Particle simulation using serial, GPU and distributed approaches

by

©Xiaoqian Men

A Thesis submitted to the School of Graduate Studies in partial fulfillment of the requirements for the degree of

Master of Engineering

Faculty of Engineering and Applied Science

Memorial University of Newfoundland

January 2015

St. John's Newfoundland

Abstract

In computer engineering, simulation is a popular and reasonable method to study scientific problems, which evaluates the motion of different objects in various sizes of simulation spaces. In order to achieve better performance, different approaches will be applied. Nowadays, both GPU and cluster show great power in parallel computing.

In this thesis, a particle simulation is formulated by following the motion of interacting particles as they move in some constrained space, colliding with each other and the walls. We compare three solutions to this problem: i) using traditional (serial) computing, ii) using general purpose computing on a graphics processing card (GPGPU), and iii) using a distributed cluster architecture and the message passing interface (MPI). Based on the experimental data gathered from the tests, the performance of the algorithms is analyzed to show how the speedup varies across different architectures and with the number of compute cores used.

Acknowledgements

I sincerely thank my supervisors, Dr. Dennis K. Peters and Dr. Claude Daley, for their intellectual guidance, constructive suggestions, constant encouragement and support for my study and thesis during the whole period of my M.Eng program.

I am grateful for the financial support received from my supervisors, the Faculty of Engineering and Applied Science, and the School of Graduate Studies (SGS) of Memorial University.

I would like to thank the instructors of Memorial University for their patient help in my course and research. I also wish to thank Ms. Moya Crocker, the Graduate office, the Faculty of Engineering and Applied Science for their administrative and technical assistance during the course of my program.

Last but not the least, I will thank my beloved parents and friends for their support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A major application of scientific problems solved by computers is scientific simulation, which is distinguished by large data and time. In order to achieve better performance, different structures, software and algorithms have been used. At first, the executing code was done on only one core in one PC, then parallel programming, which can divide the work among many cores on a multiprocessor or even several PCs was introduced. Nowadays, programmers find that new applications like clusters or graphics processing units will contribute a lot to operations of massive data. What's more, the combination of cluster and graphics processing unit is expected to be more effective.

1.1 Ice Simulation and Motivation

The aim of the research, which is supported by the Sustainable Technology for Polar Ships and Structures project (referred to as STePS^2), is to look further into the best algorithms and architectures for ice simulation. The goal of ice simulation is to describe the behaviour of a ship operating in pack ice and colliding with ice floes in one area. The aim of STePS^2 project is to gain a good understanding of interactions between ice and steel structures. The ice floes are affected by currents, wind, and

interact with land, ships and so on. Figure 1.1 shows the ice simulation view. The results are found by a discrete time-stepping simulation, which runs much faster than real-time and provides a guide and measure for planning ice management activities, in which an ice capable vessel is used to break up or disperse a drifting ice field so as to lessen the impact on another vessel, such as an oil rig. In his PhD work, Shadi Alawneh [3] has tried a serial version using CPU and a parallel version using GPU. The results and some former research by the others show that GPU has more computing power than CPU for this type of simulation. Facing a small ice field of 456 ice floes, the parallel algorithm on the GPU shows the speedup up to 77 times compared to the serial algorithm on the CPU. His design reduced the simulation time of this ice field from over 88 minutes to about 68 seconds. Detailed information like the model of vessel, different ice fields, and ice conditions is discussed in the chapter 6 of his PhD work. While the test has used small ice fields, large ice fields, will require more powerful tools to achieve desired quicker results.

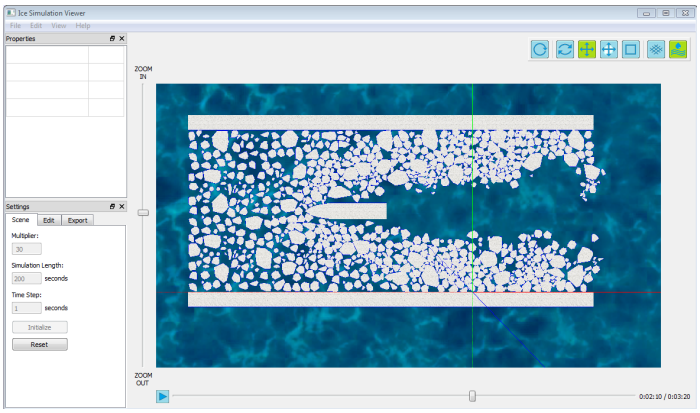


Figure 1.1: Ice Simulation Viewer [4]

As is illustrated in figure 1.1, an ice field consists of many ice floes colliding with each other. The main computing structure of the ice simulation is the interaction of objects moving in different sizes of area and the main idea of our research is applying

new parallel design to the ice floe simulation. So we choose to study the particle simulation problem as a step on the way to solving the ice floe simulation problem. The physical calculation between particles is simply and we can focus more on the parallel designs of algorithms and architectures. Our particle simulation has same simulation model in two dimensions and similar movement model as the ice floe simulation, only with much less complex physics calculation between moving objects. Therefore we think the relative performance of algorithms and architectures used in the particle simulation will translate to that work. In the particle simulation, I investigated different computing architectures and algorithms.

Previous work has investigated a serial version and a GPU version of ice simulation. In order to achieve different performance, different parallel structures are applied in this work. In addition to the serial version and the GPU version, we also investigate the particle simulation on a CPU cluster, using different algorithms.

1.2 Parallel Computers

The N body problem was first studied in the context of astronomy, to simulate the motion of celestial bodies. This problem needs to calculate the forces between all the bodies and the problem is O(n^2) where n is the number of particles. For large problem sizes, if n equals 10^{11}(same as the stars in the galaxy), the computation would take a long time on a single CPU. Even an extremely optimistic figure to compute once is 10^{-6} seconds, it would almost take 10^9 years for one iteration. Better algorithms were invented to reduce the problem complexity, such as Barnes-Hut algorithm, which makes this problem O($nlog_n$) [34]. This algorithm is described in chapter two. The N body problem is widely applied in chemical and biological areas at the molecular level and takes an enormous time. Similar problems can be solved in a reasonable time

on a small scale, but they also need to be calculated multiple times under different conditions to get different results. Researchers have tried different ways to reduce the computing time, such as introducing better algorithms, applying powerful hardware, and changing simulation structures.

Instead of sequential program running on one processor, parallel programming can be used to address this problem. Programmers can not only program on multiple processors in one computer, but also code on multiple computers in a network simultaneously. There are two basic types of parallel computers, shared memory multiprocessor and distributed-memory multicomputer [34]. Figure 1.2 shows the structures of the single processor and the shared memory multiprocessor. The whole task can be divided into several threads or processes. However, from the point of view of the memory, all the threads or processors still need to access to the main memory.

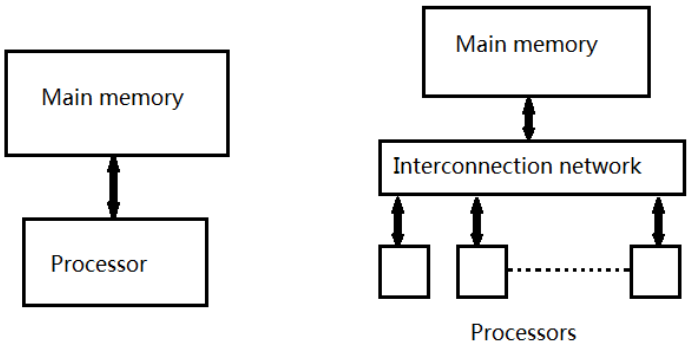


Figure 1.2: Conventional computer with one processor and shared memory multiprocessor system

Different from the shared memory multiprocessor system, distributed-memory multicomputer do not share the whole memory. Figure 1.3 shows the structure of the distributed-memory multicomputer. Each computer owns their local main memory which is not accessible to other computers. Computers communicate with each

other by sending messages through the interconnection network. Such multiprocessor systems are usually called message-passing multiprocessors, or simply multicomputers [34].

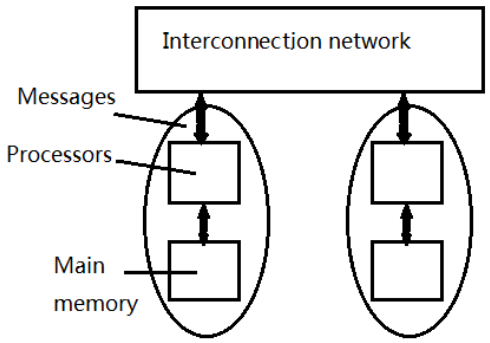


Figure 1.3: Message-passing multiprocessor/multicomputer system [34]

Programmers can use specific message-passing programming techniques to declare the parallel section. OpenMP and MPI are two popular kinds of programming interfaces to write code using threads to access the main memory. They are also compatible with C/C++ and FORTRAN. Our particle simulation in this thesis uses MPI and C to program the code on a distributed-memory multicomputer cluster. Message passing interface (MPI) is a standardized and portable communication protocol that is widely used to write message passing programs. MPI is a kind of programming model and its aim is to serve for communication between processes. Both OpenMP and MPI have high portability, so they can be used on almost all operating systems.

The Message-passing model takes advantage of synchronization mechanisms to do parallel programming for serial programs, but is not convenient in sharing the data. Data for running the program cannot be shared to all computers and they must be copied to each computer's local main memory. Sometimes, it is not necessary to copy the entire memory, and only the relevant parts of the memory required for a parallel

process to perform its task should be copied. If the amount of data is huge, a lot of time will be wasted in copying. If a processor accesses the data not in its local memory location, message passing will occur to pass data between processors. One drawback is that accessing data in a remote location will make a great delay. There are two algorithms tried on the CPU cluster, the master-slave algorithm and the heartbeat algorithm. The former one copies the whole memory to the slave nodes and the latter one distributes parts of the whole memory. Both are described in detail in chapter three.

1.3 General Purpose GPU Computing

Beyond being used only for processing graphical information, graphics processing units or GPUs are a powerful general purpose programming hardware. Many programmers use GPU to solve scientific problems, which is known as general purpose GPU computing or GPGPU. GPGPU has the advantage of more quickly solving computational problems than CPU, such as numerical problems and parallel programming. The reason is that GPU uses a many-core structure. Even though a computer can have several cores, the number is limited. Intel has shown the prototype of a processor with 80 cores [17]. The large difference in the number of cores between CPU and GPU can be seen from figure 1.4. A GPU can consist of thousands of smaller cores. GPUs are designed for SIMD (Single instruction stream multiple data stream) processing and the work flow acts as a pipeline program. More about SIMD and other architectures are discussed in chapter two.

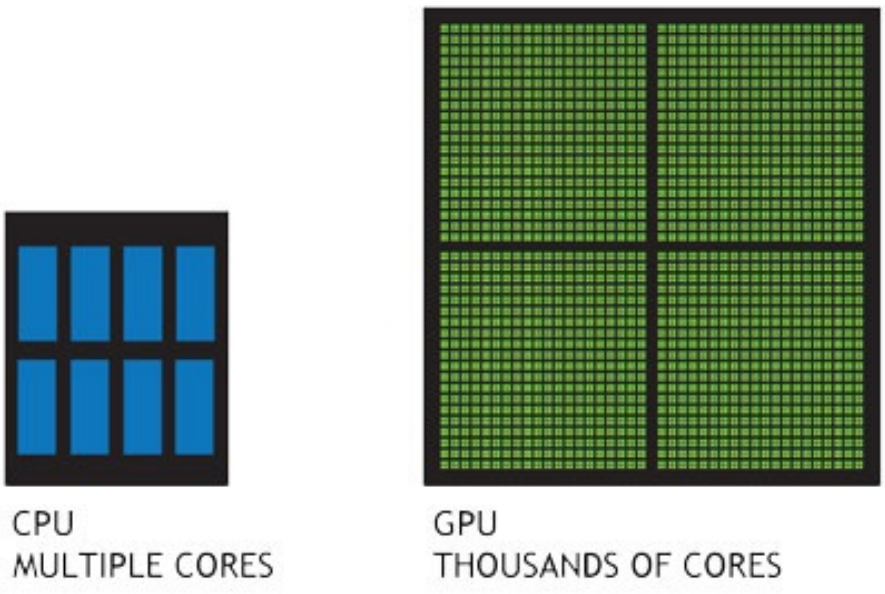


Figure 1.4: GPUs have thousands of cores to process parallel workload efficiently [23]

A programming model used on GPU is called CUDA, Compute Unified Device Architecture, which provides a parallel computing platform for programmers gaining benefits from both CPU and GPU. Programmers can code in C,C++, and FORTRAN combining with CUDA. After the appearance of CUDA, programmers can programme conveniently using GPU. More benefits are shown by using CUDA and GPUs to solve large, complex and massive data, especially simulations.

1.4 Purpose and Organization

We studied the relative performance of three architectures and three algorithms for solving problems similar to the ice floe simulation problem. We implement the particle simulation using different approaches, analyze and compare the performance, discuss how the different approaches effect the results, propose some better methods to improve the performance of ice floe simulation and possible extensions to the methods.

The organization of this thesis is the following five chapters:

Chapter 1 presents the general background, objective, and scope of the research work.

Chapter 2 provides a detailed review and applications of different ways and algorithms to do simulation, GPU structure and the CUDA model, parallel and distributed computing, and the MPI communication model.

Chapter 3 describes the theories and simulation environment used in the particle simulation. The designs of GPU program and cluster program are also explained.

Chapter 4 presents the implementation of the particle simulation on a cluster using the master-slave method and the distributed memory method.

Also implementations of the particle simulation on GPU using the particle method and the grid method are described. One is using threads to control the particles, the other is using grid to control the particles. The two methods were tried using the CPU and their performance is compared.

Chapter 5 shows the discussion of the different results using three different approaches, how different methods affect the results, and the possibility of combining the advantages. As well, the summarization and conclusions of the present work are stated, and future work is described.

Chapter 2

Literature Review

This chapter contains background information for particle simulation, different parallel designs of particle simulation, two approaches used, which are most relevant to our particle problem, and some related work.

2.1 Particle Simulation

A particle simulation is the simulation of a dynamic system, predicting the motion of a group of particles. The particles can move by given velocities or under the influence of certain kinds of driving forces, like gravity, heat, and pressure. The problem of particle simulation covers a vast area, ranging from celestial body simulation to kinetic theory of gases and chemical reactions. Particle simulations can be classified to two types by the simulation methodology: event-driven simulation and time-driven simulation [8].

2.1.1 Event-Driven Simulation

This kind of simulation was first introduced by Alder and Wainwright [5]. The particles in the system collide with each other, which needs to be taken into consideration.

They tried the simulation of hard spheres moving and colliding. The first algorithm used was simple and computed in three steps.

1. After the start, the simulator computes all the time of next collisions in the system for all particle pairs. Choose the smallest time T_{min}.

2. Compute all the particles' properties to T_{min}.

3. Update the properties of the two colliding particles.

The simulation repeats the above three steps, which is known as the event-driven algorithm. The simulation is controlled by the collisions and the whole properties are changed after collisions one by one.

For this original algorithm, the amount of computation is very large. The simulator first calculates all the collision pairs for all the particles, then chooses the earliest one. This amount of computation will be repeated for every collision. After further study, Alder and Wainwright concluded that some of this calculation is redundant [5].

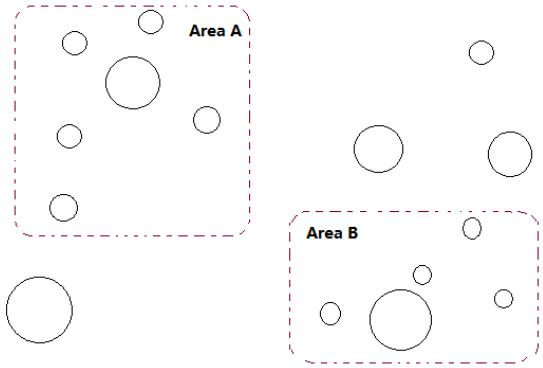


Figure 2.1: The distances between particles are different.

In a particle system, distances between particles can be short or long. As illustrated in figure 2.1, some particles in area A are far from particles in area B. A collision, which happens far away in area A, can not affect the particles in area B.

Therefore, the collision times for particles in area B do not need to be recalculated in the next collision cycle, because the collision times will stay the same. In order to solve this problem, the event queue is introduced [2]. This queue is used to save all the collision times and save a lot of computation time. After every collision, only the particles involved in the collision need to be recalculated and their old collision times will be discarded, but others' will remain. This way, a lot of computation is done away with.

Perumalla et al. [25] worked on reversible simulations of elastic collisions which make full use of the event queue. They developed a new algorithm to recover the pre-collision state from the post-collision state of the system, with essentially no memory overhead. They did the experiment and compared the results with CPU and GPU. It turns out that when the ratio of computation to memory operations becomes higher and architectures compare to themselves, GPU shows much more efficiency than CPU facing large scale of data.

After using the event queue, Alder and Wainwright suggested another method [5]. In the first step, the collision times are calculated for all particle pairs. But for a particle pair, if the two particles are very far from each other, they may not collide until the end of the simulation, so there is no need to calculate the collision time for this pair. The cell method is applied to help. Using this method, a cube containing all the particles is divided to a matrix of small cubes, called cells. Each particle is located in a cell. Then the computation scope of collisions can be reduced by the area. The simulator only computes the collisions for particles in neighbouring cells. This technique imposes additional memory use in that the particles' cell must be tracked. The figure 2.2 shows the structure of the cell method. The black particle needs to calculate the collision times with the particles located in the gray neighbouring cell.

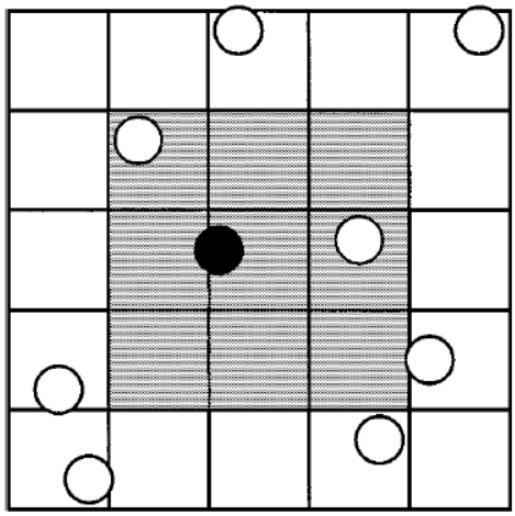


Figure 2.2: The black particle only computes collision times with particles in the 9 shaded cells [30].

The idea of the cell method can not only apply to event-driven simulation, but also to time-driven simulation. For our work, the grid-method applied to CPU and GPU holds the same idea with the cell method, which is dividing the space. Event-driven simulation is useful when there are collisions. On the other hand, if there is no collision, time-driven simulation is another choice.

2.1.2 Time-Driven Simulation

Unlike event-driven simulation, in time-driven simulation the predicted time of collision is not used to determine when calculations are done. Both kinds of simulations can have driving forces, like gravitational force, heating and so on. But the main control factor for time-driven simulation is the time step, which is a duration of time given by the system. The whole system is updated every time step. The basic algorithm is computed in two steps.

1. After the start, the simulator computes all the driving forces for all particles in

the system.

2. The simulator uses the new forces and the time step to calculate the data and update the properties.

For this original algorithm, the amount of computation is large, but the accuracy is high compared to Barnes-Hut algorithm, which is discussed next. For every time step, the simulator calculates the forces of each particle with all the other particles. This part of computation is necessary to provide the right track of particles. When the number of particles is very large, the computation time is also relatively long. The original algorithm will be changed when it is applied in different simulations.

In the time-driven simulation, the system updates after discrete time-steps. Therefore, choosing an appropriate time step is important. In some simulations like ice floe, stars, fog, heat, protein and so on, these objects are moving under certain equations of motion and their shape are like round or spherical particles, each equation will have an appropriate time step by calculation. Some application with similar framework are discussed next and time-driven simulation has been chosen for our research.

Here two kinds of algorithms are introduced, in which their main ideas are similar to those in our work.

2.1.2.1 Barnes-Hut algorithm

As mentioned before, the original algorithm for time-driven simulation is not efficient and improvements are necessary. The first idea is to compress the space and reduce the original amount of calculation.

The most widely used algorithm for compressing the space is the Barnes-Hut algorithm, which was introduced by Josh Barnes and Piet Hut. For the original algorithm, if a system has n particles, the computation should be done for every n particle with

all *n-1* particles for each time step and the complexity of this problem is $O(n^2)$. The Barnes-Hut algorithm reduces the complexity to $O(n \log n)$. The algorithm is described by Ravindra M and V Chaithanya [20].

The Barnes-Hut algorithm uses hierarchical approximations to do the simulation. If a group of particles have enough distance from particle i, then the force between the group and particle i can be seen as the force between one particle j and particle i. The quality of particle j is the total quality of the group and the position is the center of the group. Figure 2.3 shows the idea for approximation.

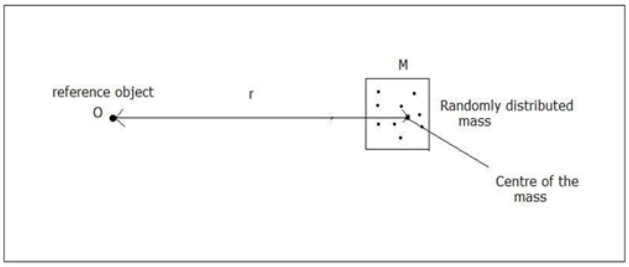


Figure 2.3: The idea of Barnes-Hut algorithm [20]

The algorithm needs to divide the space using quad-tree on 2D or octree on 3D. A quad-tree and octree are both tree data structures. In the quad-tree, each internal node has exactly four children and octree has eight children. They are most often used to partition a two-dimensional or three-dimensional space by recursively subdividing it into four quadrants or eight octants. If the space has more than one particle, the space will be divided again until there is only one body in each space. The Barnes-Hut algorithm reduces complexity and compute time, but also reduces the accuracy of results.

This algorithm is widely used in N body simulation, which mainly describes the movement of celestial bodies like stars and planets. One limitation of this algorithm is that the main force should be due to attractive forces between the particles, like

gravity or magnetism. This algorithm is not fit for our work because the particles in our simulation have no attractive forces and move under fixed velocities in the absence of collisions. Winkel et al. [35] used this algorithm to do N body simulation under an efficient parallelization strategy using MPI-Pthreads, and applied it to laser–plasma interaction and vortex particle methods. The approach proved to have excellent scalability on different data sets and an advanced load balancing strategy on a cluster. Several other researchers have investigated applying these algorithms to other problems. Grama et al. [13] also tried different parallel formulations of particle simulation based on the Barnes-Hut tree algorithm. Spurzem [31] described the direct force N-body simulation and the methods for gravitating systems of star clusters and astrophysics. He focused on the astrophysical application using Fokker-Planck equation and parallel implementation. Ananth Grama [13] introduced a new scheme to do N body simulation using this algorithm. Rather than shipping data to processors needing them, they tried to ship computation to processors where data reside. Their formulation proved to have less communication than the original scheme.

2.1.2.2 Cell-List Method and Neighbor-List Method

Apart from compressing the space, another idea to reduce the complexity of the problem is to reduce the amount of calculation for each particle. There are two algorithms that do this: the cell-list method and the neighbor-list method. These two methods can either be used individually or together in molecular dynamics simulation.

Particle simulations play an important part in molecular dynamics (MD) simulations which have become a standard tool for the investigation of biomolecules [15]. MD simulation is based on the principle of molecular movements, which are modeled by calculating the change over some time of position, velocity and acceleration, then calculating the thermodynamic quantities from the results, and analyzing to compute

quantities for the next time step. It is widely applied in physics, chemistry, biology, material and medicine areas.

Hansson [15] reports that today, MD simulation is performed for three main reasons. First, it shows the insight of bio-molecular structures on different timescales. Second, MD simulations calculate thermal averages of molecular properties. Based on the ergodic hypothesis, a single molecule and its surroundings can be simulated for some time to get time-averaged molecular properties, which can experimentally measure ensemble averages. Third, MD can be used to find which conformations of a molecule or a complex are thermally accessible. This method can be used for exploring conformational space. MD simulations always depend on their own equations of motion and potential-energy functions which lead to different movement tracks, therefore most of them are time-drive simulations.

Using these two methods on MD simulation, the cell-list method divides space into smaller space. The molecule in local space has zero force with far space(this distance can be set appropriately). The neighbor-list method will set a neighbor list for each molecule which stores the information of its neighbor molecule. The neighbor-list method's algorithm complexity is high, for the simulation needs to calculate the force between all molecules. By combining the two algorithms together, the simulator first divides space, then sets neighbors, each cell only takes its own cell and nearer cells into consideration. In figure 2.4, the left picture indicates the combination of the two algorithms and the right one shows the neighbor-list method. The r is the distance for the deciding neighbors.

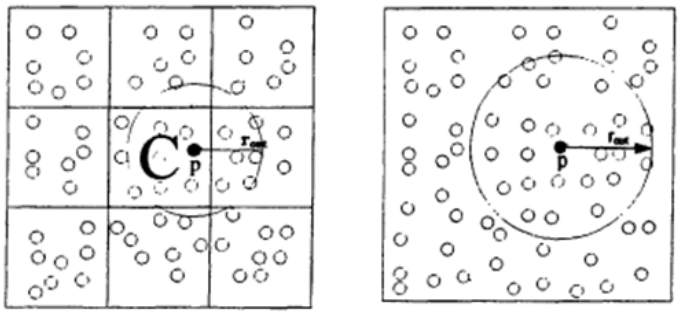


Figure 2.4: The cell-list method and the combination of the two algorithms

2.1.3 Collision Detection Technology

In a particle simulation with spherical particles, one key point is how to find the collision between particles. The general idea is checking the positions of particles, to see if a collision occurs. Either in two dimension or three dimension, if the length of the line segment joining the center points of two particles are longer than double the radius, there is no collision, otherwise a collision happens. In a discrete time-step simulation, a collision is also depends on the choice of time-step.

The cell method is another efficient way to solve particle simulation. Within this method, the simulation space is divided into smaller cells. The system will search and sort the particle positions in order to find the right cell. This procedure is called grouping of particles into cell structures. Checking collisions between particles changes to checking cell positions. This method is similar to the grid-based method [14] and the cell-list method. But when using the cell-list method in MD simulation, the length of a cell should be equal or bigger than truncation radius(if the distance between two molecules is longer than truncation radius, there is no acting force between them). The cell method used in our research can set the length of a cell freely, as long as the length is longer than particles' radius. The implementation of cell structures is

discussed in detail in [33].

Another effective method is mentioned in [19]. As illustrated in figure 2.5, the system needs to calculate the distance l_i between the beginning of system coordinates X_0 and the point of a particle which lies on the sphere at the particle [19]. The overlap of particles is needed to check the collision, which has proven effective in both two dimension and three dimension simulation.

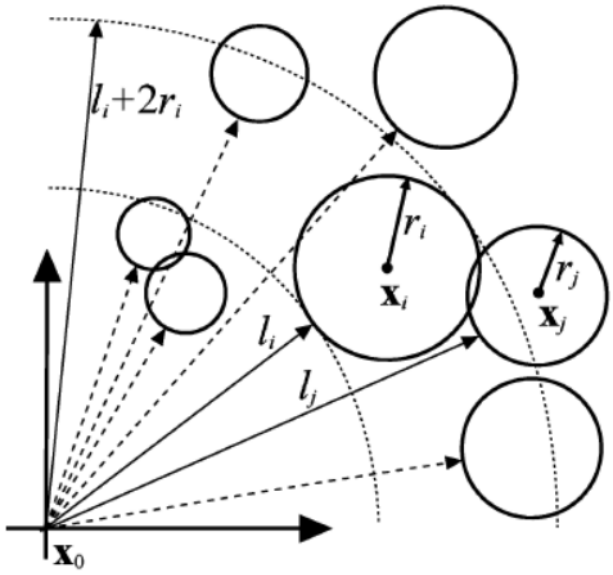


Figure 2.5: Scheme illustrates algorithm with notations [19].

Researchers also found methods for detecting collisions with non-spherical particles. FAN et al. [12] applied an octree-based method to solve this problem in large-scale particles system using GPU. Their result shows an optimization on particles colliding with both sides of the geometry surface, while the accuracy of the method is limited by the maximum subdivision level of the octree structure.

Some research of the collision of other geometry also introduced useful methods. In Sul's research [32], they solve the 3D triangle-to-triangle collision problem by changing it to a simple 2D point-in-triangle problem using matrices. By partitioning the space

using voxels, it is easy to find collision pair triangles. The results prove the advantage of using matrices to solve collisions. Choi et al. [11] also use matrices to solve the collision problems of ellipsoids.

2.2 Parellel Programming

2.2.1 Parallel Programming Design

Parallel programming is a form of computation in which calculations can be carried out simultaneously [34]. A large problem can be divided into smaller ones, which can be solved concurrently. The main aim of parallel programming is distributing the workload. Flynn created a taxonomy to classify the parallel computers, depending on the instruction stream and data stream [27]. Figure 2.6 shows different computer architectures.

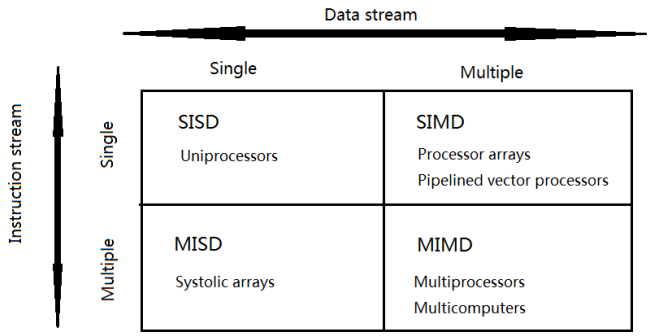


Figure 2.6: Flynn's taxonomy of computer architectures [27].

A computer with one processor that executes a single stream of instructions is called a single instruction stream-single data stream(SISD) computer. A multiprocessor system in which each processor executes the same instruction using different data simultaneously, can be classified as single instruction stream-multiple data

stream(SIMD) type. Processors arrays and pipelined vector processors are examples of this type.

The MISD type refers to computers with multiple instruction streams, but only a single data stream. Flynn's description of MISD computer is "a pipeline of multiple independently executing functional units operating on a single stream of data, forward results from one functional unit to the next" [27].

A system in which each processor executes different instructions using different data, is called multiple instruction stream-multiple data stream(MIMD) type. The shared memory and message-passing multiprocessors and multicomputers are all examples of this type. In MIMD computers, there are two programming structures: one is multiple program multiple data(MPMD) structure and the other one is single program multiple data(SPMD) structure. The former has different programs run on each processor, while the latter one has only one program copied by all processors.

For our particle simulation, a MIMD programming environment is chosen as CPU cluster, and SPMD programming structure is applied. Each processor holds a different data memory copying from the source data and the source program can be executed by certain processors depending on the identity of processor.

The general procedure of designing a parallel program has the following steps [34].

1. **Divide** Divide the task into smaller ones.

2. **Communication** Consider the communication between each small task.

3. **Combine** According to the small task's locality, some small tasks can be combined.

4. **Mapping** Allocate all the small tasks to processors

2.2.2 Parallel Design for Particle Simulation

For our particle simulations, the main computation time costs are collision detection and update, which are similar to the molecular dynamics simulation. The parallel algorithms can be classified into the following three types [26].

1. **Task parallelism** Task parallelism, also known as function parallelism or control parallelism [34]. For a particle system of N particles, the simulator evenly distributes the particles among P processors. Every processor will deal with N/P particles. Particles are allocated randomly, so every processor should keep the record information of all the particles. When every processor finishes an iteration, the new information should be communicated to all processors to make sure that every processor has the latest information of all the particles. This design is easy to program and the load is perfectly balanced on each node. The drawback is that the communication time for the processors is more for the global information updating. This design is suitable for shared memory structure.

2. **Force parallelism** Consider the particle system without collisions: the main computation will be the force computation like potential energy. For every particle, the force is the sum of the force computed with all the other particles.

$$F(i) = \sum_{j=1}^{n} f(i,j)$$

Variables *i,j* stand for particle *i,j*. Expression $f(i,j)$ indicates driving force between particle i and j. Expression $F(i)$ is the sum of $f(i,j)$ with j from one to n and n stands for the number of particles. Since the force of one particle is a summation, rather than distributing the particles, the simulator can evenly

distributes the summation computing and gather the results, which fits for some specific occasions.

3. **Space parallelism** This method divides the whole simulation space into several subspaces and the size of the subspaces can be either all the same or variable. Each process can control one or more subspace. Consider the particle system with collisions: the main computations will be collision detection, updating velocities and positions, and sometimes force computation is also needed. Using this method, every processor deals with one subspace and the particles inside this area. For this design, every processor should only hold the data of particles belonging to its area. The program needs a topology to support the mapping of areas to processors. One additional computation is the tracking of particles passing the boundary, receiving information from and sending information to neighbor processors. The global communication is local, which will offer better performance and will be suitable for large particle simulation, especially on the cluster.

For our work, methods of task parallelism and space parallelism are both applied and tested.

2.3 Parallel Architectures

As mentioned in 2.2.1, a MIMD programming environment can be either a single computer with multiple processors or multiple computers linked by a network. So there are two basic types:

1. Shared memory multiprocessor.

2. Distributed-memory multicomputer.

2.3.1 Shared Memory Multiprocessor

The shared memory multiprocessors can not only provide a general parallel environment, but also a shared memory structure. In a shared memory multiprocessor system, there is a single address space which stores all the data and can be accessed by all processors. Wilkinson and Allen describe programming: "Programmers can make processors execute its own program or code sequence from the shared data(Typically, all processors execute the same program)" [34]. There are many platforms which can provide shared memory structure, like a multi-core computer. Programmers can use specific programming languages to declare shared variables and parallel code sections. The key to programming shared memory is controlling the access of shared data. Each time, the shared data can only be modified by one instruction.

Using shared memory is a convenient way to program, but there are shortcomings. Small shared memory multiprocessors can achieve good performance, but when there is a large amount of processors, it is difficult to implement the hardware to achieve fast access to all the shared memory by all the processors [34]. Some programming issues are also important, like the control in accessing the shared memory and avoiding deadlock.

2.3.2 Distributed Memory Multicomputer

Using a distributed memory architecture, distributed computing and distributed shared memory approaches can both be applied. In distributed computing, each process sees its own memory space, but can only access information held by other processes via message passing. In distributed shared memory approach, all processes see a global shared memory, even though the actual memory is distributed. For our work, distributed computing on a CPU cluster is chosen. By using this approach, each comput-

ing node has its own memory with different addresses, and appears in the physically distributed memory as a single memory, as illustrated in figure 2.7.

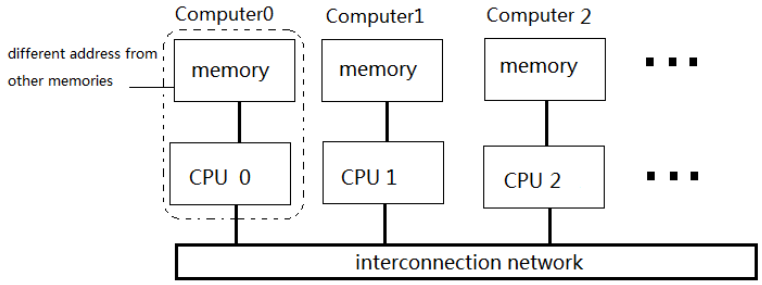


Figure 2.7: Distributed memory system

When a processor needs to access the data located in other processors, message passing must be used. However, the distributed memory system has some disadvantages because the interconnection between stand-alone computers is much slower than the interconnection within a shared memory multiprocessor system [18].

Bagrodia [8] presented a unifying framework for distributed simulation, including discrete-event and continuous simulation. He presented a new algorithm and experimental results on a cluster, which provide a efficient parallel design for distributed simulation. Chidester and George [10] explained a distributed simulator targeting the multiprocessor architectures using Message Passing Interface(MPI). The paper analyzed the performance on a CPU cluster of different parallelization designs, such as distributed, centralized, blocking and found that speedups were largely affected by cache accessing. Gregory [6] analyzed paradigms for process interaction in distributed programs and provided possible solutions to most common distributed programs.

There are many distributed algorithms discussed in [6], each of them fits for dif-

ferent kinds of problems. For our particle simulation, the heartbeat algorithm is chosen. This algorithm is useful for many data parallel iterative applications [7]. The following pseudo-code shows the outline of the heartbeat algorithm. Suppose there is an array and several workers. Each worker deals with a part of the data.

```
for process worker[i=1 to numworkers] do
    declarations of local variables;
    initialize local variales;
    while computation is not done do
        send values to neighbours;
        receive values from neighbours;
        update local values;
    end while
end for
```

When the data is divided among processors and the updating of each processor depends on its own data and neighbors' data, then a heartbeat algorithm shows advantage. It applies to applications like image processing, solving partial differential equations, and simulations of phenomena. The send and receive actions will perform communication between processors and create barriers to control the algorithm. For ice simulation, the simulation space can be divided into smaller parts, which are almost independent of the other parts, and each part deals with movements in its own area including its edges. A heartbeat algorithm will work well because ice simulation can be done efficiently by cross-cell communication, which decreases overhead.

The heartbeat algorithm is also widely used in networks, especially in managing network access and interaction. From the property of the heartbeat algorithm, if all processors keep on communicating with their neighbors, every processor can

eventually receive information from all other processors.

2.4 Message Passing Interface

Message passing interface (MPI) is used for communication between processes in parallel programming. The basic communication model is sending and receiving messages between processes, which is also called point to point communications. There are two types of point to point communication, blocking and non blocking.

Using the blocking function, the system won't return until the specific operation is completed or the data is buffered safely. Blocking functions need the recipient process to confirm that transmitting is complete. The MPI_Send and MPI_Recv are two basic blocking functions, which are a pair and used together. When MPI_Send returns, the stored data has been fully transmitted to, and received by the recipient process and any change done to the stored data at the sender cannot effect the data existing at the receiver. When MPI_Recv returns, the data is fully received and can be used directly. While the non-blocking functions always return immediately, other operations are done by system in the background. The non-blocking function use 'I' to distinguish with others. MPI_Isend and MPI_Irecv are the non-blocking pair functions for send and receive.

After calling non-blocking functions, a program must call MPI_Wait or MPI_Test to check the status of the operation. Unlike with blocking functions, before the completion of the operation, the stored data is unsafe because of the possibility of conflict with other communication. The non-blocking functions do not need the recipient process to confirm, so they can be used together in parallel programming to overlap the computation and communication. Both types are applied in this work. MPI provides four communication modes: standard communication mode, buffered communication

mode, synchronous communication mode, and ready communication mode. They are different in managing buffer and ways of synchronization between sender and receiver. Together with blocking and non-blocking functions, MPI produces eight kinds of send operations and only two kinds of receive operations. A programmer can choose the best mode to solve the problem. In our work, only the standard communication mode is needed.

2.4.1 Communication Time

MPI is the most widely used communication interface for cluster-type parallel/distributed computers [21]. Of course, the performance is a key topic of MPI program. As for a MPI program, some factors should be taken into consideration, the communication time between processes and the communication mode used. MPI Messages not only contain data information, but also consist of "envelope" information, such as tag, communicator, source, destination, the message length and other implementation specific information [36]. Considering the scheme of transferring data, there are four types of data [18] [36].

- Short: Data is transmitted within the message envelope.

- Eager: Data is transmitted without the message envelope and does not need acknowledgement from the receive process.

- Rendezvous: Data is transmitted without the message envelope for specific receiver's request.

- Get: Data and envelope are read directly by the receiver with special methods, such as shared memory.

The selection of data transfer scheme can contribute a lot in choosing blocking mode or non-blocking mode. For our research, two parallel methods are used on CPU cluster. The master-slave method can either copies the whole data array or data segment, which depends on what the slave process requires. For our implementation, the slave processes copy the whole data and return contiguous data segments which can be integrated by the master process. However, when using the heartbeat algorithm, each processor holds different data and sends the specific and non-contiguous data to the others. For contiguous data, the standard communication mode is enough, while for non-contiguous data, a temporary data buffer is required. The system will copy the specific data into buffer memory and all send and receive operations are done between buffers. The operation of creating buffer memory, handling the order of operations, processing the operations, and releasing buffer memory are completed by communication middle ware. In order to improve the efficiency, the computation time on each computing node and the communication time for middle ware can both be reduced.

Data consolidation Sending an array of one hundred items once is certainly much faster than sending one item one hundred times [36]. To reduce the number of messages sent, data consolidation is important. MPI provides three basic methods for data consolidation: the count argument, derived data types, and pack function pairs. MPI provides derived data types, which can be used to represent any collection of data items and are convenient for transmitting different data together. For example, assume there are three numbers to be sent: a double value, an integer value, and a float value. MPI can derive a new data struct that contains three values with count equaling three. MPI_Pack and MPI_Unpack functions are useful in sending and receiving non-contiguous data. The system uses a pack function to pack the non-contiguous into contiguous memory and store the packed data in a declared buffer

memory.

In addition to the communication pattern, the system topology will also contribute to the performance. The topology of processors can be mapped to the divided simulation area. Derived data types and pack function are both applied in our work. A new particle data types is used to represent the collection of different properties, like velocities, positions, and weights. When processors update, all information are updated by copying the memory once. As mentioned above, When the heartbeat algorithm is applied, non-contiguous data occurs. A processor collects all non-contiguous data in the buffer and calls pack functions to send and receive from the other processors.

Le and Rejeb [18] introduced a model to address the cost of middle-ware communication inside the memory, and the cost of interconnecting within the network. Their results shows its great ability of evaluating performance. Brandfass et al. [9] described a procedure for optimizing MPI communication by reordering the MPI ranks. They created a mapping of MPI processes to CPU cores and used distance matrices to distinguish the processes in each same computing node or not. Their results showed that the communication between processes on the same computing node is usually much faster than that on different nodes.

2.5 Graphics Processing Units (GPU)

2.5.1 The Development of GPU

Graphics Processing Units are designed for highly parallel and data intensive computing. Before the appearance of GPUs, all the work load were taken over by CPUs. The early GPUs were simple, leaving most work to the CPUs. TMS34010, the first microprocessor with on-chip graphics capabilities, was released in 1986, which could run general-purpose code [2]. In 1987, X68000 created by Sharp, was powerful for home

use, with a 65536 color palette [2]. By 1995, all major PC graphics chips had 2D acceleration support [2]. Beginning with the appearance of Nvidia's GeForce256 in 1999, the computation of 3D display is fully managed by GPUs and the efficiency is largely improved. These improvements free CPUs from much of the complex computation and work load.

Now GPUs are most widely used for two aspects, one is entertainment and the other is computing. The Nvidia company contributes a lot in GPU programming and produces several series of GPUs for specific purpose. The Nvidia Tesla series are designed for GPU computing. They are used to accelerate the computation of scientific applications due to their computing power. The Tesla K80 GPU Accelerator, for example, has thousands of compute cores and can perform up to 2.91 Teraflops of double-precision floating point computation [23]. The Nvidia Quadro series provides a highly compatible platform to conduct work in many subject areas. They show excellent performance in designing digital products, dealing with image and energy, managing media and so on [1]. They also support the GPU programming with CUDA programming language.

2.5.2 GPU Structure and CUDA Technology

As previously mentioned, because of the large number of computing cores, GPUs show great capability in computing. The reason is the different design in structure. As illustrated in figure 2.8, there are more transistors in GPUs than CPUs to deal with data processing rather than cache and control. So GPUs are less well suited to algorithms with complex data structures or logic. GPUs show an advantage of solving problems with large amounts of parallel data to be processed with similar steps and high density computing.

Figure 2.8: Different structures of GPU and CPU [24]

One parallel programming model used on GPUs, called CUDA (computing unified device architecture), reduces the difficulty of general purpose computation and improves the efficiency of the GPUs [23]. The role of CUDA is to treat the CPU as a terminal, and the GPU as a device to run the task; which can be highly threaded. So CUDA is a combination programming model of CPU and GPU. CUDA is the extension of C language. Programmers can use C, C++, FORTRAN and other high level programming language to program on the GPU. The CUDA code has two parts: the serial part running on the CPU and the parallel part running on the GPU, which is called kernel functions.

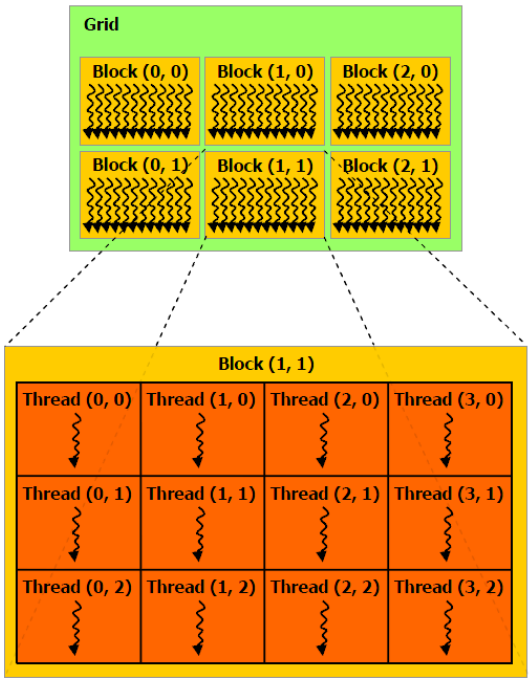


Figure 2.9: The grid block and thread block [23]

As illustrated in figure 2.9, each kernel has a grid and a grid has many thread blocks, each containing several threads. In each block, the threads work at the same time though barrier synchronization and shared memory, which are visible to the block. Thread blocks can be one, two or three dimensional.

2.5.3 Compute Capability

The compute capability is a standard to measure the performance of each GPU card, which consists of a major revision number and a minor revision number. The major revision number indicates the core architecture and minor revision number indicates minor improvement. Figure 2.10 shows all the compute capabilities of GPU cards and some of their attributes.

For our work, the GPU card used is Quadro FX 1800 card, which was provided by the project and its compute capability is 1.1. This card does not support double-

precision floating-point numbers, so the experiments used single-precision floating-point numbers. The single-precision does not influence our research much. In order to give a relevant comparison with CPU, both data types were tested on the CPU, which are discussed in chapter 4.

Feature Support	Compute Capability						
(Unlisted features are supported for all compute capabilities)	1.0	1.1	1.2	1.3	2.x	3.0	3.5, 5.0
Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)	No	Yes					
atomicExch() operating on 32-bit floating point values in global memory (atomicExch())							
Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)	No		Yes				
atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())							
Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)							
Warp vote functions (Warp Vote Functions)							
Double-precision floating-point numbers	No			Yes			
Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)	No				Yes		
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())							
__ballot() (Warp Vote Functions)							
__threadfence_system() (Memory Fence Functions)							
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Synchronization Functions)							
Surface functions (Surface Functions)							
3D grid of thread blocks							
Unified Memory Programming	No					Yes	
Funnel shift (see reference manual)	No						Yes
Dynamic Parallelism							

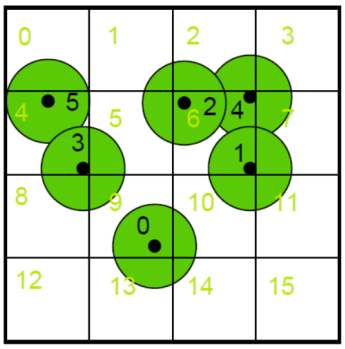
Figure 2.10: Different compute capabilities [24]

2.5.4 GPU Application on Particle Simulation

Due to the great computing power of GPUs, different kinds of simulation are applied on the GPUs. Because of the stream processing paradigm, most simulations can share the same parallel scheme with multi-core computing. Rather than develop better algorithms, how to do the parallel design is the main question.

Despite the general particle method used for particle simulation, Green [14] introduced the grid-based method using a fast radix sort, which is applied in our simulation. This method is similar to the cell-list method mentioned above, which also divides the space. The grid-based method needs to build the grid for particles. The system

can quickly access to the certain particle with the help of the fast radix sort. The radix sort is provided by the CUDPP library, which is described in [24]. Firstly, the system calculates the hash value for each particle based on its cell id and an unsorted list of particles is shown in figure 2.11. Secondly, the system uses the radix sort based on particles' hash values to produce a sorted list by cell id. Thirdly, the system uses a thread per particle and compares its index of located cell with that of the previous particle, whether two particles are in the same cell is shown. As a result, the first particle in a given cell compares with other particles along the list until the last particle. The computation is computing the particles in same cell and neighbor cells.



Index	Unsorted list (cell id, particle id)	List sorted by cell id	Cell start
0	(9, 0)	(4, 3)	
1	(6, 1)	(4, 5)	
2	(6, 2)	(6, 1)	
3	(4, 3)	(6, 2)	
4	(6, 4)	(6, 4)	0
5	(4, 5)	(9, 0)	
6			2
7			
8			
9			5
10			
11			
12			
13			
14			
15			

Figure 2.11: The grid using sorting and particles' list in cells [14]

2.5.5 GPU Cluster

Our work is done on a CPU cluster and a single GPU. Rather than implementing on a single GPU, distributed parallel programs running on a set of GPUs will be a great improvement. In a way, GPUs are the most powerful hardware on computing, and a cluster consisting of GPUs will combine computation power effectively. So the idea of applying GPU clusters is more and more popular, which combines both advantages of GPUs and cluster. A GPU cluster is a powerful computing platform which consists of a number of CPUs and a number of GPUs working together. All devices link each

other by interconnecting with different topology architectures.

A GPU cluster can be either one of the following two types: heterogeneous or homogeneous. The former one consists of different GPUs(model, make) and the latter one consists of same GPUs. A cluster has a great advantage over a multiprocessor system, the clusters can incorporate new processors at low cost and expand its scale easily by adding computers, disks and other resources. Almost all parallel algorithms can be applied on the cluster. In a GPU cluster, the work flow on a computing node follows the same programming model on a single GPU. The CPUs will still act as the host and the GPUs act as the device. Except for the original duties of allocating work from the host to the device, CPUs need to control communicating and transferring data between CPUs. A CPU server can be linked with more than one GPU. The programming idea of a GPU cluster is fully loading the GPU with computation and freeing the CPU to handle communications and workload management [22].

Kindratenko [16] and his partners presented a technique for building and running GPU cluster in HPC environment. With the increasing number of cores, efficient mechanisms for sharing GPUs among multiple cores is a significant need. Showerman [29] introduced a GPU cluster structure which also proved to be power efficient. Yang [37] tried a general parallel model using CUDA, MPI and OpenMPI on a GPU cluster, which partition loop iterations according to the number of compute nodes of the cluster. The results shows that the combination of computing power is a better choice facing some parallel problems. Zhang and Mueller [38] introduced a General-Purpose Data Streaming Framework on the GPU cluster.

Chapter 3

Methods and Approaches

In this chapter, different methods and approaches applied in the particle simulation and designs of the experiments are described.

3.1 Simulation Environment

The general particle simulation problem is to simulate the behaviour of interacting particles in a constrained space, colliding with each other and the walls. All the particles move under the influence of physical properties and the total momentum and total kinetic energy are unchangeable when collision happens. So the laws of conservation of momentum and conservation of kinetic energy can be applied to update the properties.

The conservation law of kinetic energy applied for two particles' collision:

$$m_i v_i^2 + m_j v_j^2 = m_i v_i'^2 + m_j v_j'^2 \tag{3.1}$$

The conservation law of momentum applied for two particles' collision:

$$m_i v_i + m_j v_j = m_i v_i' + m_j v_j' \tag{3.2}$$

In a two-dimensional model, the particles are considered as balls with a certain radius. The goal of the simulation is to show the performance of different algorithms. A simple X11 user interface is used to display the interactions of the particles. Figure 3.1 shows a screen shot of one moment during the particle simulation. Each line of dots represents a history of a single particle.

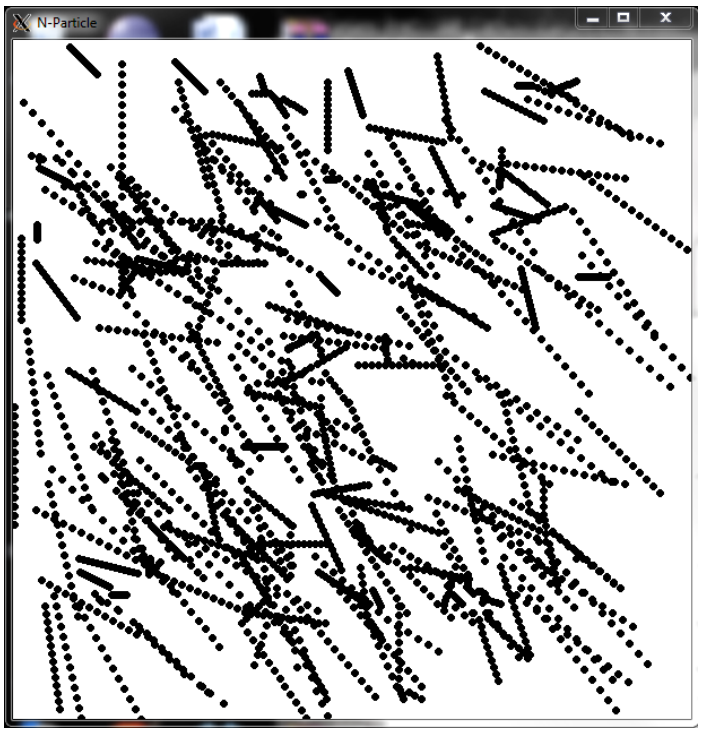


Figure 3.1: A screen shot of one moment during the particle simulation

3.1.1 Collision Configurations and Constraints

Consider a system that consists of N identical hard spheres of radius R in a two-dimensional area undergoing elastic collisions in two types: (1) single particle-wall

collisions, (2) particle-particle collisions. Each particle's velocities are along the two coordinate axis, V_x and V_y. The first type is straightforward to reverse, modelled by changing the sign of the velocity component that is orthogonal to the wall. If a particle faces a corner which has two walls simultaneously, all the appropriate velocity components change their sign. The second type follows the dynamic principle.

When collision happens without friction, the collision's line of action is perpendicular to the tangential line of two particles. When velocities are along the line of action, it is called a "direct impact". When the line of action passes the center of collision body, it is called a "central impact". Collisions of particles whose mass is uniformly distributed are all central impact. When velocities are along the line of action, it is calculated as the one dimensional collision and when velocities are not along the line of action, the collision is called a "oblique impact". On this occasion, we need to decompose the velocities. If components of velocity has the same direction with that of the line of action, they are only involved in the collision and other components of velocity which are perpendicular to the line of action are not. Figure 3.2 shows an example of oblique impact.

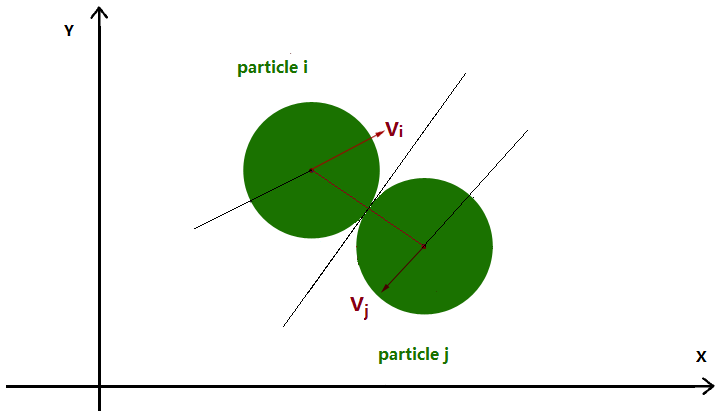


Figure 3.2: A collision of two particles

Let v_i and v_i' be the pre-collision velocity and post-collision velocity of particle i . For every pair of particles i and j that are in collision, their kinetic energy E and momentum P stay the same. Based on the unchangeable energy and momentum, we can compute the new velocity according to formulas 3.1 and 3.2.

For the simulation that is a time-driven simulation, the system will check the collision pair every time step. First, the distance between each particle and every other particles is calculated. If the distance is bigger than 2*radius, there is no collision; otherwise, a collision happens. When collisions are found, new velocities are computed by the collision function. If two particles are moving along the same line, formulas 3.1 and 3.2 can be applied directly, but particles' collisions are happening in every direction, like the figure 3.2. Because of this, the decomposition of velocities should be done first. As mentioned before, if the velocity is perpendicular to the line linking two particles, it will not change at all. The following shows the procedure to deal with the collision between particles.

if distance < radius*2 **then**

/* Collision exists */

```
double velocity_i, velocity_j;
double coordinate_i, coordinate_j;
double angle, sina, cosa;
```

/* Get velocities and positions for particle i and particle j; */

/* Calculate the degree of the line linking two centres of particle *i,j*;*/

/* Calculate the components of velocities of both particles which are along linking line; Calculate the components of velocities of both particles which are perpendicular to the linking line;*/

```
double newv_i=((m_i–m_j)*v_i+2*m_j*v_j))/(m_i+m_j);
```

```
    double newv_j=newv_i+v_i-v_j;
  /* Update local values;*/
  /* Calculate the components of new velocities of both particles which are along
  linking line;*/
  /* Calculate and update new velocities for both particles;*/
      double newposition_i=oldposition_i+ newvelocity_i*time_step;
      double newposition_j=oldposition_j+ newvelocity_j*time_step;
  /* Update new positions for both particles.*/
end if
```

3.2 Simulation Approaches and Architectures

There are five approaches applied for particle simulation: CPU, GPU, and CPU cluster (three architectures).

3.2.1 CPU

The particle simulation running on CPU is a basic serial program that used double data type and float data type to provide a comparison with other approaches.

3.2.2 General Purpose Graphics Processing Units and CUDA

As mentioned in chapter 2, streaming processing is the model of how GPGPU programming works and CUDA, is designed to help. CUDA, the computing unified device architecture, was developed by NVIDIA in 2007. This model reduces the difficulty of general purpose computation and improves the efficiency of GPU [23]. The role of CUDA is to provide a path from CPU to GPU, and to help GPU managing and

computing data. CUDA makes CPU work as a terminal, while it makes GPU work as a device to run the task. Figure 3.3 shows a general overview of CUDA. Different applications using DirectX, OpenCL, or other computing functions access GPU through the CUDA drive to finish their tasks effectively.

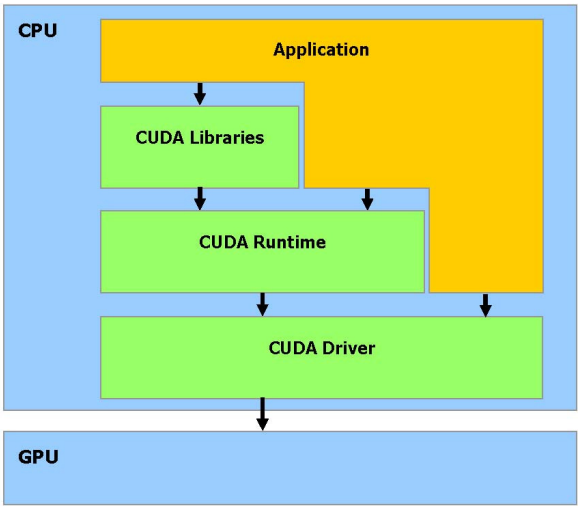


Figure 3.3: Overview of CUDA [24]

CUDA is a combination programming model of CPU and GPU, so the code has two parts: the serial part running on the CPU and the parallel part running on the GPU, which is called the kernel. When a program starts, the execution first takes place on the CPU(host). When a kernel function is invoked, the execution is moved to the GPU(device).

3.2.2.1 Framework

Figure 3.4 shows the high-level flow of particle simulation. At the beginning, CPU initializes the particle data(position, velocity, weight,radius) and the simulation parameters; then the initial data is copied from CPU to GPU. GPU will take control of the whole simulation and launch the kernel function to update the properties of

particles. After the simulation is done, GPU will return the particle data back to CPU.

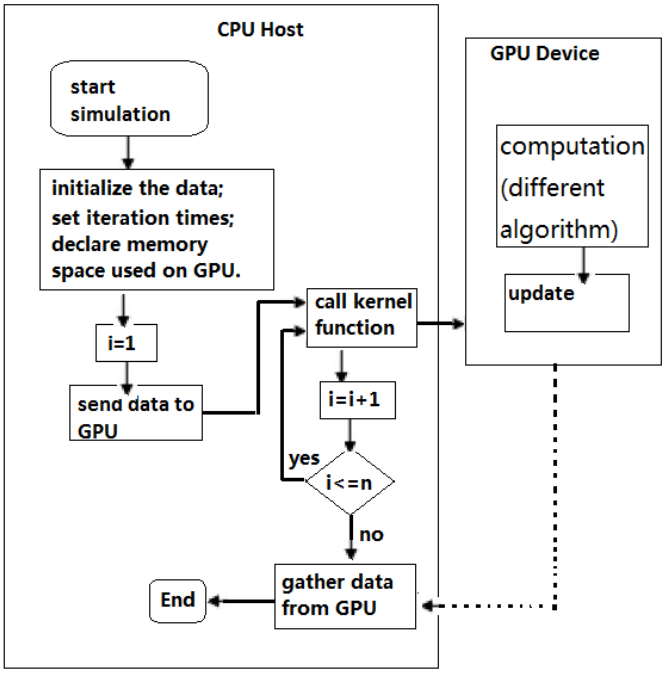


Figure 3.4: Simulation flow

3.2.3 CPU Cluster

A CPU cluster is a parallel computing platform consisting of many computing nodes and each node can perform the same or different tasks at the same time. The whole task can be split up into smaller ones using a master node and send them to different slave nodes. After the work is done, all slave nodes will return results to the master node, which is the basic processing model of the cluster. The communication is transmitted between a master node and slave nodes. Also the communication can be transmitted between slave nodes. Computing nodes find each other by the node ID. In the cluster, each computing node holds different memory.

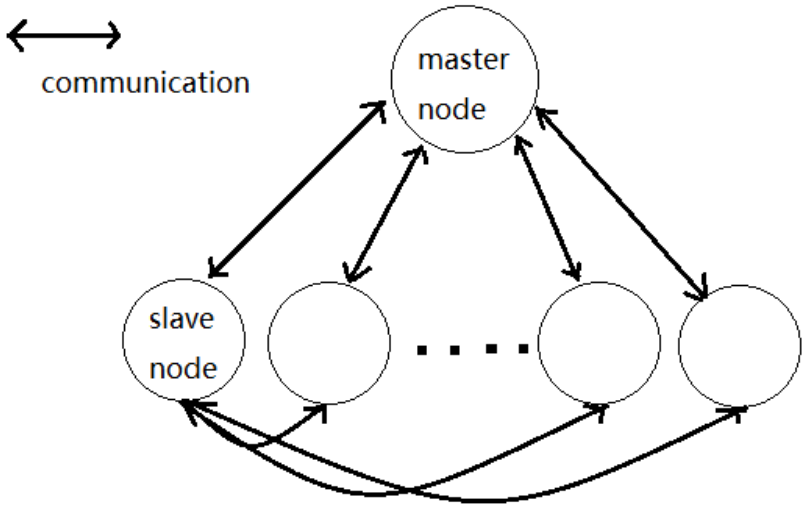


Figure 3.5: Communication between cluster computing nodes

Message passing interface(MPI) is a standardized and portable interface that is widely used to write message passing programs [21]. It is therefore an obvious choice for CPU-cluster computing using a distributed parallel model. As mentioned in chapter two, MPI can be applied to reconstruct the nodes to a certain topology in two dimensions and each node controls its area according to its position in the topology. Figure 3.6 shows the communication between neighbour nodes for a two-dimension topology.

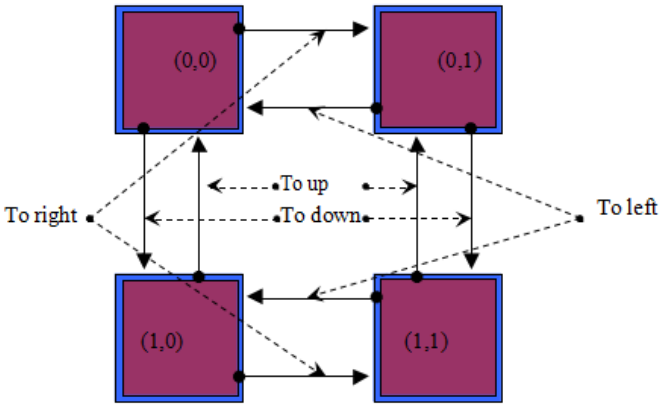


Figure 3.6: Communication for a two-dimension topology of four nodes

In order to make full use of this architecture, we try to distribute the particle data by dividing the simulation space and transmit the data of each subspace to particular nodes.

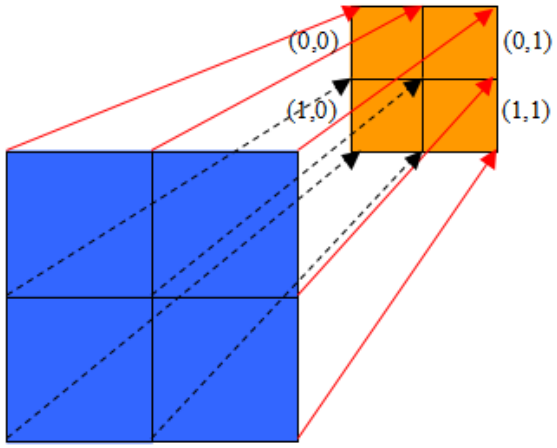


Figure 3.7: The mapping of space to virtual processors

Figure 3.7 shows the example of dividing the simulation space to four parts and their mapping to virtual processors.

3.3 Methods and Algorithms

There are three methods applied for different approaches in the particle simulation: the particle method, the grid method and the heartbeat algorithm.

3.3.1 Particle Method

Particle method is one of the basic types of particle simulation, which calculate the properties of a set of particles as they move [14]. This method has several advantages.

- The particles only perform computation(collision) when necessary.

- The particles' data structure is simply the position and velocity of each particle, which is an efficient way of representing the simulation state. The system can access all the particles though their index.

This method is applied on CPU, GPU and cluster. It is relatively easy to parallelize particle system for GPU which makes it possible to use one thread per particle. The kernel links the thread ID with particle ID. If there are n particles, GPU will call n threads. Each thread will search all the possible collision for its particle. On the cluster, a master-slave method and the heartbeat algorithm were applied to do parallel designs for the particle method. Rather than using one thread per particle, the master-slave method make one process control a group of particles. Therefore, the number of particles can be distributed evenly by the number of processes used. The heartbeat algorithm focus on parallelizing the simulation space and will be discussed later.

3.3.1.1 Particle Method on GPU

Using this method, one kernel is designed to call the collision function. This kernel focuses on the computing part of the serial code, that uses one thread per particle, finds the possible collisions with other particles, then computes new velocities, positions for the particle, and updates the new results. Figure 3.8 shows the procedure of the simulation.

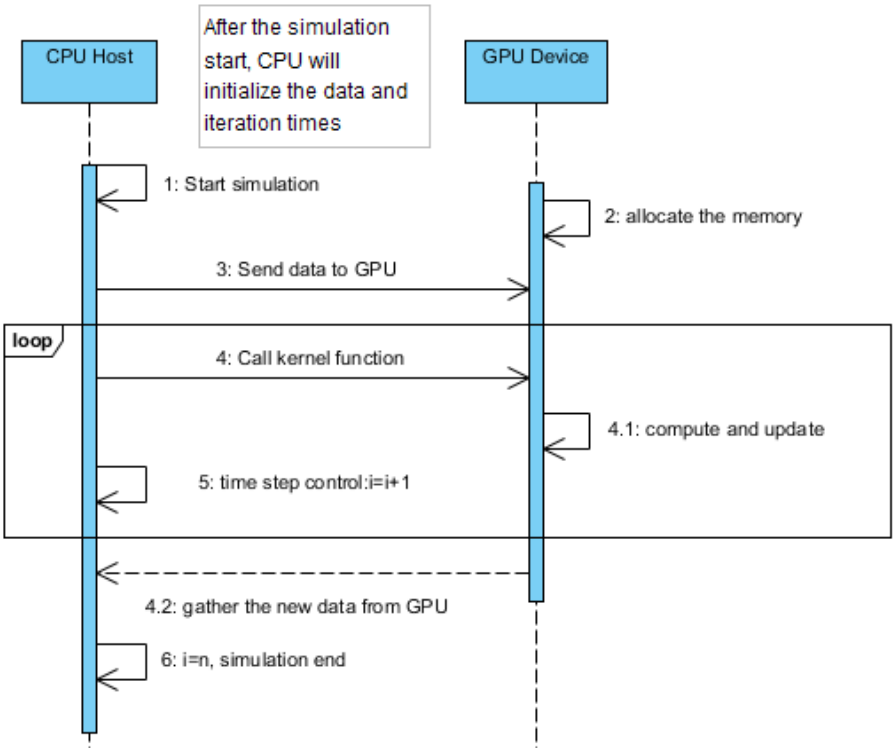


Figure 3.8: The procedure of the simulation using particle method

3.3.1.2 Particle Method on the Cluster

On the cluster, the particle data is stored in arrays. Each process can access the particle data according its index. Using the master-slave method, each process knows its segment of particles. When the updating of each segment is finished, the master process will gather the results from all slave processes. After one iteration is over, the

master process will send the latest information to all the slave processes again.

3.3.1.3 Heartbeat Algorithm

To fit the distributed memory system, a heartbeat algorithm [34] is applied to do a distributed design on the CPU cluster. Since there is a large amount of particle data, a distributed architecture may be a good way to speed up the performance. As mentioned in chapter two, the heartbeat algorithm is described as a network of many nodes. The actions of each node are like the beating of a heart: first expand, sending the information out; then contract, gathering new information in. This kind of interaction has also been called the wave algorithm[34] which spreads the information out like wave. This type of algorithm can be used especially to solve parallel iterative computational problems.

Using this algorithm and distributed computing, the original constrained area is divided into several physical subsections. Each node controls one subsection, and communicates only with its neighbor nodes, which reduces the communication. For each node, it only computes particles in its space, updating the velocity and position at each time step. For particles that cross the boundaries of the space, information about position, velocity, and weight will be sent to the appropriate neighbor. Each node will also receive the information about the particles passing into its space from neighbor nodes. Fig 3.9 presents the dividing space and the moving particles in a two-dimensional topology of 16 nodes. Particle 'a' will be sent up to neighbor node 4 from node 5 and particle 'b' will be sent to left neighbor node 9 by node 13. Particles "c" and "d" will both be sent to node 7 from node 3 and node 10 at the same time. Particle 'c', because it covers four nodes' spaces, will also be sent to node "6" and "11".

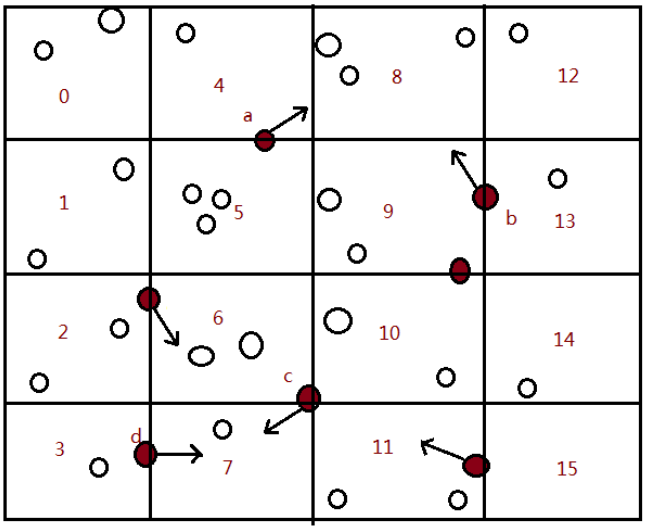


Figure 3.9: The example of dividing space with 16 nodes

The procedure of this approach has the following steps:

1. When the program starts, the master node will initialize all the particle data and make a topology according to the number of nodes, then send the appropriate data to other nodes.

2. Every node will check its neighbors and controlled area , then receive the particle data. They will produce their own list to store the particles that exist in their spaces.

3. When the preparation work is done, the heartbeat algorithm starts. Each of the nodes compute the particles only in its area. When a particle comes in the area or goes out of the area, that node will produce a list to store them and send them to the right neighbor according to its position.

4. Each node will wait for the others until one iteration is done. After all iterations are finished, every node will end their process.

3.3.2 Grid-based Method

In contrast to the particle method, the grid-based method calculates the properties of the simulation at a set of fixed points in space [14]. Using this method, a uniform grid subdivides the simulation space into a grid of uniformly sized cells, and the cell size is the same as the size of particle(double its radius). So a particle can cover maximum four grid cells in a two-dimension space. As each particle is located in only one grid cell based on its center point, in order to process collisions, we need to examine the particles in the neighbouring cells and the cell where it is located(3*3=9 in total)to check if there exists collisions. This method allow us to sort them by their grid index. Compared with the particle method, the grid-based method requires less work dealing with collisions, but more work when building the grid. This method is only applied on CPU and GPU.

3.3.2.1 Grid Method on GPU

Using the grid-based method, three kernels are designed. The first kernel calculates the hash value for each particle based on its index of the stored array. The second kernel rearranges all the particle data into sorted order using radix sort, and finds the start particle and the end particle of each cell. The third kernel does the collision computation of particles in each cell and neighbor cell. Figure 3.10 shows the procedure of the simulation.

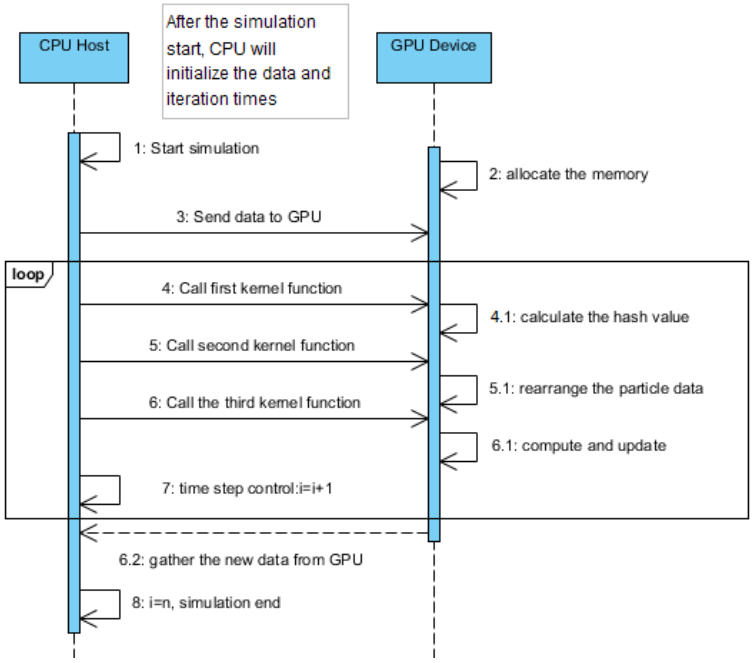


Figure 3.10: The procedure of the simulation using grid method

3.4 Experimental Design

The following experiments were completed. The results are provided in chapter four.

On the CPU, in order to get a relevant comparison, the particle method and the grid method were used on both float data type and double data type.

On the GPU, the two methods were tested. Also, by gradually changing the grid size of the original grid method, this changing grid size method was tried after.

On the cluster, all the experiments could be classified as two kinds: simulations on a fixed space and simulations on a fixed number per computing node. The former one was tried to compare the results with that on the CPU. The latter one was designed to analyze the performance of the heartbeat algorithm and the communication time used among the cluster. For the first kind, the master-slave method was tried. Then, the distributed approach using the heartbeat algorithm was performed. For

the second kind, based on the heartbeat algorithm, we gave each computing node a fixed number of particles and did the experiments on both one dimension topology and two dimension topology. The following figure shows the collaboration using the heartbeat algorithm.

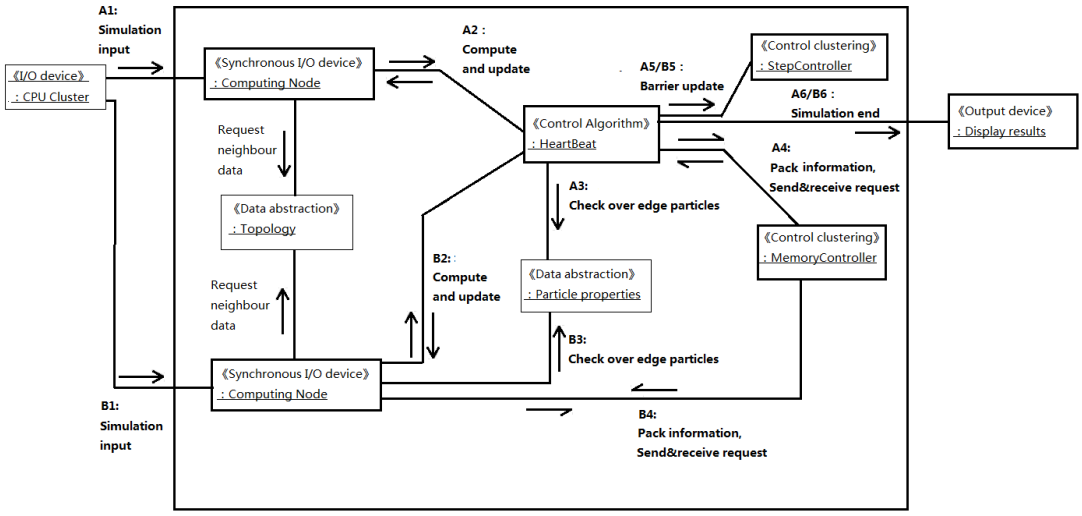


Figure 3.11: Concurrent collaboration diagram for the heartbeat algorithm

In the collaboration diagram, there are two computing nodes to be as an example. Executions A and B show two command flows on two different computing nodes. Command A1 and B1 are actually the same with each other. The simulator builds the topology according to the number of nodes used. Each node will know its area covered, neighbors and their reference ID. After computing the particles in its area(command A2,B2), each node will check if any particle across the edge(command A3,B3). If yes, the simulator will provide buffer memory to control the communication(command A4,B4). Nodes can pack necessary particle properties in the buffer memory, like positions and velocities, then send or receive the package which is hold by the buffer. When all the nodes finish, the simulator will update the time step(command A5,B5)

until the simulation ends(command A6,B6). All the nodes follow the same command from one to six.

The following table shows the outline of all the experiments.

Table 3.1: The outline of experiments

Experiment	Architecture	Method	Data type
CPU general	CPU	Particle method	Float data type
	CPU	Particle method	Double data type
	CPU	Grid method	Float data type
	CPU	Grid method	Double data type
GPU general	GPU	Particle method	Float data type
	GPU	Grid method	Float data type
GPU specific	GPU	Changed grid method	Float data type
Cluster general	CPU cluster	The master-slave method	Double data type
Cluster specific A	CPU cluster	The heartbeat algorithm with a fixed number of particles	Double data type
Cluster specific B	CPU cluster	The heartbeat algorithm with a fixed number of particles per node	Double data type

Experiments "CPU general" and "GPU general" stand for the particle method and the grid method used on two architectures. "GPU specific" uses the grid method with changed sizes of grid. "cluster general" means applying the master slave method on the cluster and "cluster specific" uses the heartbeat algorithm. In the chapter 4, there are comparisons between CPU and CPU cluster, CPU and GPU, and themselves. To keep the consistency, CPU cluster uses double type data. Results can not be compared directly between CPU cluster and GPU. While there may be little difference in execution speed on a single CPU between data types "double" and "float", on a multi-computer cluster, data has to be moved in and out of memory and shared between different computers over a network connection. As double data type variables occupy more memory, they take longer to transfer between networked computers.

While the double data type holds higher precision than the float data type. Therefore, the performance of CPU cluster is automatically at a disadvantage.

Chapter 4

Experiment Results

This chapter describes the results of all the experiments.

4.1 CPU General versus GPU General

For the program running on CPU, we set the numbers of particles as 2500, 5000, 6500, 7500, 8500, and 10000. This range of numbers are related to the area size(X*Y) as 300*300, 300*600, 400*585, 400*675, 600*600 to keep the same density as 78.5%. If the density is low, the frequency of collision is low and if the density is too high, particles will be too crowded to move. These area sizes are easy to divide between different numbers of computing nodes when the distributed algorithm is applied on the cluster. The whole simulation is running for 1000 iterations. The radius of particles is set as 3, and the simulation area changes with the number of particles to keep the same density.

The type of GPU card used for the GPU experiments is Quadro FX 1800, which has 64 processor cores, and compute capability 1.1. This card only supports the single-precision float type and does not support the double-precision double type. In order to compare the performance of GPU version and CPU version, we do the simulation

in both data types.

Figure 4.1 shows the computation time of the particle simulation on CPU and GPU. The dark blue line and the red line in the picture indicate the different data types used in the serial particle version: double type and float type. The green line and purple line indicate two data types used in the serial grid version and these two lines almost overlap with each other. The light blue line and the orange line shows the results of the two different methods used on GPU. As is shown, with the increasing number of particles, GPU is much faster than CPU. For the two different data types running on CPU, double type is more than four times quicker than float type with particle method, while using grid method, speeds are nearly equal. The differences are frequency of math calculation. As is known, the double type is usually faster running than the float type especially facing complex math calculation. In the particle method, the simulator needs to get the square root of the distance for every particle pair. While, using the grid method, the frequency of this calculation is largely reduced. As a result, the computation time costs similar using grid method on two data types. In order to prove this, the square root function was removed from the particle method and the computation time for two data types are almost the same. Table 4.1 shows more detailed data.

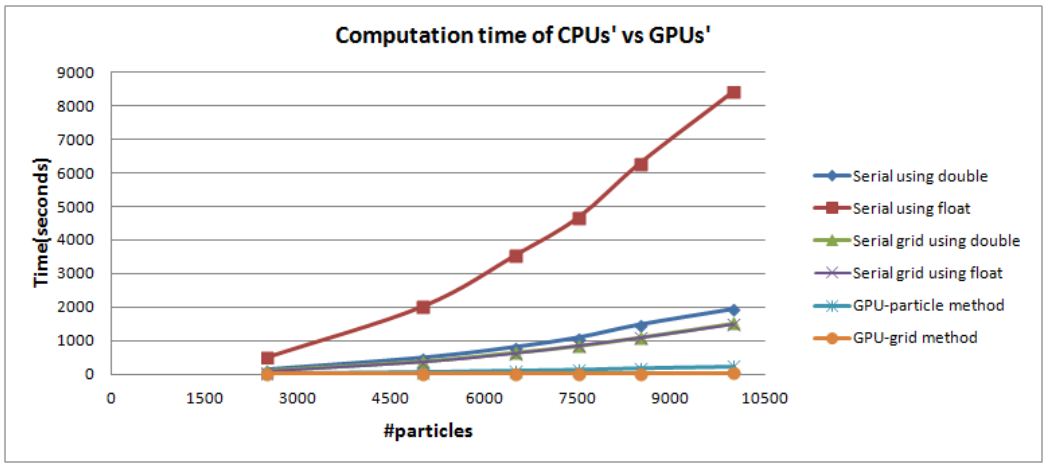


Figure 4.1: Computation time for CPU and GPU

Table 4.1: Computation time for CPU and GPU (seconds)

Number of particles	Serial particle using double	Serial particle using float	Serial grid using double	Serial grid using float	GPU particle method	GPU grid method
2500	126.50	523.51	97.02	95.04	21.19	13.56
5000	483.33	2035.43	380.39	375.95	66.87	23.39
6500	808.29	3574.80	644.09	636.99	110.51	30.98
7500	1088.20	4684.29	845.00	861.15	140.40	33.17
8500	1477.33	6310.81	1100.61	1102.89	187.58	36.55
10000	1939.58	8436.60	1520.05	1505.29	239.06	41.42

4.1.1 Speedups for CPU and GPU

Figure 4.2 shows speedups of two GPU methods compared to the serial particle version(float type) and the serial grid version(float type). The particle method used on GPU shows speedups from 24.6 to 35.2 on six cases, while the grid method's speedups range from 7.0 to 36.3. Speedups between the serial particle method and the grid method ranges from 5.5 to 5.6. It is easy to see that facing a large amount of data, the grid-based method will achieve better performance than the particle method. This is due to the number of processor cores (64) on the GPU card that has been

used in my work. Each particle is handled by one core, but in cases where there are more than 64 particles, one core must handle more than one particle. Therefore, the performance is limited by the number of processor cores and the grid method does not have this limit.

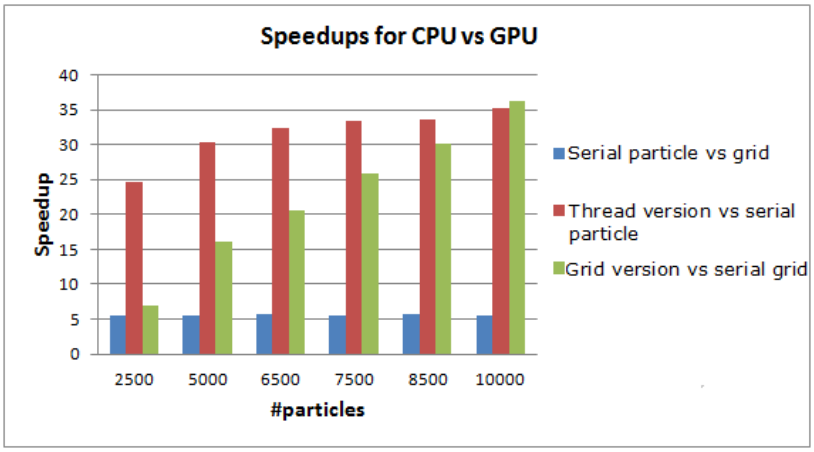


Figure 4.2: Speedups compared with GPU version and serial version

Table 4.2: Speedups compared with GPU version and serial version

Number of particles	Serial particle vs grid	Thread version vs serial particle	Grid version vs serial grid
2500	5.5	24.6	7.0
5000	5.4	30.4	16.0
6500	5.6	32.3	20.5
7500	5.4	33.3	25.9
8500	5.7	33.6	30.1
10000	5.6	35.2	36.3

4.2 GPU Specific

One problem is that the performance of the grid-based method is often, but not always, better than that of the particle method. So, as opposed to the original grid-

based method, the cell size is changed from particle size to half of the space size. With the change of the cell size, the number of grids change too. As is shown in figure 4.3, dash lines indicate dividing the grids.

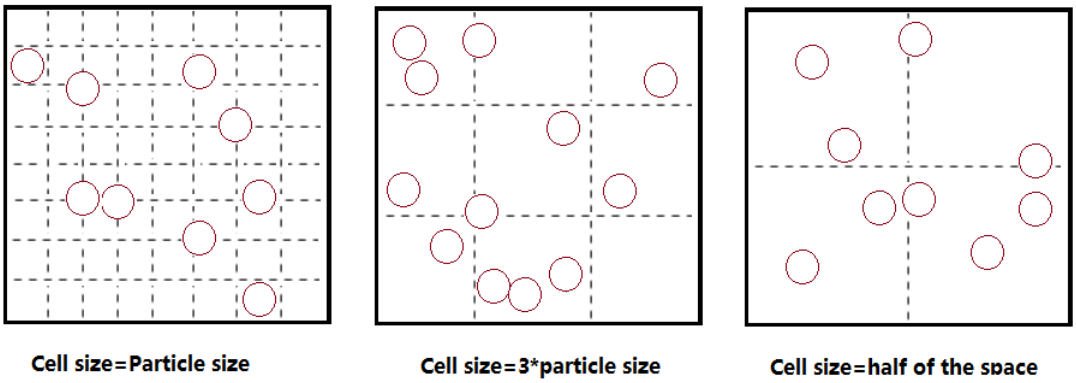


Figure 4.3: Cell size changes from small to large

Figure 4.4 shows the computation time of the grid method with different numbers of grids.

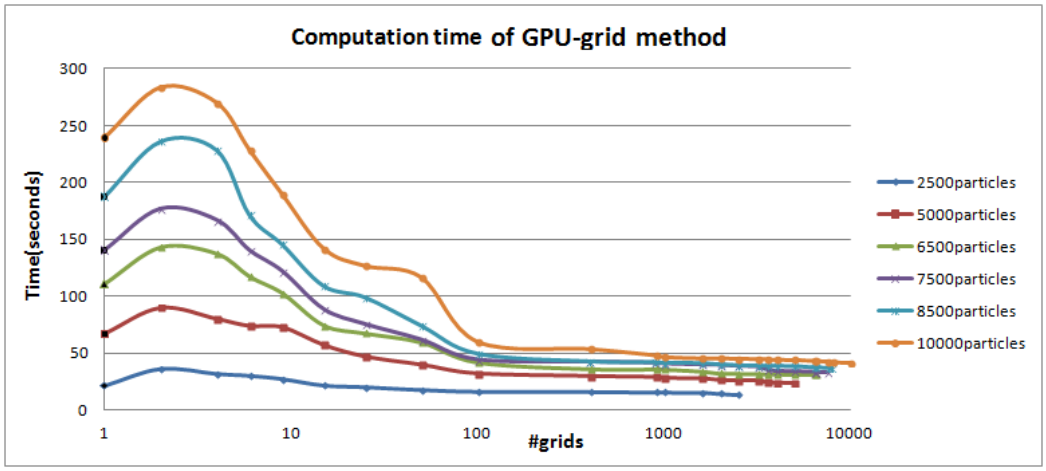


Figure 4.4: Computation time using particle method and grid-based method

Six lines indicate six cases and numbers. The starting black dot at one grid

is the computation time of the particle method and the end dot of each line is the computation time of the grid-based method, which is the cell size equal to the particle size. Middle dots represent the trend of increasing grid cells. When the number of cells are smaller than 6 to 9, the computation time is longer than for the particle method since, with a small number of cells, a grid cell and its neighbors can cover all the space, then a particle still needs to check the collision with all the other particles. When a grid cell and its neighbors cannot cover all the space, the performance is increasing obviously, because of the rapid reduction of computation. However, the performance began to flatten when the number of cells coming near maximum cells.

Table 4.3: Computation time using particle method and grid-based method (seconds)

Number of grids	2500 particles	5000 particles	6500 particles	7500 particles	8500 particles	10000 particles
2	35.53	89.88	142.68	176.92	236.41	283.17
4	31.45	80.18	137.20	167.27	228.86	269.89
6	29.79	73.96	117.16	140.67	171.72	228.37
9	27.07	72.84	102.58	122.34	145.64	189.27
15	21.61	57.22	74.33	88.98	109.18	141.75
25	20.19	46.94	67.28	75.74	99.11	126.63
50	17.85	39.82	59.39	62.20	74.22	116.54
100	16.38	32.04	41.86	44.65	49.59	60.04
400	16.13	29.75	35.95	42.79	42.70	53.58
900	15.65	28.87	35.74	41.84	41.92	48.05
1000	15.55	28.01	35.61	41.04	41.55	46.55
1600	15.43	27.60	33.81	40.24	41.17	45.42
2000	14.49	26.27	32.05	39.91	40.09	45.38
2500	13.56	25.91	31.88	39.02	39.42	45.13
3200		25.76	31.57	38.41	38.94	44.83
3600		24.00	31.52	35.83	38.72	44.62
4000		23.89	31.43	34.52	38.64	44.14
5000		23.39	31.02	34.31	38.51	44.01
6400			31.00	33.79	37.46	43.22
6500			30.98	33.68	37.40	43.02
7500				33.17	37.06	42.78
8500					36.55	42.11
10000						41.42

4.3 Cluster General

For the parallel design on the cluster, the master-slave method was used at first. The master node will send all the data to all the slave nodes. For each slave node will divide the data evenly and take over its particular segment of the data, but each node still need to compare its segment to all the other particles. As a result, each node should hold the information of all the particles. Then all nodes will do the computation to update their own segment and return the new information of their segment to the master node. For the next iteration, the master node will send the new data to all nodes again and every node will hold the latest information of all the data. The test is done in two cases, 5000 and 10000 particles. The number of computing nodes changes from four to fifty-six to provide comprehensive results.

The CPU cluster used is the STePS^2 HPC cluster [28], which consists of a head node and sixteen compute nodes. The processor used on each compute node is Intel(R) Xeon(R) E5520, and all nodes operate using Intel_ x86_ 64 Linux with toolkit provided by ROCKS 5.4. This cluster uses a job scheduler (Torque) so that each job has dedicated resources and load on the cluster head is irrelevant. There are two forms of nodal interconnects: Ethernet(1 GBit/s) and Fabric(40 GBits/s Infiniband). The former is used for all non-MPI internodal communication, like transfer, ssh and so on. The latter is used only for message passing via the MPI software. Infiniband switch served all cluster nodes.

From figure 4.5, the algorithm gets the best performance at the lowest point of each line. After that point, the performance drops gradually because of the increase cost of communication time between computing nodes.

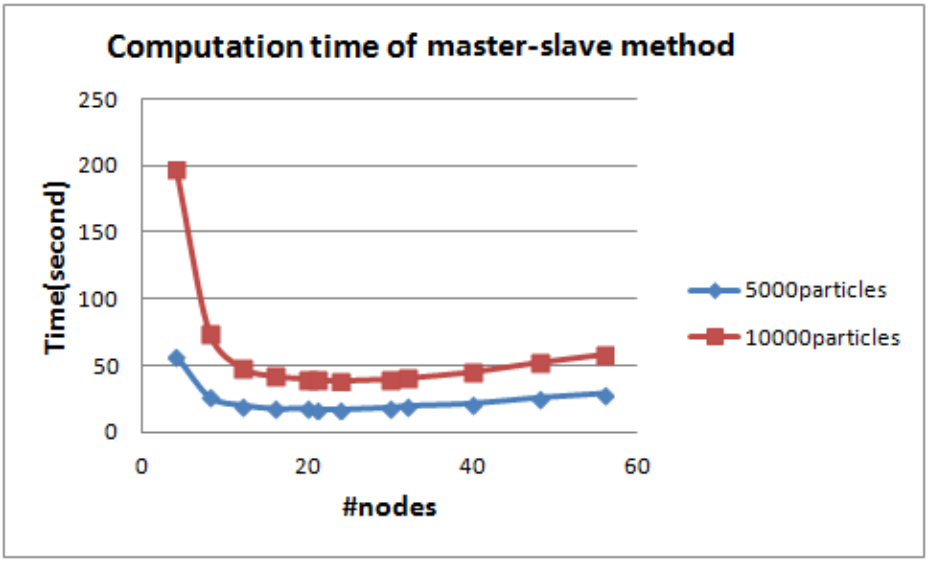


Figure 4.5: Computation time for master-slave method

Table 4.4: Computation time for master-slave method (seconds)

Number of nodes	5000 particles	10000 particles
4	56.98	197.21
8	26.86	74.50
12	20.70	48.08
16	18.16	41.80
20	18.42	39.79
21	17.73	39.16
24	17.56	38.44
30	19.09	39.70
32	20.23	40.50
40	22.07	45.09
48	26.34	52.21
56	29.46	58.22

4.4 Cluster Specific A

We set the numbers of particles as 2500, 5000, 6500, 7500, 8500, and 10000, which are the same as the number used in the former experiments. The whole simulation

is also running for 1000 iterations. Figure 4.6 represents the computation time using the heartbeat algorithm.

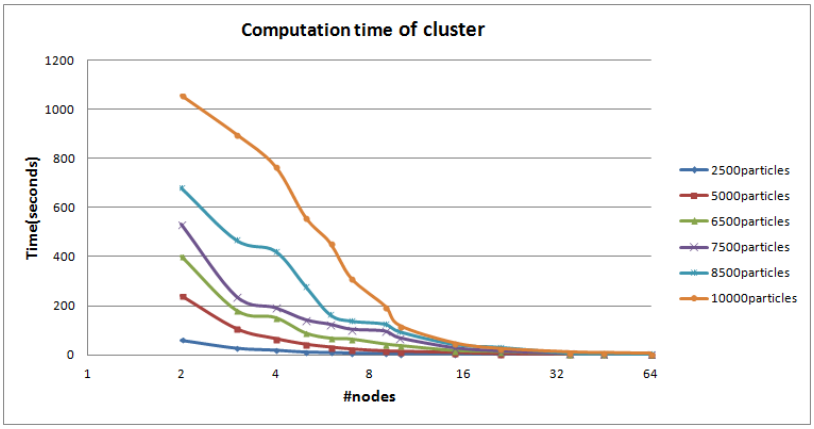


Figure 4.6: Computation time for heartbeat algorithm

The computing nodes vary from 2 to 64. It is obvious that the more computing nodes, the better the performance. Here a log scale made the figure easier to understand than a linear scale. There is no log scale on time because it would emphasize differences in time that are insignificant. When the number of nodes is bigger than 2, the computation time decreases rapidly. When the number of nodes is bigger than 20, the computation time decreases slowly.

4.4.1 Speedups for CPU and CPU cluster

Figure 4.7 shows speedups comparing the distributed version and the serial particle version. With the growing number of particles, the increase of the slope begins to slow down. Note that in Figure 4.7 the trend of the blue line doesn't follows similar directions as the others. From the experiment data, it is found that when using the heartbeat algorithm, the computation time changes a lot under the same condition. Because the number of particles controlled by each node changes a lot using the heartbeat algorithm, and a new iteration will not start until all the nodes finish their

Table 4.5: Computation time for heartbeat algorithm (seconds)

Number of nodes	2500 particles	5000 particles	6500 particles	7500 particles	8500 particles	10000 particles
2	60.38	240.57	400.61	530.84	681.63	1057.68
3	26.85	106.32	180.92	237.99	469.00	898.87
4	18.24	65.76	151.34	191.73	423.84	767.99
5	10.36	42.71	88.52	142.78	280.31	559.90
6	8.25	31.08	67.58	124.15	164.14	454.67
7	5.70	23.06	63.68	104.33	138.14	310.98
9	4.20	14.90	43.86	96.65	125.77	196.68
10	3.44	13.33	37.59	69.48	95.03	118.60
15	3.06	8.99	17.60	28.83	43.20	47.20
21	2.35	4.17	13.17	16.03	31.05	24.59
35	1.12	2.52	4.66	5.91	7.60	10.42
45	0.84	2.10	3.20	4.54	5.29	7.62
64	0.73	1.23	1.94	2.22	3.38	4.76

work. Then, the faster nodes will wait for the slower ones. The performance is better when the number of particles controlled by each node are almost the same. This conclusion is also proved more detailed in the next experiment, cluster specific B. The computation times showed in the figure 4.6 are average times after calculating. For the blue line, long computation times occupy more than short ones and the increase of performance slows down.

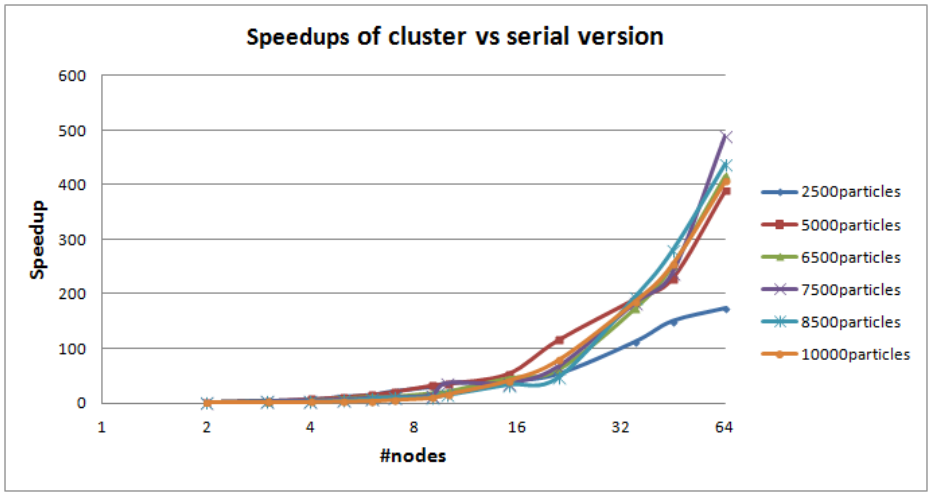


Figure 4.7: Speedups compared distributed version and serial particle version

Table 4.6: Speedups compared distributed version and serial particle version

Number of nodes	2500 particles	5000 particles	6500 particles	7500 particles	8500 particles	10000 particles
2	2.0	2.0	2.0	2.0	2.1	1.8
3	4.7	4.5	4.4	4.5	3.1	2.1
4	6.9	7.4	5.3	5.6	3.4	2.5
5	12.2	11.5	9.1	7.6	5.2	3.4
6	15.2	15.5	11.9	10.4	8.9	4.2
7	22.1	21.0	12.6	11.2	10.6	6.2
9	30.0	32.3	18.4	15.6	11.7	9.8
10	36.6	36.2	21.4	37.7	15.5	16.3
15	41.1	53.7	45.9	37.7	34.1	41.2
21	53.7	115.8	61.6	67.8	47.5	78.8
35	111.9	190.9	173.3	184.0	194.3	186.0
45	149.8	228.9	252.5	239.6	279.2	254.4
64	173.1	389.5	416.4	490.0	436.9	406.5

4.5 Cluster Specific B

Experiment nine is done with a fixed number of particles in a fixed space. It is found that with a fixed number of particles and a fixed number of nodes, the computation

time changes significantly as the number of nodes does not change. The reason is that the number of particles existing in one node changes a lot due to the random distribution of particles and the computation time for each node also changes, nodes that finishing earlier will wait for the slower ones. Load balancing isn't appropriate here because "moving" particles between nodes will introduce more overhead. In order to observe the influence of communication time between computing nodes, I tried a fixed number of particles per node. I make every computing node control 1000 particles and change the number of computing nodes, so the whole number of particles is equal to the number of nodes times 1000. You need to address why load balancing isn't appropriate here (because "moving" particles between nodes will introduce more overhead).

Also considering the number of neighbors, we tried two kinds of topology: one-dimensional and two-dimensional. Experiment ten was carried out by varying the number of nodes to compare the performance. Depending on the number of nodes, we make a one-dimensional (linear) or two-dimensional (grid) topology.

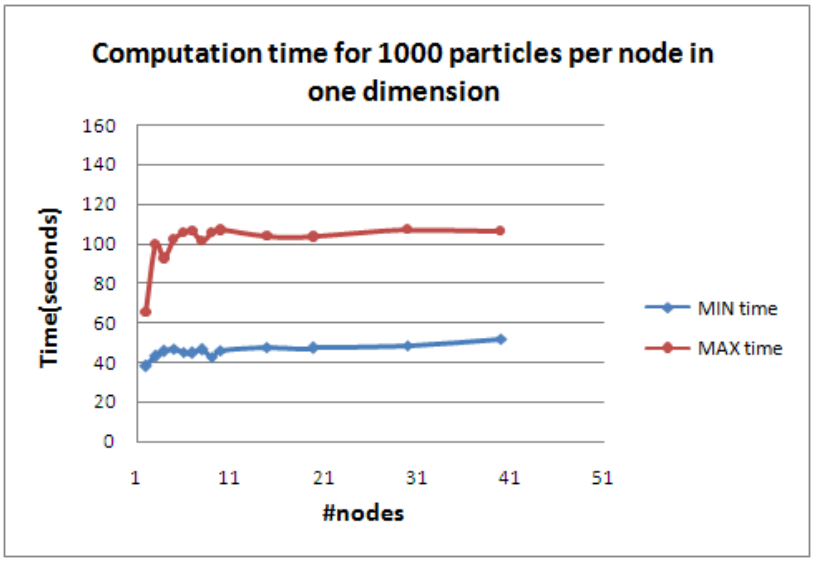


Figure 4.8: Computation time in one dimensional topology

Figure 4.8 shows the range of computation time for 1000 particles per computing node in a one dimension topology. The blue line indicates the minimum computation time for the whole experiment of 1000 iterations and the the red line indicates the maximum computation time for the whole experiment of same iterations. From the picture, we can see that the computation time ranges from 40 to 100 seconds. Each result is produced by more than 10 experiment runs. The reason is that after the heartbeat algorithm starts, the number of particles that exist in one node changes, maybe more or less than 1000. If one node finishes computing earlier than the others, it needs to wait until the last node finishes. So that the time is changeable. Given 1000 particles, the computing time is 38 seconds using only one node with no communication and the computing time for 2000 particles in the same condition, 152 seconds.

Table 4.7: Computation time in one dimensional topology (seconds)

Number of nodes	MIN time	MAX time
1	38	
2	38.2911	65.2715
3	43.2357	99.7295
4	45.7749	92.6292
5	46.9007	102.4702
6	45.4013	105.6367
7	44.7699	106.5086
8	46.6809	101.2028
9	42.7735	105.531
10	46.2103	106.8025
15	47.7044	103.6966
20	47.2502	103.5919
30	48.5195	107.0664
40	52.0298	106.3271

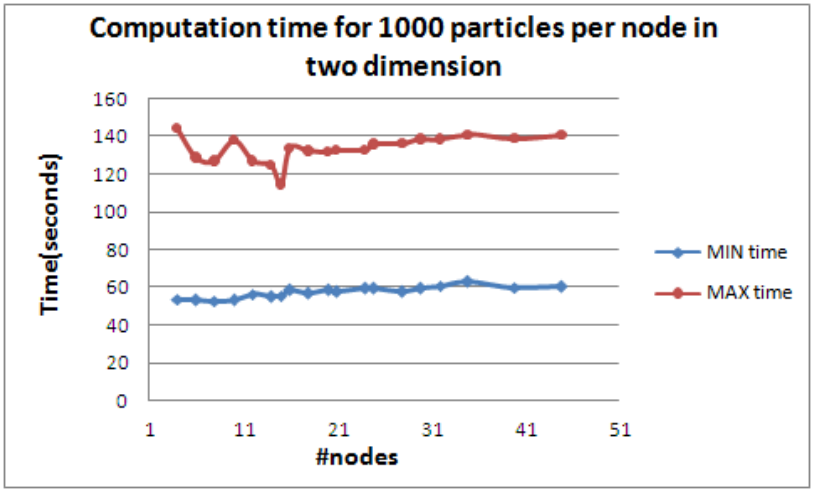


Figure 4.9: Computation time in two dimensional topology

Figure 4.9 shows the range of computation time for 1000 particles per computing node in a two dimension topology. It is shown that the lowest point and the highest point of each line are both higher than the last picture in the one dimension topology. For a two dimension topology, each node may have three to eight neighbors rather than only left and right neighbors in the one dimension topology, the communication time is increasing.

From the above two pictures, the computation time changes a lot for the number of particle changes in a specific space. Since the number of particles per node is fixed, adding nodes increases the total number of particles. The fact that the computation time does not increase significantly suggest that a distributed heartbeat approach will scale very well except for the problem. If we reduce 1000 iteration times to 100 iteration times, the particles per node will not change significantly.

Table 4.8: Computation time in two dimensional topology (seconds)

Number of nodes	MIN time	MAX time
1	38	
4	53.7612	144.1483
6	53.1705	128.8348
8	52.6811	126.9911
10	53.2773	138.1135
12	56.3989	126.8814
14	55.0163	124.7148
15	55.4917	114.4466
16	58.5433	133.7274
18	56.6631	132.3635
20	58.6632	131.9641
21	57.3664	132.7649
24	59.6642	133.043
25	59.2021	135.8864
28	57.5963	136.5483
30	59.6601	138.5521
32	60.8828	138.5523
35	63.1712	141.0312
40	59.7905	139.0524
45	60.3779	140.9597

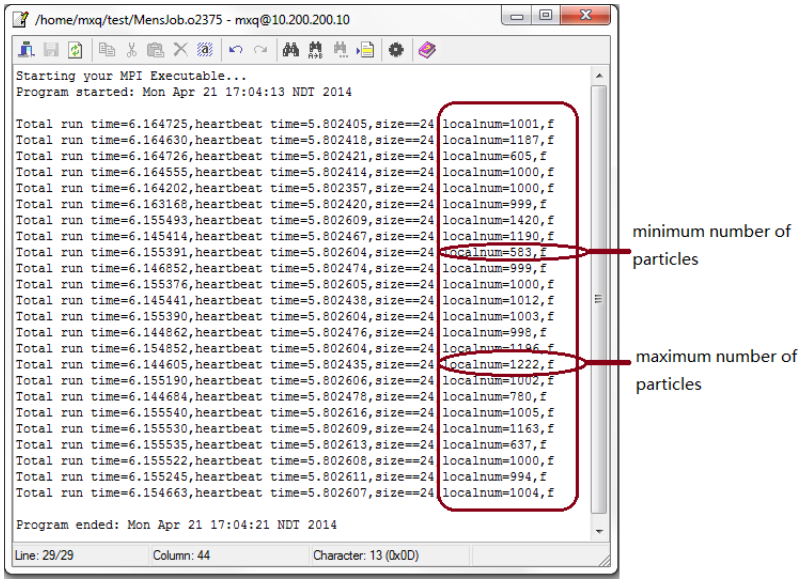


Figure 4.10: Example results running with 24 nodes

Figure 4.10 shows an example of results running with 24 nodes. It is shown that when the simulation finished, the local number of particles ranged from 583 to 1222.

Figure 4.11 represents the computation time for 100 iterations in two different topologies. The blue line shows the heartbeat running time for 1000 particles each node in one dimension and the red line shows the results in two dimension.

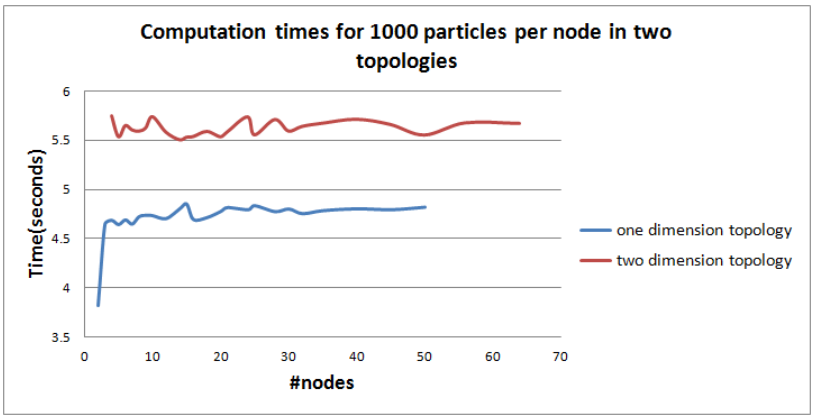


Figure 4.11: Computation time in two dimensional topologies

Two flat lines in figure 4.11 shows that the communication time between nodes is almost the same though the number of nodes vary. Also, if a node has more neighbors, the communication time is longer. In the same topology, no matter how the number of nodes change, the communication time will be almost the same. This is because the network topology is such that the number of neighbors for each node does not increase as the number of nodes increases. But the communication time also depends on how many particles move between a node and its neighbor. For our particle simulation, the data transmitted between nodes is relatively small. There are some factors that have influence on the communication time. First, particle's properties such as small radius and velocities. Second, simulation settings such as short time step. Short time step make particles move short distance slowly every time step. After tracking the communication between nodes, it is found that there was at most a dozen of particles

moving across the boundary every time step and most time, there was only several particles or even zero particle across the boundary. As a result, the nodes will not keep communicating with all the neighbors all the time. If a lot of particles keep across the boundary all the time, the communication time may be different.

As for applying the ice simulation, ice floes almost cover all the simulation space and an ice floe is big enough to cover more than one node. Every time step ice floes move a lot and the simulator should make all nodes communicate to keep update. The amount of communication will be very large, such as updating all ice floes, and transmitting information of ice floe that crosses borders to several neighbors. This is another problem of workload balancing, which will be studied further in the future work.

Chapter 5

Conclusion

Powerful programming techniques are playing an important role in solving scientific problems, especially scientific simulations. When programmers choose different platforms, like CPU, GPU, cluster and so on, different algorithms and programming languages are chosen to fit the hardware. In order to consider appropriate architectures and algorithms for the GEM Ice Simulation, which is supported by the Sustainable Technology for Polar Ships and Structures project(referred to as STePS^2), performance of a similar two dimensional simulation of simple particles was evaluated using a variety of hardware architectures and algorithms

This thesis deals mainly with a two dimensional particle simulation to understand a planar motion system under the influence of given velocities in a constrained area. This particle simulation was executed using three different approaches: CPU, GPU, and CPU cluster. Programs were written in C with CUDA and MPI. Some programming paradigms can be applied on both ice-flow simulation and similar planar motion systems, some algorithms can be applied on almost all the hardware. The first goal in the thesis is studying the performances of the different architectures and algorithms for a group of two dimensional moving objects across a wide area. The second goal is

to study the advantages and disadvantages of the techniques developed to solve this kind of problems along with how different approaches affect the results.

5.1 Contributions

The two dimensional particle simulation was executed using three different approaches: CPU, GPU, and CPU cluster. The scale of the simulation was relatively small and the iteration time was up to 1000 times. For parts of the experiments, there were up to 10000 particles and the density of particles was set as 78.5% of the area. For the other experiments, the number of particles were decided by the computing nodes used on the CPU cluster.

On the CPU, the particle simulation used the serial program with two methods: the particle method and the grid method. To provide a relevant comparison, the code also ran with two data types: the float data type and the double data type. From the results, it is shown that the time required for the particle method using two data types changed a lot. But the time required for the grid methods were almost the same. In the particle method, the main math calculation is to get the square root as the distance for every particle pair. While, using the grid method, the frequency of this calculation is largely reduced. So the proportion of math calculation in a simulation is proved as one important factor to affect the simulation time.

On the GPU, two different parallel designs were built to fit two methods. The computation time of the particle method changed a lot with the change of the number of particles. While for the grid method, the computation time changed little. Because the former one used one thread control one particle to find the collision. The latter one costed time most in building grids and finding collision in a grid and its neighbour grids. So the number of particles do not have much influence on the simulation time

using the grid method. Due to the different programming architectures, the grid method is shown to fit better for this simulation, especially with a large number of objects in a constrained area. On the other hand, the particle method is fit for simulating small number of objects. Based on the grid method, the grid size also needed to be set appropriately. If one grid and its neighbors can cover most of the simulation area, the computation will have overlap and results were worse than the particle method.

Using the CPU cluster, a master-slave design was built first. This design is shown to have a bottleneck on the performance of the CPU clusters with large number of computing nodes, which meant for this master-slave paradigm, there was a best efficiency due to the communication time and computation time on each computing node. Given different amounts of the data, the bottleneck was different. This paradigm is fit for the simulation with a high proportion of the computation and the bottleneck is far away to reach. The distributed algorithm showed much better performance than the master-slave design for it reduced the memory held by each processor directly. Using the distributed design, the workloads on each computing node are different and change all the time during the simulation, so the simulation time is not stable. To prove different workloads affect the distributed design, a fixed number of particles per computing node and smaller iteration times was applied after rather than a fixed number of particles in a fixed area. As a result, this paradigm is thought to be more suitable for the simulation with same workload on each computing node. The results were produced by this small scale of particle simulation, if the simulation are large enough, the results are expected to be almost the same for the basic simulation framework stays the same. For this distributed design, the main idea is distributing the memory. The performance will not change much when the simulator increase the number of computing nodes and the number of particles together as long as keeping

the same workload on each computing node. It is also shown that the communication time between nodes is almost the same though the number of nodes vary.

5.2 Discussion and Limitations

Using different methods on the GPU, the computing performance was different. The grid-method was always better than the particle method when the number of particles and grids were large. For our experiments, when the number of grids built was so small that the performance of the grid method was worse than the particle method. The GPU card used did not support the double data type, so all the experiments used the float data type.

To compare the performance with the CPU cluster and the GPU, their results are both compared to the results of CPU. The particle method used on the GPU shows the speedups of 24.6, 30.4, 32.3, 33.4, 33.6, and 35.2 compared to particle method used on the CPU. The grid method shows the speedups of 7.0, 16.0, 20.5, 25.9, 30.1, and 36.3. The speedups of the distributed design used on the CPU cluster changes with the number of computing nodes. Start with two computing nodes, the speedups are 2.0, 2.0, 2.0, 2.0, 2.1, and 1.8. When the number of computing nodes is more than fifteen, the average speedup is more than 40. The distributed design shows the best performance in a way for this particle simulation and the main factor is the number of computing nodes used.

This two dimensional particle simulation model is suitable for other two dimensional planar motion systems' simulations. In a way, the figure of moving objects can be different, the framework can be reused. While there is no mutual attraction or repulsion force considered in our simulation, like gravity or electromagnetic. If there are more driving forces, the parallel part should be redesigned due to different

computations. As in the ice-flow simulation, all ice-flows are resting and will gain velocities after the ship goes through. Particles are given fixed velocities at the start and move with elastic collisions. No gravity, acceleration are taken into consideration, so the computation of dealing with collisions is not complex. Since the moving particles are spheres, the method for detecting collisions is simple. If the moving object is polyhedral, the function will be more complex and can also be added to the parallel part.

5.3 Future Work

For this work was done in a small scale of the data, in order to solidify the conclusion, experiments with greater computation power should be taken out, such as bigger cluster or advanced GPU cards.

Compared with the event-driven simulation, the accuracy of the time-driven simulation is relatively low. In this particle simulation, the time step was set to let the particle move about one diameter. During simulation, particles may overlap each other or collide with others without calculating the collision. This problem can be solved using a event-driven simulation or shorter time step. As mentioned in chapter two, the event-driven simulation will store all the collision times and none will be missed. A comparision of event-driven simulation with time-driven simulation for this problem will be of interest.

To improve the interconnects and software, the application of the new technology and greater computation power should be considered, like GPU cluster. According to the conclusions, the framework and distributed algorithms used on the CPU cluster can also be applied on a GPU cluster. A GPU cluster combines both the advantages of the GPU and the cluster, which are expected to show greater efficiency.

The methods used have many ways to be implemented, like using grid method on the GPU. There are different ways to build up the grid, like constructing matrices and trees. In our work, radix sort is used to build the grid. Different ways produce different parallel designs, which also have influence on the efficiency of the whole simulation.

Rather than using a simple particle system, the ice-flow model or other irregular objects can be applied. The equations of their motion will be more complex and there are also many driving forces which have influences on the system, such as heat, gravity, and pressure.

Bibliography

[1] Nvidia home page, 2014. `http://www.nvidia.com/page/home.html`.

[2] Graphics processing unit, 2015. `http://en.wikipedia.org/wiki/Graphics_processing_unit`.

[3] Shadi Alawneh. *Hyper-Real-Time Ice Simulation and Modeling Using GPGPU*. PhD thesis, Memorial University of Newfoundland., St. John's, NL, May 2014.

[4] Shadi G. Alawneh and Dennis K. Peters. Ice simulation using GPGPU. In *14th IEEE International Conference on High Performance Computing and Communication*, pages 425–431. IEEE Computer Society, 2012.

[5] B. J. Alder and Thomas K. Wainwright. Molecular motions. *Scientific American*, 201(4):113–126, oct 1959.

[6] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):50–90, mar 1991.

[7] Gregory R. Andrews. *Foundations of Multithreaded, Parallel and Distributed Programming*. Addison Wesley, 2000.

[8] R. Bagrodia, K. M. Chandy, and Wen Toh Liao. A unifying framework for distributed simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(4):348–385, oct 1991.

[9] B. Brandfass, T. Alrutz b, and T. Gerhold. Rank reordering for mpi communication optimization. *Computers & Fluids*, 80:372–380, 2013.

[10] Matthew C. Chidester and Alan D. George. Parallel simulation of chip-multiprocessor architectures. *ACM Trans. Model. Comput. Simul*, 12(3):176–200, 2002.

[11] Yi-King Choi, Jung-Woo Chang, Wenping Wang, Myung-Soo Kim, and Gershon Elber. Continuous collision detection for ellipsoids. *IEEE Trans. Vis. Comput. Graph*, 15(2):311–325, 2009.

[12] Wenshan Fan, Bin Wang, Jean-Claude Paul, and Jia-Guang Sun. An octree-based proxy for collision detection in large-scale particle systems. *SCIENCE CHINA Information Sciences*, 56(1):1–10, 2013.

[13] Ananth Y. Grama, Vipin Kumar, and Ahmed Sameh. Scalable parallel formulations of the barnes-hut method for n-body simulations. In *Proceedings of Supercomputing'94*, pages 439–448, Washington DC, nov 1994. IEEE.

[14] Simon Green. Particle simulation using cuda. *Nvidia*, 2012. `https://developer.nvidia.com/resources`.

[15] Tomas Hansson, Chris Oostenbrink, and Wilfred F van Gunsteren. Molecular dynamics simulations. *Structural Biology*, 12:190–196, 2002.

[16] Volodymyr V. Kindratenko, Jeremy J. Enos, Guochun Shi, Michael T. Showerman, Galen W. Arnold, John E. Stone, James C. Phillips, and Wen mei Hwu. Gpu clusters for high-performance computing. In *IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, 2009.

[17] David B. Kirk and Wen mei W. Hwu. *Programming massively parallel processors hands-on with CUDA*. Morgan Kaufmann Publishers, pub-MORGAN-KAUFMANN:adr, 2010.

[18] Thuy T. Le and Jalel Rejeb. A detailed mpi communication model for distributed systems. *Future Generation Computer Systems*, 22:369–278, 2006.

[19] Jacek Leszczynski and Mariusz Ciesielski. Effective algorithm for detection of a collision between spherical particles. In *Computational Science - ICCS 2004, 4th International Conference, Kraków, Poland, June 6-9, 2004, Proceedings, Part II*, volume 3037 of *Lecture Notes in Computer Science*, pages 348–355. Springer, 2004.

[20] Ravindra M and V Chaithanya. Barnes-hut algorithm implementation in parallel programming world. *Peoples Education Society Institute of Technology*, 2009.

[21] MPI Forum. MPI2: A message passing interface standard. *Intl. J. High Performance Comput. Appl.*, 12(1/2), Spring/Summer 1998.

[22] Gabriel Noaje. Cpu-gpu cluster design, experimentations, performances. *SUPELEC and UPB*, 2008.

[23] Nvidia. Cuda architecture overview v1.1, 2009. `https://developer.nvidia.com/resources`.

[24] Nvidia. Cuda programming guide v2.3.1, 2009. `https://developer.nvidia.com/resources`.

[25] Kalyan S. Perumalla and Vladimir A. Protopopescu. Reversible simulations of elastic collisions. *ACM Trans. Model. Comput. Simul*, 23(2):12, 2013.

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

[27] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.

[28] Bruce W.T. Quinton and Anthony Kearsey. Benchmarking of three parallelized implementations of ls-dyna on a hpc server cluster. *Memorial University of Newfoundland.*

[29] Mike Showerman, Jeremy Enos, Craig Steffen, Sean Treichler, William Gropp, and Wen mei W. Hwu. EcoG: A power-efficient GPU cluster architecture for scientific computing. *Computing in Science and Engineering*, 13(2):83–87, March/ April 2011.

[30] Hersir Sigurgeirsson, Andrew Stuart, and Wing-Lok Wan. Algorithms for particle-field simulations with collisions. *Journal of Computational Physics*, 172(2):766–807, September 2001.

[31] Rainer Spurzem. Direct n-body simulations. *Computational and Applied Mathematics*, 109(1/2):407–432, September 1999.

[32] In Hwan Sul. Fast cloth collision detection using collision matrix. *International Journal of Clothing Science and Technology*, 22(2/3):145–160, 2010.

[33] E. Tijskens, H. Ramon, and J.De Baerdemaeker. Discrete element modelling for process simulation in agriculture. *Journal of Sound and Vibration*, 266(3):493–514, September 2003.

[34] Barry Wilkinson and Michael Allen. *Parallel programming - techniques and applications using networked workstations and parallel computers (2. ed.)*. Pearson Education, 2005.

[35] Mathias Winkel, Robert Speck, Helge Hübner, Lukas Arnold, Rolf Krause, and Paul Gibbon. A massively parallel, multi-disciplinary barnes-hut tree code for extreme-scale N-body simulations. *Computer Physics Communications*, 183(4):880–889, 2012.

[36] LC Workshops. Mpi performance topics, 2014. `https://computing.llnl.gov/tutorials/mpi_performance/`.

[37] Chao-Tung Yang, Chih-Lin Huang, and Cheng-Fang Lin. Hybrid CUDA, openMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications*, 182(1):266–269, 2011.

[38] Yongpeng Zhang and Frank Mueller. GStream: A general-purpose data streaming framework on GPU clusters. In *Proc. International Conference on Parallel Processing (40th ICPP'11) USB*, pages 245–254, Taipei, Taiwan, sep 2011. IEEE Computer Society.