Efficient Random Number Generator for Mesoscopic Simulations of Brownian Motion

Qiaowei Wang

August, 27 2010

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2010

Abstract

The pseudo-random number generation is a widely used technique in computational physics fields. It is achieved by generating a sequence of numbers with no obvious pattern according to some predefined algorithms. The sequence of numbers generated usually works for a specific problem with the desired statistical properties.

The “Fluctuating Hydrodynamics” [1] model simulates the movement of particles in fluid with the help of pseudo random number generators. The particles in the model will get a random velocity and move around in a fixed environment to the simulated effect of thermal noise. The pseudo random number generator is responsible for the random walk of all the particles. For the computational method Lattice Boltzmann [2] used in the simulation, 11 or 15 random numbers per lattice per timestep will be required. Therefore, the generation of the random numbers will affect the efficiency of the whole simulation to a large extent.

The implementation of the project is an extension of an existing C code of “Fluctuating Hydrodynamics” simulation developed by Kevin Stratford, Edinburgh Soft Matter and Statistical Physics Group and Edinburgh Parallel Computing Centre [3]. The random number generation section of the simulation will be mainly concentrated on.

In the project, a number of alternative methods for uniform random number generation are implemented in terms of both quality and efficiency. And then, a technique named discrete random number generator published by Anthony J.C Ladd [4] is implemented, which get a nearly 10 times speedup than the current used normal random number generator in the simulation. In addition, a parallel version of the discrete random number generator is implemented and analysed.

Contents

List of Tables

List of Figures

Acknowledgements

I’d like to thank my supervisor Kevin Stratford for his patient guidance during the project time.

I’d also like to thank my friends Haojun Yang, Qiaotian Xu, and other classmates for their help over the year.

Qiaowei Wang

Chapter 1: Introduction

This chapter will briefly introduce the history of liquid simulation, the mathematical methods used for the “Fluctuating Hydrodynamics” simulation [1] and the relevant random number generation problems.

1.1 History of Liquid Simulation

The liquid simulation has a long history, from the earlier theories and experiments simulation to the modern computer models. The purpose of the simulation is to model and analyse the behaviours of fluid molecules.

The earlier experiments simulation use real objects like metal balls to represent molecules. This old approach is hard to simulate the interactions between molecules, and the gravity effect will eventually affect the simulation. Thus, modern computer simulation uses a mathematical way instead to do the liquid simulation. There exist several computational algorithms for the computer liquid simulation.

An earliest computer liquid simulation method is the ‘Monte Carlo’ simulation [Metropolis et al. 1953] [5], which works fine to model the molecules. The MC simulation was extended by adding the function of simulating thermodynamic data when implemented on Lennard-Jones interaction potential [Wood and Parker 1957] [5].

Another method is called “Molecular Dynamics” [5], which models a lot of molecules with its dynamic properties. Molecular Dynamics is used to solve a classical equation of motion, such as the Newton’s equations, for a large number of molecules.

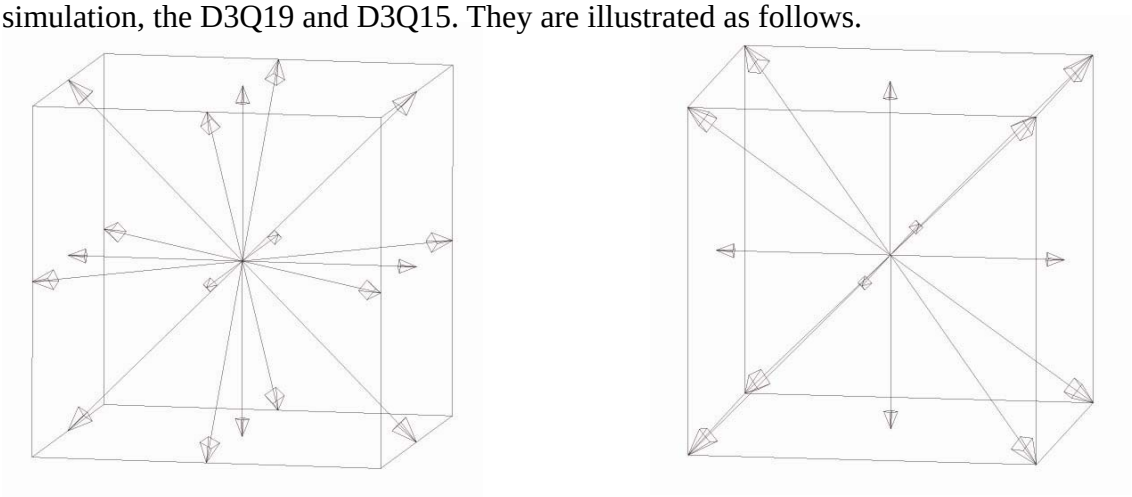
A comparatively new simulation method for computational fluid dynamics systems is the “Lattice Boltzmann” method [2] (knows as LB). LB method is more easily to be used for complex fluid system than the two conventional methods described above. It simulates the propagation and collision of fictive particles in a fixed environment by using discrete fluid lattice. It is good at dealing with complex scenarios, such as the boundaries, interactions between particles, and parallelization of the algorithm.

1.2 Lattice Boltzmann and Brownian Motion

The particles will move according to the principle named “Brownian Motion” in “Fluctuating Hydrodynamics” simulation [1]. The Brownian Motion theory can be known as random walk to some extent, which describes the random movement of fluid molecules, such as water and gas.

The fluid simulation method used in the existing simulation is the Lattice Boltzmann method [2]. In LB method, fluid is viewed as a large number of fluid points in a three dimension environment. Each fluid point is modelled in a velocity lattice (cube), which can travels to a finite number of predetermined directions according to the lattice type.

There are basically two types of LB lattice used in the “Fluctuating Hydrodynamics” simulation, the D3Q19 and D3Q15. They are illustrated as follows.



D3Q19 D3Q15

Figure-1 Two types of lattice used in LB method, D3Q19 and D3Q15

D3Q19 and D3Q15 are named by the structure of fluid point lattice. The “D3” refers to the number of dimensions, and the “Q19” and “Q15” refer to the number of velocity vectors. **Figure-1** shows, the D3Q19 have 18 velocity vectors and a 0 vector, and D3Q15 have 14 velocity vectors and a 0 vector. The velocity vectors are the possible move directions of each fluid point.

The particles' Brownian Motion and the Lattice Boltzmann method can be simply illustrated by the graph below.

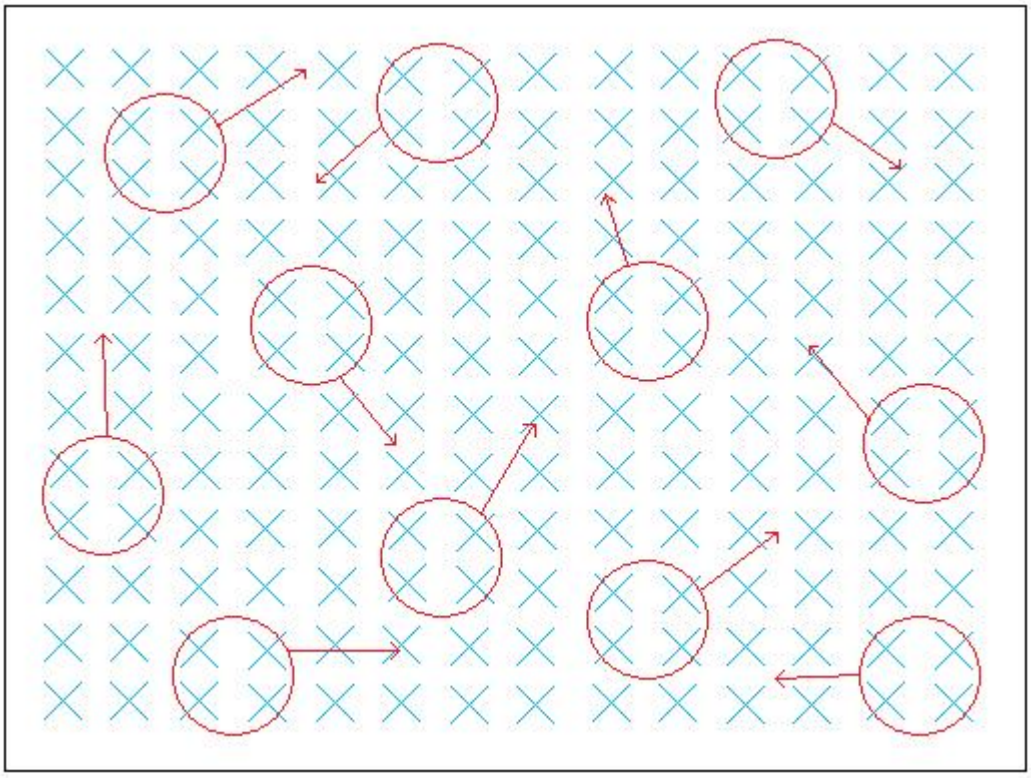


Figure-2 LB method in modelling the particle's Brownian Motion

Figure-2 shows the particles' random walks in the fluid. The particles modelled are usually the smoke particles, which are represented by the red circles. The fluid point lattices are represented by the blue folks. To simulate the real particles' behaviours, the fluid points should move randomly with the effect of thermal energy. All the random movement of fluid points surround the red particle will work together to finally determine the move direction (red arrow) of the particle every timestep. Thus, at last all of the particles move randomly everywhere similar to what they actual do in natural.

1.3 Random Number Problems

In Lattice Boltzmann method, a function f (x, v, t) is used to represent the probability of moving to a specific direction for a fluid point. The parameters x, v and t are the attributes of each fluid point, which means the point in position x will hold the velocity v at t time. For each fluid point lattice in the simulation, a series of random numbers are required to calculate the attributes for the f (x, v, t) functions every timestep. Therefore, the quality of the random number generator in the "Fluctuating Hydrodynamics" model should be considered carefully.

Further more, the total number of fluid point lattice in the simulation is huge. And then the time cost on the progress of random number generation will have a large impact on the overall performance of the simulation. Therefore, not only the quality, but also the efficiency of the random number generator used in the model should be considered. .

In the following chapters, we will introduce more concepts of random number generator and implement some techniques to solve the current random number generation problems of the “fluctuating hydrodynamics” simulation.

Chapter 2 gives some literatures on both principles and techniques of random number generators. Chapter 3 explains the current efficiency problems of random number generation, gives a possible technique solution (discrete random number), and provides some basic requirements for the project. Chapter 4 illustrates the specific execution of the current “Fluctuating Hydrodynamics” simulation program with its outputs. Chapter 5 checks the availability of possible uniform random number generators with some standard tests. Chapter 6 implements the different technique to replace the current normal random number generator. Chapter 7 analyses the performance of the discrete random number technique in a variety of statistical methods. Chapter 8 explains the implementation of parallelisation with the corresponding performance. Chapter 9 summarises the works.

Chapter 2: Literature Review

This chapter illustrates the basic concepts of pseudo random number generation and describes the technique of discrete random number generator [4]. Further more, some important rules that should be followed when implementing random number generator are advised.

2.1 Basic Concepts of Pseudo-Random Number Generator.

All the random numbers used in modern machines are generated by random number generators (known as RNGs), which are usually implemented by some artificial methods. However, the random numbers produced by computer are predictable and repeatable, which are not true “random number” by mathematical definition. In other words, the random numbers commonly used in computational physics fields are so called “pseudo random number”.

The “pseudo random number” means that the numbers generated by the system supplied RNGs always follows a deterministic algorithm. Among these algorithms, the most commonly used one is the linear congruential equation. The corresponding mathematical equation can be defined as follows.

$$I_{j+1} = a \times I_j + c(\text{mod})m \quad [6]$$

From the equation above, one random number could be generated each iteration. Given I_j as an existing random number, the next random number I_{j+1} can be given by the right hand equation. Other variables in the equation are positive integers (known as seeds) which should be predefined by users. The (mod) signal is the modulo operation which finds the remainder of division of c by m. In fact, the function will eventually repeat itself within a fixed period, which is usually less than m. The equation can generate random values between 0 and RMAX (m-1). Thus the seeds m, a, and c should be carefully chosen to meet the particular requirements of random numbers in a specific circumstance.

2.2 Advised Rules in use Random Number Generators

To correctly use RNGs to meet a specific scenario, a number of simple and sensible rules are provided as follows.

2.2.1 Do not use system generators

The system-supplied RNG is a convenient method of generating random numbers. However, it is easy to overestimate the ability of the system-supplied ones when using them for solving complex computational problems.

The fact is, almost all of the standard generators have some serious defects. From the test results of standard random number test library, TestU01 [7], we could find how many tests they have failed. These standard system-provided generators include Java.util.Random, C-library rand, and so forth. Therefore, never use the easy achieved one, if your program does need to generate good quality random numbers without confusion.

2.2.2 Use a good generator

It is a fact that the machine can never generate true random numbers by the mathematical definition, unless the machine gets some errors. However, there is an available practical definition of machine random number can be used in application.

The practical definition of pseudo randomness is that the numbers should not be distinguishable from a source of true random numbers in a given application. [8]

Now that the true random numbers by definition are unlikely to exist in computer, then the standard of a good pseudo random number generator is to pass the rules of practical definition.

To determine the quality of a RNG according to the practical definition, a series of properties should be checked, such as whether they have obvious pattern. In order to verify these properties, some standard libraries are provided. The libraries usually provide some standard quality tests functions, including binary rank test, bitstream test, overlapping sum test, redundant test and so forth.

Take the overlapping sum test as an example. [8] It generates several random sequences U(1),U(2),…of uniform variable, and then calculate overlapping sums, S(1)=U(1)+…+U(100), S2=U(2)+…+U(101),… . Finally it will get a particular value from these overlapping sums. The range of this value will mostly determine the quality of the corresponding RNG according to some invisible standard mathematical rules in the test.

Definitely, the more tests the RNG can pass, the better it is.

2.2.3 Build the RNG into your code.

A sensible way of using random number generators is to combine different good RNGs' features into your own code. Some RNGs may have flaws when implemented alone. However, by doing combination of several RNGs in a suitable way, the defects in any one of the generators may possibly be covered by others. Nowadays, the combining method is being considered as a good practice in designing RNGs by many experts in this field.

2.2.4 Properly seed the generator.

Any random number generator can run into problems because of improper seeding. For the simulation of "Fluctuating Hydrodynamics", a poor choice of seeds may result in a random sequence with unwanted correlation, which is bad for the simulation. Therefore the seeding process is an important step that should not be ignored in RNG implementation.

One way to seed a RNG is to take the date by calling the system time. This way is not repeatable and easy to be achieved. However it is bad when the program need to get absolutely the same result for the performance tests under the same environment. Thus, always give the RNG an integer seed is a comparatively safe approach.

2.2.5 Warm up generator

The same seed will generate absolutely the same sequence of numbers, and a possible solution is to run the RNG several times before actually getting a random number. This may avoid the problems caused by using the same seed. In addition, it may also help to avoid the fluctuation of the generator at the first unstable situation of machine.

2.3 Techniques in RNG

This section explains some basic RNG (random number generator) techniques by codes, which include the concept of uniform distributed RNG and normal distributed RNG. In addition, it shows the efficiency problems of the current normal random number generator in "Fluctuating Hydrodynamics" simulation.

2.3.1 Uniform Distributed RNG

The most widely used RNG is the uniform distributed one. The name “uniform” means the probability of generating each number in a given range (usually 0 to 1) is equal. A simplest uniform generator “rand” is shown as follows [6].

```
int rand(void)
{
next=next*1103515245+12345;
return (unsigned int) (next/65535)%32768;
}
```

It is a C programming language code which similar to the liner congruential equation theory. Next is the random number we need. The range of random numbers be generated could be modified by mod the RMAX values. The RMAX above is 32768, thus the uniform RNG rand will generate random numbers from 0 to 32767 (RMAX-1, described in 2.1).

2.3.2 Normal (Gaussian) Distributed RNG

The uniform RNG is not able to meet all of the requirements, since the random number generation is widely used in computational fields like physics and mathematics. Therefore, other distributions are generated to solve different problems. The possible distributions are Exponential, Normal (Gaussian), Gamma, Poisson, and Binomial. These different distributed random numbers can be implemented by doing transform operations based on the basic uniform one. Take the normal distributed RNG for instance. The mathematical equation for standard normal distribution is:

$$p(x)[dx] = \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}} [dx] \qquad [6]$$

And the corresponding normal distribution graph is:

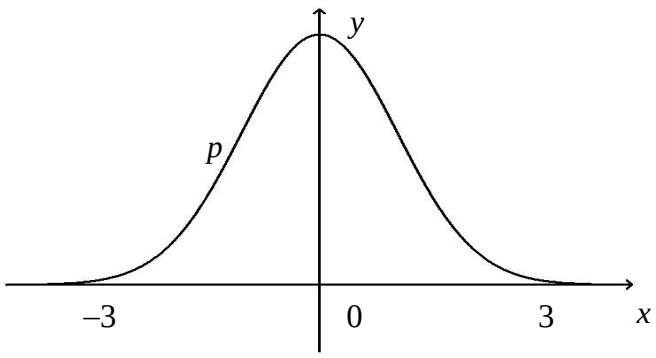


Figure-3 The normal distribution graph

X axis represents the possible random number values and Y axis represents the probability of generating a specific number. The standard normal distribution is symmetrical about y axis. The probability of generating a number that located within the x-axis range (-3, 3) is more than 99.7 percent. Other numbers located in the range $(-\infty, -3)$ and $(3, \infty)$ only hold the probability of less than 0.3 percent.

A commonly used method for normal (Gaussian) distribution is the Box-Mueller transformation [6], which returns numbers with the mean of 0.0 and the standard deviation of 1.0 (known as N(0,1)). The N(0,1) distribution is the standard normal distribution by mathematical definition. The transformation is based on the equations below.

$$y_1 = \sqrt{-2\ln x_1}\cos 2\pi x_2 \qquad [6]$$

$$y_2 = \sqrt{-2\ln x_1}\sin 2\pi x_2$$

$$x_1 = \exp[-\frac{1}{2}(y_1^2 + y_2^2)]$$

$$x_2 = \frac{1}{2\pi}\arctan\frac{y_2}{y_1}$$

For the first two equations, x1 and x2 are random numbers produced by uniform RNG. y1 and y2 are normal distributed random numbers we need. The concrete mathematical principles are explained in lecture notes [9] of computational method. A simple illustration of it is as follows.

If standard normal distribution y1 and y2 follows bivariate normal distribution, then the joint density of (y1, y2) has radial symmetry $radius^2 = y_1^2 + y_2^2$, in which the R^2 (radius) will follows the exp(1/2) distribution. Using this recipe and taking its square root R to obtain a range, we can obtain two independent samples from the standard normal distribution by selecting a random angle uniformly from (0, 2π), as $\theta = 2\pi$ x (x follows uniform distribution). Then the y1 and y2 can be calculated according to the equations y1=R$\cos\theta$ and y2=R$\sin\theta$.

In brief, one normal random number will be generated by producing two uniform random numbers.

The corresponding Gaussian rand function is as follows [6].

```
#include <math.h>
 double gaussrand()
 {
 double x, y, r;
 while(r>=1.0||r==0.0)
     {
          x=2.0*uni_dblflt()-1.0;
          y=2.0*uni_dblflt()-1.0;
          r=x*x+y*y;
     }
 r=sqrt((-2.0*log(r))/r);
 return x*r;
 }
```

The code is written in C programming language. The x, y variables are the two uniform random numbers we need to get a normal one, and r is the radius. The uni_dblflt() is a uniform RNG which can generate double float random numbers. The square root operation is a mathematical transformation of the equations described above.

2.3.3 The Transformation Problem

The Box-Mueller transformation do generate normal random numbers, whereas it is too expensive for the heavy condition check of the while loop in the code. For one normal random number generation process, the Box-Mueller transformation need at least two uniform random number generator calls, 7 multiply/divide operations, 3 addition operations and a square root function call. In fact, the time to generate a normal random number will be even more when considering the while loop.

The while loop conditions according to the Box-Mueller transform code is, x*x+y*y=r, and r range is (0, 1]. It is explained below in a graphic way by geometric definition.

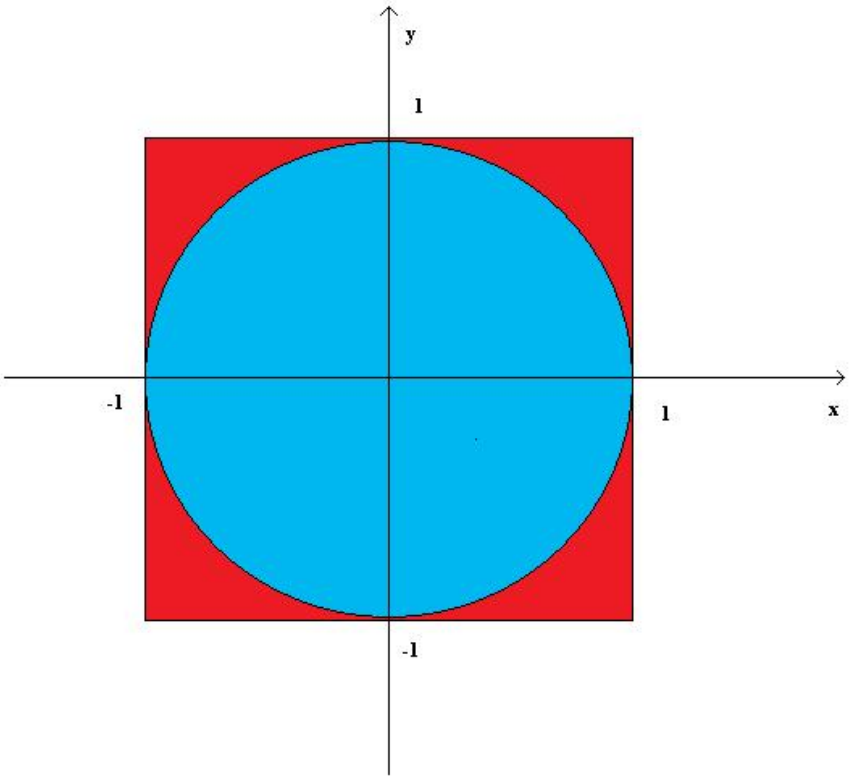


Figure-4 The while loop explanation in Box-Mueller transformation

Figure-4 shows a square and an inner circle. The square is constructed by all the possible points generated by uniform random number x [-1, 1] and y [-1, 1] separately. The inner circle is constructed by the range of r (0, 1]. The blue circle area are the accept parts which can pass the while loop. The four red triangle-like areas at corners are the unaccepted parts, which mean to loop again.

Only when the point defined by the uniform random number x and y located inside the blue circle area and both x and y do not equal to 0, the while loop can pass. However, if the point is located in the red parts, 4 multiple operations, 3 reduction operations and a condition check will be implemented again according to the Box-Mueller code described above.

Through the theory of probability, we can calculate P, the possibility of passing the while loop in one time is πr^2 (the area of circle) divide $4r^2$ (the area of square), which equals to 0.785 approximately. From the theory of probability, the probability of x or y located at a specific point is 0 percent, thus we can neglect the 0 point situation when doing the probability calculation.

The expected times to pass the while loop for this problem can be found in the geometry distribution, which equals to 1/p = 1.27. It means a normal Box-Mueller transformation should experience 1.27 times while loop. The extra 0.27 times will cost more repetition work time when an order of magnitude normal random numbers are need.

A much faster way of generating Gaussian deviates (the Ziggurat method) has been proposed by Marsaglia & Tsang[10], whereas it is no implemented for our project for its complexity. The next chapter will show how the current normal random number generator affects the efficiency of the whole simulation progress.

Chapter 3: Current problems and possible solution

This chapter demonstrates the current efficiency problems of the simulation program in random number generation and provides a possible technique solution named Ladd's discrete random number generator [4].

3.1 Problems and Targets

The simulation of "Fluctuating Hydrodynamics" [1] is based on the theory of Brownian Motion. For the particular Lattice Boltzmann method [2] used, each fluid lattice needs exactly 11 or 15 normal random numbers to determine the random velocity and move direction per timestep.

Once modelling the whole program, each processor will generate 32*32*32 lattices, more than 10 normal random numbers per lattice per timestep are required, and then 100000 steps will be implemented. As a result, there are totally around 3.3E10 normal random numbers will be generated for each processor. Thus the efficiency problems are crucial to the modelling process, and it will entirely be determined by the quality and birth rate of both the uniform and normal random number generators.

The current uniform RNG be used is a certain old generator of Pierre L'Ecuyer [11] (name is not described in code), which include a lot of pointer operations and not easy to use. In addition, the complex operations make it inefficient. The current normal RNG be used is the Box-Mueller transformation which generates normal random number based on uniform random number. As described in Chapter 2, it is such an expensive way that more than 20 percent of the total runtime is cost by producing and transforming random numbers in the simulation.

Obviously, the quality of both the uniform and normal RNGs will determine the efficiency. Thus it is necessary to find better uniform and normal RNGs to release the pressure of the random number generation for the whole simulation. Our target is to reduce its time cost to less than 10 percent of the total runtime, and in the best situation, less than 1 percent.

3.2 Latest Technique

There is a fast discrete random number (known as DRN) generator for stochastic simulations published by Anthony J.C. Ladd [4]. It is an approximate normal random number generator which will get the same statistical results to the normal RNG. Thus it is possible to use the DRN to replace the current normal RNG for the "Fluctuating Hydrodynamics" simulation. However, the practicability should be verified by its performance after the implementation progress.

The discrete random number generator is also called 8-state RNG. We assume the 8 states as $-a_4$, $-a_3$, $-a_2$, $-a_1$, a_1, a_2, a_3 , and a_4. All the random numbers are represented by 3 binary bits, which will only get one of the predefined 8 values from a lookup table. The relationship between the 3 binary bits and the 8 states is shown in the lookup table below:

3 Binary Bit	**Represented States**
000	- a_4
001	- a_3
010	- a_2
011	- a_1
100	a_1
101	a_2
110	a_3
111	a_4

Table1- Relationship between 3 binary bits and 8 states in Ladd's technique

The probability function of generating each 3-bit number is as follows [4]

$$p(x) = \frac{1}{2^m} \sum_{i=1}^{2^{m-1}} [\delta(x + a_i) + \delta(x - a_i)]$$

The $\delta(x)$ is called Dirac delta function that have two states, either 1 or 0. When the parameters in Dirac delta function is 0, then $\delta(x)$ equals to 1, otherwise, it equals to 0. By matching the mean variance and forth momentums of this distribution with the standard normal distribution, the value of a_i can be calculated. The m is chosen to be 3, thus we can solve the x with its 8 states. The corresponding discrete distribution is plotted as follows.

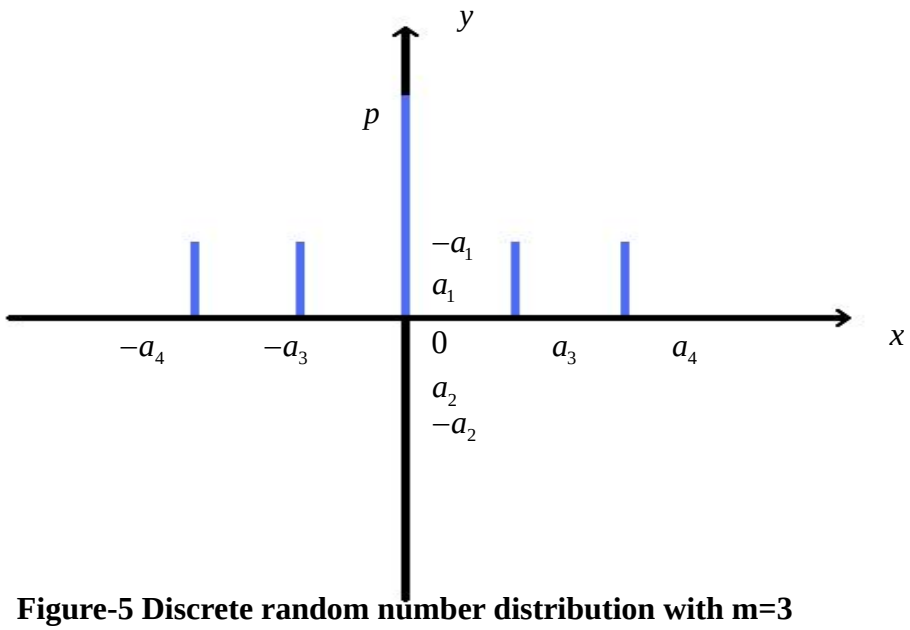


Figure-5 Discrete random number distribution with m=3

The y axis represents the probability of generating each x. The blue bars represent the actual probability. The 8 values of a_i from left to right are $-\sqrt{2+\sqrt{2}}$, $-\sqrt{2-\sqrt{2}}$, 0, 0, 0, 0, $\sqrt{2-\sqrt{2}}$,and $\sqrt{2+\sqrt{2}}$ separately. It can be calculated that the generator will generate 0 with probability of 50%, and the other four values with probability of 12.5% respectively. The DRN generator can be simply viewed as a RNG which will only generate five possible values.

The main concept of implementing the DRN technique is to use 3-bit to represent a random number from the lookup table. Thus for each 32-bit integer, at least 10 of the 3-bit discrete random number can be obtained. The main processes are as follows.

First, generate a 32-bit random integer by any random number generator. Second, remove the redundant 2 bit, and then do a 10 times loop to get the 10 3-bit random numbers. Third, return the 10 DRNs sequentially according to the corresponding values in the lookup table.

In brief, we can get 10 random numbers from just one uniform RNG call in Ladd's DRN method. If it is implemented correctly, much more time will be saved comparing to the expensive Box-Mueller transformation.

3.3 Basic Requirements

According to the problems and techniques described above, some basic requirements for the project are summarised.

1. Find random number generator that can generates good pseudo random numbers with the consideration of both quality and efficiency. The RNG that need to found is uniform distributed.

2. Do sufficient tests and carefully analyse the statistical data of the chosen uniform RNGs, in which the correctness tests should be mainly concerned. Then choose the best one for the project according to the performance of quality and efficiency.

3. Implement the Ladd's DRN alone, do some correctness test and implement it into the whole program, and then check the performance.

4. Parallelise the progress of DRN generator to check the performance in advance.

5. If time allowed, do more optimisation for the simulation program to reduce the execution time further.

Chapter 4: Structure of Current Simulation for Random Number Generation

The existing “Fluctuating Hydrodynamics” simulation program [3] is written in C programming language. This chapter will illustrate the random number generation related code, the execution of whole simulation and some options in execution.

4.1 Code Review

They are mainly two files, collision.c and ran.c related to the progress of random number generation.

The simulation can model basically two modes of fluid, the single fluid and the mixed fluid from two single fluids (known as double fluid) and Collision.c file provides two standalone functions to make the choice when executed. The two fluid modes just different in workload and do the similar parameters’ calculation for each fluid lattice point. The calculated parameters include the density of particles in the fixed area, the force in each direction, the three dimension velocity and so forth.

When the constructor (initialisation) of the Collision.c file is implemented, the single or double fluid modes will be chosen, and then the execution will follow the steps below when random number generation is turn on.

1. Generate a random symmetric 3*3 matrix for the stress (force) calculation of fluid lattices, which requires exactly 6 random numbers.
2. Enter into a “ghost mode” that needs several random numbers to calculate fluid lattices’ velocity. The specific random numbers required depend on the lattice type. D3Q19 lattice need 9 random numbers and D3Q15 lattice need 5.

A variable named NVEL will recognise the lattice type in other files and be used in the two steps calculation above. For example, NVEL=15 for D3Q15 lattice and NVEL=19 for D3Q19 lattice. The total random numbers required for each lattice situation is equal to NVEL-1-NDIM, and the NDIM is the dimension numbers. Thus for each D3Q19 lattice, 15 (19-1-3) random numbers are required, and for each D3Q15 lattice, 11 (15-1-3) random numbers are needed.

Ran.c supplies all the random number generators for the collision part, such as serial and parallel uniform generators, serial and parallel normal generators. The original uniform random number generator used in the simulation is the Pierre L'Ecuyer’s one [11], and the normal random number generator is the Box-Mueller method described in previous chapters.

4.2 Execution of “Fluctuating Hydrodynamics” Simulation

The execution of the current simulation of “Fluctuating Hydrodynamics” has several steps.

The first step is to read an input file named input.ref to initialise all the relevant properties of the fluid point lattices and then output the initial setting. The input.ref file contains all the critical input values for the simulation, such as the number of lattice Boltzmann timesteps to run (cycles), the time of restart, the size of the system in lattice units, the temperature value, and so forth.

The initialisation operation output is shown in Appendix B**,** which shows the number of user parameters, the total size of fixed environment, the boundary type, the processors decomposition, the communicator of parallelisation, the initial random number seed, the gravity values of 3 dimension, the lattice type used, the simulation mode used and other correlated physic values.

The second step is to execute the main modelling process. The results of critical attributes such as velocity of particles with its expected values are outputted each timestep, in which the unit for the timestep is cycles and the output frequency is 50 cycles. In this step, the particles are stayed in a 32*32*32 environment in the simulation with the velocity of 0. A sample of 50 cycles result is given in Appendix B.

The final result of one time simulation will output the total execution time, the time cost for main functions (both maximum and minimum) and the corresponding functions call times in the simulation. An example of 500 cycles’ execution on the NESS computer is shown below, which uses the GNU project C and C++ compiler (version 4.1.2) with the optimisation flag O3 turning on.

```
Completed cycle 500
Timer resolution: 1e-06 second
Timer statistics
Section:          tmin          tmax          total
Total:          39.480        39.480         39.480    39.480000 (1 call)
Timestep loop:   0.070         0.090         39.340    0.078680 (500 calls)
Propagation:      0.000         0.010          3.520    0.007040 (500 calls)
Collision:         0.050         0.070         30.190     0.060380 (500 calls)
Lattice halos:     0.000         0.010          3.100     0.003100 (1000 calls)
phi gradients:     0.000         0.010          2.380     0.004760 (500 calls)
Ludwig finished normal
```

The precision of time is 1e-06 second

The output shows the total execution time for a 500 cycles' simulation is 39.48 seconds. The propagation, collision, and phi gradients functions are called 500 times, and the Lattice halos swap function is called 1000 times. In these functions, the collision is the main part related to the operation of random number generation.

4.3 Options in Executions

The simulation provides some different modes to meet a variety of scenarios in modelling. These modes can be chosen by turning on or off the corresponding function flags in compiling time. The commonly used ones are:

D_D3Q19, D_D3Q15, be used to choose the lattice type in Lattice Botzmann method.

D_NOISE_, be used to determine whether implement the random number generator when modelling, which is crucial for our extension.

D_SINGLE_FLUID_, be used to simulate the single fluid mode instead of the default double fluid modes.

As described in Chapter 2, the particles in natural are move randomly because of thermal effect. By turning the D_NOISE_ flag on, the random number generator will give random velocity to fluid lattices for the particles' movement and calculate the corresponding integrate temperature in the fixed modelling environment. The corresponding 50 cycles' execution output is shown below when turning on the D_NOISE_ and D_D3Q19_ flags.

```
TEST_momentum [x, y, z]
[total][5.41928e-15, 7.00481e-15, 2.31065e-15]
[fluid][5.41928e-15, 7.00481e-15, 2.31065e-15]
Wall net:     0.0000000000e+00     0.0000000000e+00     0.0000000000e+00
Velocity stats:
[ umin ][ -0.0191008 -0.0166607 -0.0180612 ]
[ umax ][ 0.0176375 0.0176788 0.018507 ]
TEST_fluid_temperature:
<v_x^2> = 2.13278e-05
<v_y^2> = 2.07335e-05
<v_z^2> = 2.10131e-05
<mv^2> = 6.30745e-05 (target: 6.4e-05)
<drho^2> = 6.36351e-05
<ghost> = 0
```

By comparing to the original non-noise 50 cycles output in Appendix B, the fluid_temperature attributes of fluid lattices are calculated out every timestep. All of the temperature unit are Lattice Boltzmann unit, which should be transformed to real unit when dealing with specific physics or mathematical problems.

The particles are simulated in a limited square area, thus the mean of all velocity <v_x> can be calculated. The attributes <v_x^2>, <v_y^2> and <v_z^2> are the mean of velocity square in the three dimension which are equal to 1/2 K_BT, in a given temperature T. The K_B is a Lattice Boltzmann constant. The rest two values <mv^2> and <drho^2> are the sum of the three mean of velocity square above and the expected value of the sum respectively. In real world, the energy provided by the thermal effect will generate force for fluid points to make them fluctuate and walk randomly.

We take another 10000 cycles run as an example to show the performance when D_NOISE_ flag used. The default simulation is run on NESS, uses the GNU project C and C++ compiler (version 4.1.2) with the optimisation flag O3 turning on. There are totally 3.3E9 Box-Mueller normal random numbers generated.

500cycles	**D3Q15**		**D3Q19**	
FLAG	**None**	**Noise**	**None**	**Noise**
Total	395.29	542.07	568.23	771.34
Collision	237.31	384.57	383.60	585.49

Table2- Execution time of sample simulation with NOISE flag (seconds)

In **Table2**, both the D3Q15 lattice and D3Q19 lattice results show the time cost when turning on the NOISE (random number generation) is (384.57-237.31)/542.07=27.2% and (585.49-383.360)/771.34 =26% of the total execution time respectively. It means the random number generation progress will add around 25% of the total execution time, which is obviously a problem to the efficiency of the whole simulation.

Chapter 5: Uniform Random Number Generator

There exist enormous uniform random number generators in the world wide range. The main rule should be abode by for making the choice is whether the generator can pass all and many other standard tests provided with the consideration of efficiency.

5.1 Available RNGs and Preliminary Consideration

Three random number generators chosen from the enormous available ones are as follows.

Numerical Recipes' RAN2 generator is proposed by Press and Teukolsky [1992] which is a modified version of CombLec88 [6]. The RAN2 generator is a particularly well-known combination generator, which combines two linear congruential generators and further randomizes by shuffling.

KISS32 generator is proposed by G.Marsaglia [8]. The KISS32 generator also combines three different generators.

CLCG4 generator is proposed by L'Ecuyer and Andres [1997] [12], which consists of four LCG generators, and it is definitely more robust but also slower than the other two RNGs.

To describe the concepts of combination generator, we take the KISS32 generator as an example. The corresponding code is as follows.

```
unsigned int x=123456789,y=362436069,z=21288629,w=14921776,c=0;
unsigned int KISS32(){
unsigned int t;
x=x+545925293;
y^=(y<<13);y^=(y>>17);y^=(y<<5);
t=z+w+c;z=w;c=(t>>31);w=t&2147483647;
return (x+y+w);
}
```

The code is written in C programming language.

As described in Chapter 2, doing combination is a good practice in designing RNGs to cover flaws when implemented alone. The KISS32 uniform RNG is a sample that combines 3 different simple generators. The state variables x, y, z, w and c are initialised seeds, t is the temporary variable. Random number X follows the liner congruential equation described in Chapter 2, Y is generated by bit movements, and W is generated by combination of both the liner congruential equation and bit movements. In the end, the generator adds up the three random numbers and returns the sum as the final random number.

To make the choice, the first factor that should be considered is the ease of implementation. Considering the number of code lines, the KISS32 generator is the simplest. Other two are difficult to implement to some extent. For instance, the CLCG4 generator has hundreds of code lines. Thus, the comparatively redundant CLCG4 and RAN2 code will not be explained and the specific codes are shown in Appendix C.

5.2 Correctness Test

The quality and efficiency of the three random number generators chosen above will be checked by the standard library TestU01 [7].

5.2.1 TestU01

TestU01 is a standard C library which gives a series of classical tests to verify the quality and efficiency of random number generators.

The tests in the library can be implemented in two ways. One is to use the predefined generator in the library and run the tests by default functions. Another way is to use an external function to pass user-defined generators (with no variables) to the acceptable predefined types, and then repeat the same process to the first way.

Some tests in the library may consist of a series of small tests. Thus, the output of the tests contains information of each small test and a final outcome to summarise the total progress. The final outcome contains the test name, the version of TestU01, the RNG name in the test, the CPU time cost, and a p-value result to show whether the test is passed.

The p-value is the output value for a specific test which will mainly judge the quality of the RNG. The usual range of the p-value is (0, 1). If the p-value is too small ($<10^{-10}$) or too big ($>1-10^{10}$), it means the test of generator fail. The generator may be suspected to be fail if the p-value is outside of the range [0.001, 0.9990].

5.2.2 Sample Test Output

Take the Small Crush tests output for LCG generator as an example. The test is run on NESS, uses GNU project C and C++ compiler, with the optimisation flag O3 turning on. The output of a small test named smarsa_BirthdaySpacings is shown in Appendix D, which shows the initialised data for the specific test, the p-value of the test, the CPU time cost and the current generator state (seed). The outputs of the total SmallCrush tests are as follows.

========= Summary results of SmallCrush =========

Version: TestU01 1.2.3
Generator: LCG
Number of statistics: 15
Total CPU time: 00:00:17.57
The following tests gave p-values outside [0.001, 0.9990]:
(eps means a value < 1.0e-300):
(eps1 means a value < 1.0e-15):

	Test	p-value
1	BirthdaySpacings	eps
2	Collision	eps
6	MaxOft	eps

All other tests were passed

The output shows the version of TestU01 is 1.2.3 and the time cost to run the SmallCrush test is around 17.57 seconds. The LCG generator passed other tests in Small Crush but fails in the three tests, BirthdaySpacings, Collision and MaxOft, with its p-value out of legal range.

Other tests in TestU01 library are Alphabit, SmallCrush, Crush, Crypt-X, Diehard, ent, fips, nist, PSEUDODIEHARD, RABBIT, SPRNG, Berlekamp-Massey algorithm and so on.

5.2.3 Test Result

The KISS32, RAN2 and CLCG4 generators have passed all of the standard tests listed in TestU01 library. Thus for the consideration of quality, all of the three generators are good enough as a uniform random number generator.

The next step is to check the efficiency of the three generators. Among those tests, there is a test named BigCrush that need more than 6 hours to finish, which can be viewed as a big challenge for RNG. At the same time, it is also the hardest quality test in TestU01. Thus, it is declared that if a RNG can pass all the BigCrush tests as quickly as possible, then the RNG is perfect enough.

The BigCrush tests outputs for the three generators are shown in Appendix D. The results can be illustrated simply as follows.

Generators	**Time cost**	**Failed tests**	**Result**
CLCG4	9 hours, 6 minutes	NONE	PASS
RAN2	7 hours, 48 minutes	NONE	PASS
KISS32	6 hours, 15 minutes	NONE	PASS

Table3- The BigCrush test result for generator CLCG4, RAN2 and KISS32

For comparison, the BigCrush tests are run for other uniform random number generators to see how many tests they may fail. The sample uniform RNG used is the LCG generator. Details are shown in Appendix D, and the result is given below:

Generator	**Time cost**	**Failed tests**	**Result**
LCG	6 hours, 57 minutes	6	FAIL

Table4- The BigCrush test result for LCG generator

Table3 shows that, all of the three generators CLCG4, RAN2 and KISS32 can pass the tests contained in the TestU01 and the BigCrush tests. However, the CPU time cost for BigCrush tests are different. KISS32 cost nearly 6 hours and 15 minutes, RAN2 cost around 7 hours and 48 minutes, and CLCG4 cost nearly 9 hours.

Therefore, although the three generators are good at quality, the CPU time performance made us believe the KISS32 is the best uniform random number generator for the “Fluctuating Hydrodynamics” simulation and thus we should use it for the following works.

Chapter 6: Implementation of Ladd's DRN

This Chapter illustrates the design and implementation of the Ladd's Discrete Random Number [4] to replace the Box-Mueller transformation and gives the corresponding correctness tests.

6.1 Design and Implementation alone

As described in chapter 3, the DRN is represented by 8 state values. Therefore, the first step for DRN implementation is initialisation. A lookup table similar to **Table1** to store the 8 states should be set up.
The C code is as follows.

```
double a3,a4;
a4=sqrt(2+sqrt(2));
a3=sqrt(2-sqrt(2));
table[0]=-a4;
table[1]=-a3;
table[2]=0;
table[3]=0;
table[4]=0;
table[5]=0;
table[6]=a3;
table[7]=a4;
```

The second step is to generate a uniform random number, remove the redundant 2 bit and then get a 3-bit DRN 10 times. [4]

```
int w=0,i,DRN;
DRN=uniform_random()>>2;
for(i=0;i<10;i++)
    {
        w=table[DRN&7];
        DRN>>=3;
    }
```

The [u&7] uses the "and" operation to transform the random 3-bit binary value to the lookup table values. The uniform random number generator that will be implemented is the KISS32 generator, which is the best choice of uniform RNGs described in Chapter 5.

As a preliminary test, 100 continuous discrete random numbers are generated. The output is listed in Appendix E, which shows the majority of the DRN generated are 0, and other four values are correct according to the DRN's principles described in Chapter 2. However, whether it is competent for the simulation need further investigation.

6.2 Correctness Test

According to the theory of normal distribution and the definition in the DRN technique [4], the DRN sequence we implemented should satisfy some common mathematical principals. The moments of distribution for the probability function p(x) should satisfy $< x^0 >=1$, $< x^1 >=0$, $< x^2 >= \sigma^2$, $< x^3 >=0$, $< x^4 >=3\sigma^4$, $< x^5 >=0$. The σ in the equations is the standard deviation of corresponding distribution. The normal random number used in the simulation follows standard normal distribution, thus the σ will equal to 1, and then the 1-5 moments of the DRN sequence should satisfy 1^{th} (mean) = 0, 2^{th} (variance) = 1, 3^{th} =0, 4^{th}=3, 5^{th}=0.

The results of 1-5^{th} moments calculated on DRN distribution are as follows.

DRN Sequence Length	**1 th**	**2 th**	**3 th**	**4th**	**5th**
100	0.173833	1.434132	0.334580	4.536180	2.590463
1000	0.047718	1.046469	0.017347	3.155356	0.318691
10000	0.007371	1.024246	-0.007279	3.082574	0.008640
1000000	0.003588	1.000165	0.001530	3.001622	-0.011841
10000000	-0.000332	1.000055	0.000055	3.000144	-0.001975
100000000	-0.000029	0.999932	-0.000113	2.999791	-0.000072
1000000000	0.000022	0.999952	-0.000011	2.999836	-0.000168

Table5- The 1-5th moments for DRN sequence

Table5 shows that, with the DRN sequence length increased, the 1-5 moments are closer to the principals. It proves that our implementation process is correct and the whole simulation could use it directly.

6.3 Alternative Correctness Test

There exists another way to check the correctness of the DRN implementation. A method listed in Ladd's paper [4] is to generate a sequence of DRN and calculate the cumulants $< x^2 (t)>$ and $< x^4 (t)>$ with different steps t. If the DRN implementation is

correct, the distribution graphs of $< x^2(t) >$ - t and $< x^4(t) >$ - t will be plotted as **Figure-6**, which shows the correlation between the DRNs. The <x2(t)> is Δ^2, the <x4(t)> is Δ^4, and the t steps are 100 and 10000 respectively.

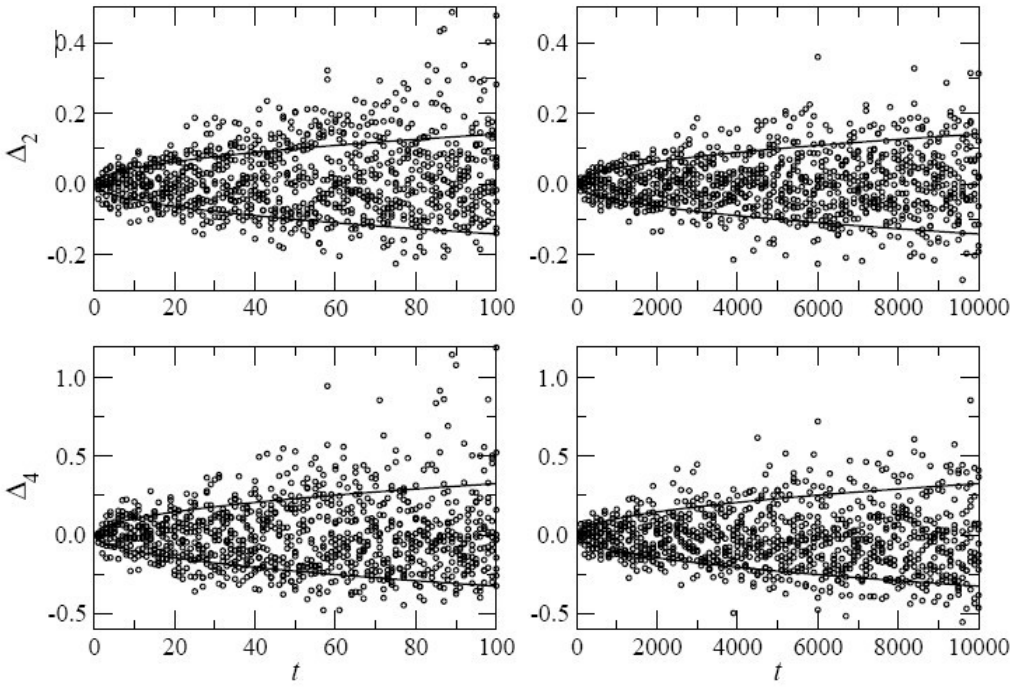


Figure-6 Distribution of $< x^2(t) >$ - t and $< x^4(t) >$ - t [4]

We have tried several ways to calculate the cumulants, whereas by no means the result looks like the **Figure-6**. Now that the DRN implementation has passed the 1-5th moments check in 6.2, it is enough to prove the correctness. This alternative test can be viewed as an option for future investigation.

6.4 Integrated Implementation

To implement the DRN into the whole simulation, basically two methods are provided. We could call them method 1 and method 2 for simple.

6.4.1 Method1 Design

The way in Method 1 is to count the total number of DRNs generated inside the function and fully get use of the 10 DRNs for each uniform integer. The progress is as follows.

1. Generate one uniform integer and 10 DRNs at the first function call, and then returns one DRN.
2. The rest times function call will sequentially get the other 9 DRNs.
3. After 10 times call, the whole generation progress will repeat again.

The corresponding C code is:

```
static int count= 0;
static double DRN_array[10];
double Method1(){
        int i;
        int DRN;
        if(count==10){
          count=0;
        }
        if(count==0){
        DRN=KISS32()>>2;
        for(i=0;i<10;i++){
        DRN_array [i]=table[DRN&7];
        DRN>>=3;
        }
        return DRN_array [count++];
        }
        return DRN_array [count++];}
```

The table array is the lookup table described in Chapter 3 for the 8 states of DRN.

6.4.2 Method2 Design

Method 2 is to let the corresponding modes (stress matrix and ghost mode, described in 4.1) in collision.c file pass a variable and an array to the DRN generator when random number required. The variable is the number of random number required for specific scenario (6 in stress matrix, 5 or 9 in ghost mode). The array is used to store the required random numbers. Method 2 will do the same implementation of the DRN, whereas the difference is that it will generate and return exactly the required number (the number should less than 10) of DRNs into the passed array.

The corresponding C code is:

```
        void Method 2(int n,double *array){
        int i,DRN;
        DRN=KISS32()>>2;
        for(i=0;i<n;i++)
        {
        array[i]=table[DRN&7];
        DRN>>=3;
        }
```

6.5 Integrate Correctness Test

To decide whether the DRN generator works correctly in the whole simulation, the force of each lattice should be checked firstly. The reason is that the force will be added by a random number to give a velocity to each fluid lattice in execution, and if the random number is wrong, then the force will be modified in a wrong way.

If correctly implemented, the sum of the force values should be the same before and after the noise (random number). In addition, the attributes related to the force calculation should also keep its value unchanged during the process, which are the density and the velocities in all x axis, y axis and z axis.

Thus, we have checked the sum of the force values, the density, and the velocity in three dimensions before and after the noise. As a result, the density and the velocity in all dimensions keep the same. However, there exists difference between the original force and the changed force, though the difference is less than $2.3*10^{-10}$. By analysing the code, we found that it is the rounding error caused by using double precision data, but not the fault of our implementation. The total execution of the “Fluctuating Hydrodynamics” simulation will generate more than $3.3*10^{10}$ number of lattice, thus the rounding error is acceptable.

Chapter 7: Performance

This chapter will check the performance of compilers combining with its optimisation flags and the two alternative methods in implementing the Ladd's technique [4] combining with its analysis. All of the tests are run on both NESS and HECTOR computers to get a trustable statistical result. The NESS we used is supplied by Sun, which uses AMD Opteron processors. And the HECTOR hardware we used is provided by Cray, which uses the Cray XT4 parallel processors.

7.1 Compilers and Optimisation flags

Before the performance check of the simulation, the C compilers to be used should be considered firstly. The compilers available on both platforms are GNU project C and C++ (known as gcc), The Portland Grup Inc. ANSI and K&R C compiler (known as pgcc). In addition, the PathScale(TM) C or C++ compiler is used on HECTOR only. The C compiler (cc) will not be used since it will point to gcc on NESS and point to pgcc on HECTOR respectively. The corresponding versions of these compilers are as follows.

On NESS:
Gcc: 4.1.2 20080704 (redhat 4.1.2-46)
pgcc 10.0-0 64-bit target on x86-64 Linux -tp k8-64e

On HECTOR:
Gcc: 4.1.2 20070115
pgcc 9.0-4 64-bit target on x86-64 Linux -tp k8-64e
pathcc GNU gcc version 4.2.0 (PathScale 3.2.99 driver)
Some simple comparisons for these compilers and optimisation flags on a variety of scenarios are provided. The different conditions and performance results are as follows.

The original simulation that turns D_D3Q19_ and D_NOISE_ flags on is used to compares the PGI and GNU compilers on HECTOR and NESS.

500cycles	**HECTOR**		**NESS**	
Compiler	**PGI compiler**	**GNU C and C++ compiler**	**PGI compiler**	**GNU C and C++ compiler**
Total	58.09	36.48	59.89	39.91
Collision	47.32	26.02	50.43	30.23

Table6- Execution for different compilers on ness and hector (seconds)

Use the same conditions to the **Table6** comparison, and implement the Method 1 & 2 to check the three compilers PGI, GNU and PathScale on HECTOR.

500cycles	**Method1**			**Method2**		
Compiler	**PGI compiler**	**GNU C and C++ compiler**	**PathScale compiler**	**PGI compiler**	**GNU C and C++ compiler**	**PathScale compiler**
Total	50.33	29.8	30.85	50.66	30.98	30.36
Collision	39.58	19.35	20.57	39.81	20.51	20.03

Table7- Execution for different compilers when method1 and 2 used on hector (seconds)

Use Method 1 with the D_D3Q19_ and D_NOISE_ flags on and the two compilers on NESS to compare the effect of optimisation flags O1, O2 and O3.

500cycles	**GNU project C and C++ compiler**			**PGI compiler**		
Optimise flags	**O3**	**O2**	**O1**	**O3**	**O2**	**O1**
Total	30.37	30.59	40.96	52.22	52.7	58.34
Collision	20.64	21.01	31.36	42.85	43.29	48.36

Table8- Execution for different compilers with different optimisation flags on ness (seconds)

From the **Table6**, **Table7** and **Table8** results, we suppose the GNU compiler is more suitable for our program than others on ness. The GNU and PathScale compilers are both good on HECTOR, and the best optimisation flag to make the execution fast is the O3. Thus, for all of the following performance tests, if not mentioned, the compiler used will be the GNU project C and C++ with the optimisation flag O3 turning on.

7.2 Original Code Performance

The simulation program is run on NESS and HECTOR. For there will be a variety of comparisons in different scenarios, the entire simulation will just run 500 cycles. The results are obtained by running the original model with different execution flags turning on and all of the results are the average results of more than 10 times execution. The performance results are as follows and the time unit is second.

On NESS:

500 cycles	**D3Q15**				**D3Q19**			
	None	**Noise**	**Single**	**Both**	**None**	**Noise**	**Single**	**Both**
Total	20.11	28.21	13.17	21.44	28.90	40.21	20.19	31.60
Collision	11.73	20.04	8.37	16.48	19.62	30.79	14.78	26.24

Table9- Original simulation execution time with different scenarios on ness (seconds)

On HECTOR:

500 Cycles	D3Q15				D3Q19			
	None	**Noise**	**Single**	**Both**	**None**	**Noise**	**Single**	**Both**
Total	20.05	25.42	13.20	18.69	28.67	36.48	20.22	27.98
Collision	11.41	16.72	7.75	13.32	18.23	26.02	14.01	21.77

Table10- Original simulation execution time with different scenarios on hectors (seconds)

D3Q15/D3Q19 means all of the fluid lattices used in the simulation are D3Q15 or D3Q19. NONE means no other flags are turned on, and the program runs the default double fluid mode. The NOISE means turn the random number generation (D_NOISE_) on, and the SINGLE means simulate only one fluid (D_SINGLE_FLUID_) instead of two. Both means turn the two flags D_NOISE_ and D_SINGLE_FLUID_ on.

The **Table9** and **Table10** only gives the total runtime and collision time because they are the critical parts related to the random number generation. It is clear that the single fluid mode will cost much less time than the two fluid normal models, because the former one gets less workload to do.

The original simulation without noise already cost some times in collision part. For both D3Q15 and D3Q19 conditions on different platforms, turn the noise on will get an extra fixed time cost. Take the D3Q15 on NESS as an example, NONE-NOISE difference is 9 seconds, and the SINGLE-BOTH difference is also around 9 seconds.

The performance on both HECTOR and NESS are similar when no noise implemented. However, when considering the effect of noise, simulation on HECTOR get a better performance than on NESS. For example, the noise mode cost extra 8 and 12 seconds from none mode for D3Q15 and D3Q19 respectively on NESS, but on HECTOR, only 5 and 8 seconds are added respectively.

7.3 Performance after DRN Implementation

The performance is compared between the implementations of method 1 and method 2 on the two platforms with the D_NOISE_ flag turning on. Other conditions are the same to the comparison in 7.2. The NONE and NOISE results from the 7.2 comparison is provided to make the performance sensible. The **Percentage** is the proportion that the extra collision time (the random number generation) accounts for the total execution time. The performance results are as follows.

On NESS:

500 cycles	D3Q15				D3Q19			
	None	Noise	Method1	Method2	None	Noise	Method1	Method2
Total	20.11	28.21	20.71	22.19	28.90	40.21	30.21	31.71
Collision	11.73	20.04	12.73	14.18	19.62	30.79	20.81	22.20
Percentage	0	29.5%	4.8%	11%	0	27.8%	3.9%	8.1%

Table11- Execution time with different scenarios for method1 and 2 on ness (seconds)

On HECTOR:

500 cycles	D3Q15				D3Q19			
	None	Noise	Method1	Method2	None	Noise	Method1	Method2
Total	20.05	25.42	19.66	21.73	28.67	36.48	29.59	30.84
Collision	10.41	16.72	11.05	12.18	18.23	26.02	19.18	20.53
Percentage	0	24.8%	3.3%	8.1%	0	21.4%	3.2%	7.5%

Table12- Execution time with different scenarios for method1 and 2 on hector (seconds)

It is obvious that both method 1 and method 2 cost much less time than the original raw NOISE mode that uses Box-Mueller transformation. Especially, the time cost by using method1 almost equals to the original NONE mode. In addition, from the **percentage** results of both **Table11** and **Table12**, it is clear that the time proportion of the random number generation progress is reduced from more than 20% to no more than 5% in method1 and no more than 11% in method2 on both platforms. It almost closes to our expectation for the random number generation described in Chapter5.

Another thing shown in the **Table11** and **Table12** is that method1 performs better than method2 on both NESS and HECTOR. To make it clearer, the specific speedup gained by the two methods comparing to the original Box-Mueller transformation are calculated from **Table11** and **Table12**. The speedup gained is as follows.

Lattice Type	D3Q15		D3Q19	
Method	1	2	1	2
NESS	8.31	3.41	9.4	4.33
HECTOR	9.86	3.56	8.2	3.4

Table13- Speedup gained by method 1 and 2 on ness and hector than Box-Mueller

From the speedup results of **Table13**, method1 do perform better, which gets almost twice speedup than method2. A possible reason is the different numbers of uniform RNG call.

As described in Chapter 6, method1 will fully get use of the uniform RNG to generate 10 3-bit DRNs, whereas method2 will also use the uniform RNG but just get the required (<10) number of DRNs out. For example, for 100 D3Q15 lattices, totally 1100 random numbers are needed (one D3Q15 lattice need 11 random numbers). Method 1 will run the uniform RNG for 1100/10=110 times, whereas the method 2 will treat it as 600 numbers for stress matrix mode (require 6 numbers) and 500 numbers for ghost mode (require 5 numbers). Thus uniform RNG in method2 will be called 600/6+500/5=200 times. Therefore, method 1 may get 200/110=1.8 times speedup than method2 regardless of other elements.

As a result, the 8 to 9 times speedup gained by method1 is close to the 10 times speedup in the theory of discrete random number [4]. To ensure the situation is repeatable for all of the possible scenarios, the two methods are executed further in other situations.

7.4 Performance on Single Fluid

This comparison is to check the performance of the two methods when turning on D_SINGLE_FLUID_. Other situations are the same to the 7.3 comparison. The NONE means just turn on the D_SINGLE_FLUID_ flag, and the D_NOISE_ mean both D_SINGLE_FLUID_ and D_NOISE_ flags are turned on. The results are as follows.

On NESS:

500 cycles	**D3Q15**				**D3Q19**			
	None	**Noise**	**Method1**	**Method2**	**None**	**Noise**	**Method1**	**Method2**
Total	13.17	21.44	14.26	23.54	20.19	31.6	21.73	30.78
Collision	8.37	16.48	9.21	18.56	14.78	26.24	16.11	25.43

Table14- Execution time when single fluid model for method1 and 2 on ness (seconds)

On HECTOR:

500 cycles	**D3Q15**				**D3Q19**			
	None	**Noise**	**Method1**	**Method2**	**None**	**Noise**	**Method1**	**Method2**
Total	13.2	18.69	14.07	22.48	20.22	27.98	20.92	29.86
Collision	7.75	13.32	8.56	16.74	14.01	21.77	14.79	23.25

Table15- Execution time when single fluid model for method1 and 2 on ness (seconds)

In **Table14** and **Table15**, the results of method1 are similar to the 7.3 comparison on both platforms. However the execution times of method2 are surprisingly strange. In principle, although method2 is not as good as method1, it should at least perform better than the original expensive Box-Mueller transformation by definition. However, the execution time of method2 performs even worse than the original one. The single fluid and double fluid are just different in work load, and the code operations of them are similar. We could not find the reason by analysing the process of random number generation. This is an interesting problem which may be caused by other elements in the simulation.

Even though the method2 is able to perform regularly on single fluid mode, it could not conquer the benefit of method 1 based on the definition and the existing results. In summary, method 1 will be chosen for the following analysis because it performs well on each situation in 7.3 and 7.4 comparison.

Chapter 8: Parallelisation of DRN

8.1 Design

The existing C code is already parallelised by using the Message Passing Interface, in which the progress of random number generation for multi-processors uses the same RNG with different seeds.

As described in Chapter 2, the same seed problem may result in bad random sequence with some unwanted correlation between random numbers. Therefore, the significant problem of the parallelisation of DRN is how to seed the DRN generator properly and carefully.

8.2 Implement Alone

According to the code explanation in previous chapters, the DRN technique [4] is also a transformation based on uniform RNG, and no seed exists in it. Thus for the parallelised seeding process, what should be concentrated on is the uniform random number seeds in the KISS32 RNG implemented in method1.

The original KISS32 generator has the seeds as follows.
unsigned int x=123456789,y=362436069,z=21288629,c=0,w=14921776.
A possible way of seed is to add or minus a small number to the x, y, z, c, and w variables immediately after the seed initialisation above to make it different. To avoid repetition, the chosen small number should multiple the processors' ranks, and then all of the processors will begin its generator with different seeds. The corresponding C code is as follows.

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
x + =rank*100;
y - = rank*100;
z + = rank*100;
c + = rank*100;
w += rank*100;
```

The small number chosen in the simulation is 100, since it just need to make the seed different.

8.3 Basic Correctness Test

Based on the seeding pattern above, the DRN generator is run for 4 processors, and each processor will generate a uniform random number sequence of 10 to check the correctness. The KISS32 generator will return a number with the range of -2^{31} to $2^{31}-1$, and the results are as follows.

Number/Rank	**0**	**1**	**2**	**3**
1	-435416739	-462449185	-502200176	-390808718
2	1870505447	42313706	252278400	-1442955152
3	1037754587	-1470209184	-619149376	-2141071960
4	-1065584380	1920012731	1375262559	-895450292
5	32571412	-445791017	-845563639	-550419037
6	595628261	1119908152	-1110272141	1636674618
7	-1382145479	752280574	-1563486669	-1856388351
8	480783889	-1630123287	1049417087	1540280032
9	1102596374	-323982351	2016874254	748050609
10	2125093149	65263119	-661982657	-465371686

Table16- Random sequence generated by 4 processors

The data in **Table16** show that the sequence generated by each processor is totally different. For the purpose of repetition, the sequence generated should be the same for each time run. Therefore, the test is run for more than 20 times and it gets absolutely identical result of **Table16**. If no other problems exist, it will be used in the whole program next.

8.4 Integrate implementation

To parallelise the discrete random number generator in the whole program, the method1 should be passed to the existing parallel normal random number generator in ran.c file as a return value. And then the seeds should be properly initialised by the method described in the initial function of ran.c file.

8.5 Parallel Performance

The performance of parallelisation is checked by calculating the average time of 20 times execution with the precision 0.01. The following records are the runtime of the simulation with different scenarios on 1, 2, 4, 8 and 16 processors. Because the Message Passing Interface should be used to run the parallel version, the GNU or PathScale compilers will no longer be used. Instead, the mpi C compiler on NESS and C compiler on HECTOR will be used, since the scalability and the speedup gained are the key points in parallelisation.

8.5.1 Performance on NESS

The runtime on NESS is easy to fluctuate, and the biggest deviation from several times run could be 4 seconds, which may be caused by the interactions between processors. The speedup calculation will only consider the collision part to check the performance of DRN generator.

The records on NESS are shown below. The **Percentage** is how much proportion the collision part accounts for the total simulation runtime with the processors numbers increased.

500 cycles run on original program, with D_NOISE_ flag turning off. Use mpi compiler on NESS, with the O3 optimisation flag turning on.

Processors	**1**	**2**	**4**	**8**	**16**
Total	72.20 ± 4.61	34.00 ± 3.25	21.50 ± 2.00	16.35 ± 1.55	8.00 ± 0.30
Collision	43.00 ± 0.50	21.48 ± 0.22	10.77 ± 0.90	5.50 ± 0.05	2.64 ± 0.01
Percentage	59.65%	63.2%	50%	33.6%	32.9%
Speedup	1	2	4	7.8	16.3

Table17- Parallel executions of the original simulation on ness (seconds)

Identical to **Table17** setup, with D_NOISE_ flag turning on.

Processors	**1**	**2**	**4**	**8**	**16**
Total	68.50 ± 0.50	40.25 ± 0.70	23.5 ± 1.50	17.50 ± 0.40	8.75 ± 0.90
Collision	51.05 ± 0.08	25.80 ± 0.10	12.90 ± 0.07	6.55 ± 0.14	3.15 ± 0.50
Percentage	74.5%	64%	55%	37.5%	36%
Speedup	1	2	4	7.8	16.2

Table18- Parallel executions of the original simulation on ness with noise on (seconds)

Implement method1 with identical setup to **Table18**.

Processors	**1**	**2**	**4**	**8**	**16**
Total	61.7 ± 0.05	35.56 ± 4.40	20.40 ± 1.80	16.00 ± 1.50	7.68 ± 0.50
Collision	44.12 ± 0.04	22.15 ± 0.10	11.00 ± 0.04	5.65 ± 0.04	2.71 ± 0.01
Percentage	71.5%	62.3%	53%	36.4%	35.28%
Speedup	1	2	4	7.81	16.28

Table19- Parallel executions of simulation implement method1 on ness (seconds)

Table17, Table18 and **Table19** show that the speedup of the collision part is ideally linear. With the processor numbers increased, the time proportion of collision part become smaller. It is because that other three functions Propagation, Lattice halos, and Phi gradients (described in Chapter 5) will account for more times. Especially, more communication overhead will occur in the halo swaps function when more processors are used.

To see the difference of collision part, we obtained the collision time of the three tables, **Table17**, **Table18** and **Table19**, without fluctuation and calculated the corresponding speedup gained by method1 for each processors situation comparing to the original Box-Mueller transformation.

Processors	**1**	**2**	**4**	**8**	**16**
No-noise	43	21.48	10.77	5.5	2.635
Noise	51.053	25.8	12.9	6.55	3.15
Noise,Method1	44.12	22.15	11	5.65	2.71
Speedup	7.19	6.69	9.22	7	6.87

Table20- Collision time comparison under different situation on ness (seconds)

In **Table20**, the noise mode increases 8.05, 4.48, 2.12, 1.05, and 0.52 seconds from non-noise section for the five processors situations respectively. By using method1, the noise mode just increases 1.12, 0.67, 0.23, 0.15, and 0.07 seconds from non-noise mode, which almost neglect the extra time cost by random number generation. It proves the method1 can get at least 6 times speedup than the original Box-Mueller one on NESS when parallelised.

An interesting thing is that with the number of processors increased, the benefit gained by method1 becomes smaller in real time, though it is reduced in proportion. For example, for one processor, from original to method1, the saved time is 51.53-44.12=7.41, and for 16 processors, it is 3.15-2.71=0.44.

8.5.2 Performance on HECTOR

For comparison, a series of records are obtained on HECTOR. Other conditions are the same to the comparison on NESS. The **Percentage** is how much proportion the collision part accounts for the total simulation runtime with the processors numbers increased.

500 cycles run on original program, with D_NOISE_ flag turning off. Use C compiler on HECTOR, with the O3 optimisation flag turning on.

Processors	**1**	**2**	**4**	**8**	**16**
Total	60.59 ± 0.05	32.70 ± 2.00	15.55 ± 0.15	7.67 ± 0.03	4.23 ± 0.13
Collision	52.30 ± 0.40	25.93 ± 0.01	12.94 ± 0.02	6.40 ± 0.01	3.18 ± 0.01
Percentage	86.3%	80%	83.2%	83.4%	75.1%
Speedup	1	2	4	8.2	16.5

Table21- Parallel executions of the original simulation on hector (seconds)

Identical to **Table21** setup, with D_NOISE_ flag turning on.

Processors	**1**	**2**	**4**	**8**	**16**
Total	70.00 ± 0.03	35.50 ± 0.70	17.82 ± 0.50	8.95 ± 0.10	4.82 ± 0.60
Collision	61.12 ± 0.15	30.62 ± 0.01	15.28 ± 0.01	7.56 ± 0.01	3.76 ± 0.01
Percentage	87.3%	86.2%	85.7%	85%	78%
Speedup	1	2	4	8.1	16.3

Table22- Parallel executions of the original simulation on hector with noise on (seconds)

Implement method1 with identical setup to **Table22**.

Processors	**1**	**2**	**4**	**8**	**16**
Total	61.86 ± 0.01	31.22 ± 0.01	15.75 ± 0.05	7.80 ± 0.02	4.20 ± 0.03
Collision	53.11 ± 0.01	26.54 ± 0.01	13.24 ± 0.01	6.55 ± 0.01	3.25 ± 0.01
Percentage	85.9%	85%	84%	84%	77.4%
Speedup	1	2	4	8.1	16.3

Table23-Parallel executions of simulation implement method1 on hector (seconds)

The speedup gained for collision part also seems linear. In addition, by comparing to the performance on NESS, the runtime and the collision proportions on HECTOR are constant, in which the fluctuation factor is less than 0.1. The reason may be that HECTOR allows multiple processors from different nodes. Thus no processors' interaction problems may affect the performance.

Similarly, for the collision part comparison, we obtained the three situations from the **Table21**, **Table22** and **Table23**, with the speedup gained by method1 calculated.

Processors	**1**	**2**	**4**	**8**	**16**
No-noise	52.30	25.93	12.94	6.40	3.18
Noise	61.12	30.62	15.28	7.56	3.76
Noise,Method1	53.22	26.54	13.24	6.54	3.25
Speedup	9.59	7.69	7.80	8.29	8.29

Table24- Collision time comparison under different situation on hector (seconds)

Table24 shows that the difference from no-noise to noise mode are 8.80, 4.69, 2.34, 1.16, 0.58 and the difference from no-noise to noise mode by using method1 are 0.92, 0.61, 0.30, 0.14, 0.07. The speedup gained by using method1 comparing to the Box-Mueller transformation is 9.59 when 1 processor used. And when more processors are used, the speedup is nearly 8 times. Thus, we could declare that the method1 will get around 8 times speedup on HECTOR when parallelised.

The patterns of the scalability on HECOTR are similar to statistical data on NESS and the total runtime on HECTOR is less than on NESS. However, the collision times on HECTOR are bigger than on NESS. A possible answer to it is the different compiler used, whereas nothing else compilers could be used for implement MPI.

Chapter 9: Conclusion

The final chapter summarises the works we have done and the specific implementation of each requirement.

9.1 Brief Summary

All of the basic requirements listed in Chapter3 have been implemented.

A good uniform random number generator is chosen which named KISS32 generator [8]. The KISS32 generator has passed all of the standard tests in the standard TestU01 library and performs the fastest among the three alternative RNGs when doing the BigCrush tests.

The Ladd's discrete random number [4] is correctly implemented into the whole "Fluctuating Hydrodynamics" simulation in two methods to replace the current expensive normal random number generator. Method 1 gained around 9 times speedup and method 2 gained around 4 times speedup than the original Box-Mueller transformation on both NESS and HECTOR computers.

The parallelised simulation that uses the method1 gets a reasonable speedup. The performance on NESS is fluctuated and the speedup gained by method 1 is varied between 6 and 9 times with the processors number changed. The performance on HECTOR is comparatively better than on NESS and the speedup gained by method 1 is stably 8 times.

A strange problem is occurred for methods 2 when implementing DRN technique [4]. We could not find the reason why the optimised way in model single fluid will perform even worse than the original expensive Box-Mueller transformation. We have checked the code and theory seriously whereas no obvious mistakes they are.

Another puzzle is occurred when doing the cumulant calculation [4] for the alternative correctness test in the DRN technique implementation. It requires to calculate the cumulants<x2(t)> and <x4(t)> per step t and the output can be illustrated by four graphs that show the correlation between the neighbour random numbers. However, by no ways we can find what the correct equations they are. It could only be an option for the correctness test of the Ladd's technique implementation in the future work.

9.2 Future Work

Some suggestions for the future optimisations of the simulation can be considered.

First, the current DRN is generated by 32bit integer, thus 10 DRNs will be generated for each integer and we have get around 6-9 times speedup. If we use a 64bit integer, then 21 DRNs will be generated for each integer and more speedup may gained if the benefit could conquer the increased loop times.

Second, the much faster Ziggurat method [10] (mentioned in Chapter 2) for normal random number generation comparing to the Box-Mueller one could be implemented in the simulation if the performance is good.

Appendix A: Post Mortem

Appendix A includes the original timetable, and risk analysis, combining with the actual progress of the project.

A.1 Timetable and Progress

The original work time plan for the project is shown in Table24.

Schedules	Deliverables	Tasks
2 weeks	Make the choice among uniform random number generators	Write. Find good uniform generators. Run tests on them. Check performance
2 weeks	Correct implementation of DRN Basic time records.	Write. Implement Ladd's discrete generator alone based on the uniform RNG, then test.
3 weeks	Basic properties records	Write unit tests for both uniform generator and DRN generator.
1 weeks	Preliminary delivery of dissertation	Write up works
2 weeks	Method for DRN implementation. Execution time, speedup and other performance records	Write. Implement DRN into the whole simulation and test. Performance checking on ness, hector
1 week	Parallel version of DRN. Execution time, speedup and other records.	Write. Parallelise the DRN and test. Performance checking on ness, hector
1 week	Final Code version for submission	Code test, comment, etc.
3 weeks	Final version of dissertation	Dissertation writing.
1 week	Any optimisation	Further optimisation if possible

Table25- Original work timetable

In the original timetable, the first step is to find a good uniform random number generator. However, in reality, the implementation of the Ladd's discrete random number technique is processed and finished firstly before the uniform random number generator finding process, for it is the main programming part in our extension progress.

The uniform random number generator finding process from enormous generators depends a lot on the information provided by the TestU01 paper [7], which also provides the standard tests for the quality and efficiency checking of uniform random number generators.

The original time given to write unit test is 3 weeks. Actually, it cost more than 4 weeks to write unit test for the correctness of both uniform generator and discrete random number technique. Thus, no spare week allowed for the further optimisation of the simulation.

The code implementation of parallelization is less than a week, and much time are spent on performance checking and recording on the two platforms.

A.2 Risk management

The original risk analysis is shown in **Table25.**

Risks	Impact	Mitigation
Technical difficulty	Medium	Allow flexibility in work plan
Problems with English writing	Medium	Get help from Native speaker
Poor back up of files	High	Backup and use CVS techniques
Lack of hardware resource	Medium	Ask for more or use other platforms
Personal problems	Low	Allow flexibility in work plan

Table26- Original risk analysis

The original risk analysis is not very comprehensive to some extent. Some other risks such as the ineffective of DRN technique should be considered, though it did not happen. Some of the risks do occurred and are solved soon.

The main board of my laptop is broken for 2 weeks before repaired. Fortunately, I could continue the work in university lab with the latest backup program.

Little problems occur in writing dissertation in English with my previous English dissertation experience. In addition, the basic grammars are checked with the help of native speaker.

We have enough budgets on both NESS and HECTOR to check the performance, and no other technique difficulties or personal problems occur.

Appendix B: Execution of Fluctuating Hydrodynamics simulation

Appendix B includes the two execution outputs of the whole “Fluctuating Hydrodynamics” simulation, the initialization output and a 50 cycles’ output.

Initialization output of the simulation execution:

```
Read 59 user parameters from input.ref
[User      ] Lattice size is (128, 128, 1)
[User      ] periodic boundaries set to (1, 1, 1)
[Default] Processor decomposition is (1, 1, 1)
Cartesian Communicator:
Periodic = (1, 1, 1)
Reorder is true
[Compute] local domain as (128, 128, 1)
[User      ] Random number seed: 8361235

External gravitational force
[User      ] gravity = 0 0 0

Model physics:
Shear viscosity: 0.100000
Relaxation time: 0.800000
Bulk viscosity : 0.100000
Relaxation time: 0.800000
Isothermal kT:      0.000021

Ghost modes
[Default] All modes (hydrodynamic and ghost) are active
[Default] Ghost mode relaxation time: 1.000000

The lattice Boltzmann model is D3Q19
Requesting 15412800 bytes for site data
Order parameter is via lattice Boltzmann
Setting phi I/O format to ASCII
Setting velocity I/O format to ASCII

Order parameter mobility M: 0.150000
[Boundary links: 0 (0 bytes)]
[User      ] Free energy: symmetric
```

Symmetric phi^4 free energy:
Bulk parameter A = -0.062500
Bulk parameter B = 0.062500
Surface penalty kappa = 0.040000
Surface tension = 0.047140
Interfacial width = 1.131371

TEST_statistics [total, mean, variance, min, max]
[rho][16384, 1, 0, 1, 1]

TEST_momentum [x, y, z]
[total][1.38778e-17, 0, 0]
[fluid][1.38778e-17, 0, 0]
[phi] -6.9646072e+00 -4.2508589e-04 8.2855946e-04 -4.9998083e-02 4.9997346e-02

50 cycles example output

Curvature statistics [t, lx, ly, lz, L1, L2, L3, alpha, beta]
50 12899.5 13411.5 inf 12895 13416.4 0 1.47583 0

TEST_statistics [total, mean, variance, min, max]
[rho][16384, 1, 0, 0.99989785, 1.0001374]

TEST_momentum [x, y, z]
[total][2.24543e-14, 1.32186e-15, 0]
[fluid][2.24543e-14, 1.32186e-15, 0]
Wall net: 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00
Velocity stats:
[umin][-7.02137e-05 -7.36595e-05 0]
[umax][6.51894e-05 6.71128e-05 1.17549e-38]
[phi] -6.9646072e+00 -4.2508589e-04 3.1065457e-04 -5.9404904e-02 6.2116536e-02
Completed cycle 50

Appendix C: Uniform Random Number Code

Appendix C include the C code of two uniform random number generator, the RAN2 [6] and CLCG4 [12].

RAN2 uniform random number generator, C code:

```
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)
double ran2(long *idum)
{
	int j;
	long k;
	static long idum2=123456789;
	static long iy=0;
	static long iv[NTAB];
	float temp;
	if (*idum <= 0) {
		if (-(*idum) < 1) *idum=1;
		else *idum = -(*idum);
		idum2=(*idum);
		for (j=NTAB+7;j>=0;j--) {
			k=(*idum)/IQ1;
			*idum=IA1*(*idum-k*IQ1)-k*IR1;
			if (*idum < 0) *idum += IM1;
			if (j < NTAB) iv[j] = *idum;
		}
		iy=iv[0];
```

```
    }
    k=(*idum)/IQ1;
    *idum=IA1*(*idum-k*IQ1)-k*IR1;
    if (*idum < 0) *idum += IM1;
    k=idum2/IQ2;
    idum2=IA2*(idum2-k*IQ2)-k*IR2;
    if (idum2 < 0) idum2 += IM2;
    j=iy/NDIV;
    iy=iv[j]-idum2;
    iv[j] = *idum;
    if (iy < 1) iy += IMM1;
    if ((temp=AM*iy) > RNMX) return RNMX;
    else return temp;
}
```

CLCG4 uniform random number generator, C code:

```
/*-------------------------------------------------------------------*/
/* clcg4.c      Implementation module                                */
/*-------------------------------------------------------------------*/
#include <stdio.h>
#include "clcg4.h"
/*********************************************************************
***/
/* Private part.                                                     */
/*********************************************************************
***/

#define H      32768                    /* = 2^15 : use in MultModM.  */

static long aw[4], avw[4],          /*      a[j]^{2^w} et a[j]^{2^{v+w}}.  */
                a[4] = { 45991, 207707, 138556, 49689 },
                m[4] = { 2147483647, 2147483543, 2147483423, 2147483323 };

static long Ig[4][Maxgen+1], Lg[4][Maxgen+1], Cg[4][Maxgen+1];
                            /* Initial seed, previous seed, and current seed. */

static short i, j;
```

```
static long MultModM (long s, long t, long M)
     /* Returns (s*t) MOD M.    Assumes that -M < s < M and -M < t < M.        */
     /* See L'Ecuyer and Cote (1991).[13]                                      */
   {
   long R, S0, S1, q, qh, rh, k;

   if (s < 0)    s += M;
   if (t < 0)    t += M;
   if (s < H)    { S0 = s;    R = 0; }
   else
      {
      S1 = s/H;    S0 = s - H*S1;
      qh = M/H;    rh = M - H*qh;
      if (S1 >= H)
         {
         S1 -= H;      k = t/qh;      R = H * (t - k*qh) - k*rh;
         while (R < 0)    R += M;
         }
      else R = 0;
      if (S1 != 0)
         {
         q = M/S1;      k = t/q;      R -= k * (M - S1*q);
         if (R > 0)    R -= M;
         R += S1*(t - k*q);
         while (R < 0)    R += M;
         }
      k = R/qh;      R = H * (R - k*qh) - k*rh;
      while (R < 0) R += M;
      }
   if (S0 != 0)
      {
      q = M/S0;      k = t/q;      R -= k* (M - S0*q);
      if (R > 0)    R -= M;
      R += S0 * (t - k*q);
      while (R < 0)    R += M;
      }
   return R;
   }

/*-----------------------------------------------------------------------*/
/* Public part.                                                          */
/*-----------------------------------------------------------------------*/

void SetSeed (Gen g, long s[4])
```

```
    {
    if (g > Maxgen) printf ("ERROR: SetSeed with g > Maxgen \n");
    for (j = 0; j < 4; j++)    Ig [j][g] = s [j];
    InitGenerator (g, InitialSeed);
    }

void WriteState (Gen g)
    {
    printf ("\n State of generator g = %u :", g);
    for (j = 0; j < 4; j++)
        {
        printf ("\n      Cg[%u] = %lu", j, Cg[j][g]);
        }
    printf ("\n");
    }

void GetState (Gen g, long s[4])
    {
    for (j = 0; j < 4; j++)    s [j] = Cg [j][g];
    }

void InitGenerator (Gen g, SeedType Where)
    {
    if (g > Maxgen) printf ("ERROR: InitGenerator with g > Maxgen \n");
    for (j = 0; j < 4; j++)
        {
        switch (Where)
            {
            case InitialSeed :
                Lg [j][g] = Ig [j][g];      break;
            case NewSeed :
                Lg [j][g] = MultModM (aw [j], Lg [j][g], m [j]);      break;
            case LastSeed :
                break;
            }
        Cg [j][g] = Lg [j][g];
        }
    }

void SetInitialSeed (long s[4])
    {
    Gen g;
    for (j = 0; j < 4; j++)    Ig [j][0] = s [j];
    InitGenerator (0, InitialSeed);
```

```
    for (g = 1; g <= Maxgen; g++)
       {
       for (j = 0; j < 4; j++)
          Ig [j][g] = MultModM (avw [j], Ig [j][g-1], m [j]);
       InitGenerator (g, InitialSeed);
       }
    }

void Init (long v, long w)
    {
    long sd[4] = {11111111, 22222222, 33333333, 44444444};
    for (j = 0; j < 4; j++)
       {
       aw [j] = a [j];
       for (i = 1; i <= w; i++)
          aw [j]    = MultModM (aw [j], aw [j], m[j]);
       avw [j] = aw [j];
       for (i = 1; i <= v; i++)
          avw [j] = MultModM (avw [j], avw [j], m[j]);
       }
    SetInitialSeed (sd);
    }

double GenVal (Gen g)
    {
    long k,s;
    double u;
    u = 0.0;
    if (g > Maxgen) printf ("ERROR: Genval with g > Maxgen \n");

    s = Cg [0][g];    k = s / 46693;
    s = 45991 * (s - k * 46693) - k * 25884;
    if (s < 0) s = s + 2147483647;    Cg [0][g] = s;
    u = u + 4.65661287524579692e-10 * s;

    s = Cg [1][g];    k = s / 10339;
    s = 207707 * (s - k * 10339) - k * 870;
    if (s < 0) s = s + 2147483543;    Cg [1][g] = s;
    u = u - 4.65661310075985993e-10 * s;
    if (u < 0) u = u + 1.0;

    s = Cg [2][g];    k = s / 15499;
    s = 138556 * (s - k * 15499) - k * 3979;
    if (s < 0) s = s + 2147483423;    Cg [2][g] = s;
```

```
    u = u + 4.65661336096842131e-10 * s;
    if (u >= 1.0) u = u - 1.0;

    s = Cg [3][g];    k = s / 43218;
    s = 49689 * (s - k * 43218) - k * 24121;
    if (s < 0) s = s + 2147483323;    Cg [3][g] = s;
    u = u - 4.65661357780891134e-10 * s;
    if (u < 0) u = u + 1.0;

    return (u);
    }

void InitDefault ()
    {
    Init (31, 41);
    }
```

Appendix D: TestU01 Results

Appendix D includes several TestU01 test [7] results for different generators. There are: One test output of Small Crush test, the passed result of RAN2, CLCG4, and KISS32 for Big Crush test and a failed result of LCG for Big Crush test.

The smarsa_BirthdaySpacings test output in Small Crush test.

```
========= smarsa_BirthdaySpacings =========
N =  1,  n = 5000000,  r =  0,  d = 1073741824,  t = 2,  p = 1

      Number of cells = d^t = 1152921504606846976
      Lambda = Poisson mean =  27.1051

---------------------------------------------------
Total expected number = N*Lambda  :  27.11
Total observed number  : 4987216
p-value of test  :  eps  *****

------------------------------------------------
CPU time used  :  00:00:01.60

Generator state:
 s = 912307078
```

The BigCrush test outputs for KISS32, RAN2 and CLCG4 uniform random number generators.

========= Summary results of BigCrush =========
Version: TestU01 1.2.3

Generator: KISS32
Number of statistics: 160
Total CPU time: 06:15:31.34
All tests were passed

========= Summary results of BigCrush =========

Version: TestU01 1.2.3
Generator: Ran2
Number of statistics: 160
Total CPU time: 07:48:28.97

All tests were passed

========= Summary results of BigCrush =========

Version: TestU01 1.2.3
Generator: CLCG4
Number of statistics: 160
Total CPU time: 09:06:25.44

All tests were passed

The BigCrush test output for LCG uniform random number generators.

```
========= Summary results of BigCrush =========

Version:          TestU01 1.2.3

Generator:        LCG
Number of statistics:  160
Total CPU time:   05:57:21.68

The following tests gave p-values outside [0.001, 0.9990]:
(eps  means a value < 1.0e-300):
(eps1 means a value < 1.0e-15):
       Test                          p-value
----------------------------------------------
68  MatrixRank, L=1000, r=0            eps
69  MatrixRank, L=1000, r=26           eps
70  MatrixRank, L=5000                  eps
71  MatrixRank, L=5000                  eps
80  LinearComp, r = 0                1 - eps1
81  LinearComp, r = 29               1 - eps1
----------------------------------------------
All other tests were passed
```

Appendix E: Single Implementation test of DRN

Appendix E includes a simple correctness test for the single Ladd's discrete random number technique [4] implementation. We have run the discrete random number generator for 100 times to check whether the 8 state values are correct, and whether the probabilities of generating each value are correct. The 100 number outputs are shown below.

-0.765367 0.000000 1.847759 0.765367 -1.847759 0.000000
-0.765367 0.000000 -1.847759 1.847759 -0.765367 1.847759
0.765367 0.000000 0.000000 0.765367 1.847759 0.765367
0.765367 0.000000 0.000000 0.000000 -1.847759 0.765367
0.000000 0.000000 0.000000 -0.765367 1.847759 0.000000
0.765367 -1.847759 -0.765367 0.765367 -0.765367 1.847759
-1.847759 -1.847759 1.847759 -1.847759 0.000000 0.765367
0.000000 -1.847759 -1.847759 0.000000 1.847759 -1.847759
0.765367 -1.847759 -1.847759 0.000000 -1.847759 1.847759
-1.847759 -1.847759 0.000000 -0.765367 -1.847759 1.847759
0.000000 -0.765367 0.765367 1.847759 -1.847759 0.000000
0.000000 0.000000 -0.765367 -1.847759 0.000000 0.000000

References

[1] R. Adhikari, K. Stratford, M.E. Cates, A.J. Wagner, Europhys. Lett. 3 (2005) 473.

[2] S. Succi, The Lattice Boltzmann Equation, Clarendon Press. Oxford (2001)

[3] J.C Desplat, 1. Pagonabarraga, and P. Bladon. Comp. Phys. Comm. *134* 273-290 (2001)

[4] A.J.C. Ladd, Computer Physics Communications, 180, 2140-2142 (2009)

[5] M.P.ALLEN AND D.J. TILDESLEY. *Computer Simulation of Liquids*, published in the united states by oxford university Press, New York (1987)

[6] William H.Press, Saul A. Teukolsky, William T. Vetterling, Brian P.Flannery, Numerical Recipes in C, the art of scientific computing, second edition. 275-290 (1999)

[7] TestU01:A C Library for Empirical Testing of Random Number Generators http://www.iro.umontreal.ca/~lecuyer/myftp/papers/TestU01.pdf

[8] David Jones, UCL Bioinformatics Unit, Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications. (unpublished)

[9] Gavin.J.Gibson. Bayesian Inference and Computational Methods (Lecture Notes). Heriot-Watt University (2009)

[10] George Marsaglia; Wai Wan Tsang."The Ziggurat Method for Generating Random Variables". Journal of Statistical Software **5** (8) Retrieved 2007-06-20. (2000)

[11] L'Ecuyer, ACM Transactions on Modeling and Computer Simulation, 87-98, (1993)

[12] L’ECUYER, P. AND ANDRES, T. H. A random number generator based on the combination of four LCGs. Mathem. Comput. Simul. 44, 99–107. (1997)

[13] P. L'Ecuyer and S.Côté, “Implementing A Random Number Package with Splitting Facilities”, ACM Transactions on Mathematical Software, 17 (1991), 98—111