University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

Computer Science and Engineering: Theses, Dissertations, and Student Research | Computer Science and Engineering, Department of

Summer 7-25-2011

MOLECULAR DYNAMICS SIMULATION BASED ON HADOOP MAPREDUCE

Chen He
University of Nebraska-Lincoln, che@cse.unl.edu

Follow this and additional works at: http://digitalcommons.unl.edu/computerscidiss

Part of the Computer Engineering Commons, and the Computer Sciences Commons

He, Chen, "MOLECULAR DYNAMICS SIMULATION BASED ON HADOOP MAPREDUCE" (2011). *Computer Science and Engineering: Theses, Dissertations, and Student Research*. 28.
http://digitalcommons.unl.edu/computerscidiss/28

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Computer Science and Engineering: Theses, Dissertations, and Student Research by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

MOLECULAR DYNAMICS SIMULATION BASED ON HADOOP MAPREDUCE

by

Chen He

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor David Swanson and Professor Ying Lu

Lincoln, Nebraska

May, 2011

MOLECULAR DYNAMICS SIMULATION BASED ON HADOOP MAPREDUCE

Chen He, M.S.

University of Nebraska, 2011

Adviser: David Swanson and Ying Lu

Molecular Dynamics (MD) simulation is a computationally intensive application used in multiple fields. It can exploit a distributed environment due to inherent computational parallelism. However, most of the existing implementations focus on performance enhancement. They may not provide fault-tolerance for every time-step.

MapReduce is a framework first proposed by Google for processing huge amounts of data in a distributed environment. The simplicity of the programming model and fault-tolerance for node failure during run-time make it very popular not only for commercial applications but also in scientific computing.

In this thesis, we develop a novel communication-free and each time-step fault-tolerant solution for MD simulation based on Hadoop MapReduce (MDMR). Through emulation of Hadoop MapReduce and introduction of a run-time program monitor, we can predict the execution time of a given size MD simulation system. We also demonstrate the performance and energy consumption improvement from implementing MDMR in a hybrid MapReduce environment with GPU hardware (MDMR-G).

To evaluate MDMR, we construct a 32 node MapReduce cluster and a run-time MapReduce program monitor. We emulate MDMR and propose a prediction formula of MDMR execution time for Map and Reduce stages. The emulation results demonstrate our formula can predict MDMR execution time within 9.1% variance. Our run-time monitor shows that MDMR can obtain high computational power efficiency for large MD simulation systems. We also build a hybrid MapReduce cluster with GPGPU. MDMR in this environment obtains 20 times speedup and reduces energy consumption 95% compared with the same size cluster without GPU accelerators.

ACKNOWLEDGMENTS

I am very grateful to my advisor Dr. David Swanson for leading me to the supercomputing research area and support not only from financial aspect, but also from life, heart and courage. He is like elder brother. He can always help me to find a reasonable and satisfied answer no matter what kinds of questions in research and life. His encouragements and supervision have made this work possible.

I want to thank my advisor Dr. Ying Lu. Her patient, careful and earnest attitude on research work establishes a role model for me.

I still want to say thank you for all administrators from Holland Computing Center, they are accommodating and I cannot build cluster for research without their kindly help. I am very thankful to my colleagues, Derek Weitzel and Brian Bockelman. Without Derek's help, I may not adapt life in Lincoln quickly. Brian's rational mind and strong mathematical background deeply affect me. You guys give me inspiration, happiness, and helps in need. I like to work with you.

Last but not least, I owe my parents, my wife and daughter all my life for their love, trust, and encouragement. This thesis is dedicated to them.

Contents

List of Figures vi

List of Tables viii

1 Introduction 1

2 Background and Related Work 5

2.1 Hadoop MapReduce 5

2.2 Hadoop Distributed File System 7

2.2.1 HDFS Architecture 8

2.2.2 Data Replication 9

2.3 Molecular Dynamics Simulation 10

2.3.1 Computational Aspects of MD Simulations 10

2.3.2 Potential Functions 11

2.3.3 Boundary Conditions 14

2.3.4 Minimum Image Convention 16

2.3.5 Integration Algorithm 16

2.3.5.1 Verlet Intergration 17

2.3.5.2 Velocity Verlet Intergration 17

2.4 CUDA 18

2.4.1 CPU+GPU structure 19

2.5 Related Works 18

3 Design and Implementation 22

3.1 MD Simulation based on MapReduce 22

3.1.1 Atom decomposition method 24

3.2 Tuning of MDMR 28

3.2.1 Hadoop Parameters 28

3.2.2 Other factors 30

3.3 Evaluation of MDMR ... 31
3.3.1 Speedup ... 33
3.3.2 Karp-Flatt metric ... 33
3.3.3 Minimum MapReduce Overhead ... 33
3.3.4 Time Complexity ... 33
3.4 Run-time Monitor for MDMR ... 36
3.4.1 Class specification ... 37
3.4.2 Buffered Negotiator ... 38
3.4.3 Monitor Metrics ... 41
3.5 MDMR in hybrid environment ... 42
3.5.1 Hybrid Hadoop ... 42
3.5.2 MDMR-G algorithm and time complexity ... 43
4 Evaluation 46
4.1 Hadoop Parameter tuning for MDMR ... 46
4.1.1 Reducer number ... 46
4.1.2 Replication ... 49
4.2 MDMR Evaluation ... 51
4.2.1 MDMR Speedup ... 52
4.2.2 MDMR Karp-Flatt Metric ... 50
4.2.3 Minimum MapReduce Overhead ... 52
4.2.4 Preduction ... 54
4.3 Run-time Program Monitor ... 61
4.4 MDMR Speedup ... 64
4.5 MDMR-G performance on Hybrid MapReduce Cluster ... 64
5 Conclusion ... 71
6 Future Work ... 73
Bibliography ... 74

List of Figures

2.1 MapReduce Framework 5

2.2 HDFS Structure 8

2.3 The Lennard-Jones Potential 11

2.4 Symmetric Electron Cloud Structure 11

2.5 Polarization of Electron Clouds 12

2.6 Periodic Images of a Central Simulation Box 13

2.7 Periodic Boundary Conditions during Atoms Crossing Over 14

2.8 Periodic Boundary Condition Code 15

2.9 Minimum Image Convention Interactions 19

2.10 CPU+GPU Architecture 19

3.1 Simulation Coordinate file 21

3.2 Hadoop Timeline on cluster 29

3.3 Run-time MapReduce program monitor data flow 33

3.4 Statistical CPU time Framework 35

3.5 Hybrid Hadoop MapReduce structure 38

4.1 Effect of Reducer Number 43

4.2 Effect of Replication number 44

4.3 MDMR Speedup 50

4.4 MDMR Speedup compared with Serial MD 51

4.5 MDMR Karp-Flatt Metric 52

4.6 Prediction vs. Actual Time 55

4.7 Prediction vs. Actual Time-2 56

4.8 Prediction vs. Actual Time-3 57

4.9 Computation Power Efficiency 62

4.10 MDMR-G Energy and Power consumption 65

4.11 MDMR-G Execution time 66

4.12 MDMR-G Speedup 67

4.13 MDMR-G Karp-Flatt Metric 67

List of Tables

2.1 LJ Reduced Units for Argon 12

3.1 Serial MD algorithm 22

3.2 MDMR algorithm 23

3.3 Java API of Mapper Interface 32

3.4 NFS vs. Local File System Execution Time 34

3.5 MDMR-G algorithm 39

4.1 Emulation Data 46

4.2 Max Variance between Prediction and Actual Time 48

4.3 Minimum MapReduce Overhead 56

4.4 Estimation of the Map stage 61

4.5 Monitor Overhead of MD simulation 62

4.6 MD simulation Efficiency 63

4.7 MDMR-G results 65

Chapter 1

Introduction

MapReduce [1] is a framework for processing huge amounts of data on distributable problems employing large numbers of computers. The simplicity of its programming model and its fault-tolerance attracts not only commercial companies but also scientists to apply MapReduce to multiple applications.

MapReduce is inspired by the *map* and *reduce* functions from functional programming. The MapReduce programming model is a data-centric model which moves the computation to data. This is different from classical distributed methods that focus on available computation resources. MapReduce has already been used in scientific computation for data-intensive applications like web page crawling, documents processing, log analysis, and so on. In this thesis, we will focus on designing and implementing Molecular Dynamics simulation [15], which is a kind of computation-intensive application, based on Hadoop MapReduce [17].

MD simulation is using computers to simulate the physical movements of atoms and molecules based on statistical mechanics. It is a kind of computation-intensive application that can be parallelized in distributed environments. Dr. Sumanth [27] has parallelized MD simulation based on Condor [37] in computing Grids [38]. However, compared with a MapReduce cluster, a computing Grid has its limitations. These limitations, to some extent, restrict MD simulation's reliability, security, and scalability.

For reliability, in the computing Grid environment used by Dr. Sumanth, computation resources may not be guaranteed. Any computing node can leave the computing Grid at any time. This problem will result in uncertainty in the execution

environment, because a computing node leaving will cause a failure of program execution. This is an inevitable property of opportunistic computing Grids. Programmers have to handle node failure by themselves in certain scenarios (eg. programs running on opportunistic computing Grid nodes have close dependency). Furthermore, there is no global file system support, and the programmer also needs to maintain a non-standard middleware to start Condor first before doing the computation. At the same time, this middleware also needs the authorization of computing nodes to create connections in the Grid. However, the MapReduce framework overcomes these problems. MapReduce has guaranteed worker nodes in a relatively closed environment (MapReduce cluster). Nodes contribute their resources barring nodes' failure. The MapReduce framework handles node failure by running replicated work on the fastest node. With the global distributed file system between nodes based on dedicated networks, the data transfer overhead can be reduced. We can expect that the MD simulation based on the MapReduce framework will be more reliable compared with opportunistic computing Grids.

For security, the opportunistic computing Grid does not have highest priority to control the computing nodes in the Grid. The computing nodes owners can monitor, interrupt, or even hack the running programs of Grid jobs. For example, if a MD simulation program is doing a highly confidentional simulation, the computing Grid cannot guarantee this program will not run on potential enemies' computers in the Grid. However, we can create a MapReduce cluster in a relatively closed environment to satisfy different security levels.

Finally, Administrative concerns, Hadoop clusters are straightforward to create and maintain. However, the Grid-based MD simulation needs middleware to initialize the computation environment. This middleware needs authorization from computing nodes. If these nodes refuse to let the middleware connect to them, the MD simulation cannot scale to these nodes. To administer those nodes, the Grid scheduler has limited priority unless the owners of computing nodes agree to follow the Grid scheduler's

administration. In a MapReduce cluster, we can easily add new nodes to a cluster not only in one data center but also across different data centers (probably not optimal but it is possible). Most importantly, Hadoop is a high profile open source project supported by an international community, while Dr. Sumanth's grid framework remains an in-house, custom effort. In this sense, MD simulation on a MapReduce cluster is easier to maintain and scale than on a computing grid.

In this thesis, we develop MDMR which parallelizes MD simulation on a MapReduce cluster. It is a communication-free and every time-step fault-tolerant implementation by employing MapReduce properties [16]. We present formulas to estimate the execution time of a given MD simulation system through MDMR. Furthermore, we create a run-time monitor which can watch the execution of MapReduce programs. This monitor can help programmers find bottlenecks in their MapReduce programs. Finally, we create MDMR-G that extends MDMR to utilize GPGPU on a MapReduce cluster. The MDMR-G obtained 20 times speedup compared to MDMR for the same MD simulation system; at the same time, we reduced energy consumption 95% compared with the same size cluster without GPU accelerators.

The rest of this thesis is organized as follows. In Chapter 2, we provide background knowledge about Hadoop MapReduce framework [17], HDFS [18], MD simulation, and CUDA [19]. In Chapter 3, we present the MDMR algorithm and its time complexity. Then we describe how we configured six main Hadoop MapReduce parameters that are closely related to MapReduce program performance. In the following section of Chapter 3, we demonstrate the MapReduce program run-time monitor mechanism and the MDMR-G algorithm. In Chapter 4, we evaluate MDMR based on twelve simulation systems containing from 1000 to 64000 atoms, and we give the coefficients of the MDMR execution time prediction formula. Furthermore, for the run-time monitor, we evaluate its overhead and estimate the MDMR computation power overhead of three MD systems. At the end of this Chapter, we show the MDMR-G evaluation based on five MD systems on a smaller hybrid MapReduce cluster with

GPGPUs embedded. In Chapter 5 and Chapter 6, we conclude with the contribution of this thesis and propose future work.

Chapter 2

Background and Related Work

MapReduce has become a standard open source parallel platform not only for the commercial Cloud but also for scientific computing. In this chapter, we first describe the Hadoop MapReduce and HDFS framework that are the platforms for our MDMR. Then we describe the background for MD simulation. Finally, our MDMR-G (MDMR with GPU accelerator) is using CUDA (Compute Unified Device Architecture) which is a parallel programming architecture based on GPU. CUDA is presented at the end of this chapter.

2.1 Hadoop MapReduce

Hadoop MapReduce is an open source project mainly supported by Yahoo! and Apache. Hadoop is a widely used cloud computing platform which contains eight subprojects including HDFS, MapReduce, HBase, Pig, ZooKeeper, Chukwa, Hive, and Common[17].

The Hadoop MapReduce structure is illustrated in Figure 2.1:

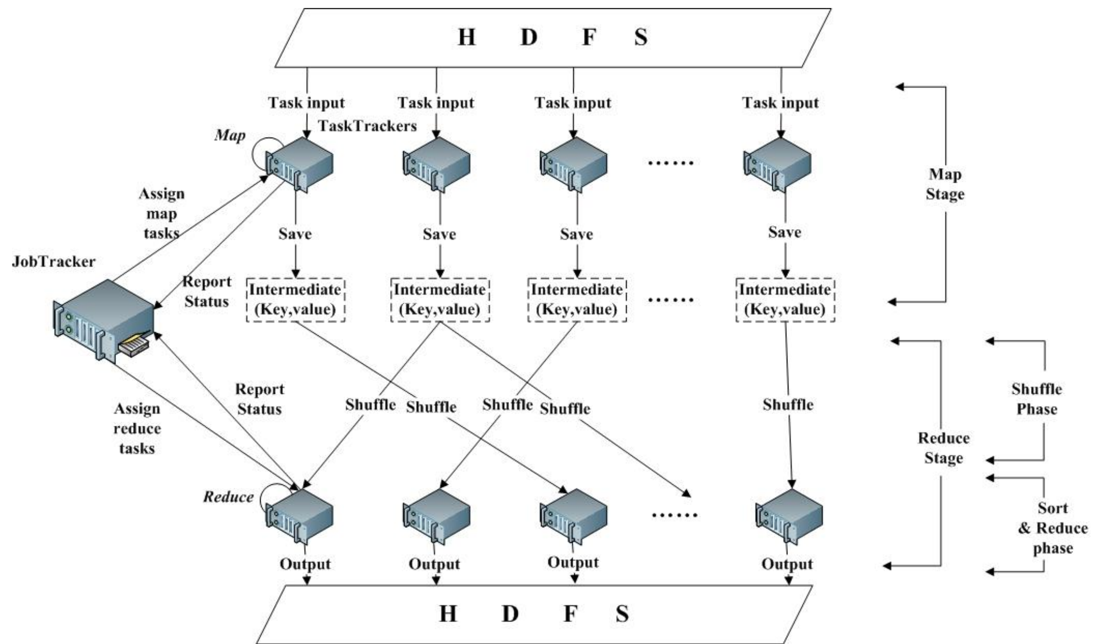


Figure 2.1: MapReduce Framework

Hadoop works as follow:

1) At the beginning, the user submits a job to the JobTracker which acts as a master. The JobTracker will divide the job's input data on HDFS into tasks when it obtains the job. After this process, the JobTracker will let the scheduler handle this submitted job. The scheduler will deploy this job into a corresponding queue (or pool if using fair sharing scheduler) according to its submission time, priority, user group or other schedulable parameters. This submitted job's tasks will be assigned to TaskTrackers when they give a heatbeat to the JobTracker if those tasks satisfy the scheduler's policy.

2) Map stage. A job enters the Map-Stage when its first map task has been assigned to a TaskTracker which will issue a new JVM to run this map task. The newly generated JVM process will read the input data in a key/value manner from HDFS and employ the *map()* function which is defined in the *Mapper* class. The *map()* function, is defined by programmers, will take the key/value pairs and produce intermediate key/value pairs which are inputs for the Reduce stage. After the *map()* function is accomplished, the intermediate key-value pairs will first be stored in TaskTracker's local memory or local

disk if memory is not enough. A system administrator can configure the size of memory that can be used to store the intermediate results considering the hardware specification and the load of clusters. In the Map stage, there is not communication between TaskTrackers; each TaskTracker does not necessarily know the existence of other TaskTrackers. Thus, there is no communication and synchronization overhead in the Map stage.

3) Reduce stage. The Reduce stage will start once the first group of map tasks finishes (the programmer can also configure the number of finished map tasks before the job will enter the Reduce-Stage). The MapReduce framework will generate one reduce task for each key of the Map stage's intermediate results by default. If necessary, the programmer can configure the reduce task number to get best performance. The reduce task is a child JVM propagated by TaskTracker. It has three phases: Shuffle, Sort and Reduce. The Shuffle phase will retrieve all the map tasks' outputs with the same key from each mapper. The Sort phase starts at the end of the Shuffle phase. It sorts the key/value pairs according to their value and send the sorted key/value pairs to the Reduce phase. At last, the user defined *reduce()* function (if not defined, the framework will run the default reduce function) will process those key/value pairs and output results to HDFS.

The MapReduce framework guarantees fault-tolerance through re-execution. In the Map stage or the Reduce stage, a failed task will be re-executed by the first available TaskTracker.

2.2 Hadoop Distributed File System

We will introduce HDFS in this section, because it is the carrier of MapReduce jobs input and output data. It is an inevitable component of the MapReduce framework.

Hadoop Distributed File System (HDFS) is designed as a highly fault-tolerant, high throughput, and high capacity distributed file system. It is ideal for storing terabytes or even petabytes of data on clusters that may be comprised of non-commodity hardware

like personal computers. The significant differences between HDFS and other distributed file systems are HDFS's write-once-read-many and streaming access models that make HDFS efficient in distributing and processing data, reliably storing and scaling large amounts of data, robustly in heterogeneous hardware and operating system environments.

2.2.1 HDFS Architecture

HDFS follows the master/slave architecture. The master node in a HDFS cluster is called the Namenode which manages the file system namespace and regulates client accesses to files. There are a number of slave nodes, called Datanodes, which store actual data in units of blocks.

The Namenode maintains a mapping table which maps data blocks to Datanodes in order to process write and read requests from HDFS clients; at the same time, the Namenode is also in charge of file system namespace operations like closing, renaming, and opening files and directories.

The Datanode stores the blocks of files in its local disk and executes the instructions like replace, create, delete, and replicate from the Namenode. Figure 2 (adopted from Apache Hadoop Project [17]) illustrates the HDFS architecture.

A Datanode periodically reports its status through a heartbeat and asks the Namenode for instructions. Every Datanode maintains an open server socket so that other Datanodes can request read and write operations; at the same time, clients access actual data on the Datanode through this channel. The heartbeat can also help the Namenode to detect connectivity with its Datanode and then replicates the blocks on a dead Datanode. In order to keep the contents of the Namenode in case of unavoidable failures, HDFS allows a secondary Namenode to periodically save a copy of data of the Namenode.

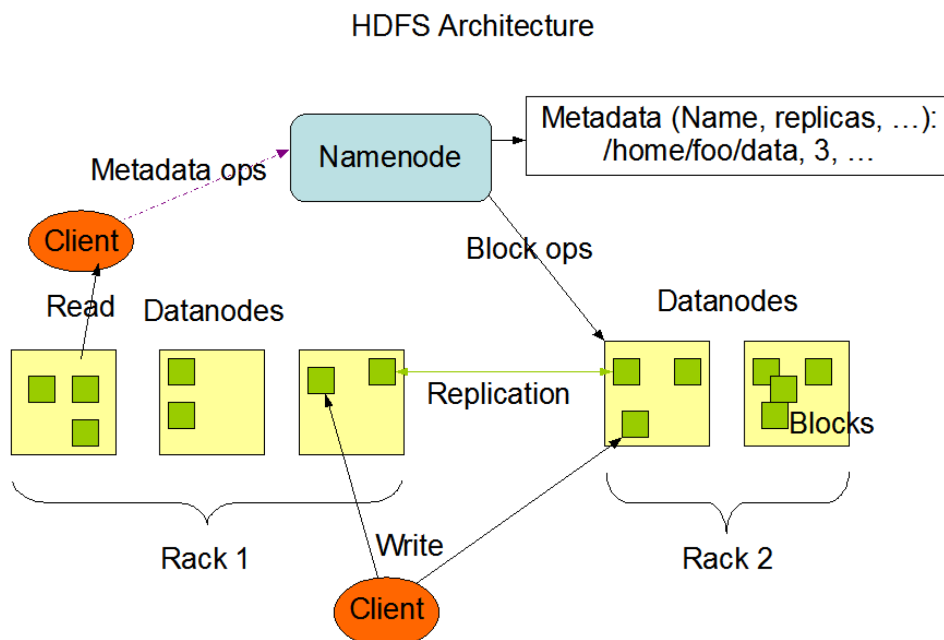


Figure 2.2: HDFS Structure[17]

2.2.2 Data Replication

HDFS can be deployed on a cluster composed of thousands of nodes. The probability of failure becomes non-negligible. This means HDFS has to handle the scenario in which some components are non-functional.

Data redundancy is a way to solve this problem. HDFS employs an intelligent replication placement policy to guarantee reliability and performance. For example, the default replication number of HDFS is 3 and HDFS will place the first replica block in a certain node. The second replica will be placed in a node that is located in the same rack of nodes where the first replica is located. Because nodes within a rack tend to connect to the same switch, the last replica will be placed in another rack to guarantee data availability even in the event that an entire rack is down. This is called rack-awareness.

2.3 Molecular Dynamics Simulation

2.3.1 Computational Aspects of MD Simulations

An MD simulation [21] performs the time integration of the differential equation 2.1 with given initial atom position and velocities. It is based on Newton's 2nd Law ($\vec{F} = m\vec{a}$). Assume we have position $\vec{p}$ and velocity $\vec{v}$ vectors $\{\vec{p}_i(s), \vec{v}_i(s) \mid i = 1,2,...,N\}$ before starting the simulation; we want to obtain velocities and positions $\{\vec{p}_i(s+\Delta t), \vec{v}_i(s+\Delta t) \mid i = 1,2,...,N\}$ in a later time.

$$\frac{\partial^2 \vec{p}_k(t)}{\partial t^2} = \vec{a}_k(t) = \sum_{i<j} \vec{p}_{ij}(t)(-\frac{1}{p}\frac{\partial u(r)}{\partial r})|_{p=p_{ij}(t)} \cdot (\delta_{ik} - \delta_{jk}) \qquad (2.1)$$

Where,

$$\delta_{ik} = \begin{cases} 1, i = k \\ 0, i \neq k \end{cases} \qquad (2.2)$$

is the Kronecker delta function and $u(r)$ is the potential function.

Forces posed on two atoms can be computed as the negative gradient of the potential in three dimensions because we simulate our system in three dimensional space.

$$F_k = -\frac{\partial V(\vec{p}^N)}{\partial \vec{p}_k} = -(\frac{\partial V}{\partial x_k}, \frac{\partial V}{\partial y_k}, \frac{\partial V}{\partial z_k}) \qquad (2.3)$$

In which $V(\vec{p}_k) = \sum_{i<j} u(\vec{p}_{ij})$ and $\vec{p}_{ij} = \vec{p}_i - \vec{p}_j$. Compared with ab initio electronic structure calculations [22] which need to solve the Schrodinger's equation at each time-step, the Classical MD simulation is less computationally intensive.

2.3.2 Potential Functions

Potential functions can be categorized as two-body potential functions if we only consider any two atoms' interaction. Similarly, the three-body potential functions will take every triplet of atoms' interaction into account. The potential energy *P(r)* is defined as the energy required moving two atoms from infinite separation to a distance *r* apart. In this thesis, we only consider two-body interactions in our simulation system.

The two body potential that we employed is the Lennard-Jones (LJ) potential (Figure 2.3). It is commonly accepted to model liquids such as argon and neon. This potential is mildly attractive when two atoms are far apart, while it becomes stronger when the two atoms are close together. We list the LJ potential and its force equations below.

$$V(p) = 4\varepsilon\left[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}\right] \tag{2.4}$$

$$F_k = 24\varepsilon[2(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}]\cdot(\frac{1}{r}) \tag{2.5}$$

The parameter ε is defined as the depth of the energy well and σ is determined by the atom's diameter. The potential energy becomes zero if the separation distance between atoms equals σ. The ε describes the strength of the interaction, freezing point and many other properties. The σ effects the structure of the material's solid state. From equation 2.4 and 2.5, we can clearly see two terms: one is $(\frac{\sigma}{r})^{12}$ which represents the short-range repulsion, the other is $(\frac{\sigma}{r})^{6}$ that models the long-range attraction. The attraction is caused by the polarization of the electron cloud of atoms. For example, if atoms are placed close to each other, the charge density fluctuations in one of the atom's electron cloud may induce the other atom's electron cloud polarization. For the non-polar neutral atoms like Nobel gases that have symmetric electron cloud structure, the attractive term is obtained from the exact quantum-mechanical solution. The

repulsion is formulated as the square of the attractive part. This makes the computation simple.

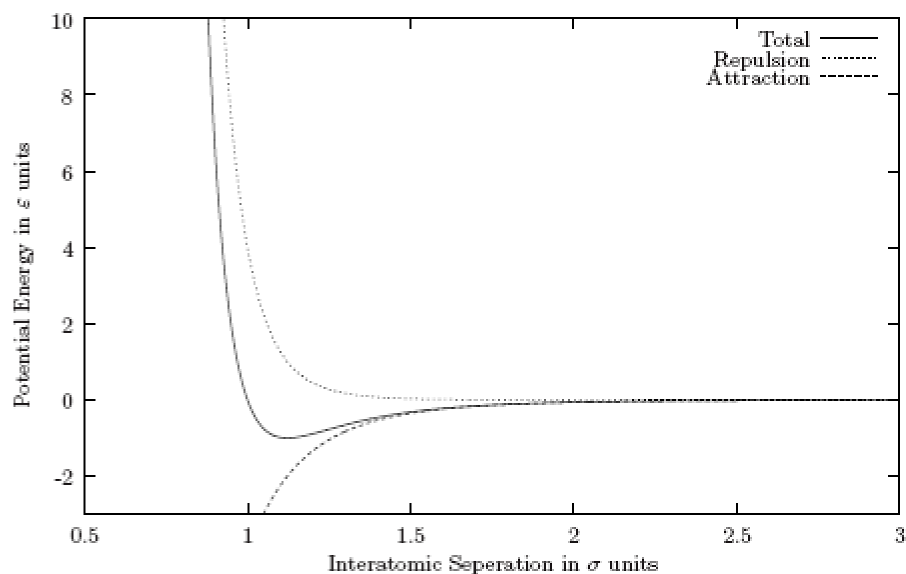


Figure 2.3: The Lennard-Jones Potential

If an atom is uncharged, its electron cloud has a unique spherical and symmetrical structure. Because of this structure, there is no charge concentrated in any particular direction. The dipole moment does not exist. An atom's electron cloud can still keep symmetrical structure, if two atoms are far enough apart. This scenario is illustrated in Figure 2.4.

Figure 2.4: Symmetric Electron Cloud Structure

This symmetrical structure may change if the atoms distance becomes closer and closer. In liquids, atoms move constantly and may collide with each other. The electron clouds of atoms lose their symmetric structure and acquire an induced dipole moment

which lasts for a very short period of time. During this time, the atoms electron clouds may exist like Figure 2.5 because the oppositely charged electron clouds result in two atoms that attract each other.

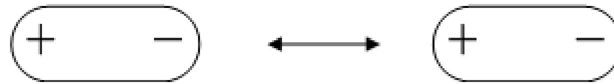


Figure 2.5: Polarization of Electron Clouds

We list the LJ reduced units for Argon in Table X. It will make the MD computation simple if we use the normalized units. For example, the unit σ can be used to normalize inter-atomic separation and ε to normalize the energy. It is very common to use $\sigma = 1$ and $\varepsilon = 1$ to simplify the computation.

It is not possible for the LJ model to model all kinds of scenarios like chemical reactions. However, it is still an important potential even with these drawbacks. The LJ model occurred in multifarious simulations where these researchers are focusing on fundamental issues rather than properties of specific materials. In this thesis, we still model our potential through the LJ method as Sumanth did in [27].

Table 2.1: LJ Reduced Units for Argon

Length	σ	=	$3.4\times10^{-10}m$
energy	ε	=	$1.65\times10^{-21}J$
mass	m	=	$6.69\times10^{-26}Kg$
time	$\sigma\sqrt{\frac{m}{\varepsilon}}$	=	$2.17\times10^{-12}s$
velocity	$\sqrt{\frac{\varepsilon}{m}}$	=	$1.57\times10^{2}ms^{-1}$
force	$\sqrt{\frac{\varepsilon}{\sigma}}$	=	$4.85\times10^{-12}N$
pressure	$\sqrt{\frac{\varepsilon}{\sigma^2}}$	=	$1.43\times10^{-2}Nm^{-2}$
temperature	$\frac{\varepsilon}{K}$	=	$120K$
Boltzmann constant	K	=	$1.38\times10^{-23}JK^{-1}$

2.3.3 Boundary Conditions

It is common to use periodic boundary condition to minimize a bulk material. The periodic box replicates the simulation box in all directions [23] through creating an infinite lattice. It is reasonable to decide how to deal with the situation that an atom reaches a boundary, because atoms lying at the same surface are very common. For instance, a 2744 atoms system can be arranged in a 14x14x14 cubic structure; there are 624 atoms at the outside surface of the cube.

Each atom from the simulation box has a periodic image in all other boxes. In order to explain the periodic box mechanism clearly, we employ a 2-dimensional version in Figure 2.6.

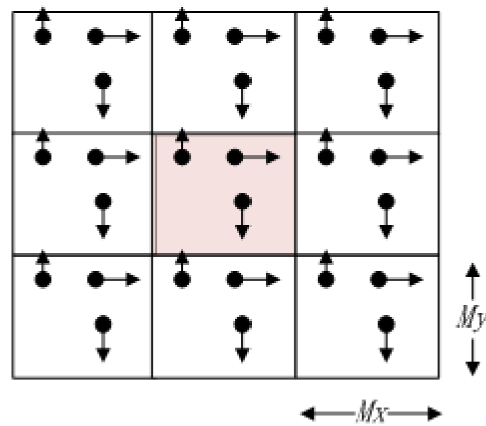


Figure 2.6: Periodic Images of a Central Simulation Box

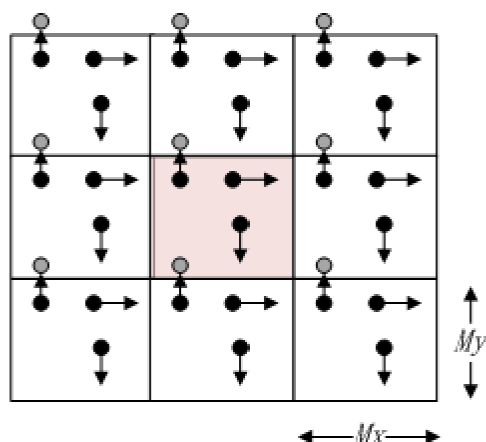


Figure 2.7: Periodic Boundary Conditions during Atoms Crossing Over

The shadow border box represents the simulation box with at least 6σ for a LJ potential. The box size should be carefully chosen for particular potential function. Since the wavelengths of the fluctuations are macroscopic, long range wavelength M (it is the simulation box length on one size) or greater can be suppressed. Then we cannot simulate a liquid near the gas-liquid critical point. However, the periodic boundary is very common and accurate if the simulation is not about liquid phase transitions (but rather equilibrium thermodynamic properties). In the periodic box, the atom number is constant, because when one atom leaves the box, another will enter from the opposite wall. Figure 2.6 and 2.7 demonstrate this process. Figure 2.7 gives some illustration on how to implement the periodic box. *Mx* is the length of simulation box in x direction; x is the x-coordinate of certain atom. These five lines code in Figure 2.8 can guarantee x is always in the range $(\frac{-M_x}{2}, \frac{M_x}{2})$.

```
1: if x > Mx/2 then
2:     x = x - Mx
3: else if x < Mx/2 then
4:     x = x + Mx
5: end if
```

Figure 2.8: Periodic Boundary Condition Code [27]

2.3.4 Minimum Image Convention

There are two constraints we need to keep if we use the periodic boundary conditions. One is the cut-off range of the potential must be no greater than the half of the simulation box length (or width, because they equals each other). The other is that atoms in the simulation box also need to interact with the periodic images of all other atoms. The first constraint guarantees that each atom interacts only with the nearest images of other atoms. It is known as the minimum image convention [22].

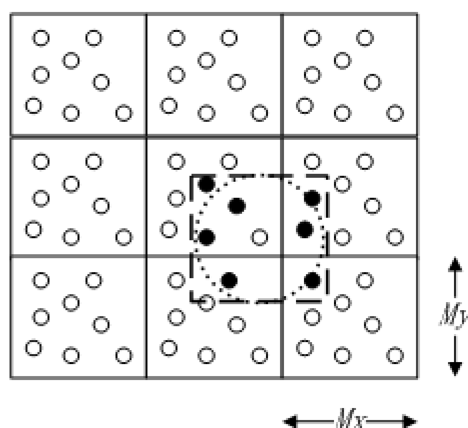


Figure 2.9: Minimum Image Convention Interactions

The white atom in the center dot circle (cut-off radius cycle) is the atom *i*.

2.3.5 Integration Algorithm

Assume the MD simulation starts at time-step *t*. What the integration algorithm does is to obtain the position of all N atoms at $t + \delta t$, where the δt is the time step length. There is not an analytical solution because of the complexity of the equations being integrated. We have to numerically solve the integration. A good integration algorithm should have five properties: computational efficiency, near optimal energy conservation, low hardware requirement, easy implementation, and accuracy of following classical trajectories [27].

The time-step δt is the crucial factor for all integration algorithms. If it is too large, the total energy is hard to keep conserved and the simulation result loses its accuracy. On the other hand, the simulation may take extraordinary long time if it is too small. We will introduce two famous integration algorithms.

2.3.5.1 Verlet Intergration

This algorithm was first created by L.Verlet[24]. The position after one time-step δt may be found from equation 2.6:

$$p(t+\delta t) = p(t) + v(t) + \frac{1}{2} \cdot a(t) \cdot \delta t^2 \tag{2.6}$$

Substituting $-\delta t$ for t gives

$$p(t-\delta t) = p(t) - v(t) + \frac{1}{2} \cdot a(t) \cdot \delta t^2 \tag{2.7}$$

If we add the equations 2.6 and 2.7, we then obtain the next time step position in equation 2.8

$$p(t+\delta t) = 2p(t) + a(t) \cdot \delta t^2 - p(t-\delta t) \tag{2.8}$$

in which $a(t)$ is the acceleration at t time and $v(t)$ is the velocity at t time.

The verlet algorithm uses no explicit velocities during the integration. It can be easily implemented with modest memory requirements.

2.3.5.2 Velocity Verlet Intergration

The velocity verlet algorithm is an improved version based on the verlet algorithm. The velocity verlet algorithm can produce next time-step position and velocity only by current information. The position and velocity formulas are listed in equation 2.9 and 2.10.

$$p(t+\delta t) = p(t) + v(t)\cdot\delta(t) + \frac{1}{2}a(t)\cdot\delta t^2 \tag{2.9}$$

$$v(r+\delta t) = v(t) + \frac{1}{2}[a(t) + a(t+\delta t)]\delta t \tag{2.10}$$

In this thesis, our MD implementation is based on the velocity-verlet integration algorithm.

2.4 CUDA

CUDA [19] (Compute Unified Device Architecture) is a parallel computing architecture designed for GPUs. It enables programmers to write C (C-CUDA) code to utilize GPUs for processing non-graphical data. C-CUDA programs are compiled using a specialized PathScale Open64 C compiler. CUDA shares the same purpose as Microsoft DirectComput and OpenCL. CUDA has been widely used to accelerate computations which otherwise take much longer time or are intractable with the current technology, e.g., molecular dynamics simulation, electronic design automation, accelerated rendering of 3D graphics, speech indexing, and physical simulations.

With a design principle different from traditional CPUs, GPUs are based on a parallel throughput architecture that is aimed at executing a large number of concurrent threads slowly, as opposed to executing a single thread very fast. CUDA provides APIs for multiple operating systems, including Windows, Linux, and recently Mac OS X. Moreover, CUDA is supported by all GPUs recently designed and manufactured by nVIDIA [25], i.e., from the G8X series onwards, including GeForce, Quadro and the Tesla product lines. nVIDIA maintains binary compatibility among different generations of their GPUs such that CUDA programs developed for the GeForce 8 series will also work without modification on all future nVIDIA graphics cards.

With a radically different design, CUDA is superior over traditional GPGPU solutions with graphics APIs. For example, CUDA supports Scattered Reads, i.e.,

programs can access memory at arbitrary addresses on both the host and device. Moreover, CUDA allows the different hardware threads on the GPUs to access a shared memory region. Lastly, CUDA has a solid hardware implementation of the floating point arithmetic, which is essential for scientific computations.

Admittedly, CUDA also suffers several drawbacks at the current stage. For instance, C-CUDA disallows the uses of recursion and function pointers, which might place a burden on programmers while developing CUDA programs in some scenarios. Although equipped with very fast internal cache memories, the GPU might suffer from the bus bandwidth and latency bottlenecks along the data-path to the CPU. Furthermore, the deep memory hierarchy and intricate internal mechanisms might have huge performance implications if CUDA programs are written without accounting for such complexities in the design. Nevertheless, we believe the advantages of massive-parallelization offered by CUDA surely outweigh the drawbacks as mentioned above in real world applications.

Besides C, CUDA has bindings for most mainstream programming languages, including C++, Java, .NET, Perl, Python, Ruby, Lua, FORTRAN, and Matlab. In this work, we focus on jCuda [26], which is the CUDA binding for the Java language, which is being actively developed with support for the most recent CUDA API. Moreover, jCuda is fully interoperable among different CUDA based libraries. Since Hadoop is implemented entirely in Java, jCuda provides a solid foundation for bringing CUDA technology into Java applications, including the Hadoop framework.

2.4.1 CPU+GPU structure

The CPU+GPU architecture is shown in Figure 2.10. We demonstrate a very simple array summation example to explain how they work. In order to distinguish arrays in main memory from GPU's global memory, we use "dev" (short for device) plus capital character to identify three arrays on GPU's global memory. First of all, the CPU allocates three arrays in the main memory, array "a" and "b" contains elements we want

to sum where array “c” is used to store the results. Correspondingly, the CPU also needs to allocate three arrays on GPU’s global memory which is the blue square in the GPU. The Main memory will copy the array “a” and “b” contents into GPU’s global memory following the CPU’s order. On the GPU side, the green squares are computation elements and the purplish red squares are shared memory for certain amount of the computation elements. Communication between shared memories should employ global memory. The computation element needs to load array “devA” and “devB” into shared memory before launching the summation.

After the summation operation, array “devC” will be stored to global memory from shared memory. The next step is to copy array “devC” to array “c” from global memory to main memory. Finally, all memory space in main memory and global memory will be recycled.

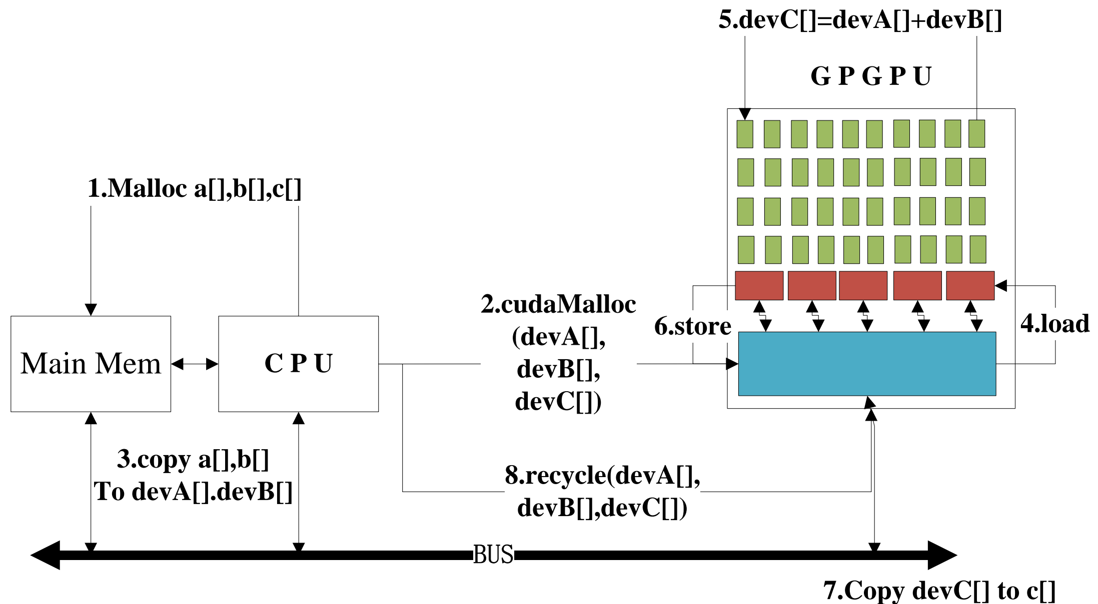


Figure 2.9: CPU+GPU Architecture

2.5 Related Work

MapReduce has been used in scientific computation in many fields. Kelvin Cardona [2] implemented MapReduce to analyze Probabilistic Neural Network data. Jaliya

Ekanayake[3] introduced MapReduce to High Energy Physics data analyses and Kmean clustering. Michael C. Schatz [4] developed *BlastReduce* based on MapReduce for processing DNA sequences and obtained 250x speedup compared with single processor *BLAST*. Weiying Shang [5] used MapReduce for mining Software Repositories. Jinguo You [6] parallelized the Close Cube Computation process with MapReduce.

MapReduce was first focused on data-intensive applications. It has been extended to some computation-intensive applications because of its fault-tolerance, simplicity of programming mode, and scalability. Bin Wu [7] proposed a general All Maximal Clique enumeration process in a distributed manner on a cluster with the help of MapReduce. Chao Jin [8] created an automatically parallelizing Genetic Algorithm platform called MRPGA built on MapReduce.

MapReduce programs debugging and profiling has grown in prominence with the increasing number of applications using MapReduce. There are some companies that have published white papers and presented their methods [9, 10]. Xu [11] and Jiaqi Tan [12] provide Log-based analysis and a debugging tool for MapReduce respectively. A real-time tracing tool for MapReduce has been created by Dachuan Huang [13].

In order to improve clusters' performance, accelerators have become the common devices to enhance CPUs' performance, to reduce energy consumption, and to speed up programs' execution. There are some MapReduce variants that can utilize accelerators to improve original MapReduce program performance. Bingsheng He [36] proposed "Mars" which is a MapReduce framework on graphics processors. Yolanda Becerra [34] has introduced an approach for exploiting the heterogeneity of a Cell BE cluster by linking an existing MapReduce runtime implementation for distributed clusters and another to exploit the parallelism of the Cell BE nodes. Jorda Polo [14] created an adaptive task scheduler which provides dynamic job allocation on hybrid MapReduce clusters consisting of nodes with accelerators.

Chapter 3

Design and Implementation

3.1 MD Simulation based on MapReduce

In this section, we will introduce how we designed and implemented MDMR. First of all, we will list challenges and constraints when we designed MDMR. They come from two aspects.

On one hand is the MapReduce specification. The MapReduce framework can easily handle Terabytes of data that have little dependency. The MapReduce framework does not allow communication among TaskTrackers in the Map stage. All data flows are in the form of key/value pairs.

On the other hand, the MD simulation data have dependency because atoms interact with their neighbors. Synchronization is needed because atoms' positions in the next time-step are decided by their current position, acceleration and velocity. Fault-tolerance should be guaranteed. The whole time-step computation result will be invalid if there is an error in processors which run in parallel. In the end, we have to verify the correctness of our MD simulation program.

We satisfied previous constraints one by one through carefully designing MDMR.

1) Data dependency. In this thesis, we disassembled the simulation system by using the atom-decomposition method [28] to satisfy data dependency. We first place a file containing all atoms' position into DistributedCache, which is a public area on

HDFS that every TaskTracker can access. Before starting a Map task, the TaskTracker has to first load the position file into its memory. In this way, no matter how we divide the atoms in a simulation system, every worker node has a copy of all atoms' positions. We also eliminated the communication between TaskTrackers in the Map stage.

2) Synchronization. In MD simulation, the next time-step atoms' positions, velocities, and accelerations are decided by the current time-step potential and kinetic energy. We regard one time-step computation as one MapReduce program; at the same time, we put all computation in the Map stage and let the Reduce stage do the synchronization, because the Shuffle phase can handle synchronization.

3) Fault-tolerance. MDMR guarantees fault-tolerance for each time-step because each time-step is a MapReduce program which can deal with node failure in run-time. Most of the current MD simulation programs' fault-tolerance mechanism is to periodically output a restart file for several time-steps. Assume a MD simulation program saves a restart file every 5 time-steps. It has to redo the previous 4 correct time-step simulations if failure happens in the 5th time-step. It wastes computation resources. However, MDMR will not move to the next time-step until it obtains the correct result for the current time-step.

4) Correctness verification. We verified our MD programs' correctness through energy conservation. That's to say, the current time-step system energy should equal the system energy in the next time-step. We first wrote a serial MD program and verified its correctness through energy conservation. We then compared the MDMR's result with this serial MD program's result. MDMR is considered correct if there is no difference between its result and the serial program's result.

We will detail MDMR's design and implementation in this chapter.

3.1.1 Atom decomposition method

Scientists parallelize MD simulation through atom decomposition, spatial decomposition and force decomposition methods [39, 40, 41]. The atom decomposition method can be used to parallelize MD simulation through dividing the input atoms' coordinates file. It does not need to know simulation system's spatial and force information. This method is easy to parallelize through the JobTracker which is the master node in MapReduce framework according.

The atom decomposition method first divides the input data to small parts and allocates them to worker nodes without considering data dependency. However, every worker node keeps a copy of all input data to satisfy data independency.

In MDMR, we place the input file in the *DistributedCache* which is a public cache for all TaskTrackers in Hadoop MapReduce. This guarantees every TaskTracker has a copy of the input file; at the same time, the JobTracker will divide input file evenly to *inputsplits* and allocate them to TaskTrackers. From the TaskTrackers point of view, each TaskTracker receives an *inputsplit*. If necessary, it can obtain other *inputsplit*s information by accessing *DistributedCache.*

In our MD simulation, we use the Velocity-Verlet method for particle velocity computation [24], the Leonard-Jones method for potential computation among particles and the atom decomposition method for the parallelization of MD. We choose the Argon atom as the object for the MD simulation. The long-range interaction (non-bonded) is the only interaction between every two atoms.

Every Argon atom has a unique ID number and can be located by three-dimensional coordinates in the simulation system. To simulate atoms behavior in a given period of time, we have to know their initial velocity and acceleration, which are both three-dimension vectors. We store these atom position, velocity, and acceleration vectors in a text file in which each line records one atom's information. Every line starts with an

atom ID number. The atom element symbol follows with the ID; and then, atom position vector, velocity vector, and acceleration vector. This is shown in Figure 3.1.

1 Ar Px Py Pz Vx Vy Vz Ax Ay Az

Figure 3.1: Simulation Coordinate file

The atom-decomposition method parallelizes the MD simulation through a “divide and conquer” algorithm. For example, if we use 3 processors to simultaneously simulate a 300 atom system, each processor will be assigned 100 atoms without considering the interaction dependency among atoms. However, each processor should keep all 300 atoms’ information in memory to maintain data independency. The serial atom-decomposition algorithm is shown in Table 3.1.

Table 3.1: Serial MD algorithm

---

for *i*th atom in the system(i from 0 to n)

obtain last time-step information

for *j*th atom in the system (j from 0 to n)

compute D_{ij} which is the distance between atom i and atom j

if $D_{ij} <= R_{cutoff}$ (distance between atom i and atom j is not bigger than potential cut-off radius)

compute Lennard-*Jones* potential energy, interactive force posed on *i*th atom from *j*th atom. Write the force vector into $F[j]$.

Loop

for *k*th element in $F[]$ (k from 0 to n)

Add force vector $F[k]$ into vector f_i

Loop

According to f_i, compute the acceleration, obtain the velocity through *Velocit-Verlet* algorithm, and the position for current time-step. Store all of them for next time-step computation.

Loop

We can easily get its time-complexity:

$$T(n) = n(a + bn + cn + d) \quad (3.1)$$

where n is the number of atoms in our simulation system. a, b, c and d are constants. They refer to time to load one atom information, computation of potential energy and force, operation of adding $F[k]$ to f_i, and next time-step data's computation time. We simplify the formula into equation 3.2.

$$T(n) = p_2 n^2 + p_1 n + p_0 \quad (3.2)$$

Based on the serial algorithm, the MDMR algorithm is shown in the Table 3.2.

Table 3.2: MDMR algorithm

Load all atoms information to *DistributedCache* before starting *Mapper*

Mapper:

Input (Key, Value): (*xyz* file's line number, single atom last time-step information)

Map method:

Read all atoms information from *DistributedCache*, for each input k-v pair (one atom), implement the Serial algorithm, but we only need to simulate portion of atoms in the system.

Output Key: atom sequential number; Output Value: single atom current time-step information

Reducer:

Input (Key, Value): (atom sequential number, single atom current time-step information)

Reduce method:

Collect all key-value pairs and store them into HDFS

Through MapReduce parallelization, we can see the time complexity reduce to

$$T(n) = \frac{p_2 n^2 + p_1 n}{m} + \varphi(m) \qquad (3.3)$$

m is the number of mappers that can simultaneously execute in our MDMR. p_1, p_2 are coefficients. $\varphi(m)$ is composed of two parts. One is the possible overhead caused by increasing the number of mappers. It is a function of m (in this thesis, we assume it is a linear function of m). The other is MapReduce framework overhead which is a constant (we find out this constant through experiments in Chapter 4) like job initialization and recycling, JVM creation, and garbage collection, etc.

MDMR guarantees fault-tolerance for each time-step because each time-step is a MapReduce program which can deal with node failure in run-time. Most of current MD simulation programs' fault-tolerance mechanism is to periodically output a restart file for several time steps. Assume a MD simulation program saves a restart file every 5 time-steps. It has to redo the previous 4 correct time-step simulations if failure happens

in the 5th time-step. It wastes computation resources. However, MDMR will not move to the next time-step until it obtains correct result of current time-step.

3.2 Tuning of MDMR

Program performance depends on multiple factors, not only hardware but also software configuration. In this section, we explore 10 factors that have influence on our MDMR's performance.

3.2.1 Hadoop Parameters

Hadoop MapReduce has abundant configurable parameters that are closely related to the program's performance. Impetus [9] and Cloudera [10] published their case studies on tuning a Hadoop MapReduce cluster. These parameters concern compression of intermediate output, speculative execution, JVM reuse, replication of data, logging, mapper/reducer number, temporary space allocation, block size, and so on. We refer to these configurations that may contribute to MDMR's performance and examine three important parameters that are the number of mappers, the number of reducers and the block replication in our evaluation. We detail 8 Hadoop MapReduce configuration parameters in this work; others follow the default setting of Hadoop 0.20.3.

mapred.job.reuse.jvm.num.tasks This parameter is in charge of the number of tasks that can be executed by a jvm. The default value is "1" which means one jvm can only run one task. However, the cost of initializing and recycling a jvm is not neglegable if one TaskTracker needs to process a large number of tasks. In order to reduce overhead, we configure this parameter to be "-1" which means a jvm can be reused by a job in a TaskTracker no matter how many tasks this job has.

mapred.child.java.opts This is the java options for a TaskTracker's child processes. The administrator can adjust according to the hardware properties and application requirement. In our clusters, each node has two single-core CPU and 4GB RAM.

Because a map task of MDMR is computation-intensive and needs to keep all atom information in memory, we configure this parameter as 1GB.

mapred.task.timeout It is the maximum time in milliseconds before a task will be terminated if it stops reporting its status, reading the input or writing output. We need to increase this limit because MDMR is a kind of computation-intensive application. The default value, which is 10 minutes, is not adequate. For example, a 27000-atom system needs more than 10 minutes to run its map tasks without any data I/O and status change. It is large enough for MDMR if we set this parameter as 1000 minutes.

mapred.output.compress This parameter is very important for a data-intensive application. It allows Hadoop to reduce I/O data size between memory and disk by compressing MapReduce job output. For example, at the end of the Map stage, mapper output data is first stored in memory before it has been dumped into disk due to the memory size limitation. The reducers do not need to access mappers disk if the mappers output can be stored in memory by compression. Nevertheless, MDMR is a kind of computation-intensive application. We do not need to waste CPU time which can be used to do MDMR simulation, compressing the job output, because its job output is smaller than the memory capacity.

mapred.task.cache.levels This defines the max level of task cache for a node. A node will cache tasks not only at node level but also at rack level if this parameter is 2. Similarly, if it is 1, the tasks cached are only at node level. It is important for data-intensive MapReduce programs to maintain the data locality. To facilitate nodes processing corresponding tasks using local data can reduce the network traffic and decrease the processing time. We assume disk access is faster than network data transferring. In our experiments, we leave this parameter as 2 which is the default setting.

mapred.map/reduce.tasks.speculative.execution These are Boolean parameters that have “TRUE” or “FALSE” options. “TRUE” means multiple instances of some “slow” map/reduce tasks may be simultaneously executed (in order to avoid exhausting

computation resources, there are two copies actually running in parallel including the original one). This is a fault-tolerance policy of the MapReduce framework. It can re-execute tasks if there are TaskTracker failures. Thus, we set these two parameters to be "TRUE".

mapred.TaskTracker.map/reduce.tasks.maximum These limit the maximum number of map/reduce tasks that will be run simultaneously by a TaskTracker. According to the MDMR specification and our cluster's hardware, we set this parameter as 2 for map tasks and 2 for reduce tasks. While MDMR is a computation-intensive application, at the same time, each node has 2 single-core CPUs. If there are more than 2 map tasks running on one node, they will compete for the CPU resource. For the reduce tasks, we keep it the same as map tasks' setting.

dfs.replication We can set the replication for each block in HDFS through this parameter. It is 3 by default. The actual number of replications can be specified when a certain file is created. The replication number is 2 in our experiment and the relationship between replication number and the MDMR execution time will be evaluated in Chapter 4.

In this thesis, we will evaluate MDMR performance based on three parameters: *mapred.TaskTracker.reduce.tasks.maximum*, *mapred.TaskTracker.map.tasks.maximum*, and *dfs.replication*.

3.2.2 Other factors

In the distributed environment, the slowest processor determines the execution time of entire parallel program. We need to balance the work load among the cluster and aim our best to let all nodes finish their work at the same time [27]. That means we should assign slower processors a smaller number of tasks and the powerful processors a larger number of tasks.

Hadoop MapReduce is based on a Master/Slave structure. Worker nodes will ask the master node for tasks once they have free slots or finish their current tasks. In a heterogeneous environment, for a given job, we can achieve the balance of slave nodes by decreasing a job's granularity to generate larger amounts of tasks. In this way, faster nodes will ask for more tasks and slower nodes may take fewer tasks. To some extent, this method has a positive effect on balancing the cluster. However, this method may introduce extra overhead if the number of tasks is arbitrarily large. We call this method "*more tasks*". In section 3.4, we develop a runtime monitor for this method to demonstrate its influence on our MDMR.

There is another way to achieve load balancing. We do not change the task number but assign different sizes of tasks to TaskTrackers according to their performance. It needs a dynamic adaptive load balancer for the current MapReduce framework; we will implement this in our future work.

3.3 Evaluation of MDMR

In this section, we evaluate MDMR as a MapReduce application composed of a Map stage and a Reduce stage. Because the Reduce stage includes data transferring in the cluster, we separate the Reduce stage into two phases: shuffle and Reduce phase. The Shuffle phase transfers the mappers' outputs to the reducer as inputs. It starts from the end of the Map stage and finishes at the beginning of the Reduce stage. MDMR's Reduce stage is different from the original MapReduce Reduce stage which consists of three phases: Shuffle, Sort and Reduce. Since the key is the atom ID and the value is this atom's coordinate information, mapping from key to value is one to one. The Classical MapReduce Sort phase which sorts all key-value pairs with the same key in order does nothing in MDMR. We can neglect this phase. Figure 3.2 shows why we divide MDMR like this. It illustrates the timeline in which MDMR simulates a 64000 atoms system using 16 mappers and 1 reducer in 1 time-step. The x-axis is time and the y-axis is the name of nodes that run the program. The blue bar means that node04 runs

setup (initialization) for this job. Because MapReduce will use a map task (or reduce task, depending on which type of task has a free slot) to setup the job, we regard setup as part of the Map stage. After the setup phase, all tasks enter the job queue and the JobTracker will schedule them to TaskTrackers. We can clearly see that mapper process time (yellow bar) on different nodes is different even though every node has been assigned one task with the same size. There are many reasons, for example, data locality, status of each node, etc. In a heterogeneous environment, the difference among yellow bars might be more significant. Work load balancing is needed if all worker nodes have different computation capability. In this thesis, we only take the homogeneous environment into account and will implement the balancer in the future.

The red bar is the Shuffle phase which starts after the Map stage. The Reduce phase is the light blue bar following the Shuffle phase. The green bar is the cleanup process which is related to job recycling. Between the Reduce phase and cleanup, there is also a gap. This is a kind of system overhead. Because once a job finished, it will enter a committing queue. The JobTracker will move a job from RUNNING queue to COMMIT queue. Before job committing, JobTracker has to report the job counter and finalize the job monitor called *JobInProgress.*

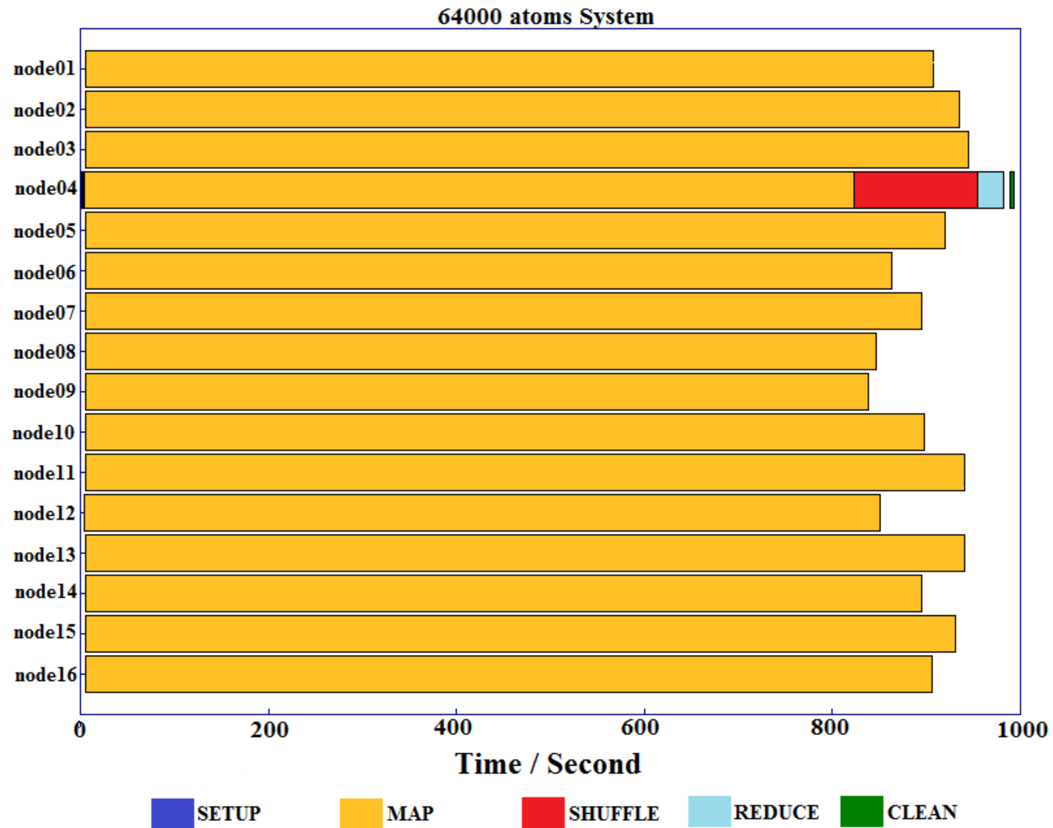


Figure 3.2: Hadoop Timeline of MDMR for 64000 atoms

3.3.1 MDMR Speedup

Speedup helps us quantify how much MDMR is faster than a corresponding sequential algorithm. It is defined as the following formula

$$S = \frac{T_{serial}}{T_{parallel}} \tag{3.4}$$

where T_{serial} is the execution time of a serial program and $T_{parallel}$ is the execution time of a parallelized program. In this thesis, we have two T_{serial}, one is the execution time of MDMR with one mapper and one reducer, the other is a totally serial MD simulation without using MapReduce framework. We can objectively evaluate MDMR in this way.

We also provide the speedup of MDMR-G (with GPGPU as accelerator) and MDMR-G vs. MDMR.

3.3.2 Karp-Flatt Metric

The speedup does not consider the parallel overhead with increasing processor number. It may overestimate speedup or scale speedup. The Karp-Flatt metric, also called experimentally determined serial fraction can provide some insights [20]. We introduce the following equation to describe the execution time spent in a parallel program.

$$T(n,p) = \sigma(n) + \varphi(n)/p + \kappa(n,p) \qquad (3.5)$$

where n is the problem size and p is the number of processors. $\sigma(n)$ is the serial portion of computation that cannot be parallelized. $\varphi(n)$ is the portion of the computation that can be executed in parallel. $\kappa(n,p)$ is the overhead that comes from the increasing processor number. The serial execution of the program does not have this part. It is simply:

$$T(n,1) = \sigma(n) + \varphi(n) \qquad (3.6)$$

The experimentally determined serial fraction e is defined as follow

$$e = \frac{\sigma(n) + \kappa(n,p)}{T(n,1)} \qquad (3.7)$$

We can use speedup to describe e and get another form of equation 3.7

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}} \qquad (3.8)$$

where ψ is the speedup on p processors.

By evaluating MDMR with the Karp-Flatt metric, we can clearly understand the parallel overhead which will contribute for formulating the MDMR execution time model. And then, we can tell whether MDMR is suitable for large MD simulation systems or not.

3.3.3 Minimum MapReduce Overhead

From the equation 3.5, we want to figure out what is the minimum of $\sigma(n)$. It is the time of system overhead even when there is no computation in MapReduce program. We employ “loadgen” which is a test example of Hadoop MapReduce framework [17]. It loads the input data and outputs them without any change. The user can specify the output data size through configuring the output data as a percentage of the input data. In order to get the minimum MapReduce overhead, we let the output data equal the input data.

3.3.4 Time Complexity

MDMR encapsulates the main computation into the Map stage. The Map stage execution time is quadratic with the number of atoms in the simulation system. Thus, the Map stage execution time is $T\ (n,m)$,

$$T_{map}(n,m) = \frac{c_2 n^2 + c_1 n}{m} + \varphi(m) \tag{3.9}$$

in which n is the number of atom in the simulation system and m is the number of mappers used in execution. We use c_1, c_2 to represent coefficients to avoid confusion from coefficients of equation (3.3). $\varphi(m)$ is the possible overhead but only in Map stage. In this thesis, we take this as constant. We will explain it in the evaluation chapter.

The Reduce stage of MDMR is in charge of data synchronization. It includes the Shuffle phase the Reduce phase. We formulate the Reduce stage as a linear process. It is a function of simulation system size.

$$T_{reduce} = c_1^r n + c_0^r \qquad (3.10)$$

Similarly, c_0^r, c_1^r are coefficients and n is the number of atoms in the simulation system.

In general, we add up equation 3.9 and equation 3.10. The total execution time formula for a given number of mappers (m is a constant number) is:

$$T\ (n) = d_2 n^2 + d_1 n + d_0 \qquad (3.11)$$

in which $d_2 = \frac{c_2}{m}, d_1 = \frac{c_1}{m} + c_1^r$, and $d_0 = c_0^r + \varphi(m)$.

3.4 Run-time Monitor for MDMR

In order to clearly understand MapReduce programs' execution, we create a run-time program monitor. It can monitor execution of any part of MapReduce programs, help a programmer to find out their programs bottleneck, and estimate the overhead of a new scheduler. It can also verify the correctness of MapReduce programs. In this thesis, we use this run-time monitor to "*more tasks*" method which balances the cluster through increasing the number of tasks.

As we discussed in section 3.2, it is necessary to evaluate the "*more tasks*" method not only in performance but also in efficiency. Therefore we create a run-time monitor for MapReduce programs. It can detect the computation-power of MapReduce programs. Its data collection and presentation processes are independent from the MapReduce framework, which can correspondingly reduce the interference on the original MapReduce program running on the same clusters.

3.4.1 Class specification

Our runtime monitor focuses on two main objects: *Mapper* and *Reducer.* We acquire the Map stage and Reduce stage computation power by inspecting the execution of *Mapper* and *Reducer*'s primary methods *map()* and *reduce()*.

Table 3.3: Java API of Mapper Interface

Method	Description
void configure()	Initializes a new instance from a JobConf
void map()	Maps a single input key-value pair into an intermediate key-value pair
void close()	Closes this stream and release any system resource associated with it

From Table 3.3, *Mapper* implementations can access the *JobConf* via the *JobConfigurable.configure(JobConf)* and initialize themselves. Similarly, they can use the *Closeable.close()* method for recycling. The framework then calls *map() (Object,Object, OutputCollector, Reporter)* for each key value pair in the *InputSplit* for that task. From the above desription, we conclude that *map()* will be called once for every key-value pair input. The following are the Java-MOP [29,30] FSM (Finite State Machine) codes:

```
fsm :
start [
      configureEvent -> running
]
running [
      mapEvent -> running
      closeEvent -> end
]
end []
```

The run-time monitor counts the number of *map()* and *reduce()* methods being called and monitors their execution time. According to the introduction of the MapReduce framework, we can confirm that the *map()* method counter must be the number of the *Mapper* class input. And the *reduce()* counter must be the number of the *Mapper* class output. By comparing the corresponding counter number with the input and output number of key-value pairs, we can confirm MapReduce programs' correctness. To get

the overhead of the run-time monitor, we can compare the program's execution with and without AspectJ code. The run-time data will be sent to the head node and the head node will load the data to its web page. In Figure 3.3, monitoring related processes have been marked with a bold line.

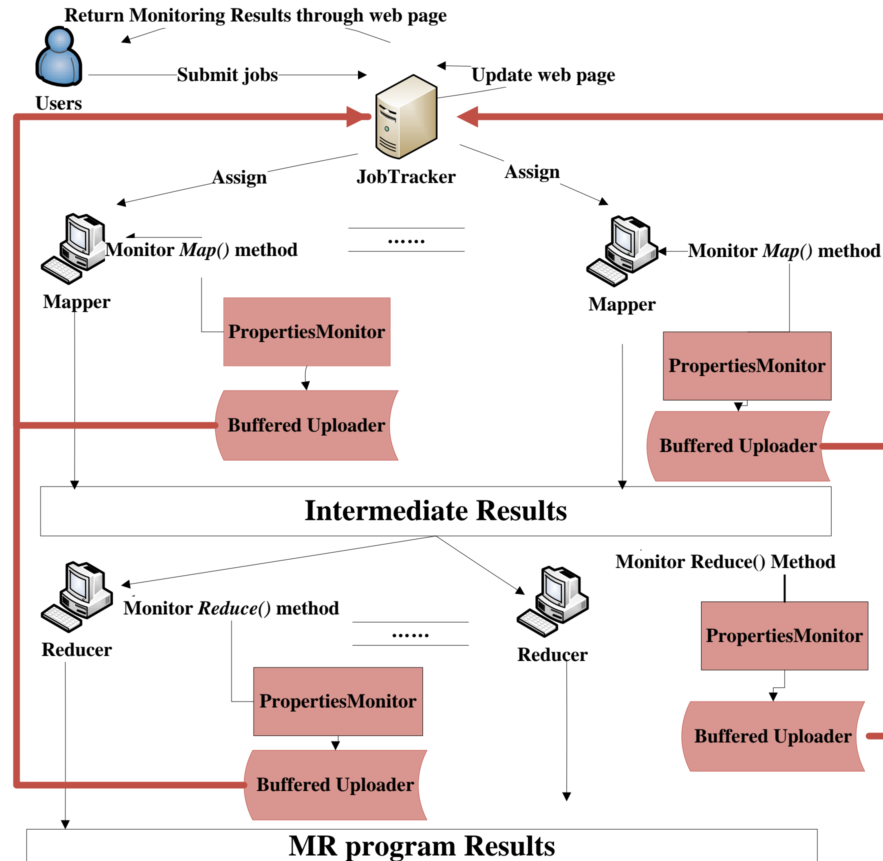


Figure 3.3: Run-time MapReduce program monitor data flow

3.4.2 Buffered Uploader

Once the monitor gets the results from the program it monitored, we need to report the data in real time. However, according to our experience, the head node may be flooded by worker nodes' requests if they report their results immediately. In order to obtain the smallest overhead, we explore 3 monitoring intermediate storage solutions: HDFS, NFS, and local file system with daemon.

Local file system with daemon has been used in this thesis (Buffered Uploader). The monitor stores intermediate results into a local file system on every worker node. The daemon on worker nodes periodically sends accumulated intermediate results to the head node through TCP connections. The other two methods have their limitations. First of all, it will introduce extra overhead into the original MapReduce program data flow if HDFS is used as storage for the monitor. For example, when we store the monitor's results to certain block on HDFS, the original MapReduce program has an I/O request on the same node. The original MapReduce program's request may be delayed.

For NFS, we can save some effort dealing with extra communications between worker nodes and the head node if every worker node can write to NFS. However, this method has its limitation. The NFS partition is mounted on a certain node which will be flooded if all other nodes simultaneously send a large number of requests to it.

Table 3.4 illustrates the relation between input file size and execution time of MapReduce program in both NFS and local file system. The execution time of program with run-time monitor increases significantly if we use NFS. However, the execution time with a local file system does not increase as fast as in NFS.

Table 3.4: NFS vs. Local File System Execution time

NFS as medium		
Input File size	No Monitoring Model	With Monitoring Model
1000 Atoms(61KB)	27.1s	29s
27000 Atoms(1.7MB)	29s	134s
Local File System as medium		
1000 Atoms(61KB)	27.1s	28s
27000 Atoms(1.7MB)	29s	29s

We can clearly understand the reason through the observation of clusters performance from Figure 3.4.

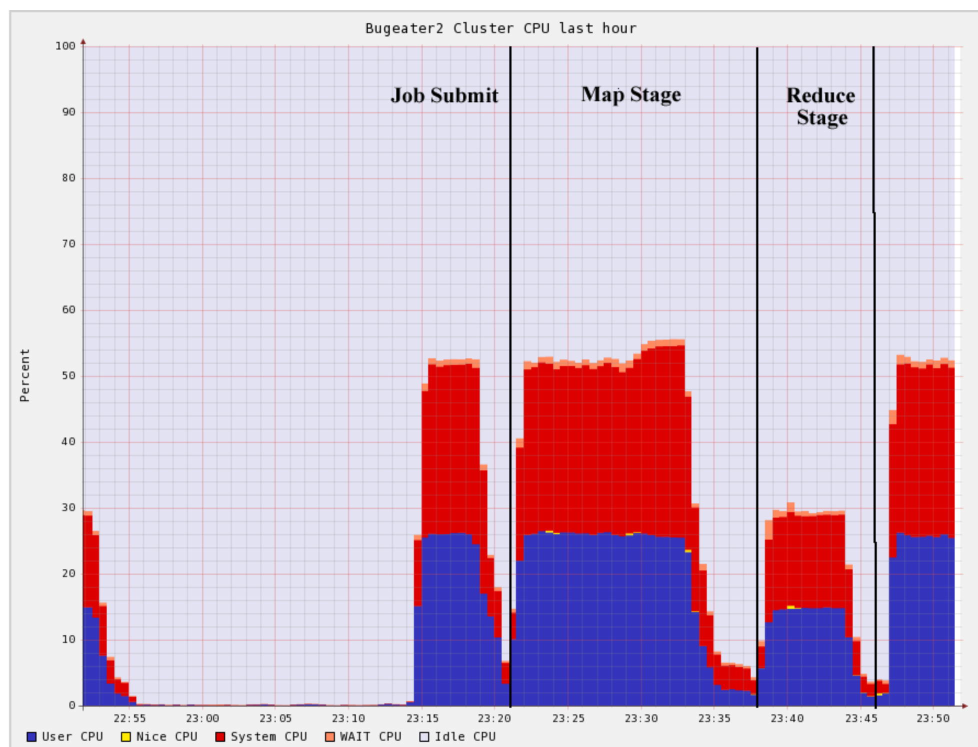


Figure 3.4: Statistical CPU time

Figure 3.4 shows processes on a MapReduce cluster obtained through Ganglia[31] which is cluster monitoring software. The red area is the system CPU time. It means CPUs do not really do user computation; on the contrary, they spent half their power to deal with system calls, such as context switch, process synchronization, etc. The system CPU time in Figure 3.4 is caused by the *nfsd* requests which were issued by our run-time monitor from worker nodes. The reason why they simultaneously occured is that the NFS partition is located on the head node of the cluster. Worker nodes make requests of the head node when there is a write or read request on NFS. The 25% CPU time is used by worker nodes' monitors NFS requests. NFS becomes a bottleneck for the monitor program.

In the MapReduce program monitor, we use a Buffered Uploader. In this manner, the worker nodes first store their monitoring result in local disk. At the same time, we

employ *crond* which is Linux system daemon which runs periodically to report data to the head node.

On the head node side, we may not update the web page for every worker node request. Because the head node is hosting the NameNode and the JobTracker simultaneously, memory overflow may occur if we use a large amount of memory to respond to worker nodes requests and frequently update the web page. The monitor's web page is updated in every one minute by *crond*.

3.4.3 Monitor Metrics

Because we collect each *map()* and *reduce()* methods' execution time, the computation power of the *map()* method is the summation of all *map()* execution time from all TaskTrackers, the same for to *reduce()* methods. In this thesis, we regard the time spent on all other parts (except *map()* and *reduce()* functions) of the MapReduce framework as overhead, like communication, synchronization, etc. Therefore, the efficiency of computation power is the ratio of *map()* and *reduce()* methods computation power and the computation power of whole MapReduce program. The formulas are listed below:

$$M_{overhead} = T_M - T \qquad (3.12)$$

$M_{overhead}$ is the monitor's overhead imposed on system. It equals the execution time of a given job with the monitor minus its execution time without the monitor.

$$C = \sum_{i=1}^{n} c_i \qquad (3.13)$$
$$c_i = p \cdot t_i$$

C is the cluster total computation power. It is the summation of computation power spends in every node in cluster. The i th node computation power c_i equals the processor number p multiplied by time t_i for those processors used for the computation.

$$\eta_C = \frac{C_{map} + C_{reduce}}{C_{total}} \qquad (3.14)$$

Computation power efficiency η_C is the ratio of computation which has been used in a user defined “useful” portion to the total computation power consumed by whole program in the cluster. We regard the time spent on *map()* and *reduce()* methods as our useful computation power. High computation power efficiency means the cluster spends a larger portion of computation power to process the user defined “useful” computation.

3.5 MDMR in hybrid environment

The MapReduce cluster is composed of PC nodes. They provide a high computation/cost ratio. In the past ten years, an accelerator has been accepted into clusters to improve their performance and power efficiency. In the Top 500 [32] supercomputers, Top-1, Top-3 and Top-4 supercomputers are CPU+accelerator architecture. At the same time, in the Green Top 500 [33], eight of the top 10 rated supercomputers are equipped with accelerators. In order to fully improve MDMR performance on a MapReduce cluster, we build a hybrid MapReduce cluster which has deployed GPGPU in every TaskTracker.

3.5.1 Hybrid Hadoop

We employ GPGPU to the Map stage because the most computation is in this stage. The data flow in our hybrid Hadoop MapReduce framework is shown in Figure 3.5.

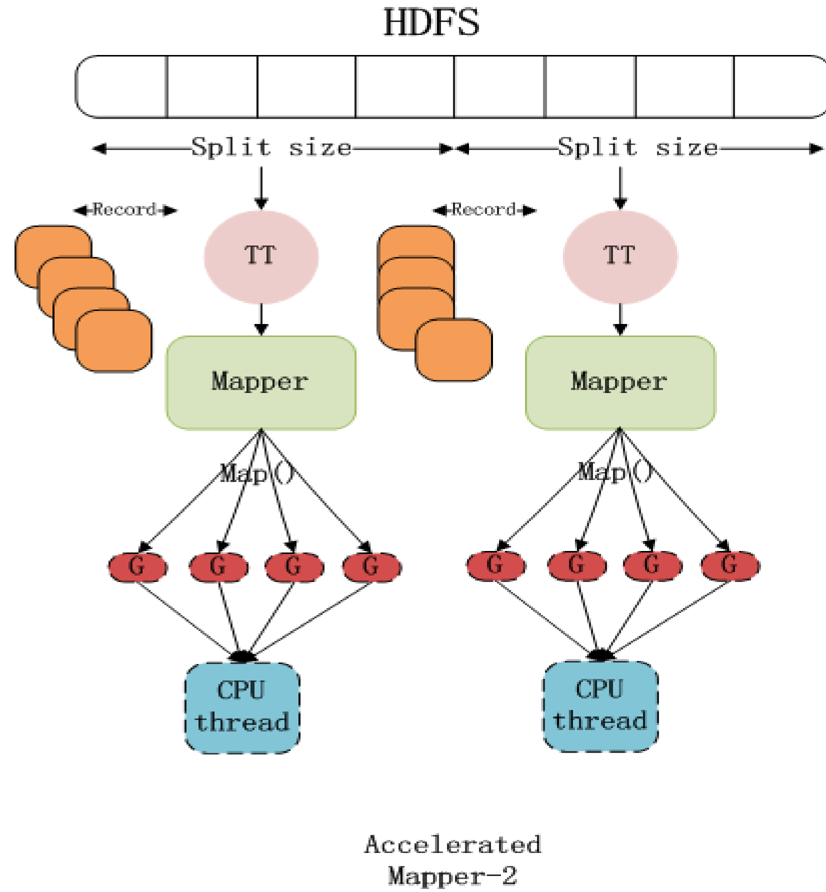


Figure 3.5: Hybrid Hadoop MapReduce structure

Section 2.4.1 illustrates that GPGPU needs a CPU to process its I/O. The hybrid MapReduce framework follows this rule. In Figure 3.5, our *mapper* is a CPU process in charge of data input, the GPGPU will help the *mapper* to parallelize the computation intensive *map()* method. After that, the GPGPU will ship the processed data to the CPU and the CPU will pass them to the *reducer*.

3.5.2 MDMR-G algorithm and time complexity

Before we introduce GPGPU into MDMR, we need to understand some specifications of the GPU. The GPGPU graphics adapter we used is Geforce 9400GT which allows one grid to run at a time. Each grid contains at most 65,535 blocks and each block can simultaneously execute a maximum of 512 threads. This means we can concurrently

have 65,535*512=33,553,920 threads. The MDMR's algorithm time complexity is $O(n^2)$. We can employ GPU to parallelize the outer loop in the *map()* method in Table 3.1 and reduce the time complexity from $O(n^2)$ to $O(n)$ if n isalways smaller than 33 million. The MDMD-G algorithm is shown in Table 3.5.

Table 3.5: MDMR-G algorithm

Load all atoms information to *DistributedCache* before starting *Mapper*

Mapper:

Input (Key, Value): (coordinates file's line number, current input atom a_i last time-step coordinates, velocity, and acceleration)

Map method:

{ Read all atoms information from *DistributedCache* to array *Total*[] ;

Send a_i and array *Total*[] into GPU global memory;

On GPGPU:{

Use *length(Total[])* GPU threads to concurrently obtain force array *F[]* ;

Sum force array *F[]* through vector summation to variable f ;

Return f to main memory;

}

Compute next time step acceleration, position, and velocity through equations 2.9 and 2.10;

Output intermediate key-value;

}

Output Key: atom sequential number; Output Value: single atom current time-step information

Reducer:

Input (Key, Value): (atom sequential number, single atom current time-step information)

Reduce method:

Collect all key-value pairs and store them into Distributed File System

---

Chapter 4

Evaluation

Our experiments are based on a MapReduce cluster, called BugeaterII, composed of 30 worker nodes. Each node has 2 single-core AMD Operon64 2.2GHz CPUs, 4GB DDR RAM, and is connected by 1 Gbps Ethernet. The capacity of HDFS on BugeaterII is about 10 TB. The replication factor on HDFS is configured as 2. The head node hardware configuration is the same as the worker node except it has 8G RAM and 2TB disk space. The Namenode and JobTracker are simultaneously running on it. BugeaterII is based on Hadoop 0.20.2 using the default First Come First Serve (FCFS) scheduler.

4.1 Hadoop Parameter tuning for MDMR

In Chapter 3, we predicted that two parameters have limited influence on MDMR, they are '*reduce.tasks.maximum*' and '*dfs.replication*'. We will verify their influence on MDMR in this section.

4.1.1 Reducer number

In MDMR, the Reduce stage takes intermediate results from the Map stage and generates the final output. The number of reducers can contribute either a positive or negative effect to the performance of the Reduce stage. An example may shed some light on this point. Assume the Reduce stage follows a slow start, no one reducer will start until the Map stage finishes. In other words, the Reduce stage does not overlap with the Map stage. Consider a data-intensive application with 1TB intermediate results

from 5 mappers. The cluster is composed of 5 homogeneous worker nodes, each of which has one network adapter. The head node only runs the master process: Namenode and JobTracker. According to the Hadoop default configuration, it will start 5 reducers to handle this 1TB data. That's to say, each node will process 204.8GB data. If we use more than 5 reducers, for example 10, each reducer will be assigned 102.4GB data. But these extra 5 reducers may compete with the original 5 reducers for the network bandwidth if the cluster allows 10 reducers to simultaneously retrieve data from the mappers. It is hard to say if the 10 reducer solution is better than the 5 reducer' one. To avoid competition, we can divide 10 reducers into two waves and only allow 5 reducers to run at the same time. However, this is not optimal either. The reducer processes need to be initialized at the beginning and recycled in the end. We introduce initialization and recycle overhead twice if there are two waves. On the other hand, from the administration point of view, the system resource is underutilized if the reducer number is smaller than the actual number of network adapters in the cluster.

Regardless, for CPU-intensive applications, the reducer number has limited effect on the performance because most of the execution time is spent not in I/O but in computation. We evaluated the effect of the reducer number on the MDMR's performance. Experimental results are shown in Figure 4.1.

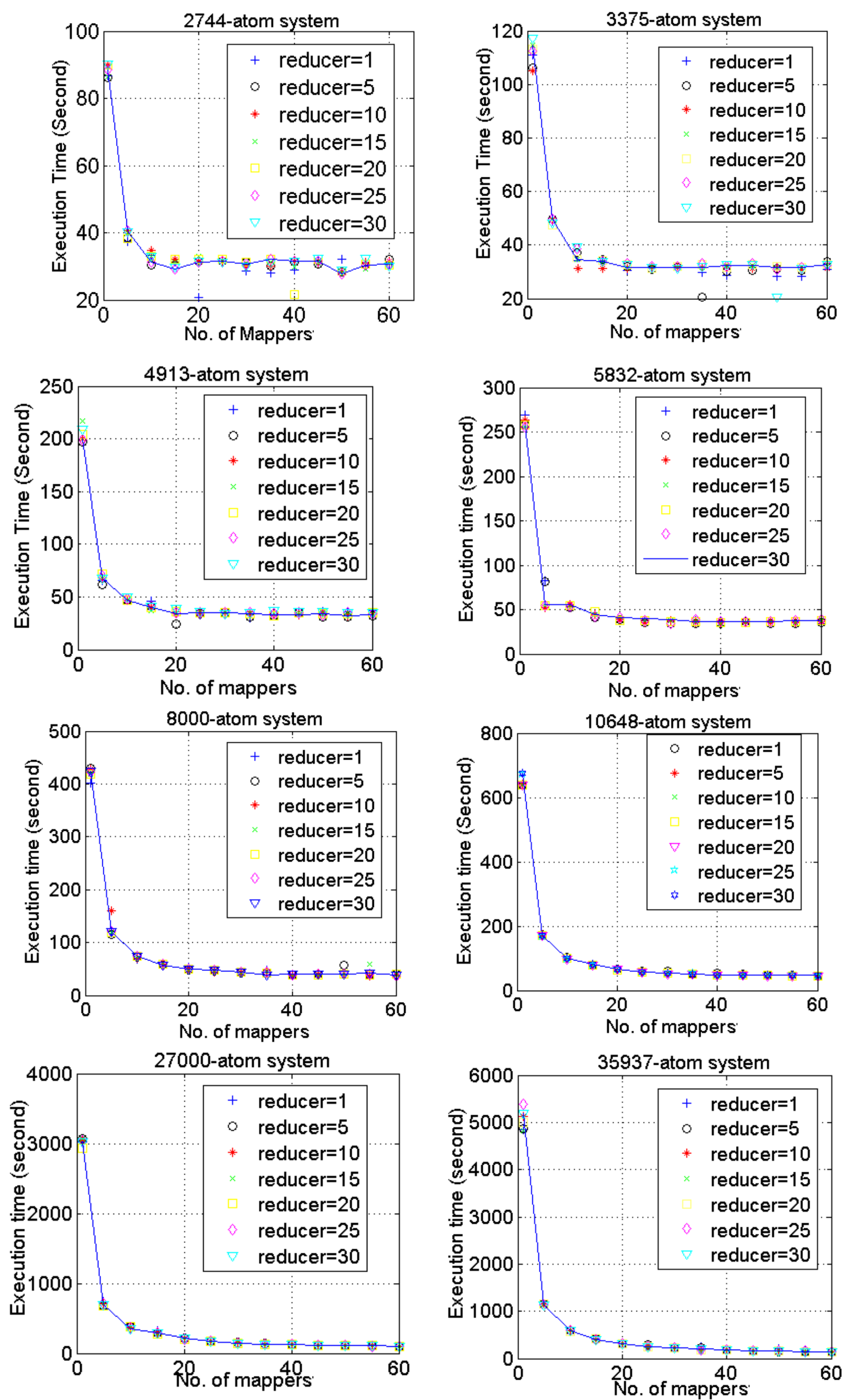
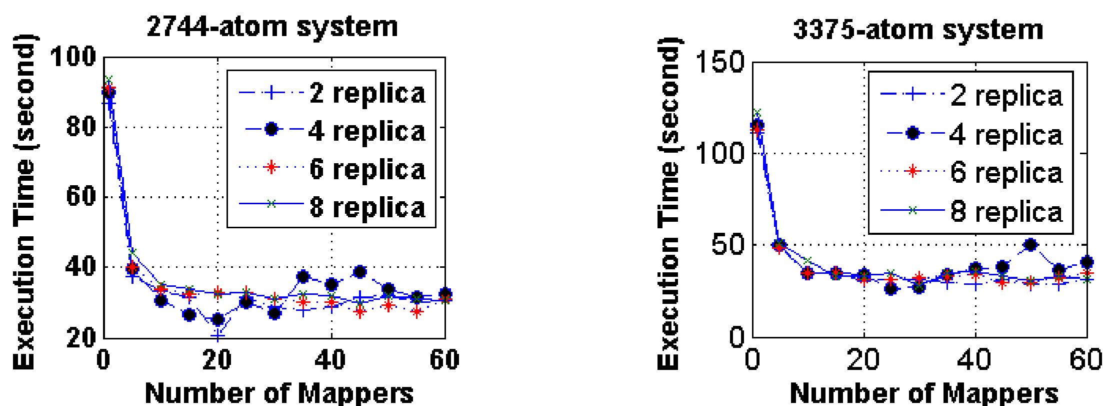


Figure 4.1: Effect of Reducer Number

From Figure 4.1, the fluctuation caused by the reducer number converges with the increasing simulation size. This verified our expectation. We will set the reducer number to 1 in our following MDMR simulations to reduce the disturbance probability caused by failure of a reducer if multiple reducers are used.

4.1.2 Replication

As we introduced in Chapter 2, HDFS has a replication mechanism to improve its data availability, load balancing and MapReduce program performance. In production clusters, some popular data may be the bottleneck of the whole distributed computing system. Multiple replications of popular data can balance the load among the system and reduce the risk of losing data. For data-intensive applications, replication can reduce the data transferring and execution time. However, MDMR is a computation-intensive application. The replication number has limited effect on it. In a 35,937 atom system, the input file is about 2.2MB. To transfer a 2.2MB file through a 1Gbps network will take 17.2ms. This is negligible compared to one time-step of simulation time which is about 12 minutes. Figure 4.2 lists the execution time of MDMR with different replication factors.



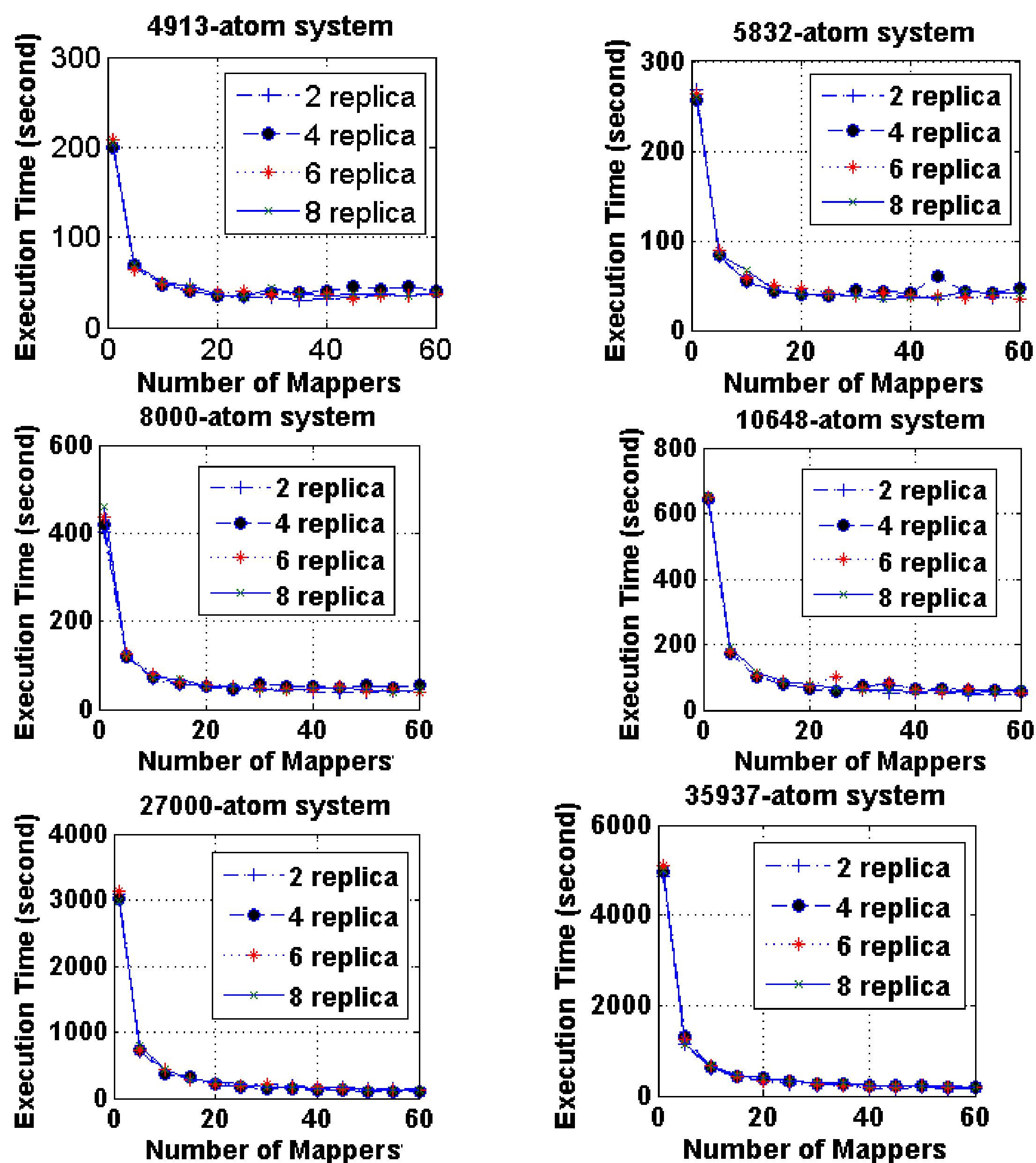


Figure 4.2: Effect of Replication number

The fluctuation trend of block replication number is similar to the situation we met in exploring reducer number effect. The replication number has limited effect on the execution time. Thus, we will set the block replication number as “2” in our following experiments to satisfy data redundancy and save disk space.

4.2 MDMR Evaluation

In this section, we evaluate MDMR with 12 MD simulation systems with atoms from 1000 to 64000. For each simulation system, we run it with 13 different numbers of mappers from 1 to a maximum of the actual CPU number in the cluster. And then, for a given simulation system with a given number of mappers, we run MDMR 3 times on a cluster and take the arithmetic average of these as our final result. The method helps us to avoid disturbance from cluster node failure or other random events.

We first obtained the speedup of MDMR and then evaluated MDMR performance through Karp-Flatt metrics. The parallelization overhead is not the critical factor affecting MDMR performance. We want to know what the minimum MapReduce system overhead is.

In Table 4.1, we show our simulation system size and the number of mappers used for evaluating MDMR. The first column demonstrates the atom number of 12 systems and the second column shows the 13 groups mapper numbers evaluated for each.

Table 4.1: Evaluation Data Configuration

Atoms in Simulation System	**Mapper used by MDMR**
1000	1
2744	5
3375	10
4913	15
5832	20
8000	25
10648	30
27000	35
35937	40
42875	45
54872	50
64000	55
	60

We used the Matlab curve fitting tool [35] which allows its user to use regression, interpolation, and smoothing modeling techniques to obtain the coefficients of a given

expression. We fit our time complexity formula with 95% confidence bounds. In order to estimate the accuracy of extrapolated results, we use the first 9 systems that contain atom numbers from 1000 to 35937 as our base line to get corresponding coefficients. We then use them to predict execution times of the 42875 atom system. Furthermore, we use the first 10 systems from 1000 to 42875 to estimate the 54872 atom system. Finally, we introduce 54872 atoms system into base groups and use these 11 systems to estimate the 64000 atom system time. The variance between the predicted time and actual execution time should decrease. In Table 4.2, we list three systems' maximum variance. t_p^m is the predicted time for MDMR with m mappers. t_a^m is the actual execution time of MDMR with m mappers. The variance decreases as we expected.

Table 4.2 Max Variance between Prediction and Actual Time

Max Variance \ System Size	**42875**	**54872**	**64000**
$\max\left(\left\lvert\frac{t_p^m - t_a^m}{t_a^m}\right\rvert\right)$ $m = 1,5,10,15,20,25,30,35,40,45,50,55,60$	7.74%	7.07%	6.69%

4.2.1 Speedup

In Figure 4.3, the base line is the MDMR execution time with one mapper and one reducer. We can see that if we use 60 mappers to simulate a 64000-atom system, the speedup is 43.7 in maximum. Another trend concerns the relation between the speedup and the simulation system size. We can see that if we use a larger system, we can get a larger speedup.

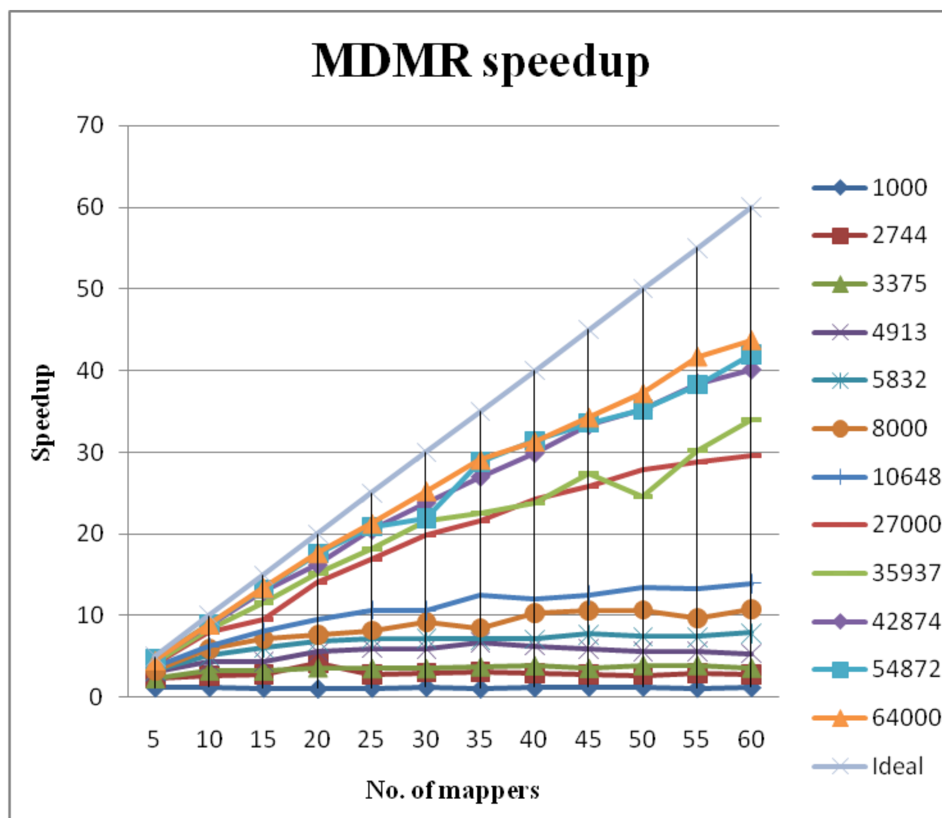


Figure 4.3 MDMR Speedup

However, we have to compare MDMR with the serial MD simulation to entirely understand MDMR performance. From Figure 4.4, the reason that we cannot obtain speedup as large as 43.7 is due to the MapReduce overhead. The overhead comes from MapReduce framework, communication, synchronization, etc. We employ the Karp-Flatt metric to reveal the parallelization overhead of MDMR in the next section.

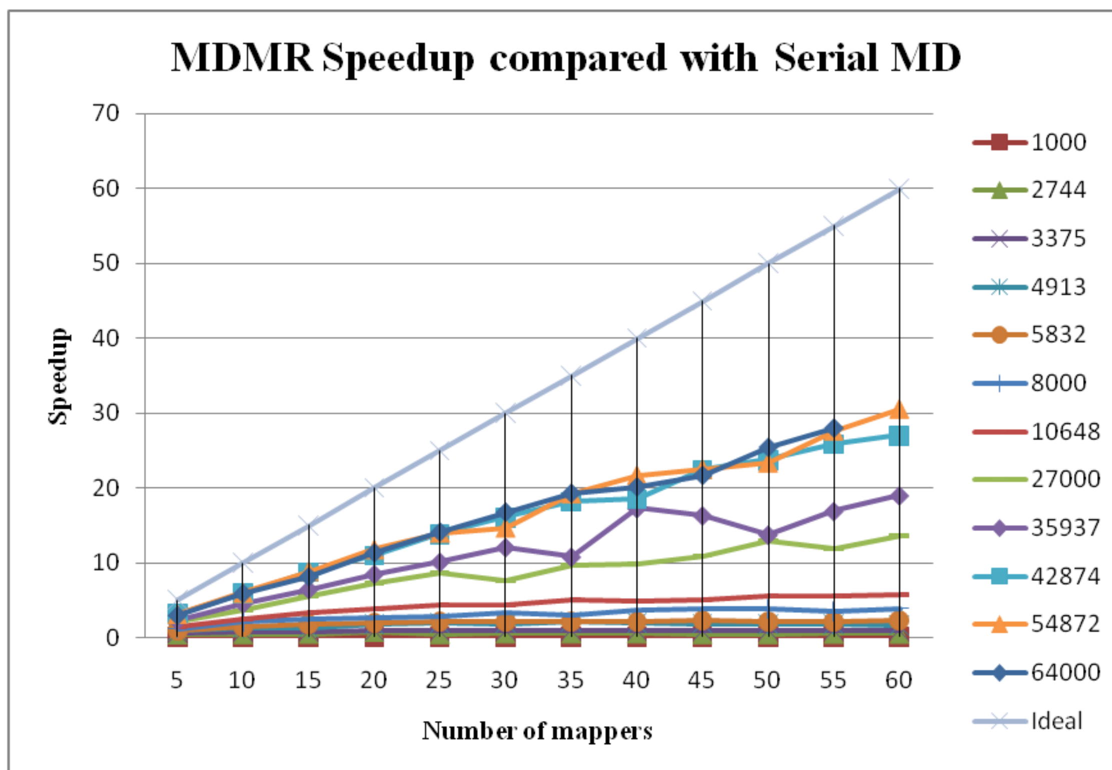


Figure 4.4 MDMR Speedup compared with Serial MD

4.2.2 MDMR Karp-Flatt Metric

The Karp-Flatt metric can help us to figure out the relation between the experimentally determined serial fraction and number of mappers in different simulation systems. In Figure 4.5, we can see that the serial percentage decreases with an increasing number of mappers and an increasing simulation system size. For a given size of simulation system, if *e* increases with the number of processors, this means this application performance decreases with the increasing number of processors due to parallel overhead. However, we can see that MDMR is a good implementation of MD simulation. In Figure 4.5, we can also see there are fluctuations. The *e value* increases in points in the curve of 8000 atom system using 35 and 55 mappers, 10648 atom system using 30 mappers, and 35937 atom system using 50 mappers. This is because the probability of node failure or error increases if we use more worker nodes. If a node is slow, it will be speculatively

executed and this will increase the execution time and thus speedup decreases. The *e value* becomes larger than the case without node failure or slow node due to impractical parallel overhead.

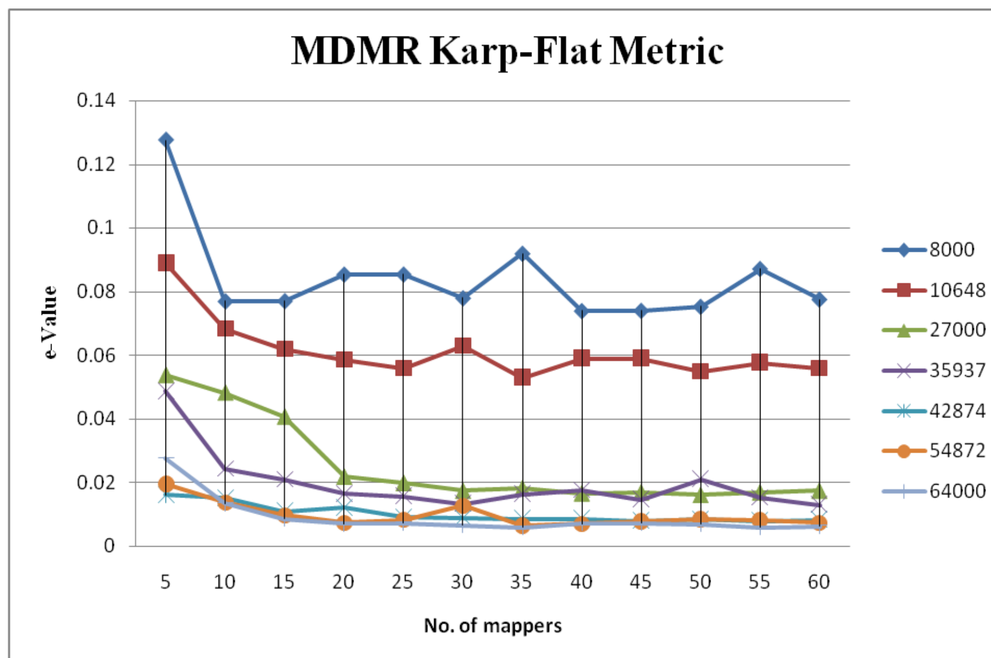


Figure 4.5 MDMR Karp-Flat Metric

4.2.3 Minimum MapReduce Overhead

In order to obtain the minimum MapReduce overhead, we employ "loadgen" to load the atom coordinates input file and output them without any change. In Table 4.3, we measure the time in seconds. An input file containing only one atom is 57 bytes. The Map stage takes 3 seconds. The Shuffle phase and Reduce phase take 7 and 10 seconds respectively. For a 64000 atoms input file (4.1MB), the overhead is the same. This is because the difference to load 57 bytes and 4.1MB data is in the millisecond level. At the time unit of seconds, we cannot find a significant difference. Next, the Shuffle phase involves multiple to multiple communications. To transfer 4.1MB data through 1Gbps Ethernet will not take 7 seconds. These 7 seconds are the system overhead. We will leave the question of why it takes this long to do the sort phase for future research.

Table 4.3 Minimum MapReduce Overhead

System	**1**	**1000**	**8000**	**27000**	**64000**
Size(Byte)	57	61K	498k	1.7M	4.1M
Map(sec)	3	3	3	3	3
Shuffle(sec)	7	7	7	7	7
Reduce(sec)	10	10	10	10	10
Total(sec)	20	20	20	20	20

4.2.3 Prediction

As we mentioned in equation 3.9 (Map stage's execution time),

$$T_{map}(n,m) = \frac{c_2 n^2 + c_1 n}{m} + \varphi(m) \qquad (4.3)$$

equation 3.10 (Reduce stage execution time),

$$T_{reduce} = c_1^r n + c_0^r \qquad (4.4)$$

and equation 3.11 (Total execution time with given number of mappers), total execution time prediction formula for a given number of mappers is

$$T\ (n) = d_2 n^2 + d_1 n + d_0 \qquad (4.5)$$

in which $d_2 = \frac{c_2}{m}$, $d_1 = \frac{c_1}{m} + c_1^r$, and $d_0 = c_0^r + \varphi(m)$.

In our experiments, we find that the curve of execution time is a perfect function of m^{-1} .($T = b_{-1} m^{-1} + b_0$, b_{-1} and b_0 are constants) if we fix n which is the simulation system size except *n=1000*. When *n=1000,* the execution time fluctuates between six seconds and twelve seconds. This is because the overhead is a relatively large constant. However, the computation time for 1000 atoms or less is not comparable with this overhead. From the above facts, we then deduce $\varphi(m)$ is a constant or linear function of

m but with very small coefficient. We do not evaluate the coefficient of $\varphi(m)$ and take $\varphi(m)$ as a constant in this thesis.

In the equation 4.5, we combine the linear coefficients and get $d_1 = \frac{c_1}{m} + c_1^r$. Through MDMR's algorithm, we can see that the Map stage dominates the total execution time of MDMR. The execution time of Reduce stage is related to the total data it collected during its execution. However, the MDMR simulation data size is relatively small. For 64,000 atoms system, the size of input or output file (input file and output file are the same size) is about 4.1MB. In this thesis, we firstly use $d_1 = \frac{c_1}{m} + c_1^r$ to simplify our evaluation and will estimate c_1 and c_2 in following paragraph.

In the 42,875 atom system, the execution time prediction formula for 1 mapper is

$$T(n) = 0.0000031698n^2 + 0.04785n + 9.34 \qquad (4.6)$$

Figure 4.6 shows the predicted time as a dot-line and actual time as a solid line. The maximum variance between actual and predict time is less than 7.74% of actual time.

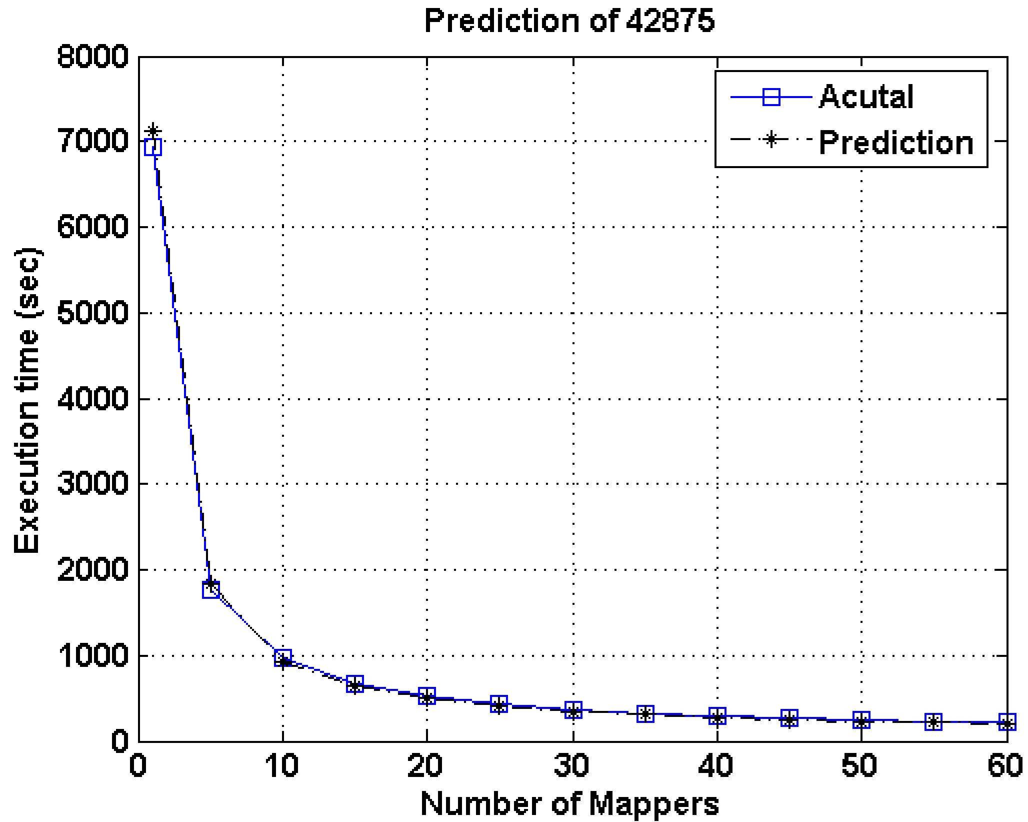


Figure 4.6: Prediction vs. Actual time

In the 54,872 and 64,000 atoms system, their execution time formulas for 1 mapper are

$$T(n) = 0.00000311n^2 + 0.04914n + 17.7 \tag{4.7}$$

$$T(n) = 0.000003n^2 + 0.05270n + 10.34 \tag{4.8}$$

Figure 4.7 and Figure 4.8 demonstrates the predict time in dot-line and actual time in solid line respectively for 54,872 and 64,000 atoms system. The maximum variance between actual and predicted time for 54,872 is less than 7.07% of actual time and for 64,000 is less than 6.69% of actual time.

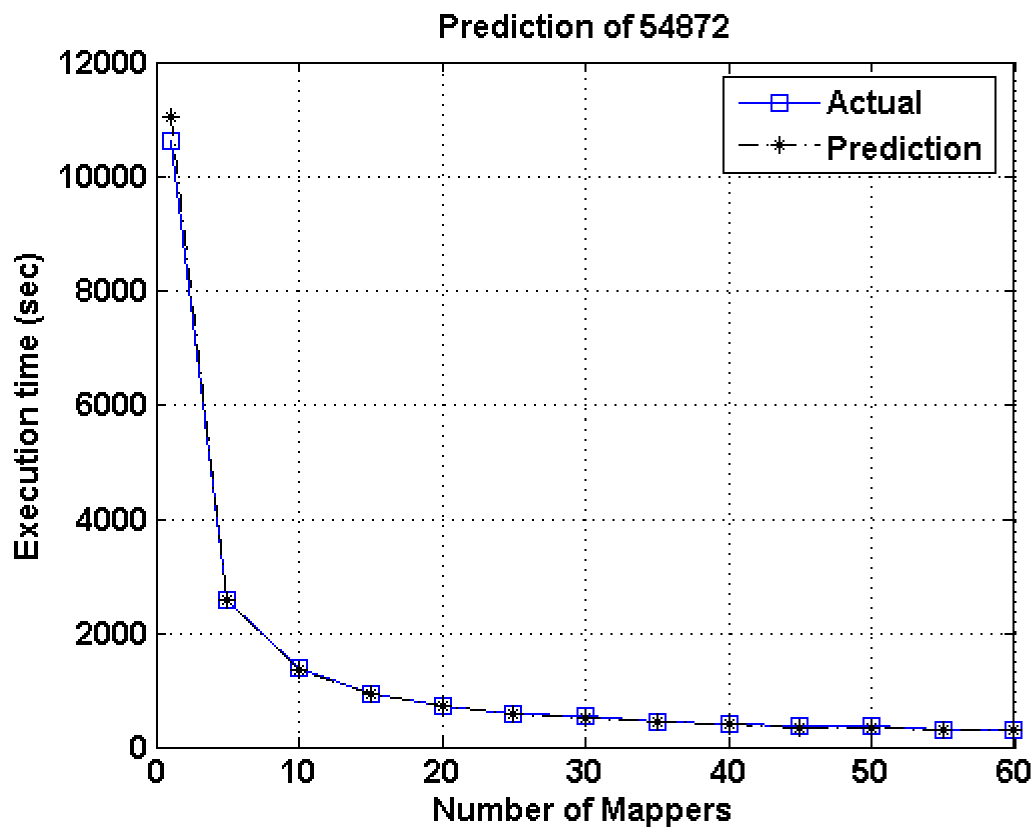


Figure 4.7: Prediction vs. Actual time-2

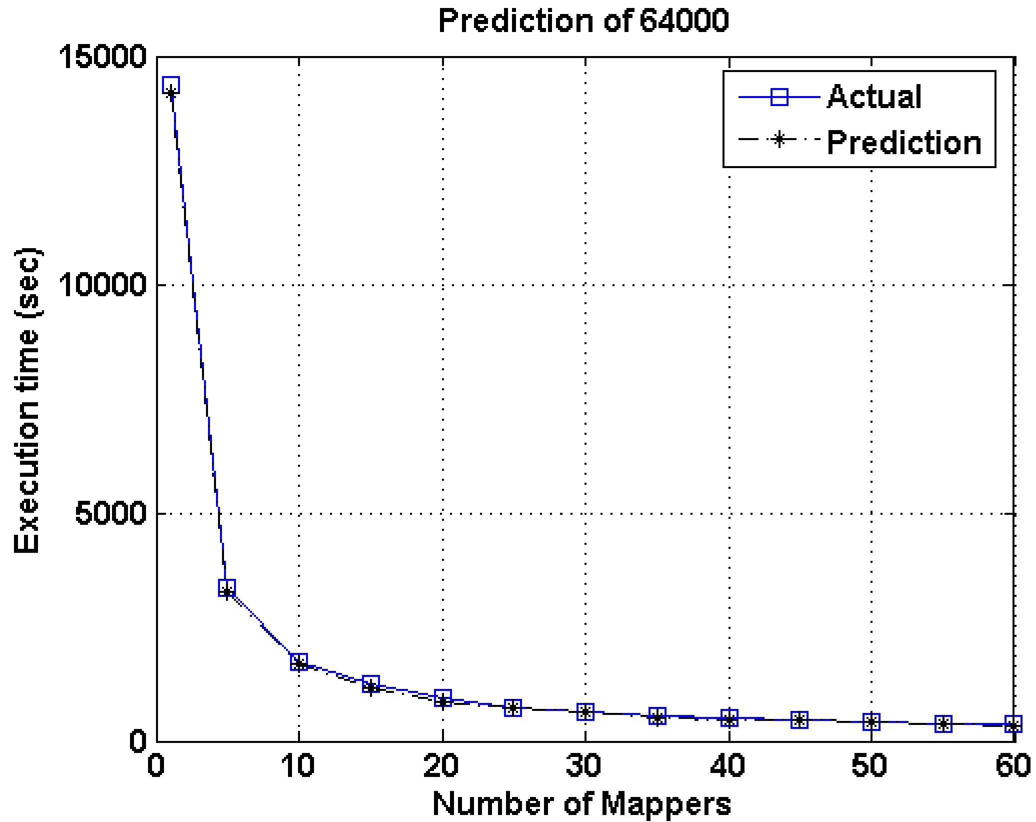


Figure 4.8: Prediction vs. Actual time-3

Because the time spent in the Map stage is a function of number of mappers and number of atoms and it dominates the total programs execution time, we will give a detailed function of execution time in the Map stage with two variables. The formula is shown below:

$$T(n,m) = \frac{c_2 n^2}{m} + \frac{c_1 n}{m} + \varphi(m) \quad (4.9)$$

We use the same method as estimating coefficient d and obtained three estimated c_1 and c_2. Table 4.4 shows the coefficient estimation results in the Map and the Reduce stage:

Table 4.4 Estimation of the Map stage

No. particles	c_1	c_2	$\varphi(m)$	c_1^r	c_0^r	Variance
42875	3.19E-06	0.02536	8	3.02E-4	4.98	17%
54872	2.93E-06	0.0338	8	2.93E-4	5.18	15%
64000	3.01E-06	0.0306	8	2.89E-4	5.04	11%
Average	3.04E-06	0.0299	8	2.95E-04	5.06	

The maximum variance is the difference between actual value and value obtained from formula with estimated coefficients divided by actual value. It is larger than our whole program estimation because we do the curve fitting twice. One is in coefficient d 's estimation. The other is in c_1 and c_2 estimation based on d 's estimation. It involves extra error. In the Reduce stage, execution time is linear with the number of atoms. c_1^r and c_0^r are the estimation of coefficients in equation 4.4. We take the average of three times estimation and use the average as our final coefficients.

$$T(n,m) = \frac{0.00000304n^2}{m} + (\frac{0.0299}{m} + 0.000295)n + 13.06 \tag{4.10}$$

For any given number of atoms in simulation system, we can obtain MDMR's total execution time by equation 4.10 within variance of 11%.

4.3 Run-time Program Monitor

The platform we used to do these experiments is composed of 11 worker nodes. Since we have 22 CPUs in evaluating our run-time monitor, the number of mappers in our test cases is the integer times 22. We chose three simulation systems that respectively contain 1000, 8000 and 27,000 particles.

The MD simulation is computation-intensive. MDMR's *map()* method may take more time than the reduce*()* method. In the MD simulation, the counter of *map()* and *reduce()* are the same as the number of input particles. Thus we do not include this result.

Table 4.5: Monitor Overhead of MD simulation

No. particles	**No. of Mappers**	T	$M_{overhead}$	**Overhead/total time**
1000	22	509	52	0.10
	44	539	18	0.03
	66	606	63	0.10
	88	695	58	0.08
	110	737	52	0.07
8000	22	1011	93	0.09
	44	1023	91	0.09
	66	1124	38	0.03
	88	1122	60	0.05
	110	1239	38	0.03
27000	22	3770	466	0.12
	44	3966	190	0.05
	66	3619	654	0.18
	88	3639	556	0.15
	110	3963	299	0.08

In Table 4.5, we obtain that with an increasing simulation system size, the overhead goes up correspondingly. The reason is that the monitoring times rise if the number of particles in the simulation system increases. If we fix the simulation system size but increase the number of mappers, the execution time also becomes longer. This is caused by the MD simulation program itself. For every *mapper*, the program needs to load all atoms information into memory, and then do the assigned atoms' simulation. Increasing the number of *mappers* is equivalent to increasing the time of loading the information of all atoms.

Table 4.6: MD simulation Efficiency

No. atoms	**No. of Mappers**	$C_{map} + C_{redu}$	C_{total}	$\eta_C = \frac{C_{map} + C_{redu}}{C_{total}}$
1000	22	7.88	77.52	0.10
	44	8.8	85.08	0.10
	66	9.32	113.0	0.08
	88	9.72	137.0	0.07
	110	9.92	153.5	0.06
8000	22	108.22	207.0	0.52
	44	112.47	231.5	0.49
	66	116.92	261.3	0.45
	88	123.85	273.8	0.45
	110	119.43	301.9	0.40
27000	22	857.8	925.3	0.93
	44	889.57	1089.	0.82
	66	907.9	1139.	0.80
	88	920.28	1203.	0.76
	110	923.23	1241.	0.74

Table 4.6 shows the computation power efficiency we defined in equations 3.13 and 3.14. With the help of AspectJ, we can easily get the computation power of the *map()* and *reduce()* functions. This is very useful for a programmer to profile and tune their MapReduce program. And the efficiency of the MapReduce program presents the trend of the best performance program. For example, in our MD simulation, the efficiency increases with the size of simulation system. That means if a programmer does not want to waste cluster's computation power, the larger system is the first choice.

In Figure 4.9, we can clearly get another useful fact. It is the relation between the number of mappers and the efficiency. With an increasing number of mappers, the efficiency goes down gradually. As we explained before, it is caused by the overhead of loading the input file. From Figure 4.9, we conclude using a smaller number of mappers to simulate a large system can obtain better computation power efficiency.

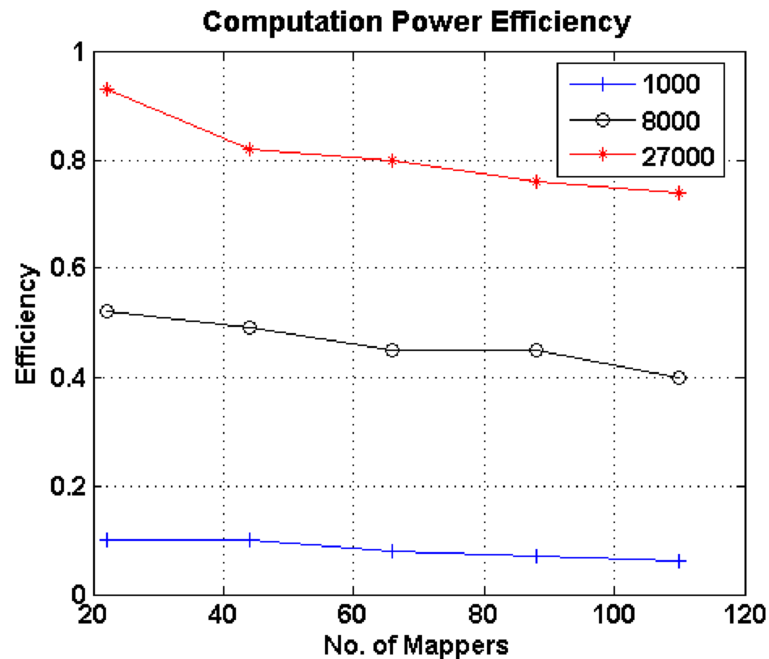


Figure 4.9: Computation Power Efficiency

4.4 MDMR-G performance on Hybrid MapReduce Cluster

In the MDMR-G (MDMR with GPU) performance evaluation, we construct a new Hadoop MapReduce cluster which contains three PCs because previous clusters nodes are old and do not have a PCI-Express slot for GPU cards. The hybrid cluster detailed information is listed below:

- Head node: 2 AMD 2.2GHz CPU, 4GB DDR RAM, 800GB HD, 1Gbps Ethernet.

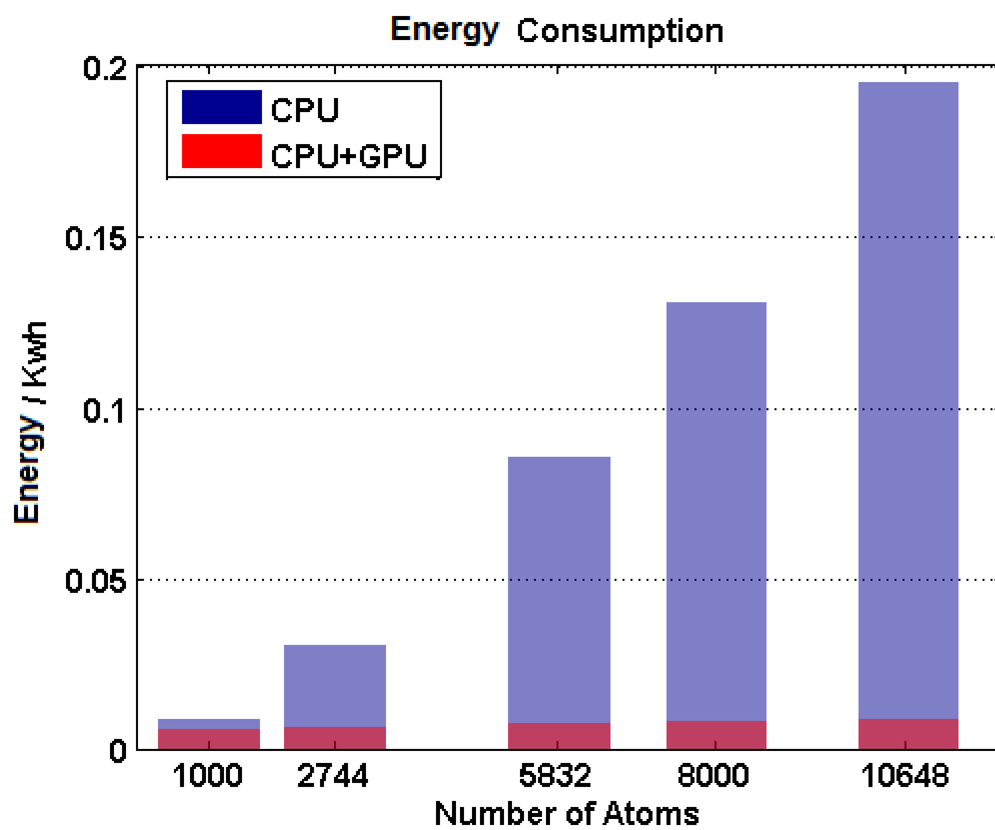
- Worker node: AMD 2.3GHz CPU, 2GB DDR2 RAM, 400GB HD, 1Gbps Ethernet
- Graphic Card: NVIDIA 9400GT 64bit 512Mb GDDR3 RAM ($20)
- Operating System: CentOS 5.5 (Linux 2.6.18, x86 64, SMP)
- Hadoop: 0.20.3 (stable)
- CUDA: Tookit 3.2 and x86 64-260.19.21 graphics driver
- Power monitor: ServerTech CWG-CDU power distribution unit

We chose five simulation systems in different sizes for our MDMR-G evaluation; each simulation system was executed 3 times to avoid randomness. In order to make the energy consumption easy to measure, we simulate every system for 10 time steps. Table 4.7 shows the execution time and energy consumption in CPU only and CPU+GPU environments.

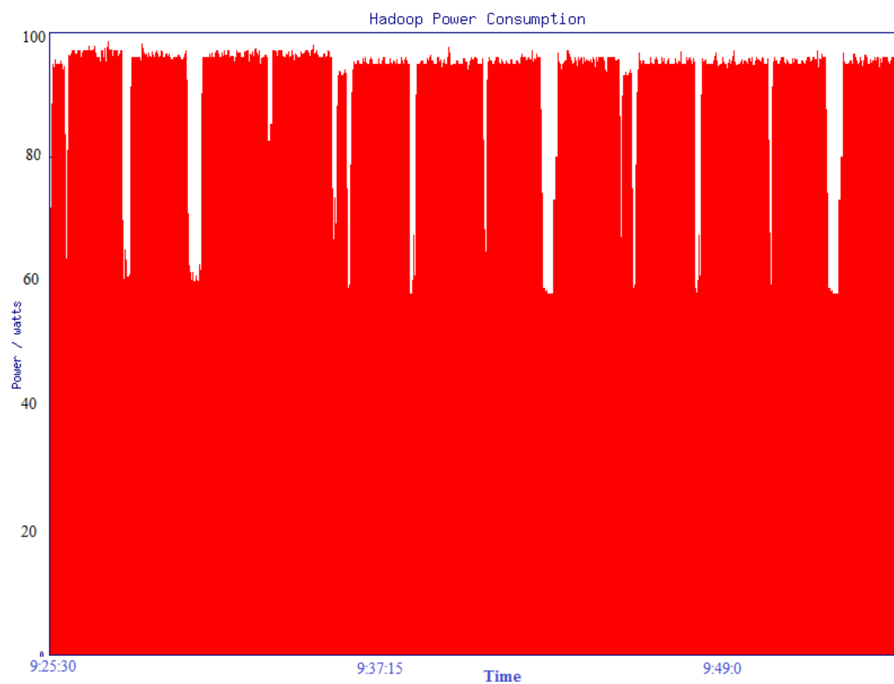
Table 4.7: MDMR-G results

Metrics\Size	**1000**	**2744**	**5832**	**8000**	**10648**
ExeTime\second(CPU)	209	617	1689	2561	3787
ExeTime\second(hybrid)	130	154	167	170	195
Energy \Kwh(Kwh)	0.0096	0.0309	0.0857	0.1305	0.1947
Energy \Kwh(hybrid)	0.0067	0.0072	0.008	0.085	0.096

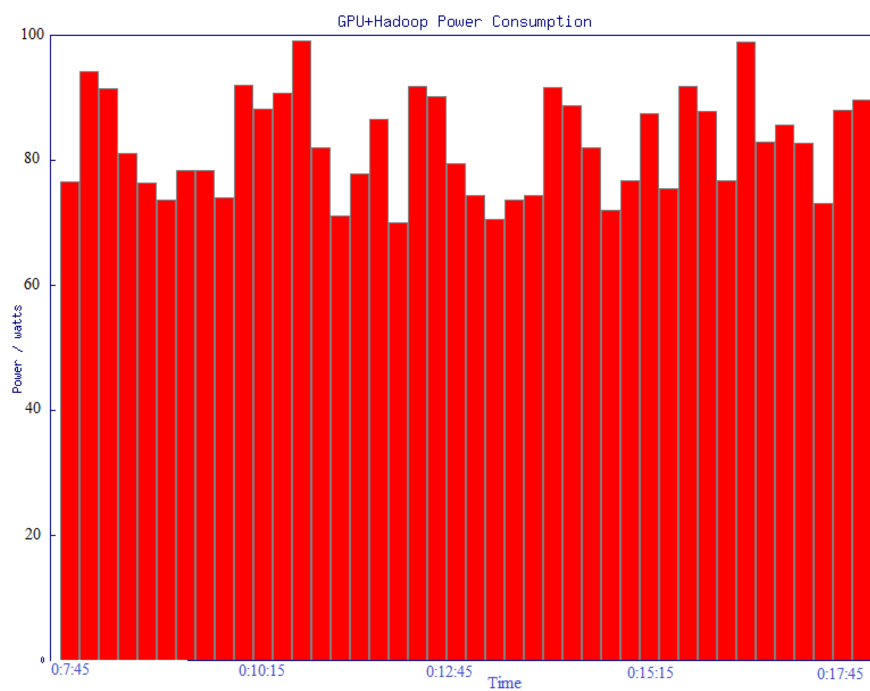
The worker node energy consumption will increase if we add an extra graphics card on the motherboard. However, the energy consumption has been reduced 95% in the simulation system with 10648 atoms because the execution time has been significantly decreased. The energy consumption Figure 4.10a verified our explanation. The Figure 4.10b is the MDMR power consumption, the lowest point in this figure is about 60 watts, which is the idle energy consumption. In the Figure 4.10c, the idle power consumption is about 70 watts; these 10 watts are caused by the newly added Geforce 9400GT graphics card.



a



b



c

Figure 4.10: MDMR-G Energy and Power consumption

The Figure 4.11satisfied our time complexity expectations.

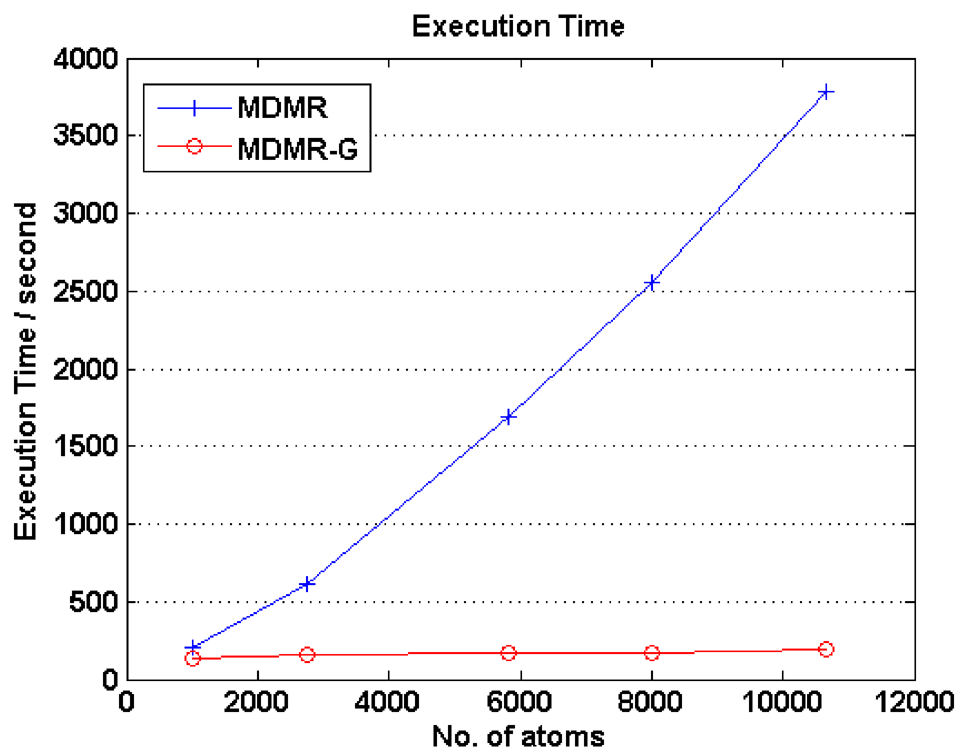


Figure 4.11: MDMR-G Execution time

The blue line is the MDMR program's execution time with quadratic trend. The red line is the MDMR-G program's execution time in linear manner.

Compared with MDMR, MDMR-G achieves promising speedup. In order to objectively evaluate MDMR-G's performance, we also obtain MDMR-G's speedup in a different simulation system. We take one mapper and one reducer on the same node with one GPGPU as our serial baseline. Figure 4.12 shows the speedup of using 3 worker nodes in our hybrid MapReduce framework.

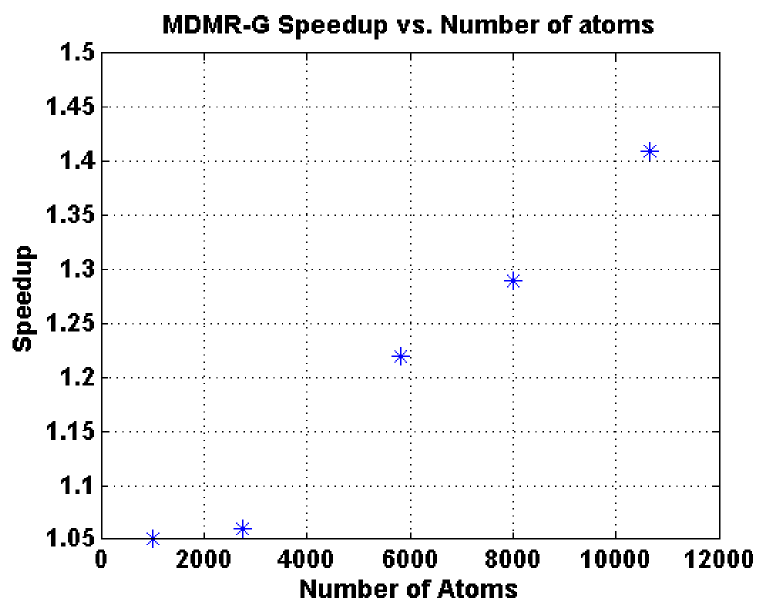


Figure 4.12: MDMR-G Speedup

And Figure 4.13 demonstrates MDMR-G's Karp-Flatt Metric value.

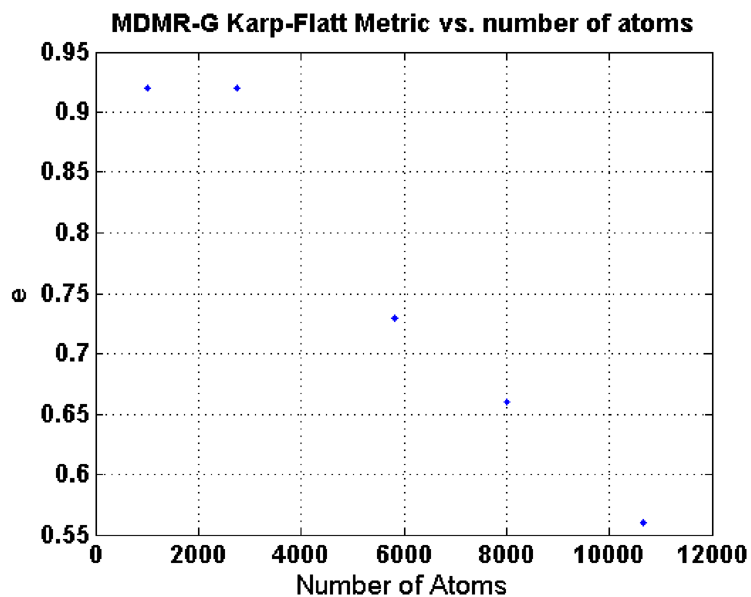


Figure 4.13: MDMR-G Karp-Flatt Metric

The Karp-Flatt Metric is also called experimentally determined serial fraction. We can conclude from Figure 4.13 that our MDMR-G is good at larger simulation system.

Because with the increasing simulation system size, the serial portion which includes the program's serial execution percentage and the parallelization overhead is decreasing.

Chapter 5

Conclusion

In this thesis, we parallelized a MD simulation called MDMR using the MapReduce programming model; at the same time, we predict the MDMR execution time by evaluating its execution based on its time complexity. We obtain 30.5 times speedup in maximum comparing with serial MD simulation using 60 mappers. Furthermore, we improve MDMR performance by introducing it into a hybrid MapReduce cluster with GPGPU. A run-time MapReduce program monitor has been developed to verify the computation energy efficiency of MDMR. We evaluate our work in previous chapters. In this chapter, we summarize our major contributions of this thesis.

The major contributions of our work are listed as follows:

1. We create MDMR which is a communication-free and every time-step fault-tolerant parallel implementation of MD simulation based on Hadoop MapReduce. We emulate the execution of MDMR and provide and evaluate the prediction formula of its execution time. Compared with serial MD simulation, MDMR achieves 30.5 times speedup in maximum using 60 mappers.

2. We create a run-time MapReduce program monitor which can monitor the execution time of *map()* and *reduce()* function, and then obtain the computational energy efficiency of a given MapReduce program. This can help a MapReduce programmer find the bottleneck of their MapReduce programs and give some hints for the improvement of their algorithm.

3. We develop MDMR-G which introduces CUDA and jCUDA to accelerate the program execution on a hybrid MapReduce cluster where each node has CUDA ready GPGPU. We achieve at most 20 times speedup comparing with the MapReduce cluster without any accelerator. MapReduce cluster energy consumption is reduced by 95%, and the speedup can be larger if larger systems are included.

Chapter 6

Future Work

As we mentioned in Chapter 3, we will develop a scheduler which can adjust the work load to make all TaskTrackers finish the tasks of a given job nearly at the same time.

MDMR-G presents its superiority not only in the execution time, but also in the energy consumption. We will focus on accelerator embedded MapReduce clusters in the future. The first step is to balance the tasks among the heterogeneous MapReduce cluster which is composed of non-GPGPU nodes and nodes with GPGPU. Secondly, GPGPU scheduling is challenging if nodes have GPGPUs with different computational power.

Bibliography

[1] D. Jeffrey, and G. Sanjay, *MapReduce: simplified data processing on large clusters.* Commun. ACM, 51(1), (2008), 107-113.

[2] K. Cardona, J. Secretan, M. Georgiopoulos, and G. Anagnostopoulos, *A grid based system for data mining using MapReduce*. Technical Report TR-2007-02,AMALTHEA, (2007).

[3] J. Ekanayake, A. Balkir, et. al. *MapReduce for Data Intensive Scientific Analyses*, 2008 4th IEEE International Conference on eScience, (2008).

[4] M.C. Schatz, *BlastReduce: high performance short read mapping with MapReduce*. *http://www.cbcb.umd.edu/software/blastreduce/.*

[5] W. Shang, Z. M. Jiang, B. Adams, and A. E. Hassan. *Mapreduce as a general framework to support research in mining software repositories (MSR)*. In MSR ’09: Proceedings of 6th IEEE International Working Conference on Mining Software Repositories, (2009) 21–30.

[6] J. You, J. Xi, P. Zhang, and H. Chen. *A Parallel Algorithm for Closed Cube Computation*. ICIS, (2008).

[7] B. Wu, S. Yang, H. Zhao, B. Wang, *A Distributed Algorithm to Enumerate All Maximal Cliques*, MapReducFrontier of Computer Science and Technology, 2009. FCST '09, (2009).

[8] C. Jin, C. Vecchiola, R. Buyya, *MRPGA: An extension of mapreduce for parallelizing genetic algorithms.* In: Press, I. (ed.) IEEE Fouth International Conference on eScience (2008), 214–221.

[9] Impetus http://www.impetus.com.

[10] Cloudera http://www.cloudera.com.

[11] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, *Mining Console Logs for Large-scale System Problem Detection*. In SysML, (2008).

[12] J. Tan, X. Pan, S. Kauvlya, et al., *Mochi: Visual Log-Analysis Based Tools for Debugging Hadoop*, HotCloud' (2009).

[13] D. Huang, et al. *MR-Scope:A Real-Time Tracing Tool for MapReduce*, HPDC 2010. (2010).

[14] J. Polo, D. Carrera, Y. Becerra, V. Beltran, J. Torres and E. Ayguadé Performance *Management of Accelerated MapReduce Workloads in Heterogeneous Clusters*, ICPP2010, (2010), 654-662.

[15] J.M. Haile, *Molecular Dynamics Simulation: Elementary Methods, 1st edition*, ISBN:0471819662, (1992).

[16] C. He, D. Swanson. *Molecular Dynamics simulation based on MapReduce*, poster section, *LCI 2010*, (2010).

[17] Hadoop, *http://www.hadoop.com.*

[18] HDFS, *http://hadoop.apache.org/hdfs/.*

[19] nVIDIA CUDA *http://developer.nvidia.com/object/cuda-3.2/downloads.html.*

[20] M. J. Quinn., *Parallel Programming in C with MPI and OpenMP*, ISBN 0-07-282256-2, (2004)

[21] S. Gupta. *Computing aspects of molecular dynamics simulations*. In J.Comp.Phys.Comm., volume 70, (1992), 243-270.

[22] K. Balusubramanian and K. Pitzer. *Ab Initio Methods in Quantum Chemistry, chapter Part I*. Wiley and Sons Ltd., New York, (1987).

[23] M. P. Allen and D. J. Tildesley, *Computer simulation of liquids.* Oxford University Press, (1987).

[24] L. Verlet, *Computer experiments on classical fluids i. thermodynamical properties of lennard-jones molecules*. In Phys. Rev., vol. 159, (1967), 98-103.

[25] nVIDIA *http://www.nvidia.com.*

[26] JCUDA *http://jcuda.org.*

[27] J.V., Sumanth, dissertation, *Adaptive Scheduling of MD Simulations in Parallel and Distributed Environments*. (2007).

[28] B. Hendrickson and S. Plimpton. *Parallel many-body simulations without all-to-all communication.* J. Parallel Distrib. Comput., 27(1), (1995), 15–25.

[29] G. Chen, U Ros, *MOP: An Efficient and Generic Runtime Verification Framework*, Object- Oriented Programming, Systems, Languages and Applications *(OOPSLA'07)*, (2007), 569-588.

[30] JavaMOP,*http://fsl.cs.uiuc.edu/index.php/MOP.*

[31] Ganglia *http://ganglia.org.*

[32] Top 500 *http://www.top500.org/.*

[33] Green Top 500 *http://www.green500.org/lists/2010/11/top/list.php.*

[34] Y. Becerra, V. Beltran, D. Carrera, M. Gonz´alez, J. Torres, and E. Ayguad´e, *Speeding up distributed mapreduce applications using hardware accelerators*, in *38th International Conference on Parallel Processing (ICPP)*, (2009).

[35] Matlab curve fitting tool, *http://www.mathworks.com/help/toolbox/curvefit/cftool.html*.

[36] Bingsheng He, Wenbin Fang *Mars: a MapReduce framework on graphics processors*. *PACT* ,2008

[37] Condor Project, http://www.cs.wisc.edu/condor/

[38] Grid Computing, http://en.wikipedia.org/wiki/Grid_computing

[39]T. W. Clark, R. v. Hanxleden, J. A. McCammon, and L. R. Scott. *Parallelizing molecular dynamics using spatial decomposition.* In Proceedings of the Scalable High–Performance Computing Conference, pages 95–102. IEEE Computer Soc. Press, 1994.

[40]Jianhui Li, Zhongwu Zhou, and Richard J. Sadus. *A cyclic force decomposition algorithm for parallelising three-body interactions in molecular dynamics simulations*. In Ni and Dongarra , pages 338–343.

[41]Steve Plimpton, Bruce Hendrickson, and Grant Heffelfinger. A new decomposition strategy for parallel bonded molecular dynamics. In PPSC, pages 178–182,1993.