dimension
29.   float ysize;         // Size in the y dimension
30.   float x0, y0;        // Lower coordinates
31.   float x1, y1;        // Higher coordinates (lower coordinate + size)
32.   int xDim, yDim;      // Number of cell in each dimension
33.   particleList** list; // Two dimensional array of cells
34. };
35.
36. typedef struct _cellList cellList;
37.
38. ////////////////////////////////////////////////////////////////
39. // CONSTRUCTOR
40. ////////////////////////////////////////////////////////////////
41.
42. // Constructor for cellList: this create cellLists with size
43. //   'xsize' and 'ysize' with origin coordinates 'x0' and 'y0' and
44. //   including the particles in 'list' if some of these particles
45. //   are not in the region of the cellList, they will not be included
46. //   in the cellList
47. cellList* cellListCreate(float xsize, float ysize, float x0, float y0,
48.                          particleList* list);
49.
50.
51. ////////////////////////////////////////////////////////////////
52. // LINKS SETTERS
53. ////////////////////////////////////////////////////////////////
54.
55. // Set the initial values of the links of the cell list
```

```
56.int cellListSetLinks(cellList* c);
57.
58.// Set the links of the cell list for calculating forces
59.int cellListSetLinksForForce(cellList* c);
60.
61.// Set the links of the cell list for updating particles position
62.int cellListSetLinksForMove(cellList* c);
63.
64./////////////////////////////////////////////////////////////////
65.// ADD AND REMOVE PARTICLES
66./////////////////////////////////////////////////////////////////
67.
68.// Add particles in 'list' to the cell list
69.int cellListAdd(cellList* c, particleList* list);
70.
71.// Add particles in 'list' to the cell list including the halo
72.int cellListAddWithHalo(cellList* c, particleList* list);
73.
74.// Accumulate forces and points of particles in 'list'
75.//   to the ones in the cell list including the halo
76.int cellListAcumulateWithHalo(cellList* c, particleList* list);
77.
78.// Add the particle 'p' to the cell list
79.int cellListAddParticle(cellList* c, particle* p);
80.
81.// Add the particle 'p' to the cell list including the halo
82.int cellListAddParticleWithHalo(cellList* c, particle* p);
83.
84.// Accumulate forces and points of particle 'p'
85.//   to the one withs same identifier in the cell
86.//   list including the halo
87.int cellListAcumulateParticleWithHalo(cellList* c, particle* p);
88.
89.// Clear the halos of the cell list
90.int cellListClearHalos(cellList* c);
91.
92./////////////////////////////////////////////////////////////////
93.// GET AND SET HALOS
94./////////////////////////////////////////////////////////////////
95.
96.// Get the up halo of the cell list. Returns an array with
97.//   the particles in the halo and sets 'size' to the array size
98.particleItem* cellListGetUpHalo(cellList* c, int* size);
99.
100.// Get the down halo of the cell list. Returns an array with
101.//   the particles in the halo and sets 'size' to the array size
102.particleItem* cellListGetDownHalo(cellList* c, int* size);
103.
104.// Get the right halo of the cell list. Returns an array with
105.//   the particles in the halo and sets 'size' to the array size
106.particleItem* cellListGetRightHalo(cellList* c, int* size);
107.
108.// Get the left halo of the cell list. Returns an array with
109.//   the particles in the halo and sets 'size' to the array size
110.particleItem* cellListGetLeftHalo(cellList* c, int* size);
111.
112.// Set the particles in the array 'halo' wich has a size 'size'
113.//   in the down halo of the cell list
```

```
114. int cellListSetDownHalo(cellList* c, particleItem* halo, int size);
115.
116. // Set the particles in the array 'halo' wich has a size 'size'
117. //   in the up halo of the cell list
118. int cellListSetUpHalo(cellList* c, particleItem* halo, int size);
119.
120. // Set the particles in the array 'halo' wich has a size 'size'
121. //   in the left halo of the cell list
122. int cellListSetLeftHalo(cellList* c, particleItem* halo, int size);
123.
124. // Set the particles in the array 'halo' wich has a size 'size'
125. //   in the right halo of the cell list
126. int cellListSetRightHalo(cellList* c, particleItem* halo, int size);
127.
128. ////////////////////////////////////////////////////////////////
129. // FORCES AND POINTS RELATED
130. ////////////////////////////////////////////////////////////////
131.
132. // Clear the forces of all the particles in the cell list
133. int cellListClearForces(cellList* c);
134.
135. // Set the partial forces of the particles in the cell list
136. //   to the total value
137. int cellListTotalToPartial(cellList* c);
138.
139. // Calculate interaction forces in the same cell for
140. //   all particles in the cell list
141. int cellListCalculateForcesSameCell(cellList* c);
142.
143. // Calculate interaction forces in sorround cells for
144. //   all particles in the cell list
145. int cellListCalculateForcesAroundCells(cellList* c);
146.
147. // Calculate the lattice points inside particles for
148. //   all the particles in the cell list
149. int cellListCalculatePoints(cellList* c);
150.
151. // Get the particles that will accumulate forces in the up
152. //   direction of the cell list. Returns an array with
153. //   those particles and sets 'size' to the array size
154. particleItem* cellListGetUpForces(cellList* c, int* size);
155.
156. // Get the particles that will accumulate forces in the down
157. //   direction of the cell list. Returns an array with
158. //   those particles and sets 'size' to the array size
159. particleItem* cellListGetDownForces(cellList* c, int* size);
160.
161. // Get the particles that will accumulate forces in the right
162. //   direction of the cell list. Returns an array with
163. //   those particles and sets 'size' to the array size
164. particleItem* cellListGetRightForces(cellList* c, int* size);
165.
166. // Get the particles that will accumulate forces in the left
167. //   direction of the cell list. Returns an array with
168. //   those particles and sets 'size' to the array size
169. particleItem* cellListGetLeftForces(cellList* c, int* size);
170.
171. // Accumulate forces from the particles in the array 'halo'
```

```
172.//   wich has a size 'size' with the ones with same identifier
173.//   in the down region of the cell list
174.int cellListSetDownForces(cellList* c, particleItem* halo, int size);
175.
176.// Accumulate forces from the particles in the array 'halo'
177.//   wich has a size 'size' with the ones with same identifier
178.//   in the up region of the cell list
179.int cellListSetUpForces(cellList* c, particleItem* halo, int size);
180.
181.// Accumulate forces from the particles in the array 'halo'
182.//   wich has a size 'size' with the ones with same identifier
183.//   in the left region of the cell list
184.int cellListSetLeftForces(cellList* c, particleItem* halo, int size);
185.
186.// Accumulate forces from the particles in the array 'halo'
187.//   wich has a size 'size' with the ones with same identifier
188.//   in the right region of the cell list
189.int cellListSetRightForces(cellList* c, particleItem* halo, int size);
190.
191.
192.////////////////////////////////////////////////////////////////
193.// MAIN FUNCTIONS
194.////////////////////////////////////////////////////////////////
195.
196.// Swapt the halos of the cell list clearing them before
197.int cellListSwapHalos(cellList* c, process* p);
198.
199.// Calculate interaction forces for each particle and lattice point
200.//   inside them.
201.int cellListCalculateForces(cellList* c);
202.
203.// Performs communications to accumulate forces of particles
204.//   in more than one subdomain
205.int cellListAcumulateForces(cellList* c, process* p);
206.
207.// Update the particles position
208.int cellListIteration(cellList* c, float inct);
209.
210.////////////////////////////////////////////////////////////////
211.// PRINTING FUNCTIONS
212.////////////////////////////////////////////////////////////////
213.
214.// Print the content of the cell list in the screen
215.void printCellList(cellList* c);
216.
217.// Print the content of the cell list with the halos in the screen
218.void printCellListWithHalo(cellList* c);
219.
220.// Print the content of the cell list in a file
221.void printCellListToFile(FILE* fd, cellList* c);
222.
223.// Print the content of the cell list with the halos in a file
224.void printCellListWithHaloToFile(FILE* fd, cellList* c);
225.
226.#endif
```

MPIWrapper.h

```
1. ////////////////////////////////////////////////////////////////////
2. //
3. // Project: Domain Decomposition for Colloid Clusters
4. // File:    MPIWrapper.h
5. // Author:  Pedro Fernando Gomez Fernandez
6. // Version: 1.5
7. // Date:    5-9-2004
8. //
9. ////////////////////////////////////////////////////////////////////
10.//
11.// Description: In this file, structures and prototypes related
12.//   to process are described. These are the only methods that
13.//   will use real MPI functions. That allow the rest of the
14.//   program to do not have to deal with real MPI functions.
15.//
16.////////////////////////////////////////////////////////////////////
17.
18.#ifndef _MPIWRAPPER_H
19.#define _MPIWRAPPER_H
20.
21.#include "particle.h"
22.#include <mpi.h>
23.
24.////////////////////////////////////////////////////////////////////
25.// STRUCTURE
26.////////////////////////////////////////////////////////////////////
27.
28.// Struct of the process
29.struct _process{
30.  int myid;                // Identifier of the process
31.  int numprocs;            // Total number of processes
32.  int x, y;                // Coordinates of the process
33.                           //   in a 2 dimensional decomposition
34.  int xDim, yDim;          // X and Y dimensions of the
35.                           //   decomposition
36.  MPI_Comm topology;       // The 2 dimensional decomposition
37.                           //   topology
38.  MPI_Datatype MPIParticle; // The datatype definition of a
39.                           //   particleItem needed to send MPI
40.                           //   messages containing this type
41.                           //   of data
42.};
43.
44.typedef struct _process process;
45.
46.
47.////////////////////////////////////////////////////////////////////
48.// INIT AND END MPI
49.////////////////////////////////////////////////////////////////////
50.
51.// Initialize an MPI program
52.void MPIWrapperInit(int argc, char *argv[]);
53.
54.// Initialize the process structure setting all its values
55.process* MPIWrapperInitProcess();
```

```
56.
57.// Finish the MPI program
58.void MPIWrapperEnd();
59.
60.
61.////////////////////////////////////////////////////////////////
62.// SENDING AND RECIVING
63.////////////////////////////////////////////////////////////////
64.
65.// Send up and array of particleItems of size 'size' and receive
66.//   from down an array of particleItems that will be returned
67.//   and the 'reciveSize' value sets to its size.
68.particleItem* SendUpReceiveDown(process* p, particleItem* send, int size,
69.                                int* receiveSize);
70.
71.// Send down and array of particleItems of size 'size' and receive
72.//   from up an array of particleItems that will be returned
73.//   and the 'reciveSize' value sets to its size.
74.particleItem* SendDownReceiveUp(process* p, particleItem* send, int size,
75.                                int* receiveSize);
76.
77.// Send right and array of particleItems of size 'size' and receive
78.//   from left an array of particleItems that will be returned
79.//   and the 'reciveSize' value sets to its size.
80.particleItem* SendRightReceiveLeft(process* p, particleItem* send, int size,
81.                                   int* receiveSize);
82.
83.// Send left and array of particleItems of size 'size' and receive
84.//   from right an array of particleItems that will be returned
85.//   and the 'reciveSize' value sets to its size.
86.particleItem* SendLeftReceiveRight(process* p, particleItem* send, int size,
87.                                   int* receiveSize);
88.
89.#endif
```

Appendix B: Workplan

Week	*Date*	*Task*
1	19-5/26-5	Get information about the project.
2	27-5/2-6	Get more information about the project.
3	3-6/9-6	Introduction presentation.
4	10-6/16-6	Analysis.
5	17-6/23-6	Analysis and Design.
6	24-6/30-6	Serial code implementation.
7	1-7/7-7	Serial code implementation.
8	8-7/14-7	MPI code for 1 particle.
9	15-7/21-7	MPI code for 1 particle.
10	22-7/28-7	Particle list implementation.
11	29-7/4-8	Particle list implementation.
12	5-8/11-8	MPI code for 2 particles.
13	12-8/18-8	Forces accumulation communication.
14	19-8/25-8	Testing correctness and memory leaks.
15	26-8/1-9	Reviewing final document.
16	2-9/8-9	Reviewing final document.

References

[1] S. Succi. “The Lattice Boltzmann Equations for Fluid Dynamics and Beyond” Clarendo, Oxford (2001)
[2] M.P. Allen and D.J. Tidesley. “Compute Simulation of Liquids”
[3] J. C. Desplat, I. Pagonabarraga and P. Bladon. “LUDWIG: A parallel lattice-Boltzmann code for complex fluids” Compt. Phys. Comms. 134 (2001)
[4] N. Q. Nguyen and A. J. C. Ladd. “Lubrication corrections for lattice-Boltzmann simulations of particle suspensions. Phys. Rev. E, 66, 046708 (2002)
[5] A. J. C. Ladd. “Numerical simulations of particulate suspensions via discretised Boltzmann equation. Part 1. Theoretical foundation” J. Fluid. Mech. 271 (1994)
[6] A. J. C. Ladd. “Numerical simulations of particulate suspensions via discretised Boltzmann equation. Part 2. Numerical results” J. Fluid. Mech. 271 (1994)
[7] “MPI: A Message-Passing Interface Standard” Message Passing Interface Forum (1995)