MOLECULAR DYNAMICS SIMULATION

By

LAKSHMIKANTH GANTI
B.Tech., Indian Institute of Technology, Madras, India.
May, 2000

A PORTFOLIO

Submitted in partial fulfillment of the requirements for the degree

MASTER OF SOFTWARE ENGINEERING

Department of Computing and Information Sciences
College of Engineering

Kansas State University
Manhattan, Kansas
2004

Approved by:

Major Professor
Dr. Virgil Wallentine

ABSTRACT

Molecular Dynamics Simulation is an extremely powerful technique which involves solving the many-body problem in contexts relevant to the study of matter at the atomic level. The method allows the prediction of the static and dynamic properties of substances directly from the underlying interactions between the molecules. Because there is no alternative approach capable of handling such a broad range of problems at the required level of detail, molecular dynamics methods have proved themselves indispensable in both pure and applied research. Molecular Dynamics Simulations are computationally very intensive and hence an ideal application of Parallel Programming concepts.

The purpose of this project is to develop software in Java that uses MD Simulation technique to simulate the interaction between atoms in a group of molecules which interact due to *Lennard-Jones* potential (or any other similar system whose motion can be simulated by stepping through discrete instants of time).

Multi-threaded programming that can be executed on more than one processor is used to improve the efficiency of the system. Different parallel algorithms based on 1) synchronization mechanism, 2) the pattern of thread creation and 3) Granularity, were implemented and performance measurements were done on them to predict the best possible combination for a system such as Molecular Dynamics Simulation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I sincerely thank Dr. Virgil Wallentine, my major professor, for giving me timely guidance, encouragement and facilities to complete the project. I also thank him for being flexible and accomodating during the course of this project.

I would like to thank Dr. Paul Smith and Dr. Mitch Neilsen for serving in my project committee.

I would like to thank Ms. Delores Winfough for helping me understand the policies and procedures of Graduation.

I would like to thank the Systems Administrators in CIS for being very prompt in providing access to the necessary servers and in trouble shooting.

CHAPTER 1: VISION DOCUMENT

1. Introduction

1.1. Motivation

Molecular Dynamics (MD) Simulation is an extremely powerful technique which involves solving the many-body problem in contexts relevant to the study of matter at the atomic level. The method allows the prediction of the static and dynamic properties of substances directly from the underlying interactions between the molecules. Because there is no alternative approach capable of handling such a broad range of problems at the required level of detail, molecular dynamics methods have proved themselves indispensable in both pure and applied research. However, Molecular Dynamics Simulations are computationally very intensive and hence an ideal application of Parallel Programming concepts. These ideas motivated me to use my knowledge of parallel programming to develop a software for MD Simulation which can run on multiple processors and hence computationally efficient.

1.2. Molecular Dynamics Simulation

Molecular dynamics [1] simulation is a technique where the time evolution of a set of atoms is followed by integrating their equations of motion. In molecular dynamics we follow the laws of classical mechanics, and most notably Newton's law: $F_i = m_i a_i$ for each atom i in a system constituted by N atoms. Here, m_i is the atom mass, $a_i = d^2r_i/dt^2$ its acceleration, and F_i, the force acting upon it, due to the interaction with other atoms.

2. Project Overview

2.1. Purpose

The purpose of this project is to develop software in Java that uses MD Simulation technique to simulate the interaction between atoms in a group of molecules (or any other similar system whose motion can be simulated by stepping through discrete instants of time).

2.2. Goals

The goals of this project are to develop robust and efficient software, enhance the usability of the system with good documentation of the design and the overall system, and make the system as self sufficient as possible and unambiguous specification of the constraints under which the system will work.

2.3. Direction

The interaction force existing between molecules considered here is called the Lennard-jones potential. The Lennard-Jones [2] potential is mildly attractive as two uncharged molecules or atoms approach one another from a distance, but strongly repulsive when they approach too close. The resulting potential is shown in Figure below. At equilibrium, the pair of atoms or molecules tends to go toward a separation corresponding to the minimum of the Lennard-Jones potential (a separation of 0.39 nanometers for the case shown in the figure below.)

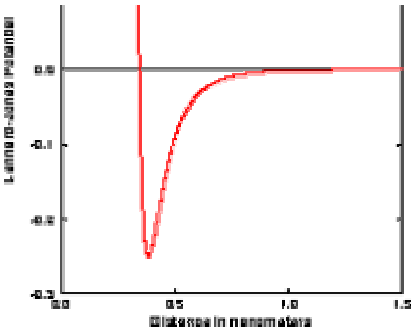


Figure 1 : Lennard-Jones potential

The potential resulting from these attractive and repulsive interactions is called the Lennard--Jones potential and is described by the following equation:

$$V_{LJ} = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \qquad (1)$$

σ and ϵ are the specific Lennard--Jones parameters, different for different interacting particles. **r** is the distance between the interacting particles. For the current system, the values of these parameters are: σ = 0.3 Nanometers and ε = 1.0 KJ/mole. The Lennard--Jones force between two atoms is given by the equation:

$$F_{LJ} = -24\epsilon \left[2\left(\frac{\sigma^{12}}{r^{13}}\right) - \left(\frac{\sigma^{6}}{r^{7}}\right) \right] \qquad (2)$$

This interaction force existing between the atoms causes them to accelerate and move. This system is simulated in small time steps. At each time step we have to calculate the Lennard-jones's force on each atom due to the interaction with all the other atoms and update its velocity and position. These interactions are effective until a certain length called the interaction length. Here it's taken as 1.0 Nanometers.

The software takes input from three different files: 1) a data file (.dat) which supplies parameters like the total number of particles, mass of each particle, number of dynamics steps etc. The file lists in a specific format the value of the parameter, the variable name used to hold the parameter and a brief description of the parameter. 2) A file in the .pdb (protein data bank) format [3] that supplies the program with the initial coordinates of the atoms in space. 3) A data file from which the initial velocities of particles in all dimensions could be read by the program. The velocities are read from this file only for testing purposes. Otherwise, the program calculates the velocities for all the atoms based on a random Gaussian distribution. Various string handling methods of the Java language are used to extract the exact numerical values from the formatted input files.

The program calculates the force on each atom due to all the atoms that are within the interaction length. This force is used to update the velocities and positions and calculate the potential energy, kinetic energy of the particles and temperature of the system at every time step. At the end of the simulation, the averages and fluctuations of each of these quantities are calculated. These values are displayed on the console for every certain number of time steps (indicated in the input data file). This data along with the averages and fluctuations is written to an output file and the final x, y, z coordinates of all the particles in system is written to a file in the pdb format.

2.4. Features

Multi-threaded programming that can be executed on more than one processor will be used to improve the efficiency of the system. These parallel programs will be implemented using different designs [4] based on,

- Synchronization mechanism i.e. Message Passing versus Barrier, Monitor etc.
- The pattern of thread creation i.e. grid shaped where each thread only communicates only with its neighbors versus a vertical pipeline where each thread communicates with its upper and bottom neighbors.
- Granularity i.e. Course-grained versus fine-grained, which is determined by the frequency of thread synchronization or communication relative to the amount of computation done.

The performance of these parallel programs will be compared to predict the best suited design for a system such as the Molecular Dynamics Simulation.

2.5 Risks

Since the project is based extensively on parallel programming, all the risks inherent to concurrent programs apply to this project. Some of them are 1) Safety – Parallel activities interfering with each other. The programmer must maintain crucial invariants to avoid this situation. 2) Liveness - The programmer must make sure that all threads don't just stop, without finishing their work. 3) Deadlock – The programmer should avoid a deadlock situation where every thread is waiting on each other to make progress, thus no progress is made. Also, one of the biggest challenges in this project is to

achieve the maximum speed-up and efficiency for the parallel programs especially when run on a distributed system.

CHAPTER 2: SOFTWARE REQUIREMENTS SPECIFICATION

1. Introduction

1.1 Purpose

The purpose of this chapter is to specify requirements to explain the behavior of the proposed software system. The audience of this chapter is physics, biochemistry and software researchers, designers, and students who are interested in applying Molecular Dynamics Simulation techniques to simulate physical systems.

1.2 Overview

The purpose of this project is to develop a software package that uses Molecular dynamics simulation techniques to simulate the interaction between the atoms in a group of molecules (or any other similar system whose motion can be simulated by stepping through discrete instants of time).

1.3 Scope

The software developed in this project will display the coordinates, velocities and the physical properties of the system such as potential energy, temperature as a function of time, starting form a given initial configuration. Algorithms and Software Patterns used might not be suitable for all kinds of systems. For example this software uses the Particle-Particle method for the simulation, which is one of the three scientific software patterns used in dynamic systems. Appropriateness of the model should be considered before using this software.

1.4 Definitions, Acronyms and Abbreviations

Molecular Dynamics Terms:

Molecular Dynamics Simulation: A technique where the time evolution of a set of atoms is followed by integrating their equations of motion.

Lennard-jones potential: An interaction potential existing between atoms which are considered here.

PDB: Protein Data Bank

Potential Energy: The energy resulting from position or configuration of an atom.

Kinetic Energy: The energy resulting from motion of an atom.

Velocity: The rate of motion of an atom in a particular direction.

Temperature: A measure of the Kinetic energy in atoms of a substance.

Cut-off Distance: Distance between the atoms above which there are no interaction forces.

Software Terms:

Pattern: Extension of Object-oriented methods of analysis and design

SLOC: Source Lines of Code

IEEE: Institute for Electrical and Electronic Engineers

SRS: Software Requirements Specifications

SQA: Software Quality Assurance

2. Overall Description

2.1 Product Perspective

2.1.1 Approach

There are three software patterns [5] [6] available for dynamic systems simulation, Particle-Particle (PP) method, Particle-Mesh (PM) method and Particle-Particle –Particle-Mesh (P3M) method. For this project, the PP method is used. Various designs for a parallel program based on 1) Synchronization mechanism, 2) the pattern of thread creation and 3) Granularity, are implemented. Performance measurements such as measuring the speed up as a function of the number of threads and granularity were done on each design.

2.1.2 Applications

There are numerous applications of Molecular Dynamics in many fields of study, for e.g. Biopolymers, Biomedicine, and Biochemistry etc. MD Simulations allow prediction of properties for novel materials, which have not yet been synthesized, and for existing materials whose properties are difficult to measure or poorly understood. The results of the current simulation of molecules are used in protein folding studies.

2.1.3 Constraints

1. Data is read from three files, which should be in a specific format.
    - The Data file (.dat) should have the value of the parameter as first ten characters, the variable used for that parameter as next eight characters and the description of the parameter on rest of the line.
    - The file from which velocities are read (.in) should display the velocity in x – direction as the first twenty five characters, velocity in y – directions as next twenty five and velocity in z- direction as the last twenty five characters in a line. All the three values for an atom should be in one line.
    - The file from which the coordinates are read (.pdb) should be in the protein data bank format for atomic coordinate files.
2. Stability and scalability issues are not mentioned extensively.

2.2 Product Functions

assign: Assigns the atoms to the partitions in a thread, depending on their coordinates.

force: calculates the force on an atom due to all the other atoms within the cut-off distance and updates the potential energy of the system.

calculateEnergies: calculates the potential, kinetic and the total energies at the current step of the simulation.

incrementVel: increments the velocity of the atoms depending on the forces of interaction at each step.

displace: displaces the atoms depending on the increment in velocity at each step.

CalculateAvgsAndFlucs: calculates the average energies and the fluctuations in energies during the entire simulation.

printEnergies: prints the kinetic energy, potential energy, total energy and the temperature of the system for every given number of steps.

2.3 User Characteristics

This Product is developed for applications in scientific computing involving molecular dynamics simulations. Therefore, the user is assumed to have necessary background in solving equations of motion and molecular dynamics simulations. It is also assumed that the user has basic computing knowledge and Java Programming background.

2.4 Assumptions and Dependencies

It is assumed that user has basic background has discussed under User Characteristics and JDK version 1.3 or above is installed.

2.5 Apportioning of Requirements

In future version implementations of the simulation programs, message passing interface could be implemented so that the program could be run on a distributed system with more number of processors to increase performance.

3. Specific Requirements

3.1 External Interface Requirements

3.1.1 User Interfaces

- Input screen to enter the program arguments such as the number of threads to be created.
- Screen to display the results of simulation.

3.1.2 Hardware Interface

As the application is developed in java, it is platform independent.

3.1.3 Software Interface

- Java JDK Library.

3.2 Classes/Attributes

The figure on the following page shows the Object Model/ Class Diagram of the project. The description of all the classes and their functions has been outlined in the Chapter 8, Component Design.

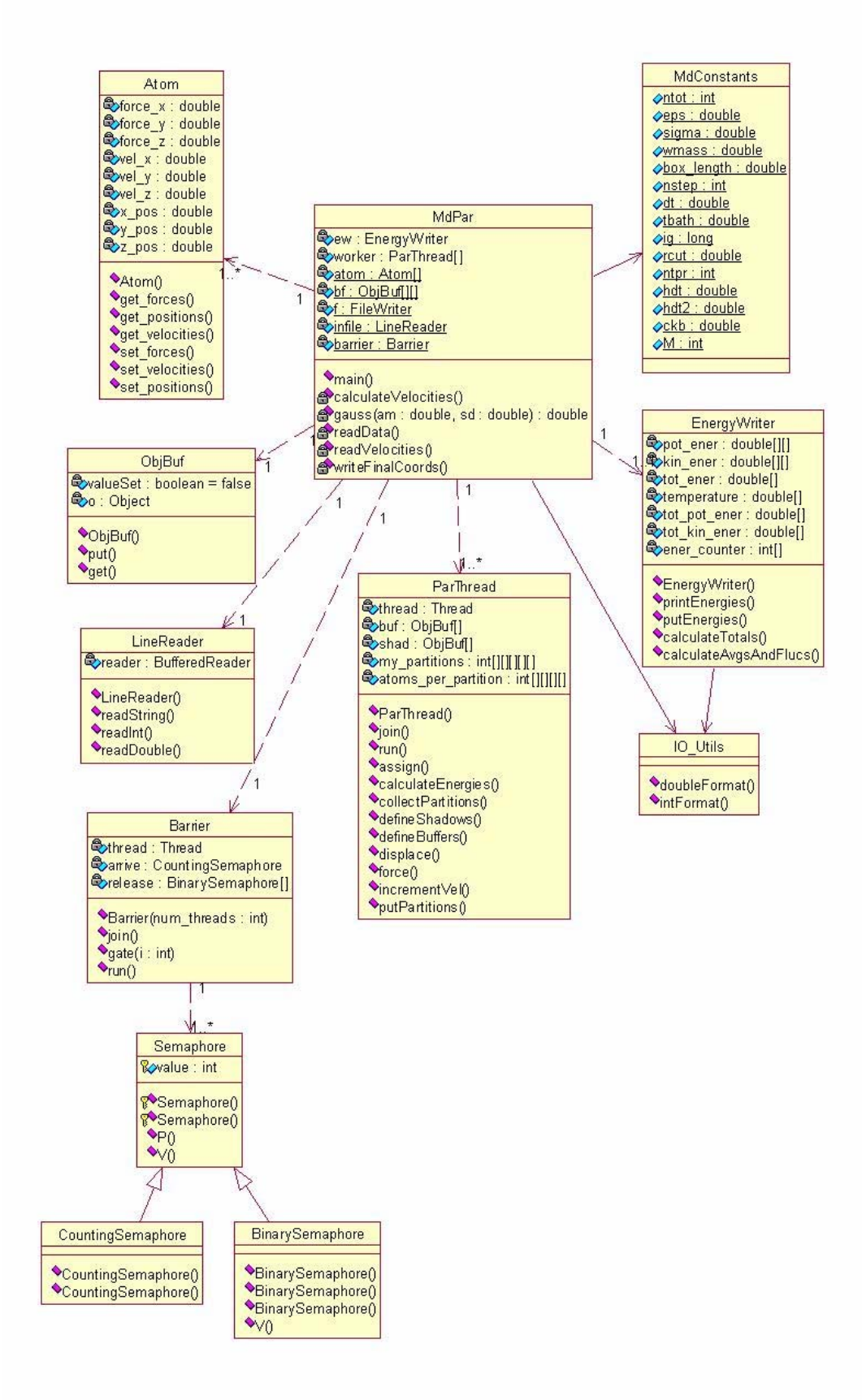

Atom
force_x : double
force_y : double
force_z : double
vel_x : double
vel_y : double
vel_z : double
x_pos : double
y_pos : double
z_pos : double
Atom()
get_forces()
get_positions()
get_velocities()
set_forces()
set_velocities()
set_positions()
MdPar
ew : EnergyWriter
worker : ParThread[]
atom : Atom[]
bf : ObjBuf[][]
f : FileWriter
infile : LineReader
barrier : Barrier
main()
calculateVelocities()
gauss(am : double, sd : double) : double
readData()
readVelocities()
writeFinalCoords()
MdConstants
ntot : int
eps : double
sigma : double
wmass : double
box_length : double
nstep : int
dt : double
tbath : double
ig : long
rcut : double
ntpr : int
hdt : double
hdt2 : double
ckb : double
M : int
ObjBuf
valueSet : boolean = false
o : Object
ObjBuf()
put()
get()
EnergyWriter
pot_ener : double[][]
kin_ener : double[][]
tot_ener : double[]
temperature : double[]
tot_pot_ener : double[]
tot_kin_ener : double[]
ener_counter : int[]
EnergyWriter()
printEnergies()
putEnergies()
calculateTotals()
calculateAvgsAndFlucs()
ParThread
thread : Thread
buf : ObjBuf[]
shad : ObjBuf[]
my_partitions : int[][][][][]
atoms_per_partition : int[][][][]
ParThread()
join()
run()
assign()
calculateEnergies()
collectPartitions()
defineShadows()
defineBuffers()
displace()
force()
incrementVel()
putPartitions()
LineReader
reader : BufferedReader
LineReader()
readString()
readInt()
readDouble()
IO_Utils
doubleFormat()
intFormat()
Barrier
thread : Thread
arrive : CountingSemaphore
release : BinarySemaphore[]
Barrier(num_threads : int)
join()
gate(i : int)
run()
Semaphore
value : int
Semaphore()
Semaphore()
P()
V()
CountingSemaphore
CountingSemaphore()
CountingSemaphore()
BinarySemaphore
BinarySemaphore()
BinarySemaphore()
BinarySemaphore()
V()
1
1..*

Figure 2: Object Model

3.3 General Requirements

- To produce a neat interface so that it is easy for the user to understand and use.
- To produce statistical data to show the speed improvements.
- To produce API documentation in JavaDoc style, explaining the classes and their corresponding methods and attributes
- To produce a user manual explaining, with detailed instructions of how to use the application.
- Design specification document explaining all the design features of the application.
- Documented source code.
- SQA plan.
- Test plan.
- Project plan details.
- Object Model, showing classes and their relationships.
- Interaction diagrams.

3.4 Performance Requirements

- To be able to calculate the physical properties at different simulation steps correctly for system of any size.
- To achieve maximum speed-up and efficiency when executed on multiple processors.
- To be able to perform time wise comparison between implementations using different synchronization mechanisms.

- Time taken to run the parallel program on a multi processor machine should be less than the time taken to run on a single processor machine.
- To be able to read data from a file specified in certain format
- To be able to handle the stability constraints on data
- Handle perturbations in data
- Minimize memory usage

3.5 Hardware and Software Requirements

- Application will be developed in Java, facilitating use of the application in any platform with JDK version 1.3 or above installed in it.
- There are no special hardware requirements to use this application although a machine with processor speeds of more than 400 MHz is recommended for improved performance.

3.6 Critical Requirements

- The system should be free from deadlock i.e. where each thread waits on each other to make progress and thus no progress is made.
- The system should be safe i.e. there should not be any miscalculations by interference of threads activities with each other.
- The system should not violate simple assertions such as a thread gets the exact number of data items and their values from a buffer, which it is supposed to get.

It will be made sure that the above critical requirements are satisfied by the system by checking the synchronization model used here formally using *Java Path Finder*.

CHAPTER 3: PROJECT PLAN

1. Introduction

The success of any project depends very much on how well a Project Plan is set up. We need to know what the standard milestones or events for the project will be and plan the project accordingly. The most successful approach in planning a project is the Iterative Planning Approach, where the software is developed on an iterative basis with specific cost and schedule guidelines. The key planning elements include the Work Breakdown Structure, Cost Estimation and the Architecture Elaboration Plan.

2. Work Breakdown Structure

The Work Breakdown Structure displays and defines the tasks to be done in each iteration phase of the project life cycle. It should clearly describe each task and the completion criteria for each task in the life cycle. All artifacts are identified in the work breakdown structure and the completion criteria are determined. The different phases and the important artifacts that are produced in each phase are listed below:

2.1 Inception Phase

The Inception phase involves the development of a prototype that would establish the feasibility of the important or risky elements of the requirements and give users an idea about how the final product will look like. This phase also involves documentation that will be presented at the first presentation. These include a Vision document along with the requirements specification, Project plan and a Software Quality Assurance (SQA) Plan. The key requirements are finalized on the approval of the documents and the

executable prototype, by the committee after the first presentation. The changes that are recommended by the committee are identified as actions items for the next iteration.

2.2 Elaboration Phase

The Elaboration phase involves the development of an architectural baseline for the software product, keeping in mind the action items identified during the inception phase. The design is drafted using the overall architecture developed in the inception phase. Critical Use cases are designed which are used to develop the second executable prototype that will be demonstrated during the second presentation. The conclusion of this phase depends upon the approval of the committee that the executable prototype demonstrates all critical use cases. Some of the critical use cases would be increasing the system size i.e. the total number of molecules, running the parallel program with number of threads equal to one and maximum number of threads that the program could handle before throwing an out of memory exception.

2.3 Production Phase

The production phase involves coding the entire simulation system to satisfy all the requirements. Since the critical use cases are satisfied during the design phase, the production phase would involve building the remaining part of the system and integrating all the components so as to check their correctness.

2.4 Testing Phase

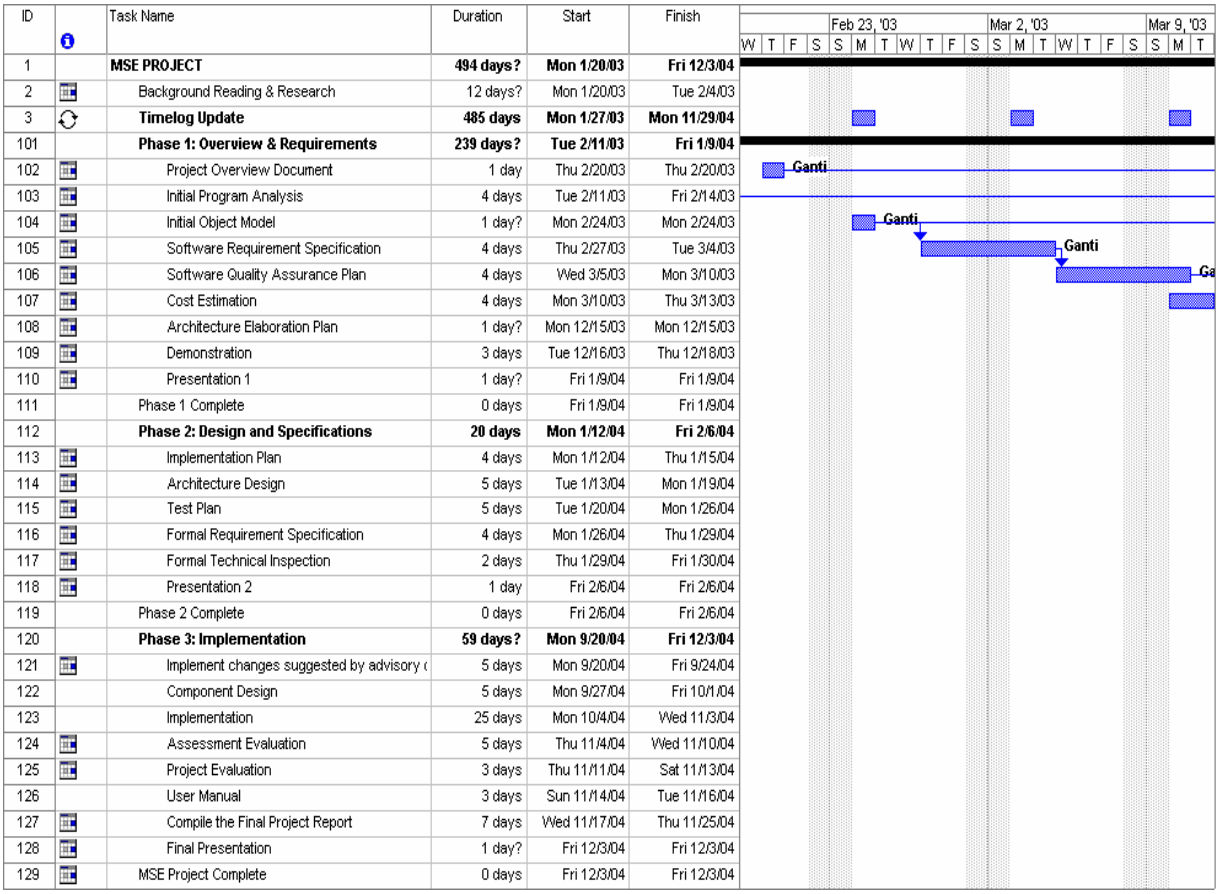
The testing phase involves testing the entire software system for correctness and performance. It is checked if all the critical use cases are satisfied. Unit testing and Integration Testing are performed during this phase. An error free software system marks the end of this phase.

2.5 Documentation Phase

This phase involves developing several artifacts that will be submitted along with the final software at the end of the third presentation. The documents include all the artifacts developed in each phase and the User Manual. The User Manual will demonstrate to the user, how to use the software and contains help and trouble shooting sections. A test report will be developed that would describe how the tests were conducted and their results. An evaluation report will also be written in this phase which would give a brief evaluation of the entire project and lessons learned. The end of the phase will be marked by the approval of the final version of the software and the documentation, by the committee.

3. Project Plan and Gantt chart

The following figure shows a snap shot of the project plan drafted in Microsoft Project and the accompanying Gantt chart.



ID		Task Name	Duration	Start	Finish
1		**MSE PROJECT**	**494 days?**	**Mon 1/20/03**	**Fri 12/3/04**
2		Background Reading & Research	12 days?	Mon 1/20/03	Tue 2/4/03
3		**Timelog Update**	**485 days**	**Mon 1/27/03**	**Mon 11/29/04**
101		**Phase 1: Overview & Requirements**	**239 days?**	**Tue 2/11/03**	**Fri 1/9/04**
102		Project Overview Document	1 day	Thu 2/20/03	Thu 2/20/03
103		Initial Program Analysis	4 days	Tue 2/11/03	Fri 2/14/03
104		Initial Object Model	1 day?	Mon 2/24/03	Mon 2/24/03
105		Software Requirement Specification	4 days	Thu 2/27/03	Tue 3/4/03
106		Software Quality Assurance Plan	4 days	Wed 3/5/03	Mon 3/10/03
107		Cost Estimation	4 days	Mon 3/10/03	Thu 3/13/03
108		Architecture Elaboration Plan	1 day?	Mon 12/15/03	Mon 12/15/03
109		Demonstration	3 days	Tue 12/16/03	Thu 12/18/03
110		Presentation 1	1 day?	Fri 1/9/04	Fri 1/9/04
111		Phase 1 Complete	0 days	Fri 1/9/04	Fri 1/9/04
112		**Phase 2: Design and Specifications**	**20 days**	**Mon 1/12/04**	**Fri 2/6/04**
113		Implementation Plan	4 days	Mon 1/12/04	Thu 1/15/04
114		Architecture Design	5 days	Tue 1/13/04	Mon 1/19/04
115		Test Plan	5 days	Tue 1/20/04	Mon 1/26/04
116		Formal Requirement Specification	4 days	Mon 1/26/04	Thu 1/29/04
117		Formal Technical Inspection	2 days	Thu 1/29/04	Fri 1/30/04
118		Presentation 2	1 day	Fri 2/6/04	Fri 2/6/04
119		Phase 2 Complete	0 days	Fri 2/6/04	Fri 2/6/04
120		**Phase 3: Implementation**	**59 days?**	**Mon 9/20/04**	**Fri 12/3/04**
121		Implement changes suggested by advisory (	5 days	Mon 9/20/04	Fri 9/24/04
122		Component Design	5 days	Mon 9/27/04	Fri 10/1/04
123		Implementation	25 days	Mon 10/4/04	Wed 11/3/04
124		Assessment Evaluation	5 days	Thu 11/4/04	Wed 11/10/04
125		Project Evaluation	3 days	Thu 11/11/04	Sat 11/13/04
126		User Manual	3 days	Sun 11/14/04	Tue 11/16/04
127		Compile the Final Project Report	7 days	Wed 11/17/04	Thu 11/25/04
128		Final Presentation	1 day?	Fri 12/3/04	Fri 12/3/04
129		MSE Project Complete	0 days	Fri 12/3/04	Fri 12/3/04

Figure 3: Gantt chart

4. Cost Estimation

In this project, Functional Point Analysis and COCOMO model [7] are used for estimating the size and cost of developing the application.

4.1 Functional Point Analysis

4.1.1 Program Features

- Outputs

Each user output that provides application-oriented information to the user is counted. In this context output refers to reports, screens, error messages, and so on. Individual data items within a report are not counted separately. For this application, we have three outputs 1) output data written to screen 2) output data written to a file and 3) final coordinates written to a file. All the outputs can be classified as simple.

- Inputs

They are each unique user data or control input that enters the application boundary and also updates (adds to changes, or deleted from) a logical internal file, data set, table or independent data item. Each input is uniquely formatted or processed portion. For this application, we have two inputs 1) a data file and 2) a file in .pdb format which has the initial coordinates. All the inputs can be classified as simple.

- Files

Each major logical group of user data or control information related to application. They may be one part of a large database or a separate file. For this application, there are four simple files: 1) md.dat, the input file from which data for the system is read, 2) init_positions, the input file from which the initial position coordinates of all the atoms in the system are read, 3) md.out, the output file to which the energies at every

given number of time steps is printed out, 4) final_positions, the output file to which the position coordinates of all the atoms of the system after the simulation are written.

- External Interfaces

All machine-readable interfaces (e.g. data files on tape or disk) that are used to transmit information to another system are counted. There are no external interfaces for this system.

- User Inquiries

An inquiry is defined as an online input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted. There are no user inquiries for this application.

4.1.2 Weights for features

	Simple	Average	Complex	Total
Outputs	4(3)	5 (0)	7(0)	12
Inquiries	4(0)	5(0)	7 (0)	0
Inputs	3(2)	4(0)	6(0)	6
Files	7(4)	10 (0)	15(0)	28
Interfaces	5(0)	7(0)	10(0)	0
$FP_{unadjusted}$				46

Table 1: Weights for features

The numbers in brackets represent the number of features for this particular application. They are multiplied by the weighting factor and added.

4.1.3 Influence Factors

System Characteristic	Influence level	
	Sequential Code	Parallel Code
Is the code designed to be reusable?	3	3
Are conversion and installation included in the design?	0	0
Is the system designed for multiple installations in different organizations?	4	4
Is the application designed to facilitate change for ease of use by the user?	3	3
Does the system require online data entry?	0	0
Does the online data entry require the input transactions to be built over multiple screens or operations?	0	0
Are the master files updated on-line?	0	0
Are the inputs, outputs, files, or inquiries complex?	0	0
Is the internal processing complex?	0	3
Does the system require reliable backup and recovery?	0	0
Are data communications required?	0	3
Are there distributed processing functions?	0	0
Is performance critical?	3	3
Will the system run in an existing, heavily utilized operational environment?	0	0
Total	13	19

Table 2: Influence Factors

Legend:

Level	Influence
0	No Influence
1	Incidental
2	Moderate
3	Average
4	Significant
5	Essential

Table 3: Legend

Sequential Code:

Process Complexity Adjustments = .65 + 0.01 * (sum of influence ratings) = .78

$FP_{adjusted} = FP_{unadjusted}$ * (.65 + 0.01 * (sum of ratings)) = 46 * (.65 + .01 * (13)) = 35.88

Source lines of code (SLOC) = FP * Language Factor (for Java) = 35.88 * 40 = 1435.2

Parallel Code:

Process Complexity Adjustments = .65 + 0.01 * (sum of influence ratings) = .84

FPadjusted = FPunadjusted * (.65 + 0.01 * (sum of ratings)) = 46 * (.65 + .01 * (19)) = 38.64

Source lines of code (SLOC) = FP * Language Factor (for Java) = 38.64 * 40 = 1545.6

* Language Factor = Estimated source lines of code per function point.

4.2 Cost Estimation by COCOMO Model

TDEV	Programmer Productivity	Development Time (Month)
Application Programs	PM = 2.4*(KDSI) 1.05	TDEV = 2.5* (PM) 0.38
Utility Programs	PM = 3.0*(KDSI) 1.12	TDEV = 2.5*(PM) 0.35
System Programs	PM = 3.6*(KDSI) 1.20	TDEV = 2.5*(PM) 0.32

Table 4: COCOMO Model

The above table lists the equations to calculate the Person Months (a measure of programmer productivity) and the development time in months for a given KLOC/KDSI* for different types of Software Programs i.e. Application programs, Utility programs and System Programs. The current software project fits into the category of an application program.

Sequential Code:

Person Month: PM = 2.4 * (KLOC) 1.05 = 2.4 * 1.4 1.05 = 3.4

Development Time (Months): TDEV = 2.5 * (PM) 0.38 = 2.5 * 3.4 0.38 = 3.9

Parallel Code:

Person Month: PM = 2.4 * (KLOC) 1.05 = 2.4 * 1.54 1.05 = 3.77

Development Time (Months): TDEV = 2.5 * (PM) 0.38 = 2.5 * 3.770.38 = 4.2

* KLOC – Kilo Lines of Code

* KDSI – Kilo Delivered Source Instructions (same as KLOC)

5. Architecture Elaboration Plan

The following activities have to be accomplished prior to the architecture presentation:

- Action Items: The action items identified during each of the phases, along with the efforts made to satisfy them, will be documented.
- Updated Vision Document: The vision document will be updated to provide a complete and adequate representation of all the requirements. A set of "critical" requirements will be identified by ranking the requirements according to importance. These modifications will be based on the recommendations made by the members of the graduate committee.
- Updated Project Plan: The project plan will detail the phases, iterations, and milestones that will comprise the project. Each deliverable will be included in the plan with estimated dates, sign-offs and evaluation criteria.
    - Cost Estimate: The document will also provide an updated estimate on the size, cost and effort required for the project implementation.
    - Implementation Plan: The Implementation plan will define the activities and actions that must be accomplished during implementation. The plan will include a Work Breakdown Structure, complete with time and costs estimates and completion criteria.

- Formal Requirement specification: The properties of the system are formally expressed using Object Constraint Language (OCL). Use of a formal language to express the requirements will help making them adequate (it will adequately state the problem at hand), internally consistent (it will have a meaningful semantic interpretation that makes true all specified properties taken together), unambiguous (it may not have multiple interpretations of interest making it true), and minimal (it should not state properties that are irrelevant to the problem or that are only relevant to a solution for that problem).
- Architecture Design: The complete architectural design of the project is documented using modeling languages such as UML. Use case diagrams, Class diagrams and sequence diagrams will be used to illustrate the architecture design of the system. Re-use of pre-existing components will be documented.
- Test Plan: A set of test cases, the types of tests that will be used for these test cases, the data that will be used for each test case and the requirement traces for each test case will be identified. The results of the test are documented.
- Formal Technical Inspection: One of the technical artifacts (design, formal requirement or executable prototype) will be subjected to a formal technical inspection by two independent MSE students – Srinivas Kolluri and Laxminarayan. An IEEE standard formal check list will be used by the inspectors. The inspectors will provide a report on their inspection results and these become a part of the project documentation.

- Executable Architecture Prototype: An executable architecture prototype will be built in one or more iterations which will address all critical requirements identified in the vision document and expose the top technical risks.

CHAPTER 4: SOFTWARE QUALITY ASSURANCE PLAN

1. Introduction

This document explains the Software Quality Assurance Plan (SQAP) for MSE project of Lakshmikanth Ganti. The project is to develop an application in Java that uses Molecular Dynamics Simulation techniques to simulate the interaction between the atoms in a group of molecules.

1.1 Purpose

Software Quality Assurance Plan (SQAP) consists of those procedures, techniques and tools used to ensure that a product meets the requirements specified in software requirements specification.

1.2 Scope

The scope of this document is to outline all procedures, techniques and tools to be used for quality assurance of this project.

This plan:

- Identifies the SQA responsibilities of the project developer and the SQA consultant
- Lists the activities, processes, and work products that the SQA consultant will review and audit
- Identifies the SQA work products

1.3 Reference Documents

- Lecture notes, CIS 748 Software Management, Dr. Scott Deloach, Spring 2002
- Lecture Notes, CIS 771 Software Specifications, Dr. John Hatcliff, Spring 2001
- Software Engineering, Roger S. Pressman, 5th Ed.
- IEEE Guide for Software Quality Assurance Planning, IEEE STD 730.1 – 1995.
- IEEE Standard for Software Quality Assurance Plans, IEE STD 730 – 1998.

1.4 Overview of the Document

The rest of the document is organized as follows:

Management: A description of each major element of the organization and a description of the SQA tasks and their relationships

Documentation: Identification of the documents related to development, verification, validation, use and maintenance of the software.

SQAP Requirements: This section defines the SQA review, reporting, and auditing procedures used to ensure that software deliverables are developed in accordance with this plan and the project's requirements.

Training: This section describes the training program for the developer.

2. Management

2.1 Organization

This tool is developed as an individual project as part of partial fulfillment of requirements for Masters in Software Engineering degree. Since there is only one

member involved, it will be the sole responsibility of the developer to review the product's usability, efficiency, reliability, and accuracy. The major professor will however conduct inspections, reviews, and walk-through on a regular basis. In addition a committee consisting of the major professor and two other faculty members will review the documents of each phase before every presentation. Major Professor's and the committee's specifications and suggestions will be used in places where quality decisions need to out-weigh development schedule decisions.

2.2 Roles

- The committee consists of Dr. Virgil Wallentine, Dr. Paul Smith and Dr.Mitch Neilsen.
- Major Professor: Dr. Virgil Wallentine
- Developer: Lakshmikanth Ganti.

2.3 Tasks and Responsibilities

The responsibilities of the developer are as follows:

- Develop the requirement specification and cost estimation for the project
- Develop the design plan and test plan for testing the tool
- Implement and test the application and deliver the application along with the necessary documentation
- Give a formal presentation to the committee on completion of the analysis, design and testing phases. The committee reviews the developer's work and provides feedback/suggestions.

- Planning, coordinating, testing and assessing all aspects of quality issues.

The responsibilities of the committee members are to:

- Review the work performed by the developer
- Provide feedback and advice

2.4 SQA Implementation in different phases

Quality assurance will be implemented through all the software life cycles of the tool's development process, until the release of the software product. The following are the quality assurance tasks for each phase of the software development:

Requirements phase: When the SRS is being developed, the developer has to ensure that it elucidates the proposed functionality of the product and to keep refining the SRS until the requirements are clearly stated and understood.

Specification and Design phase: Due to the great importance for accuracy and completeness in these documents, weekly reviews shall be conducted between the developer and the professor to identify any defects and rectify them.

Implementation phase: The developer shall do code reviews when the construction phase of the Tool begins.

Software testing phase: The developer shall test each case. The final product shall be verified with the functionality of the software as specified in the Software Requirements Specification (SRS) for the Tool.

Through all these phases of the software development, the following shall also be conducted to improve the software quality:

- Develop and generate SQAP: Generate a finalized SQAP plan

- Communication and Feedback: The developer is encouraged to freely express disagreements, suggestions and opinions about all aspects of the weekly process of software development.
- Internal audits and evaluations: The Major professor and the committee are expected to do auditions and evaluations at the end of each phase in the project.

3. Documentation

In addition to this document, the essential documentation will include:

1) The Software Requirements Specification (SRS), which

- Prescribes each of the essential requirements (functions, performances, design constraints and attributes) of the software and external interfaces
- Objectively verifies achievement of each requirement by a prescribed method (e.g. Inspection, analysis, demonstration or test)
- Facilitates traceability of requirements specification to product delivery.
- Gives estimates of the cost/effort for developing the product including a project plan.

2) The Formal Specification Document, which gives the formal description of the product design specified in Object Constraint Language (OCL).

The Software Design Description (SDD)

- Depicts how the software will be structured

- Describes the components and sub-components of the software design, including various packages and frameworks, if any.
- Gives an object model that is developed using Rational Rose highlighting the essential classes that would make up the product.
- Gives a sample interaction diagram, showing the key interactions in the application. This should also be a part of the object model.

3) Software Test Plan: Describes the test cases that will be employed to test the product.

4) Software User Manual (SUM)

- Identify the required data and control inputs, input sequences, options, program limitations or other actions.
- Identify all error messages and describe the associated corrective actions.
- Describe a method for reporting user-identified errors.
- Documented Source Code.

The following documents will be provided at the end of each phase by the developer:

Phase 1: Objectives

- Project Overview
- Requirements Specification

- Cost analysis
- Project plan
- Software quality assurance plan

Phase 2: Architecture

- Implementation Plan
- Formal Requirement Specification
- Architecture design
- Test plan

Phase 3: Implementation

- User Manual
- Assessment Evaluation
- Project Evaluation
- References
- Formal Technical Inspection Letters

Appendix

- Source code

4. SQA Program Requirements

4.1 **Standards**

- Document standards – MSE Portfolio

- Coding standards – Java 1.4
- Coding Documents standards – Java Documentation
- Test Standards – IEEE Standard for software test documentation

4.1. Metrics

- LOC - lines of code is used to measure the size of the software

4.2. Software Documentation Audit

Quality Assurance for this project will include at least one review of all current work products in each stage of development (Requirement, Design, and Implementation). The reviews will assure that the established project processes and procedures are being followed effectively, and exposures and risks to the current project plan are identified and addressed. The review process includes:

- A formal presentation at the end of each development phase (Requirement, Design and Implementation). All current work products are presented to the committee members for review.
- A managerial review by the advisor periodically to ensure the work generated is in compliance with project requirements.
- Reviews by the committee after each presentation.

4.3. Requirements Traceability

The SRS will be used to check off the deliverables. The Project Review will ensure that each of the requirements mentioned in the SRS is met by the deliverables.

4.4. Software Development Process

The software development process involves three stages: 1) Requirements phase, 2) Design phase (this phase also involves the development of the product prototype and 3) Implementation and testing phase. During each phase, the Major Professor and the committee will review the deliverable documents. The developer would incorporate modifications suggested by the committee. This would ensure quality of the software product.

4.5. Project Reviews

The Committee will perform a review at the 3 stages of the project as described in the section above. This review will determine whether the requirements have been met for the deliverable, check that the product meets the requirements, ensure that the SQA plan has been adhered to, verify the performance of the software and ensure that acceptance testing is carried out. In addition the developer will conduct a Formal Technical Review after the design phase. A design checklist will be used and the developer will check to see whether his/her design meets the checklist criteria.

4.6. Testing and Quality Check

Testing will be carried out in accordance with the Software Testing Plan (STP). Testing documentation will be sufficient to demonstrate that testing objectives and software requirements have been met. Test results will be documented and discussed in the final phase of the project.

5. Training

The following courses taken by the developer at Kansas State University and Research experience under the guidance of Dr. Virgil Wallentine and Dr. Paul Smith will provide the required training.

- CIS 540: Software Engineering –1
- CIS 740: Software Engineering – 2
- CIS 748: Software Management
- CIS 771: Software Specification
- CIS 625: Parallel Programming

CHAPTER 5: ARCHITECTURE DESIGN

1. Introduction

The purpose of this document is to describe the architecture design of the Molecular Dynamics Simulation tool that will capture the requirements as outlined in the requirements specification section. The document will outline the goals, key design principles along with class diagram and sequence diagrams.

2. References

IEEE STD 1016-1998, “IEEE Recommended practice for Software Design Description”.

3. Definitions and Abbreviations

- SDD: Software Design Description
- Molecular Dynamics Simulation: A technique where the time evolution of a set of atoms is followed by integrating their equations of motion.
- Lennard-jones potential: An interaction potential existing between atoms that are considered here.
- PDB: Protein Data Bank
- Potential Energy: The energy resulting from position or configuration of an atom.
- Kinetic Energy: The energy resulting from motion of an atom.
- Temperature: A measure of the kinetic energy in atoms of a substance.

- Cut-off distance: Distance between atoms above which there are no interaction forces.

4. Goals

The overall goal of the system is to calculate the final coordinates of all the atoms, the energies and the temperature of the system after simulating through a certain number of time steps. This depends on doing the following tasks correctly at each time step: 1) Calculate the force on an atom due to every other atom within the interaction distance 2) Use the force calculated above to calculate the increment in velocity and displacement of the atom.3) Calculate the potential energy contributed by each atom in the system.4) Calculate the total Kinetic energy of the system and the temperature of the system using the kinetic energy.

5. Key Design Principles

5.1 Algorithm

The algorithm used here is called the Particle-Particle (PP) method. Here, the state of the physical system at some time t is described by the set of atom positions and velocities $\{X_i (t), V_i (t); i = 1, Np\}$. The time step loop updates these values using the forces of interaction and equations of motion to obtain the state of the system at a slightly later time t + DT as follows:

1. Compute forces.

    Clear potential and force accumulators

    $V := 0$

    for i = 1 to Np do

        $F_i := 0$

    Accumulate forces

    for i = 1 to Np – 1 do

    for j = i + 1 to Np do

    Find force F_{ij} of particle j on particle i

    $F_i := F_i + F_{ij}$

    $F_j := F_j - F_{ij}$

    Find the potential energy contribution

        $V = V + V_{ij}$

2. Integrate equations of motion

    for i = 1 to Np do

        $Vel_{inew} := Vel_{iold} + (F_i/m_i)DT$

        $X_{inew} := X_{iold} + Vel_iDT$

3. Update time counter

    $t := t + DT$

Repeated application of the time step loop is used to follow the temporal evolution of the system.

The equations for calculating F_{ij} and V_{ij} are called the Lennard-Jones equations for calculating potential and force and are given as follows:

$$V_{LJ} = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \qquad (1)$$

σ and ε are the specific Lennard--Jones parameters, different for different interacting atoms. r is the distance between the interacting atoms. The values of these parameters for the system under consideration are: σ = 0.3 nanometers and ε = 1.0 KJ/mole. The Lennard--Jones force between two atoms is given by the equation:

$$F_{LJ} = -24\varepsilon \left[2\left(\frac{\sigma^{12}}{r^{13}}\right) - \left(\frac{\sigma^{6}}{r^{7}}\right) \right] \qquad (2)$$

5.2 Design

Performance is a key issue in computationally intensive systems such as the one being programmed here. The majority of the computation in the code occurs in the calculation of force on each atom due to every other atom. The fact that there will be no interacting forces between atoms whose separation is greater than a specific cut-off distance is taken into consideration and the following model is designed for calculating forces, which improves performance.

Each of the atoms is assigned to cubical partitions whose length is equal to the cut-off distance for interaction, depending on their spatial configuration (x, y, and z coordinates).

The ideal number of partitions in the current system would be 8x8x8, since the length of the simulation box is 8.09202 nm and the cut-off distance is 1.0 nm, and the length of each partition would be 1.0115025 nm. Each partition is uniquely identified by three indices. For example if each partition is of length 1, then the partition which is identified by partition (0,0,0) holds the atoms whose coordinates are such that 0<=x<1, 0<=y and 0<=z<1. The following diagram shows how the whole system of atoms is assigned to partitions. The partitions are connected like a torus interconnection system to accommodate the periodic boundary conditions property of the simulation system. So, partition (0,2,0) is the left neighbor of partition (0,0,0). Similarly partition (2,0,0) is the top neighbor of partition (0,0,0). The direction of the positive z-axis is into the paper.

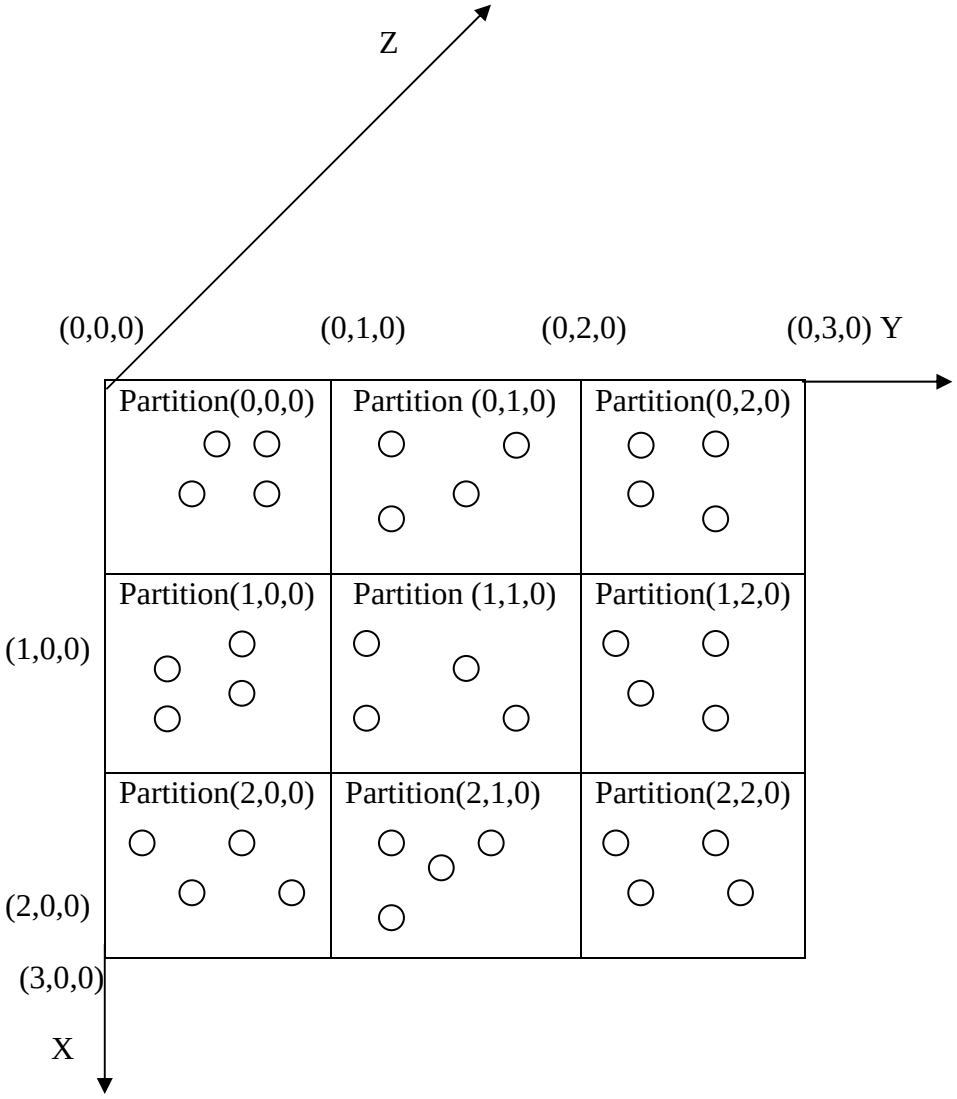


Figure 4: Partitions of the system

The efficiency in this model is due to the fact that now we need to consider the interactions only between atoms in the neighboring partitions instead of calculating the distance between each pair of atoms and checking if it is less than the interaction distance. So the above algorithm for calculating force and potential energy is slightly modified to accommodate for this model and is as follows:

```
Initialize forces and potential energy
for partition1  = 1 to n
   for partition2   = 1 to n
    {
       check if the partitions are neighbors
        {
            for i  = 1 to number of atoms in the partition1
             {
                initialize force accumulators: sfx  = 0, sfy = 0, sfz = 0
                 for j  = 1 to number of atoms in the partition2
                  {
                     check if atom number in partition1 > atom number in partition2
                      {
                         check if distance between the atoms < cut-off distance
                          {
                              pot = pot  + vlj;
                              fxj  = fxj + fx; fyj  = fyj + fy; fzj = fzj + fz;
                              sfx = sfx + fjx; sfy = sfy + fjy; sfz = sfz + fjz;
                          }
                      }
                  }
                fxi = fxi – sfx; fyi = fyi – sfy; fzi = fzi = fzi – sfz;
       }
     }
```

fx, fy and fz are the interaction force between two atoms in x, y and z directions and vlj is the interaction potential between two atoms calculated according to the Lennard-Jones equation (eqns 1 & 2). The symmetry of the forces of two bodies on each other is

exploited here by examining each pair of bodies just once (note the check “check if atom number in partition1 > atom number in partition2”).

5.3 Design considerations for a parallel program

The current project is an ideal application of parallel programming owing to the intense computational nature of the molecular dynamics simulations. Parallelizing the program and running it on multiple processors significantly reduces the time taken for the simulation, since the work is shared by multiple threads each running on different processors.

The simplest design of a parallel program from the above sequential code would be to distribute the partitions equally between all the threads. At the end of iteration, each thread has to communicate with its neighboring threads by passing all its bordering partitions which will be required by its neighboring threads to run the simulation of its own partitions. Various designs for a parallel program based on 1) Synchronization mechanism, 2) the pattern of thread creation and 3) Granularity, are explained below:

5.3.1 Design based on Synchronization Mechanism

Synchronization between threads is a very important issue to be considered carefully here. It has to be ensured that each of the threads is in sync with the other threads for the computation to be accurate. Two possible mechanisms where

synchronization can be achieved are by 1) message passing between threads and 2) using a barrier to stop all the threads, at the point where synchronization is required.

a) Message Passing

The message passing between the threads is carried out using Bounded Buffers. Each bounded buffer is represented by an object of the Java class "Objbuf" which is described in detail later in the class diagram section of this document. Two unique bounded buffers exist between each pair of neighboring threads: one to put the objects to be transferred and one to get them. The threads with which a thread communicates directly are referred to as neighboring threads. The number of neighboring threads and the mechanism of message passing depend on the pattern of thread creation, which is explained in detail in the next section. Message passing is used only when the neighboring threads need to synchronize with each other.

b) Barrier Synchronization

A Java class called "Barrier" is created which is used for synchronizing all the threads at a given point. A common Barrier is shared between all the threads. Whenever all the threads need to stop at a point and synchronize, each of the threads calls the gate () method in this class. This is used only when all the threads in the system need to stop and synchronize. The Barrier class provides synchronization through the use of the Semaphore Classes (Binary and Counting Semaphores).

5.3.2 Design based on pattern on thread creation

The pattern in which the threads are arranged has no effect on the barrier synchronization mechanism since when a barrier is used we intend that all the threads be stopped irrespective of how they are arranged. However, it affects the message passing mechanism since the number of neighbors for a thread and who they are is changed based on the pattern in which they are arranged. Two possible patterns are 1) 3-D grid shaped – where each thread communicates with its twenty six neighbors and 2) Vertical pipeline – where every thread communicates only with its upper and lower neighbors. The system requires that the connections are based on torus inter-connection system i.e. in the case of Vertical pipeline, the lower neighbor of the bottom most thread is the top most thread and vice versa. The following sub-sections describe the creation of threads and message passing between them for each pattern:

a) 3-D Grid shaped

A three dimensional array of threads is created with each thread uniquely represented by three indices. Since there are 8x8x8 partitions, it would be only possible to create an array of 2x2x2 or 4x4x4 threads with 4x4x4 or 2x2x2 partitions assigned to each thread. Creating 8x8x8 threads and assigning one partition per thread will cause an excessive overhead in thread creation which increases the execution time enormously. Since there is a 3D grid of threads, each thread has to communicate with its twenty-six neighboring threads. So each thread should have twenty-six buffers associated with it. Each thread has access to a static four-dimensional array of bounded buffers. It has the dimensions [M][M][M][26]. The first three dimensions are the same as the thread

identifiers. The fourth dimension determines the other thread, which it is associated to i.e. top or left etc. The following mapping helps to determine the neighboring thread's coordinates with relative to the coordinates of the current thread and the value of the fourth dimension of the corresponding bounded buffer. The naming convention is T: Top, B: Bottom, L: Left, R: Right, O: Outer, I: Inner. So the neighboring threads are these and their combinations. E.g. TRO represents the Top-Right-Outer thread.

Neighboring thread.	X coordinate	Y coordinate	Z coordinate	Fourth dimension
T	-1	same	same	0
B	+1	same	same	1
L	Same	-1	same	2
R	Same	+1	same	3
O	Same	same	-1	4
I	Same	same	+1	5
TL	-1	-1	same	6
TR	-1	+1	same	7
TO	-1	same	-1	8
TI	-1	same	+1	9
BL	+1	-1	same	10
BR	+1	+1	same	11
BO	+1	same	-1	12
BI	+1	same	+1	13
LO	Same	-1	-1	14
LI	same	-1	+1	15
RO	same	+1	-1	16
RI	same	+1	+1	17
TRO	-1	+1	-1	18

TRI	-1	+1	-1	19
TLO	-1	-1	-1	20
TLI	-1	-1	+1	21
BRO	+1	+1	-1	22
BRI	+1	+1	+1	23
BLO	+1	-1	-1	24
BLI	+1	-1	+1	25

Table 5 : Bounded Buffer Coordinates in 3D Grid System

Each thread has an array of buffers of size 26 called “buf []” to get the bordering partitions of the neighboring threads and an array of buffers called “shad []” to put its bordering partitions into them so that the corresponding thread will fetch them. These buffers and shadows should be properly defined so that the buffer of a thread is the same as the shadow of one of the neighboring threads. For e.g. the top shadow of a thread should be the bottom buffer of the thread’s top neighbor. It is also important that the correct partitions are transferred to the corresponding neighbors. For e.g. the top layer of the partitions array is to be transferred to the top neighbor.

b) Vertical Pipeline

A one dimensional array of threads is created with each thread identified by a unique index. The idea here is to assign layers of partitions to each thread rather than a 3-D array of partitions to each thread. Since there are 8 layers of partitions with 8x8 partitions in each layer, the number of threads could be 1,2 4, or 8 with 8,4,2,1 layers assigned to each thread respectively. In this pattern, each thread has only two neighbors associated with it. So each thread should have two buffers associated with it. Each thread has access to a static two-dimensional array of bounded buffers. It has the dimensions

[M][2]. The first dimension is the same as the thread identifier and the second dimension determines the other thread, which it is associated to i.e. top or bottom. The following mapping helps to determine the neighboring thread's coordinate relative to the coordinate of the current thread and the value of the second dimension of the corresponding bounded buffer. The naming convention is T: Top, B: Bottom.

Neighboring thread.	First Dimension	Second Dimension
T	-1	0
B	+1	1

Table 6: Bounded Buffer Coordinates in Vertical Pipeline system

Each thread has an array of buffers of size 2 called "buf []" to get the bordering partitions of the neighboring threads and an array of buffers called "shad []" to put its bordering partitions into them so that the corresponding thread will fetch them. These buffers and shadows should be properly defined so that the buffer of a thread is the same as the shadow of one of the neighboring threads. For e.g. the top shadow of a thread should be the bottom buffer of the thread's top neighbor. It is also important that the correct layer of partitions is transferred to the corresponding neighbors. For e.g. the top layer of the partitions array is to be transferred to the top neighbor.

5.3.3 Design based on Granularity

Granularity is determined by the frequency of thread synchronization or communication relative to the amount of computation done. It is expected that the more the computation per thread relative to the communication, the more speed up is achieved by the parallel program since the over head in communication is reduced. Granularity is

increased or decreased by altering the number of partitions assigned to a thread. For a system of fixed size, the more the number of threads, the less the granularity is. Granularity can also be increased by increasing the system size with a fixed number of threads. Measurements of the speed up with different levels of granularity will be performed.

The pseudo code for the run method of a thread i.e. what each thread will do in parallel is as follows:

For time step = 1 to number of iterations

{

1) assign the atoms to the partitions that belong to this thread depending on their spatial configuration

2) put the bordering partitions in the corresponding shadows.

3) collect the bordering partitions of the neighbors from all the buffers.

4) calculate forces.

5) increment velocities and calculate displacements.

6) Calculate energies due to the contribution of atoms in this thread's partitions and send them to energy writer class.

}

6. Class Diagram

The following figure represents the class diagram for the Simulation program and the subsequent sections explain in detail the purpose of each class in the class diagram:

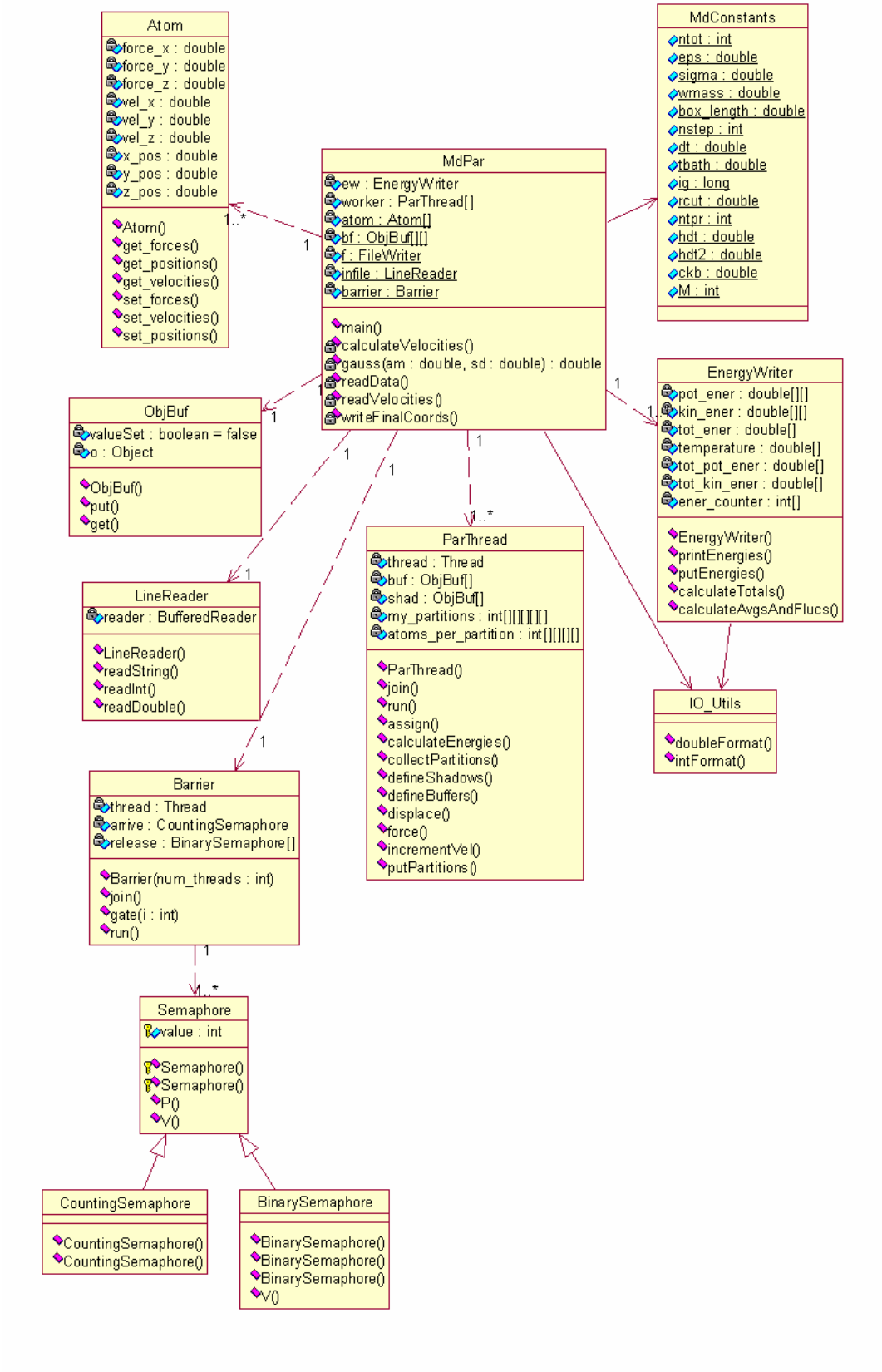

Atom
force_x : double
force_y : double
force_z : double
vel_x : double
vel_y : double
vel_z : double
x_pos : double
y_pos : double
z_pos : double
Atom()
get_forces()
get_positions()
get_velocities()
set_forces()
set_velocities()
set_positions()
MdPar
ew : EnergyWriter
worker : ParThread[]
atom : Atom[]
bf : ObjBuf[][]
f : FileWriter
infile : LineReader
barrier : Barrier
main()
calculateVelocities()
gauss(am : double, sd : double) : double
readData()
readVelocities()
writeFinalCoords()
MdConstants
ntot : int
eps : double
sigma : double
wmass : double
box_length : double
nstep : int
dt : double
tbath : double
ig : long
rcut : double
ntpr : int
hdt : double
hdt2 : double
ckb : double
M : int
ObjBuf
valueSet : boolean = false
o : Object
ObjBuf()
put()
get()
EnergyWriter
pot_ener : double[][]
kin_ener : double[][]
tot_ener : double[]
temperature : double[]
tot_pot_ener : double[]
tot_kin_ener : double[]
ener_counter : int[]
EnergyWriter()
printEnergies()
putEnergies()
calculateTotals()
calculateAvgsAndFlucs()
ParThread
thread : Thread
buf : ObjBuf[]
shad : ObjBuf[]
my_partitions : int[][][][][]
atoms_per_partition : int[][][][]
ParThread()
join()
run()
assign()
calculateEnergies()
collectPartitions()
defineShadows()
defineBuffers()
displace()
force()
incrementVel()
putPartitions()
LineReader
reader : BufferedReader
LineReader()
readString()
readInt()
readDouble()
IO_Utils
doubleFormat()
intFormat()
Barrier
thread : Thread
arrive : CountingSemaphore
release : BinarySemaphore[]
Barrier(num_threads : int)
join()
gate(i : int)
run()
Semaphore
value : int
Semaphore()
Semaphore()
P()
V()
CountingSemaphore
CountingSemaphore()
CountingSemaphore()
BinarySemaphore
BinarySemaphore()
BinarySemaphore()
BinarySemaphore()
V()
1
1..*

Figure 5: Class Diagram

6.1 Class Atom

An object of this class represents each atom in the simulation system. It has the forces, velocities, and coordinates in all directions as its attributes and has get and set methods for each of these attributes.

6.2 Class IO_Utils

This is a helper class, which has methods for formatting an integer or a double to be outputted to a file in a specified pattern.

6.3 Class LineReader

This is a helper class which has methods that browse through a file and reads it line by line for a string, double or integer input.

6.4 Class ObjBuf

This is the class, which provides the communication between the threads. Each object of this class represents a buffer where threads can put or get an array of objects. As already seen before in this document, there's a unique pair of get and set buffers between each pair of threads. It has synchronized methods for putting and getting an array of objects. Thus, at most one thread can be inside these methods at a time. Synchronization is provided by means of a Boolean variable that is initially set to false. There's a check on

this variable in each of the methods. So, a thread waits in the get() method if the buffer is empty and likewise waits in the put() method if the buffer is full.

6.5 Class EnergyWriter

Since the energy calculations are distributed over multiple threads and non-neighboring threads could be at different time steps at a time, we need a class that collects the energy contributions from each thread at a particular time step and adds them together to get the totals. This is the class that does this work. It has methods for a) collecting the potential and kinetic energies form the threads at each time step b) calculating the averages and fluctuations for each of the physical quantities at each time step i.e. potential energy, kinetic energy, total energy and temperature c) printing the energies and temperature at every specified number of time steps to a file in a specified format.

6.6 Class ParThread

This is a Java Thread class and has methods to do the following tasks which are called in the standard run() method.

- Assign atoms to the partitions belonging to itself based on the atom's coordinates
- Put the bordering partitions of this thread in all of the neighboring shadows
- Get the bordering partitions of all the neighbors from buffers

- Calculate forces on the atoms of the partitions belonging to this thread due to every other atom with in the interaction distance.
- Increment the velocities and displace the atoms of the partitions belonging to this thread.
- Pass the kinetic energy and potential energy contribution due to the atoms of the partitions belonging to this thread, to the EnergyWriter class at each time step.

6.7 Class MdPar

This is the main class, which is responsible for starting the simulation. It creates an array of threads, joins them and records the time taken for the entire simulation. This class is also responsible for 1) initializing the coordinates and velocities of the Atoms with the data read from the velocity and coordinate input files, 2) initializing the values of the physical constants required in the simulation with the data read from the md.dat input file and 3) write the final positions of the Atoms after simulation to the coordinates output file.

6.8 Class MdConstants

This is the class which holds all the constant values such as eps, the Lennard-Jones parameter, and is used by most of the classes.

6.9 Class Barrier

This class is used for synchronizing all the threads at a given point. A common Barrier is shared between all the threads. Whenever all the threads need to stop at a point and synchronize, each of the threads calls the gate () method in this class. This is used only when all the threads in the system need to stop and synchronize. If we need only the neighboring threads to stop and synchronize, the ObjBuf class serves the purpose in addition to communicating objects between the threads. In our system it is observed that the atoms did not move more than twice the length of the partition during a simulation with 100 iterations. That means a thread needs to wait only for the neighboring threads and synchronize. So an ObjBuf could be used instead of a Barrier. The Barrier class provides synchronization through the use of the Semaphore Classes (Binary and Counting Semaphores).

7. Use Cases

The primary use cases in the system are listed below and explained with the help of sequence diagrams.

7.1 Read Data from input files

The readString method of the LineReader class is used to browse the files line by line to get the input as a string. These strings are then parsed appropriately to extract the

desired data e.g. velocities or coordinates in double format and other data values in integer format.

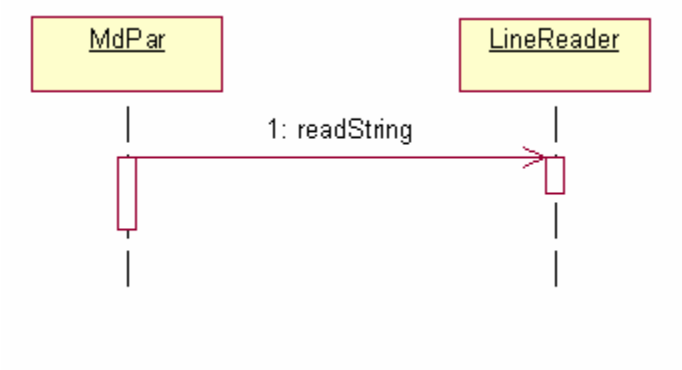


Figure 6: Sequence Diagram, Read Data From Input Files

7.2 The sequence diagram below illustrates the following use cases, which involve method calls in the same class itself.

- Assigning atoms to a partitions of a thread depending on their spatial coordinates
- Put bordering partitions in all the thread's shadows for its neighboring threads to collect.
- Get bordering partitions from all the neighboring threads.
- Calculate forces on atoms in the partitions of the current thread
- Increment velocities and displace the atoms of the partitions of the current thread

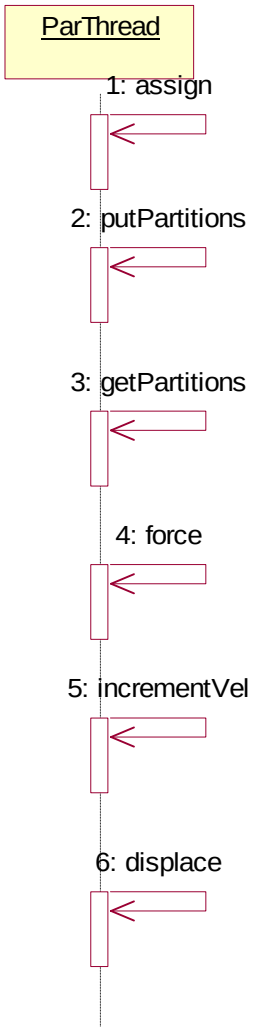


Figure 7: Sequence Diagram

7.3 Calculate and print energies

The threads communicate with the EnergyWriter class to put the kinetic and potential energy contributions of the threads atoms at each iteration step by calling the putEnergies() method. This method checks at each step if all the threads have communicated their energy contributions and makes a call to the calculateTotals()

method which calculates the total energy contributed by all the threads and calls the printEnergies() method which prints the energies to the console and a file at every given number of steps.

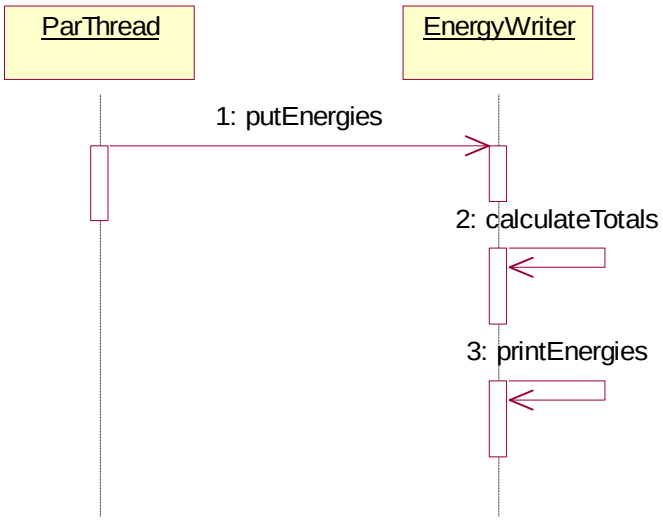


Figure 8: Sequence Diagram, Calculate and Print Energies

7.4 Calculate averages and fluctuations of energies and temperature and write them to a file.

The main class (MdPar) calls the calculateAvgsAndFlucs() method of the EnergyWriter class, which calculates the averages and fluctuations of the potential energy, kinetic energy and the temperature and calls the printEnergies() method to output them to the console and a file.

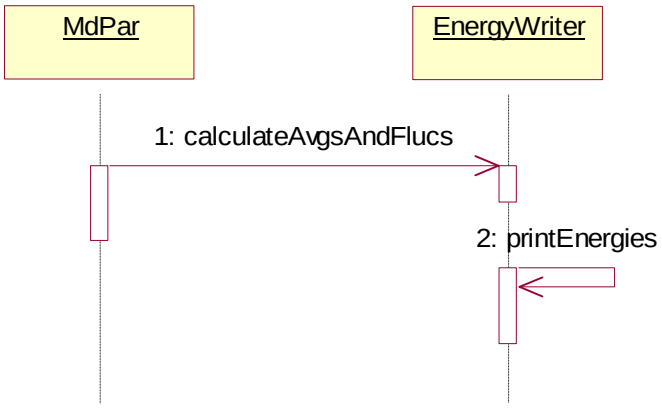


Figure 9: Sequence Diagram, Calculate Averages and Fluctuations

CHAPTER 6: FORMAL REQUIREMENTS SPECIFICATION

1. Introduction

The purpose of this document is to present the process of formal specification and verification of the synchronization technique used in this project. Java Path Finder (JPF) has been used to formally specify and verify the synchronization properties of the system.

2. Java Path Finder

The Java Path Finder [8] is a translator from a subset of Java 1.0 to PROMELA, the programming language of the SPIN model checker. This tool is designed to establish a framework for verification and debugging of Java programs based on model checking. It simplifies the verification of Java programs by obviating the need to manually reformatting the program into a different notation (e.g. PROMELA or OCL), in order to analyze the program. This system is especially suited for analyzing multi-threaded Java applications, of which the current project is an example. The system can find deadlocks and violations of Boolean assertions stated by the programmer in a special assertion language.

3. Model

Synchronous communication between the threads is carried out using Bounded Buffers. Two unique bounded buffers exist between each neighboring pair of threads, one to put the objects to be transferred and one to get them. Each thread is modeled as a Producer-Consumer i.e. it is a producer as well as a consumer at the same time. Since the simulation is modeled using a three dimensional grid of threads, each thread has twenty

six neighbors and a buffer associated with it. The model with communication between all the twenty six threads has an enormously huge state space and very intensive in terms of time and resources. So the following prototype is proposed for the communication and has been specified using JPF:

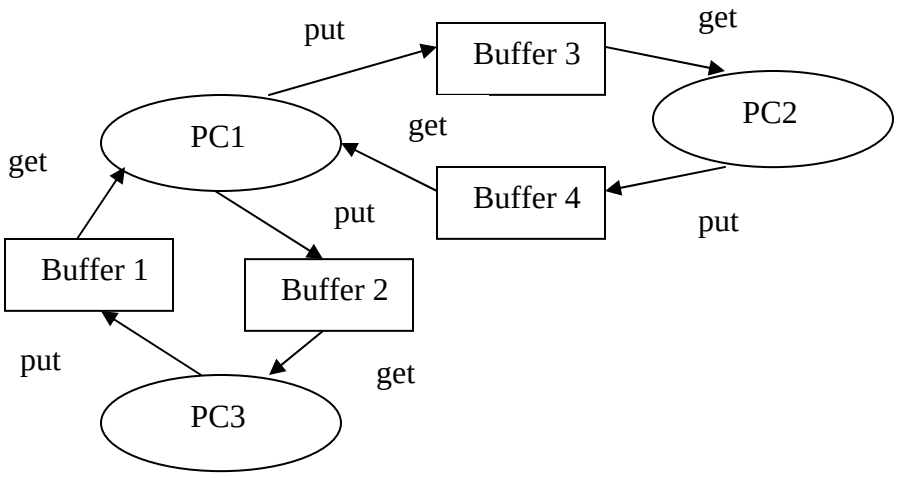


Figure 10: JPF Model

4. JPF Specification

The Java code for the verification of the synchronization technique used in the project can be found in Appendix A. JPF checks for deadlocks, assertion violations and uncaught exceptions. The Assertion made here is that none of the neighboring threads are more than one step ahead or one step behind than the simulation step of a given thread. In other words, this means that the computations for step t+1 of a thread are dependent on the step t of all of its neighbors and thus ensure that the threads are never out of synchronization.

5. JPF Result

The above model is verified by JPF for its safety properties and assertion violations in all the possible states that can be reached. The following is the result by JPF:

```
==========================================
 No Errors Found
==========================================

----------------------------------
States visited       : 8,970,992
Transitions executed : 27,423,628
Instructions executed: 864,227,005
Maximum stack depth  : 720
Intermediate steps   : 1,181,428
Memory used          : 1.1GB
Memory used after gc : 1.05GB
Storage memory       : 59.43MB
Collected objects    : 27,051,899
Mark and sweep runs  : 25,795,428
Execution time       : 3:06:53.591s
Speed                : 2,445tr/s
----------------------------------
```

CHAPTER 7: TEST PLAN

1. Test Plan Identifier

MSE – TP 01

2. Introduction

The purpose of this document is to outline the plan for testing all the critical use cases and functionality of the Molecular Dynamics Simulation tool. The document will also describe the tools and environment used to test the software.

3. References

The following documents are used for reference:

- Software Requirements Specification
- Architecture Design

4. Test Items

The following features are to be tested:

- Read Data from files
- Read Program Arguments
- Formatting values for output.

5. Features not to be tested

The communication between the threads is not to be tested again, since we already validated the code for communication using Java Path Finder and ensured that it is free from deadlocks and uncaught exceptions.

6. Approach

The specific requirements specification is used as a guide to test the above-mentioned features of the software.

6.1 Read Data from files.

The software should read data from the files in a specific format i.e. integer, string or a double. Exceptions should be raised appropriately whenever a wrong format or a blank line is encountered.

6.2 Read Program Arguments:

The program should catch exceptions in the program arguments and throw an appropriate error message. For e.g., the program has an argument, the number of threads, which can be only 1,2,4 or 8.

6.3 Formatting values for output

The methods used for formatting the values for output should be tested for correctness. They should properly format the values raising exceptions for invalid inputs.

6.4 Functional Testing

The program is tested for correctness i.e. it should give the same results when run on any number of threads.

6.5 Performance Requirements Testing

All performance requirements will be tested against their requirements described in the software requirements specification document.

7. Item/Pass Fail Criteria

The software should be able to pass all the tests for all the features and performance requirements as described in the Software requirements Specification document. Each feature will be considered passed if it satisfies the corresponding requirement and failed if the expected behavior is not met or if any exceptions are raised.

8. Suspension Criteria and Resumption Requirements

8.1 Suspension Criteria

If any of the above features are tested and the test fail or are not satisfactory, testing will be suspended till the bug is traced or corrected. While testing new versions of the software, testing will be suspended if any of the features of the previous release fails the test.

8.2 Resumption Requirements

Testing will be resumed when all the functions listed above work adequately and correctly. When testing for new releases, testing will resume when all the features of the previous release are considered passed.

9. Test Deliverables

The following artifacts are produced after tests are conducted on the simulation software.

- Test Plan
- Test cases and results

10. Environment

All the tests will be conducted on sunflower.cis.ksu.edu and blackeye.cis.ksu.edu which are UNIX machines with Java -1.3 installed on it.

CHAPTER 8: COMPONENT DESIGN

1. Introduction:

The purpose of this document is to outline the design of all the components (classes) of the software and the interaction between them necessary to achieve the desired results. The objective of the project is to develop a parallel program for the Molecular Dynamics simulation of a group of atoms acted upon by an interaction force called the Lennard-Jones force of interaction. The following sections explain in detail all the classes and their functions .The Object Model is used as a reference to explain the functionality of each class.

2. class Atom:

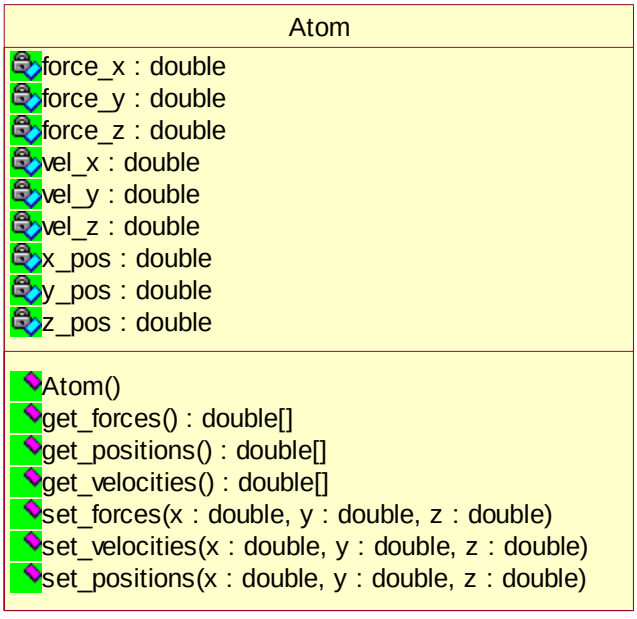


Figure 11: Class Atom

An instance of this class represents an atom in the system. It holds the values of the forces, velocities and coordinates in all the directions. The methods of this Atom class are used to set or get the velocities, forces or coordinates of an atom at any instant of time. The detailed description of the methods of this class is as follows:

set_positions

```
public void set_positions(double x,
                          double y,
                          double z)
```

Sets the coordinates in x, y and z directions
Parameters:
x - - The x coordinate
y - - The y coordinate
z - - The z coordinate

---

set_velocities

```
public void set_velocities(double x,
                           double y,
                           double z)
```

Sets the velocities in x, y and z directions
Parameters:
x - - The velocity in x direction
y - - The velocity in y direction
z - - The velocity in z direction

---

set_forces

```
public void set_forces(double x,
                       double y,
                       double z)
```

Sets the forces in x, y and z directions
Parameters:
x - - The force in x direction
y - - The force in y direction
z - - The force in z direction

---

get_positions

```
public double[] get_positions()
```

Gets the coordinates in x, y and z directions
Returns:
positions - the x, y and z coordinates packed in an array

---

get_velocities

`public double[] get_velocities()`

Gets the velocities in x, y and z directions

Returns:

velocities - the x, y and z velocities packed in an array

---

get_forces

`public double[] get_forces()`

Gets the forces in x, y and z directions

Returns:

forces - the x, y and z forces packed in an array

3. class Barrier

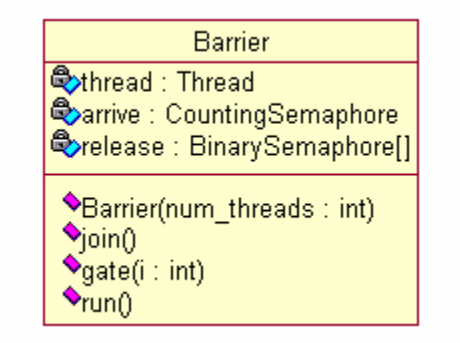


Figure 12: Class Barrier

This class is mainly used for synchronization purposes. A common barrier is shared between all the threads. This is used only when it is required that all the threads stop. If not, the ObjBuf class is used for synchronization between the neighboring threads. Whenever all the threads need to stop at a point and synchronize, each of the threads calls the gate() method in this class. The gate method releases the threads once all the threads arrive. The detailed description of the methods of this class is as follows:

join

`public void join()`

Joins the Barrier thread

---

gate

`public void gate(int i)`

Stops the thread

Parameters:

i - - thread identifier

---

run

`public void run()`

This is the method that is started when the current thread is instantiated

Specified by:

run in interface java.lang.Runnable

4. class BinarySemaphore

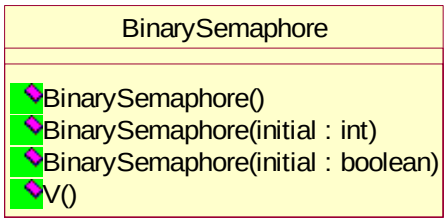


Figure 13: Class BinarySemaphore

This class extends the Semaphore Class and is an implementation of the Binary

Semaphore i.e. this semaphore can have only two values.

5. class CountingSemaphore

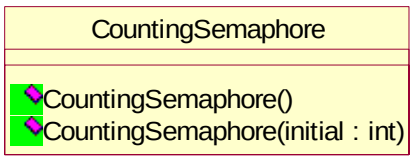


Figure 14: Class CountingSemaphore

This class extends the Semaphore Class and is an implementation of the Counting Semaphore which can have any number of values.

6. class EnergyWriter

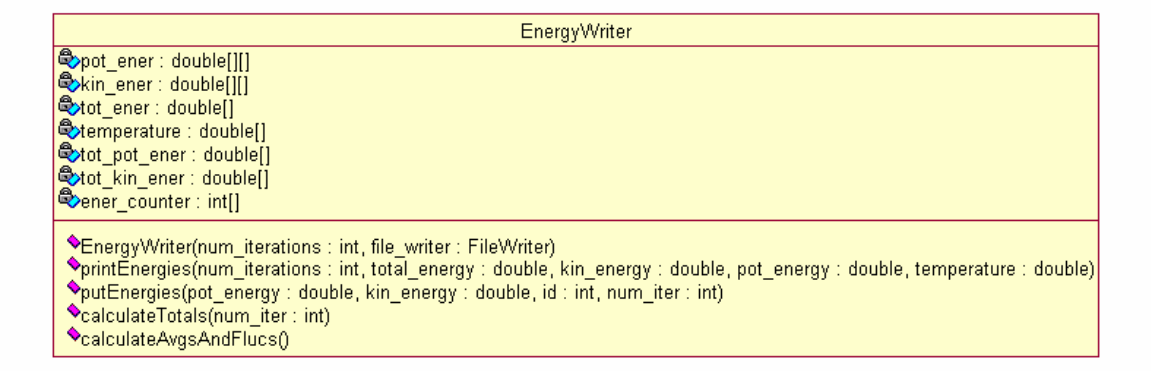


Figure 15: Class EnergyWriter

This class is responsible for collecting the individual contributions of energies from all threads at each iteration step, calculate the total energies, averages, fluctuations and temperatures, and display them to the console as well as write to a file after every given number of steps. The detailed description of the methods of this class is as follows:

putEnergies

```
public void putEnergies(double pot_energy,
                        double kin_energy,
                        int id,
                        int num_iter)
                 throws java.lang.Exception
```

Collects the energy contributions from the individual threads
Parameters:
pot_energy - - potential energy contribution of a thread
kin_energy - - kinetic energy contribution of a thread
id - - thread index
num_iter - - the current iteration step of the thread
Throws:
java.lang.Exception

---

putAtomCount

```
public void putAtomCount(int count,
                         int id,
                         int num_iter)
```

This method is for only debugging purposes, to make sure that an atom is assigned only to one thread in each iteration

Parameters:

count - - Number of atoms that the thread holds

id - - thread index

num_iter - - the current iteration step of the thread

---

calculateTotals

```
public void calculateTotals(int num_iter)
                     throws java.lang.Exception
```

Sum the individual contributions of each thread and calculate the total energy and temperature of the system

Parameters:

num_iter - - the iteration at which these energies and temperature are calculated

Throws:

java.lang.Exception

---

calculateAvgsAndFlucs

```
public void calculateAvgsAndFlucs()
                           throws java.lang.Exception
```

Calculate Averages and Fluctuations of Energies and Temperatures over the length of the simulation

Throws:

java.lang.Exception

---

printEnergies

```
public void printEnergies(int num_iterations,
                          double total_energy,
                          double pot_energy,
                          double kin_energy,
                          double temperature)
                   throws java.lang.Exception
```

Write energies and temperature to a file

Parameters:

num_iterations - - the iteration step at which these are written

total_energy - - total energy of the system

pot_energy - - potential energy of the system

kin_energy - - kinetic energy of the system

temperature - - temperature of the system

Throws:

java.lang.Exception

7. class IO_Utils

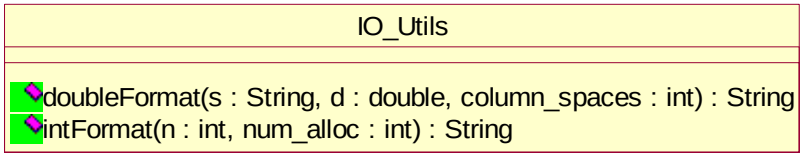


Figure 16: Class IO_Utils

IO_Utils is a helper class which has the methods for formatting numbers in a specific decimal format. This is useful while printing out energies and the final coordinates. The detailed description of the methods of this class is as follows:

doubleFormat

```
public static java.lang.String doubleFormat(java.lang.String s,
                                             double d,
                                             int column_spaces)
```

formats a double number in the required format
Parameters:
s - - the string pattern representing the format e.g. "###.##"
d - - the double that needs to be formatted
column_spaces - - the number of spaces to be allocated for writing to file or console
Returns:
to_return - the formatted number as a string

---

intFormat

```
public static java.lang.String intFormat(int n,
                                         int num_alloc)
```

formats an integer to be outputted in a specified pattern
Parameters:
n - - the integer to be outputted
num_alloc - - the number of spaces to be allocated while writing to file or console
Returns:
to_return - the formatted number as a string

8. class LineReader

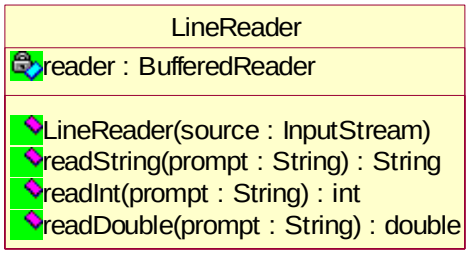


Figure 17: Class LineReader

LineReader is a Helper Class which reads a line of input from the given input stream. This is used to read the positions, velocities and values of the constants used for computation, from input files. The detailed description of the methods of this class is as follows:

readString

```
public java.lang.String readString(java.lang.String prompt)
```

reads a string input
Parameters:
prompt - - the prompt used to ask for input
Returns:
s - the string entered

---

readInt

```
public int readInt(java.lang.String prompt)
```

reads an integer input
Parameters:
prompt - - the prompt used to ask for input
Returns:
input - the integer entered

---

readDouble

```
public double readDouble(java.lang.String prompt)
```

reads a double input
Parameters:

prompt - - the prompt used to ask for input
Returns:
input - the double entered

9. class MdConstants

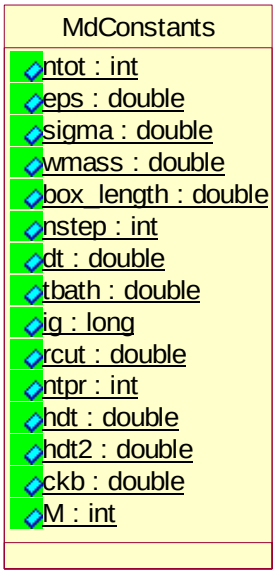


Figure 18: Class MdConstants

This class holds the variables that are central to all classes and have constant values.

10. class MdPar

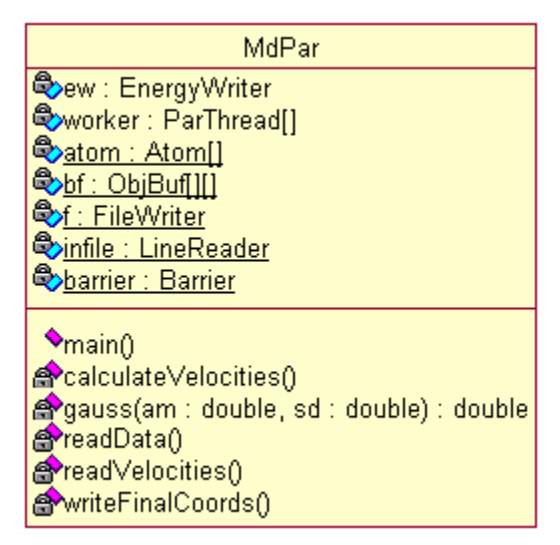


Figure 19: Class MdPar

This is the class which drives the parallel program to simulate a group of atoms acted upon by a force called the Lennard-Jones force of interaction. It instantiates the threads each of which are assigned a number of partitions and responsible for simulating i.e. calculating the forces and displacing the atoms according to the forces, the atoms in the partitions assigned to it. . This is the main class, which is responsible for starting the simulation. It creates an array of threads, joins them and records the time taken for the entire simulation. This class is also responsible for 1) initializing the coordinates and velocities of the Atoms with the data read from the velocity and coordinate input files, 2) initializing the values of the physical constants required in the simulation with the data read from the md.dat input file and 3) write the final positions of the Atoms after simulation to the coordinates output file. The detailed description of the methods of this class is as follows:

main

```
public static void main(java.lang.String[] args)
```

---

readData

```
private static void readData()
```

Read Values from md.dat values and assign them to the values in the Constant class

---

readCoordinates

```
private static void readCoordinates()
```

Read the coordinates from the positions file and assign to the atoms

---

readVelocities

```
private static void readVelocities()
```

Reads the velocities from the file vel.in, this method is called only for testing purposes

---

calculateVelocities

```
private static void calculateVelocities()
```

Calculates and sets the velocities of the atoms based on a random Gaussian distribution

---

gauss

```
private static double gauss(double am,
                            double sd)
```

assigns velocities to the atoms randomly based on the Gaussian distribution
Parameters:
am - - the mean of the distribution
sd - - the random number seed
Returns:
r - the velocity assigned

---

writeFinalCoords

```
private static void writeFinalCoords()
```

Write the Final Coordinates of each atom of the system, as a result of the simulation, to a file

11. class ObjBuf

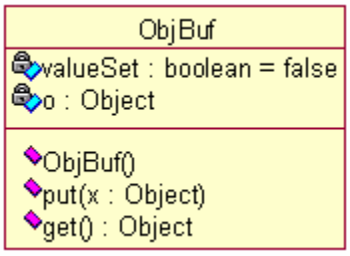


Figure 20: Class ObjBuf

This class represents a buffer in which a thread can put an array of partitions or get an array of partitions from it. The put and get methods are synchronized, thus enabling synchronized communication between the threads. Each pair of neighboring threads shares two instances of this class, one for putting the partitions and another for getting the partitions. The put buffer for a thread will be the get buffer for its neighbor and vice-versa. This class is used while transferring the partitions of a thread to all its neighboring threads at each iteration step. The detailed description of the methods of this class is as follows:

put

`public void put(java.lang.Object x)`

Put an array of Partitions
Parameters:
x - - An Object

---

get

`public java.lang.Object get()`

Get an array of Partitions
Returns:
x - An Object

12. class ParThread

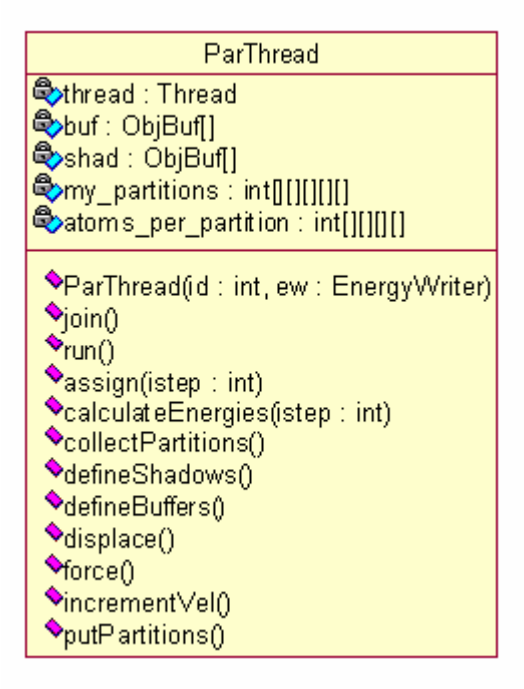


Figure 21: Class ParThread

An instance of this class represents a thread which simulates the atoms in the partitions that are assigned to it. The detailed description of the methods of this class is as follows:

join

```
public void join()
```

Joins the current Thread

---

run

```
public void run()
```

This is the method that is started when the current thread is instantiated
Specified by:
run in interface java.lang.Runnable

---

assign

```
public void assign(int istep)
```

Assigns the atoms to the partitions of the current thread depending on their coordinates

---

incrementVel

```
public void incrementVel()
```

Increment the Velocity as a result of the changes in the Forces for each atom in the partitions of the current thread

---

defineShadows

```
public void defineShadows()
```

Define the Shadows to which the current thread will put its border partitions for its neighbors to Read

---

defineBuffers

```
public void defineBuffers()
```

Define the Buffers from where the current thread gets the border partitions of all its neighboring threads

---

putDummy

```
public void putDummy()
```

Put a dummy object into the neighboring shadows. This is only used for synchronization between neighboring threads

---

getDummy

```
public void getDummy()
```

Get a Dummy Object from the neighboring buffers. This is only used for synchronization between neighboring threads

---

force

```
public void force()
```

Calculate the forces and potential energy due to the interaction between the atoms, for each atom of the current thread

---

calculateEnergies

```
public void calculateEnergies(int istep)
```

Calculate the energy (Kinetic and Potential) contributions of the current thread and report them to the EnergyWriter Class

---

displace

```
public void displace()
```

Update the coordinates of the atoms of the current thread's partitions as a result of the interaction forces

13 class Semaphore

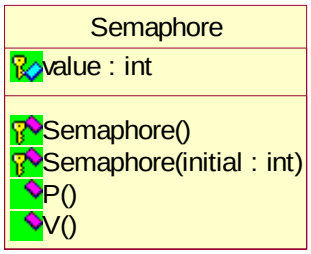


Figure 22: Class Semaphore

This is an abstract class representing a Semaphore, which is a classic method for restricting access to shared resources in a multi threaded environment. This is extended by BinarySemaphore and CountingSemaphore depending on the number of values it can have. The detailed description of the methods of this class is as follows:

P

```
public void P()
```

P stands for Dutch "Proberen", to test. This method busy-waits until a resource is available whereupon it immediately claims one

---

V

```
public void V()
```

V stands for Dutch "Verhogen", to increment. This method simply makes a resource available again after the process has finished using it.

CHAPTER 9: ASSESSMENT EVALUATION

1. Introduction

The purpose of this document is to outline the testing done on the project and the results of testing. The following sections describe the three types of testing done for this project, 1) Testing of the features of the program such as reading the input, 2) Functional testing such as checking if the program gives the same output when executed with any number of threads in the parallel program and 3) Performance testing which actually tests the performance of the parallel program i.e. the speed up achieved when running on multiple processors.

2. References

The following documents are used as a reference:

- Test Plan
- Software Requirements Specification

3. Feature Testing

The following features of the program are tested:

- **Read Data from files:** The program reads a line at a time as a string from the input file and parses the string and then applies the string functions to convert into an integer or a double. So it is very important that the form at is exactly as expected by the parser. The program throws appropriate

errors when an incorrect format is encountered. The table at the end of this section lists the test cases where this feature of the program is tested.

- **Read program arguments:** The program has two arguments, 1) The number of threads which can be 1, 2, 4 or 8 for a small system and 1,2,4,8 or 16 for a large system. The limit is 8 and 16, since the number of partitions in one direction is 8 for a small system and 16 for a large system. 2) An integer which specifies what system to simulate, 0 for small system and 1 for a large system. The program has to catch exceptions and throw an error, if the program has an incorrect number of arguments, or invalid arguments. The table at the end of this section lists the test cases where this feature of the program is tested.
- **Formatting values for output:** The program outputs the total energy, potential energy, kinetic energy and the temperatures at each step in a fixed format i.e. each of these fields are printed with a fixed spacing between them. If the length of one of the fields is larger than the spacing, an exception should be thrown. This in a way also indicates that the values are incorrectly calculated by the program, since usually the lengths of the fields fall within the space allocated for them. As a test case, the forces of each of the atoms are not initialized in the force () method of the program. So, the values of the potential energy and kinetic energy keep increasing until they are no longer small to be printed out in the desired format. This test case is shown in the following table.

The following table describes the test cases and the results:

Test Unit	Test Case	Result
Read Data from files	md.dat input file with line 9 in incorrect format	Error : Line 9 in the file md.dat is not in the format expected.
Read Data from files	vel.in input file with line 38 in incorrect format	Error : Line 38in the file vel.in is not in the format expected.
Read Data from files	init_positions input file with line 256 in incorrect format	Error : Line 256 in the file init_positions is not in the format expected.
Format values for output	Forces are not initialized.	Error: The values of the fields are too large to be printed.
Read program arguments	java mdpar/MdPar	Usage: java mdpar/MdPar arg1 arg2 where arg1: number of threads = 1,2,4 or 8 for a small system = 1,2,4,8 or 16 for large system arg2: system identifier = 0 for small system = 1 for large system
Read Program arguments	java mdpar/MdPar 16 0	Error: Incorrect number of threads for the system chosen Usage: java mdpar/MdPar arg1 arg2 where arg1: number of threads = 1,2,4 or 8 for a small system = 1,2,4,8 or 16 for large system arg2: system identifier = 0 for small system = 1 for large system
Read Program Arguments	Java mdpar/MdPar 4 3	Error: Incorrect choice for system identifier Usage : java mdpar/MdPar

		arg1 arg2 where arg1: number of threads = 1,2,4 or 8 for a small system = 1,2,4,8 or 16 for large system arg2: system identifier = 0 for small system = 1 for large system

Table 7: Test Cases and Results

4. Functional Testing:

The result of the program varies from system to system and for each run, since the velocities are randomly generated using a Gaussian distribution. To verify that the parallel program which is run with 1, 2, 4 and 8 threads is producing the same results each time, the velocities are read from a file "vel.in", instead of generating randomly. The velocities are read from the file only for testing purposes. The results obtained are checked against a standard sequential program written for the same purpose and with the similar inputs. The following is the result for a system of 6860 particles and the input files md_small.dat, init_positions_small and vel.in. The values are printed out for every 10 time steps.

Time Step	Time(ps)	TE	PE	KE	TEMP
0	0	11537.58	-13958.42	25496	298.05
10	0.1	11538.62	-14032.79	25571.42	298.93
20	0.2	11537	-13912.55	25449.55	297.51
30	0.3	11537.44	-14058.29	25595.73	299.22
40	0.4	11537.62	-13931.52	25469.13	297.74
50	0.5	11536.81	-14045.47	25582.27	299.06
Averages					
50	0.5	11537.17	-13997.07	25534.24	298.5
Fluctuations					
50	0.5	1.07	66.48	65.95	0.77

5. Performance Testing

This section describes the performance measurements done on the program, i.e. the speed-up as a function of number of threads and granularity is measured and plotted, for each of the designs that have been come up with while working on the project and also explains the key concepts of each design as an attempt to justify the performance.

- Initial Design: 3-D grid shaped pattern of thread creation and synchronization by message passing through bounded buffers and each thread is assigned to a partition owing to code simplicity. So there are 512 threads. The program took far longer to run when running as 512 threads, which is justified due to the huge overhead in creating so many threads. The number of threads and the system size has been hard coded into the system so there is no possibility of running the program with different arguments.
- Design I: 3-D grid shaped pattern of thread creation and synchronization by message passing through bounded buffers and multiple partitions are assigned to each thread. Since there are 8x8x8 partitions, it would be only possible to create an array of 2x2x2 or 4x4x4 threads with 4x4x4 or 2x2x2 partitions assigned to each thread. Creating 8x8x8 threads and assigning one partition per thread will cause an excessive overhead in thread creation which increases the execution time enormously. For a larger

system similar number of threads can be created with more granularities. The following tables give the speed-ups and efficiencies for different granularities and number of threads. The speed-up is defined as the ratio of the time taken to run on one thread to the tine taken when running on multiple threads. The efficiency is defined as the ratio of the speed-up to the number of processors (equal to 4 in this project).

Less Granularity (System size = 6860 Atoms)

Number of Threads	Time Taken	Speed-up	Efficiency
1	179625	--	--
8	174393	1.03	25.75
64	216415	0.83	20.75

Table 8: Speedup (Design I, fine grained)

More Granularity (System size = 54880 Atoms)

Number of Threads	Time Taken	Speed-up	Efficiency
1	1726549	--	--
8	1676261	1.03	25.75
64	2105547	0.82	20.5

Table 9: Speedup (Design I, coarse grained)

The reasons why the speed up is very low is that though multiple partitions are assigned to one thread, there is a significant amount of message passing which involves copying large 3 dimensional arrays. Also, the number of available processors is only four, so there is a considerable amount of thread switching especially with 64 threads. The amount of

copying is still huge for the larger system, hence the slight decrease in speed-up when compared to the smaller system. The plot of speedup vs. number of threads for different granularities is as shown (In the legend, Fine Grained implies system with less granularity i.e. the smaller system and Coarse Grained implies more granularity i.e. the larger system):

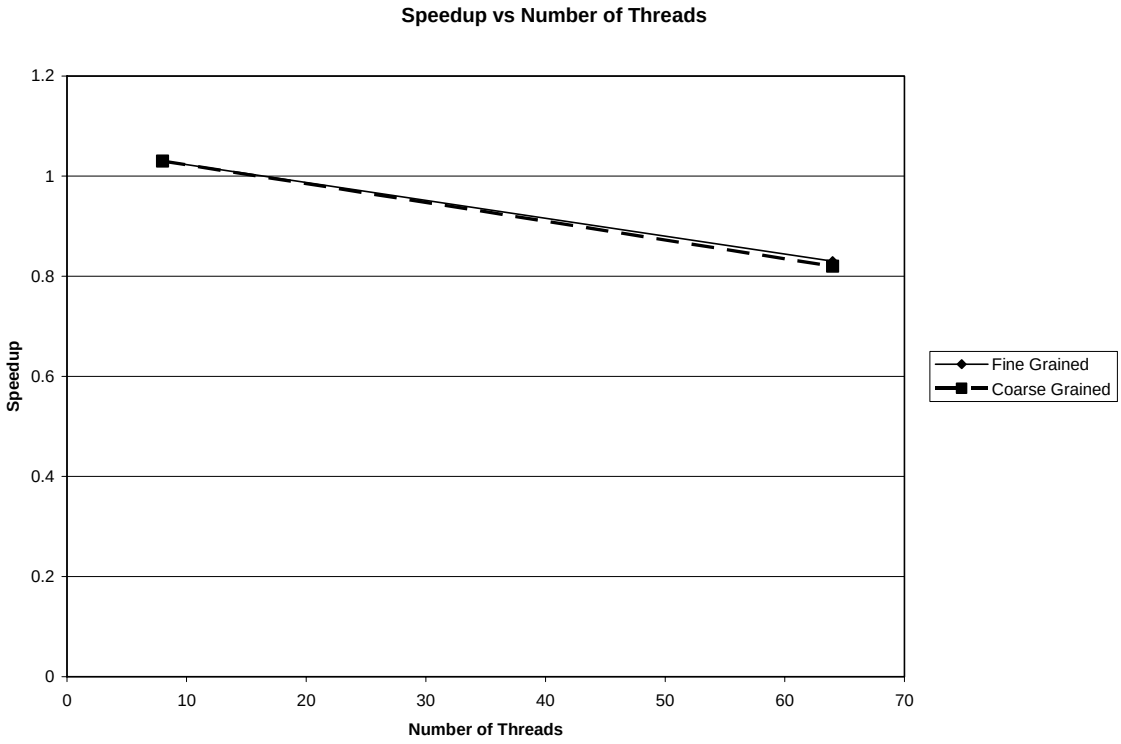


Figure 23: Plot of Speedup vs. no of threads (Design I)

- Design II: Vertical pipeline shaped pattern of thread creation and synchronization by message passing through bounded buffers and multiple partitions are assigned to each thread. The idea here is to assign layers of partitions to each thread rather than a 3-D array of

partitions to each thread like in the previous design. Since there are 8 layers of partitions with 8x8 partitions in each layer, the number of threads could be 1,2 4, or 8 with 8,4,2,1 layers assigned to each thread respectively. Similarly, for the larger system, the number of threads could be 1,2,4,8 or 16 with 16, 8,4,2,1 layers assigned to each thread respectively. The following tables give the speed-ups and efficiencies for different granularities and number of threads.

Less Granularity (System size = 6860 Atoms)

Number of Threads	Time Taken	Speed-up	Efficiency
1	170062	--	--
2	158936	1.07	26.75
4	104796	1.62	40.5
8	126911	1.34	33.5

Table 10: Speedup (Design II , fine grained)

More Granularity (System size = 54880 Atoms)

Number of Threads	Time Taken	Speed-up	Efficiency
1	1699526	--	--
2	1559198	1.09	27.25
4	982384	1.73	43.25
8	1196849	1.42	35.5

Table 11: Speedup (Design II, coarse grained)

The speedup has been improved considerably compared to the previous design, especially with the number of threads equal to 4. This is because, though the synchronization mechanism is message passing, the amount is copying is far less than the previous model since only one layer of partitions is copied rather than copying the whole three dimensional array

of partitions. The speedup with four threads is the highest since the number of processors available are four. The speed up with eight threads is a little less due to thread switching. Also an increase in speedup is observed with the larger system due to more granularities. It is slight since the amount of copying is also increased when moving to larger system. The plot of speedup vs. number of threads for different granularities is as shown (In the legend, Fine Grained implies system with less granularity i.e. the smaller system and Coarse Grained implies more granularity i.e. the larger system):

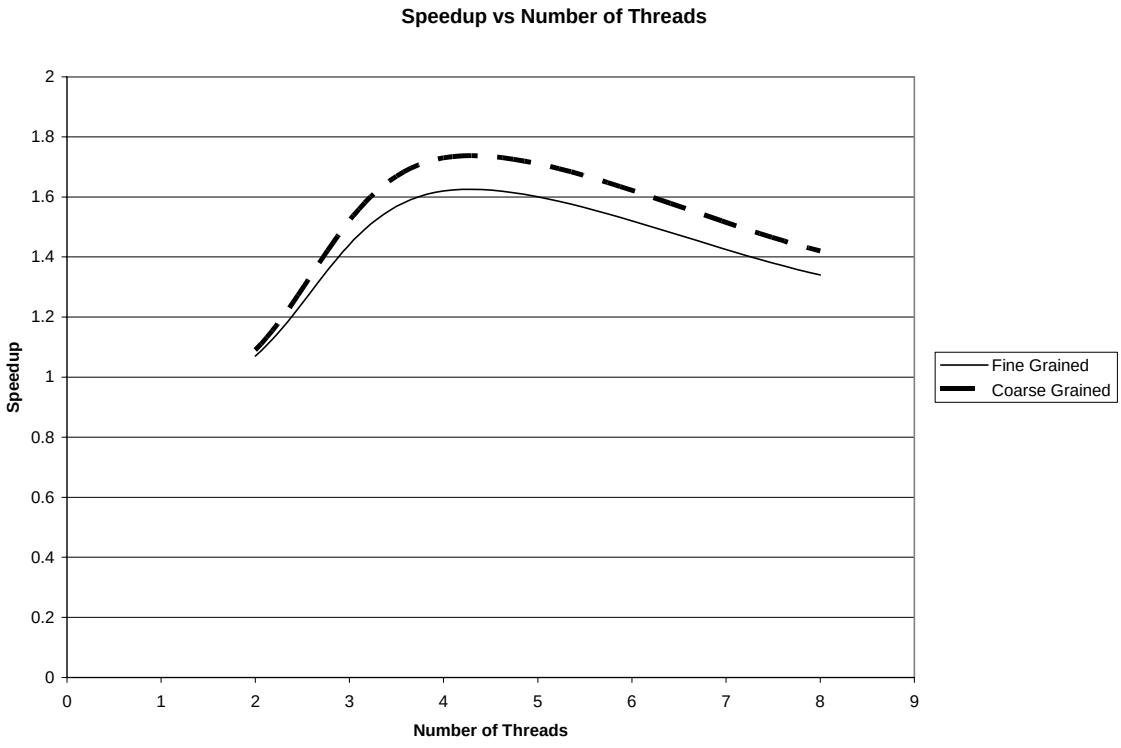


Figure 24: Plot of Speedup vs. no of threads (Design II)

- Final design: Vertical pipeline shaped pattern of thread creation and synchronization by Barrier. Bounded buffer is also used to synchronize

whenever only the neighboring threads need to stop and synchronize. No messages are passed. The Barrier is used when all the threads need to stop and synchronize. The number of threads that could be created and the number of layers of partitions assigned to each are the same as in the previous design. The following tables give the speed-ups and efficiencies for different granularities and number of threads:

Less Granularity (System size = 6860 Atoms)

Number of Threads	Time Taken	Speed-up	Efficiency
1	162653	--	--
2	137841	1.18	29.5
4	62800	2.59	64.75
8	66935	2.43	60.75

Table 12: Speedup (Final Design, fine grained)

More Granularity (System size = 54880 Atoms)

Number of Threads	Time Taken	Speed-up	Efficiency
1	1684963	--	--
2	1306172	1.29	32.25
4	591215	2.85	71.25
8	640670	2.63	65.75

Table 13: Speedup (Final Design, coarse grained)

The speedup has been improved considerably compared to the previous design. This is because the synchronization mechanism is changed from bounded buffer to barrier. Thus, there is no message passing and hence no copying of huge data structures at each step. Instead the data structures are static which each of the threads can access and the critical sections are guarded by barriers. Also an increase in speedup is observed with the

larger system due to more granularities. The maximum speed up that could be achieved is only 2.85. A possible reason for this is that all the four threads are not really running independent of each other. They will have to stop at different points to synchronize with each other. In this design, all the threads will have to stop twice and the neighboring threads have to stop four times to synchronize during a single time step. So, delays occur and there is a decrease in speedup. The plot of speedup vs. number of threads for different granularities is as shown:

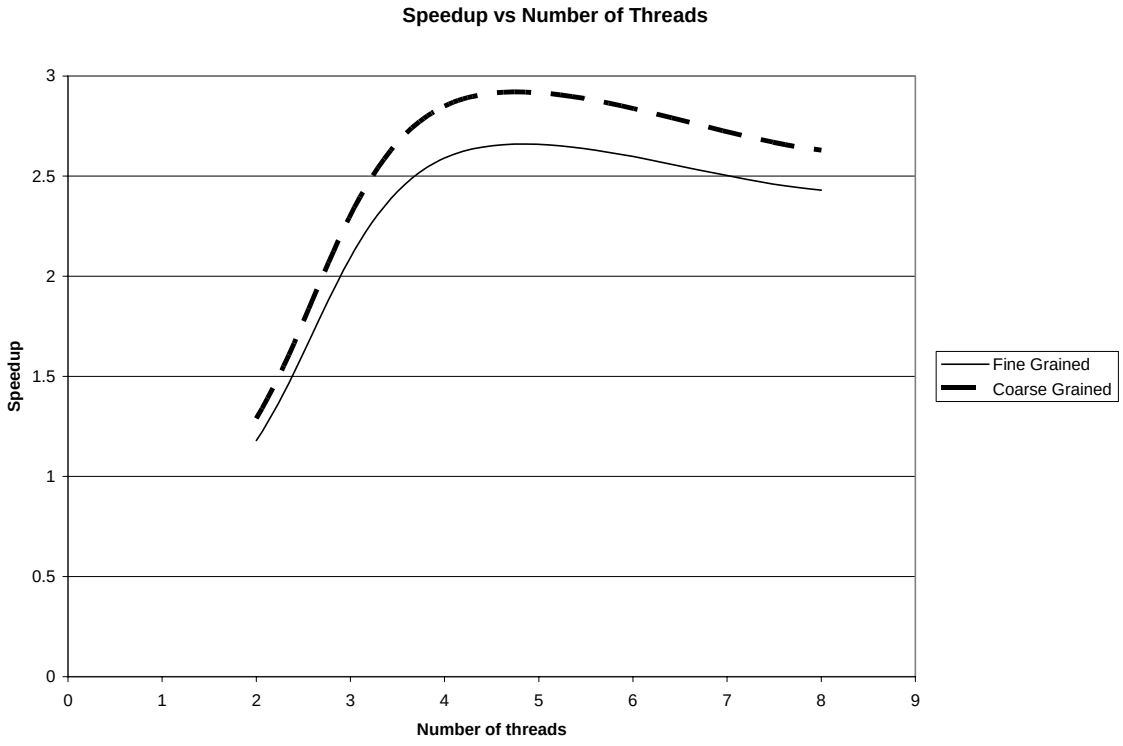


Figure 25: Plot of Speedup vs. no of threads (Final Design)

Notes:

- It was ensured that none of the users are logged on to the system except the Developer while timing the programs. And the %CPU of the user averaged around 94%.
- All the I/O operations in the program were commented out for timing the programs.
- All the timings are for 50 iteration steps in the simulation.

CHAPTER 10: USER MANUAL

1. Introduction

The purpose of this project is to develop software in Java that uses MD Simulation technique to simulate the interaction between atoms in a group of molecules which interact due to Lennard-Jones potential (or any other similar system whose motion can be simulated by stepping through discrete instants of time). Multi-threaded programming that can be executed on more than one processor will be used to improve the efficiency of the system. Various designs for a parallel program based on 1) Synchronization mechanism, 2) the pattern of thread creation and 3) Granularity are implemented and performance measurements are done to calculate the speed-up for all the implementation of each design.

The following sections describe the data formats, the usage of the program, user commands, the system configuration and how to carry out the performance measurements.

2. Data Formats

2.1 Input Data Format

The software has two input data files, the coordinates input file (init_positions_small for the system with 6860 atoms and init_positions_big for the system with 54880 atoms) and the data input file (md_small.dat for the system with 6860 atoms and md_big.dat for the system with 54880 atoms). The format for the data input file should be as follows:

6860	NTOT	number of particles
1.0000	EPS	epsilon (kJ/mol)
0.3000	SIGMA	sigma (nm)
20.0	WMASS	mass (amu)
8.09202	BOX	box length (nm)
50	NSTEP	number of dynamics steps
0.0100	DT	time step (ps)
300.0	TBATH	initial temperature (K)
17841	IG	random number seed
1.00	RCUT	LJ cutoff distance (nm)
10	NTPR	print energies every NTPR steps
1000	NTWX	write coordinates every NTWX steps
1000	NTWE	write energies every NTWE steps

Basically the constraint is that the values should occupy at most the first 10 characters of each line and the order of the property – value pairs should be preserved i.e. NTOT should always be the first line, followed by EPS etc.

The format of the coordinates input file should follow the pdb (protein data bank) format. For more information on the pdb format, please refer http://www.umass.edu/microbio/rasmol/pdb.htm. The first few lines of the coordinates input files are as follows:

```
ATOM      1  B   B      1     -20.512   7.215  33.289  0.00  0.00
ATOM      2  B   B      2     -22.849   1.471 -25.234  0.00  0.00
ATOM      3  B   B      3      40.181 -34.958  10.244  0.00  0.00
ATOM      4  B   B      4      27.884  19.728 -18.005  0.00  0.00
ATOM      5  B   B      5     -18.798  15.000   5.051  0.00  0.00
ATOM      6  B   B      6      -1.832  10.136  39.148  0.00  0.00
ATOM      7  B   B      7     -29.855  18.356  22.965  0.00  0.00
```

For the purpose of this project, the constraints are 1) the total number of lines in this file should be equal to the number of atoms in the simulation system and 2) the x, y and z

coordinate values should be between the 26th and 38th character positions, 39th and 46th character positions, 47th and 54th character positions respectively.

For testing purposes, the velocities have to be read from the file “vel.in” (this is only for the system with 6860 particles). The total number of lines in this file should be equal to the number of atoms in the simulation system and the format is shown as below (the first few lines in the file are shown):

```
-0.468415343794807          0.366741958309217          0.798061712445195
 2.630747657094827E-002 -6.747689914408322E-002 -7.582933435198365E-002
-0.617412010567999          0.339550740206594         -0.167975555032691
 0.585295550026653         -0.191146826899770          0.293272860486950
 6.131226309386512E-002 -0.182546512969571         -0.365545674040731
 0.106814174639356         -0.107245268069441         -0.423193642978872
```

Basically, the constraint is that the each of the velocities in a line should be within the allocated number of characters for it which is 24, i.e. the x coordinate of velocity should be within the first 24 characters, and the y coordinate of velocity should be within the next 24 characters and so on.

2.2 Output Data Format

There are two output files for this program:

final_positions: The final coordinates in all directions (x, y and z) of each atom are written to this file after the end of the simulation. The format of the file is similar to the coordinates input file.

md.out: The energies and the temperature at every given number of time steps are written to this file. Also written at the end of the file are the averages and fluctuations of

the energies and temperature during the entire simulation. The following shows a sample output file when we ask the program to print energies and temperatures every ten steps:

Time Step	Time (ps)	TE	PE	KE	TEMP
0	0	11537.58	-13958.42	25496	298.05
10	0.1	11538.62	-14032.79	25571.42	298.93
20	0.2	11537	-13912.55	25449.55	297.51
30	0.3	11537.44	-14058.29	25595.73	299.22
40	0.4	11537.62	-13931.52	25469.13	297.74
50	0.5	11536.81	-14045.47	25582.27	299.06
Averages					
50	0.5	11537.17	-13997.07	25534.24	298.5
Fluctuations					
50	0.5	1.07	66.48	65.95	0.77

3. Using the System

3.1 Changing the Inputs

The Parameters that one would want to change frequently are the:

- Total number of atoms in the system
- Total number of simulation steps
- Size of the simulation system i.e. the length of the simulation box
- The interval (number of steps) at which to print energies and temperatures

To change the number of atoms in the simulation system, edit the input data file and change the value of NTOT accordingly. We also need to make sure that the coordinates input file has the exact number of entries corresponding to the total number of atoms in the system. In the current project, an initial coordinates file is given which corresponds to

a system with 6860 particles and a simulation box length (represented by BOX in the input data file) of 8.09202 nm. We also need a larger system for higher granularity i.e. the computation per thread relative to the communication between the threads, to measure the speed up as a function of granularity. So, a small program “replicator.java” is written to replicate the system of 6860 particles into a bigger system with 54880 particles i.e. eight times the size of the original system. Correspondingly the parameter BOX has to be changed to 16.18404 nm i.e. twice the length of the original simulation box.

To facilitate ease of use and since there are only two sizes of the system we wish to simulate, the input data file is split into two files:

1) md_small.dat, which is read by the program when simulating the small system. The values of NTOT and BOX in this file are always 6860 and 8.09202 respectively.
2) md_big.dat, which is read by the program when simulating the big system. The values of NTOT and BOX in this file are always 6860 and 8.09202 respectively.

This obviates the changing of the input data file every time we switch between simulation of small system and big system. The other parameters such as the number of simulation steps should be changed via the corresponding input data file though.

The total number of simulation steps can be changed via the parameter NSTEP in the input data file. We might need to increase the number of simulation steps speculating for

an increase in speed-up, since there is an overhead in creating the threads and a lengthy simulation might significantly reduce the effect. The default value for NSTEP is 50.

The NTPR parameter in the input data file has to be changed to change the interval (number of steps) at which to print energies and temperatures. The default is 10 i.e. the energies and the temperature of the system are printed to the console and a file after every 10 steps. It is advisable to keep this value as high as possible, since the frequent I/O operations reduce the efficiency of the system.

3.2 User Commands

3.2.1 To compile the source code:

1) Navigate to the src directory of the corresponding folder (sequential for the sequential program and parallel for the parallel program).

2) Run the following command:

java –d ../classes mdseq/.java* -- for compiling the sequential program

java –d ../classes mdpar/.java* -- for compiling the parallel program

3.2.1 To generate javadoc documentation API:

1) Navigate to the src directory of the corresponding folder (sequential for the sequential program and parallel for the parallel program).

2) Run the following command:

javadoc –d ../docs mdseq/.java*

-- for generating API for the sequential program

java –d ../docs mdpar/.java*

-- for generating API for the parallel program

3.2.1 To run the program

1) Navigate to the src directory of the corresponding folder (sequential for the sequential program and parallel for the parallel program).

2) Execute the following command for running the sequential program:

java mdseq/MdSeq

3) Execute the following command to run the parallel program:

java mdpar/MdPar arg1 arg2

Where

arg1: number of threads

= 1, 2, 4 or 8 for a small system

= 1,2,4,8 or 16 for large system

arg2: system identifier

= 0 for small system

= 1 for large system

Note: The above commands are stated assuming that the user installs the source code on a system according to the instructions in the System configuration and Installation section and the directory structure is preserved.

3.3 Performance Measurements

One of the Objectives of the project is to examine the speed-up of the parallel program as a function of the number of threads and the granularity for each design as stated in the Architecture design document. To conduct the performance measurements the program is run with different arguments as shown in the previous section and the time taken to run is noted. The number of threads to run can be varied by changing the first argument and the granularity in each run can be varied by choosing to run the smaller system or the larger system which requires changing the second argument.

4. System configuration and Installation

The program could be run on any multi processor system with Java-1.3 or higher installed in it. To install Java please visit http://java.sun.com . The user commands listed in the previous section were run on a UNIX terminal. The source code is available from the project's website at http://www.cis.ksu.edu/~ganti/mse_pro.htm in the form of a zip file. Extract the contents to a working directory. The directory structure is explained as follows. The top level directories separate the parallel program (parallel) and the sequential program (sequential). Inside each directory:

- The src folder has the actual source code.
- The classes folder has the compiled classes
- All the necessary input files are placed in the resources folder
- All the output files will be placed in the results folder
- All the generated documentation is in the docs folder

There is an additional folder named utils folder at the top level apart from the parallel and sequential folders which has the source code for the replicator program that generates an initial coordinates file for the bigger system (initial_positions_big) by taking the initial coordinates file for the smaller system (initial_positions_small) as an input. The user has to preserve the directory structure for the user commands listed in the previous section.

5. Useful Tips for running parallel programs

- It is often useful to know if the all the threads in the parallel program are indeed utilizing all the processors available in the system. To determine this, a utility for UNIX systems called top is used. Running the top command in a terminal gives the usage statistics of the processors of the system. This is what top might look like if you had a multithreaded application which was taking up a large amount of cpu time on all processors. The main thing to look for is that if the %CPU does (in this case 96, 96, 94.1, and 90.5) add up to over 100, then all the available processors are being utilized.

```
PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  Command
706 ganti     18   0  6264 6264 4252 R 96.0  0.2   0:05.20 java
751 ganti     18   0  6256 6256 4256 R 96.0  0.2   0:03.18 java
616 ganti     17   0  6264 6264 4252 R 94.1  0.2   0:10.85 java
661 ganti     15   0  6264 6264 4252 R 90.5  0.2   0:08.58 java
```

- When timing the programs it is important that the CPU is exclusively used by the program to give accurate results. That is, the user needs to make sure that there are no other programs currently running that take the CPU processing time. The top command is useful for this purpose also. It lists all the processes that are being executed currently.

- Some operating systems are set to run the threads as native by default i.e. the threads spawn all the processors available. But in case where it is not so, the –native option of java is to be used.

CHAPTER 11: PROJECT EVALUATION

1. Introduction:

This document describes the review of the project in terms of the problems encountered, accuracy of the estimations, usefulness of the reviews and the methodologies used. It also describes the evaluation of the project for whether it accomplishes the ideas presented in the initial overview.

2. Problems Encountered

2.1 JPF

JPF is a tool for verification and debugging of Java programs based on model checking. A lot of learning and researching had to be done by the developer to figure out the options offered by the tool for the verification of the program. The default options take a very long time to complete the verification owing to the huge state space of the program.

2.2 Debugging Parallel Programs

Owing to the inherent complex nature of parallel programs, the developer had to spend a lot of time debugging for the implementation of each design. The developer had to implement different parallel algorithms before arriving at the final design which is best suited for Molecular Dynamics Simulation.

2.3 Limited availability of systems

The developer had access to only one multi processor system and has to wait for times when no users are logged on, to time the execution of the parallel program with varying number of threads. It is important that the CPU is used entirely by the developer's program for the speed-up and efficiency results to be correct. Also, the system often rebooted with only one processor accessible.

2.4 Limited processing power of available systems

The available four processor systems sunflower.cis.ksu.edu and blackeye.cis.ksu.edu have a limited processing power. Each processor is only in the range of 400 MHz – 500 MHz. Running the simulation with the larger system (i.e. 54880 atoms) took as long as 35 minutes.

2.5 Breaks

The developer had to take long breaks in the duration of the project due to unforeseen reasons which made it difficult to recapitulate the work done in the previous phase.

3. Accuracy of the Estimates

3.1 Lines of Code

The following table shows the estimated LOC and the actual LOC for the sequential and the parallel program:

	Estimated LOC	Actual LOC
Sequential	1435	504
Parallel	1545	1271

Table 14: Estimated and actual LOC

The reason for the wide variation in LOC of sequential program could be that it is too simple to be estimated. The number of LOC for the parallel program is quite close to the estimate.

3.2 Cost Estimation

The cost estimation was done using the functional point analysis and the COCOMO model. The total time taken for the sequential program and the parallel program is 3.9 + 4.2 = 8.1 months. The duration of the project is approximately 7 months which is quite close to the estimate.

4. Lessons Learnt

4.1 Methodology

On completion of the project, the developer realizes the usefulness of following the software methodologies and the life cycle. The MSE portfolio served as a useful guide throughout the project. The developer believes that this project experience has equipped him with a better understanding of a software life cycle and will be a guiding factor in future software projects.

4.2 Reviews

The usefulness of the reviews and the feedback is an important lesson learnt in doing this project. Besides getting valuable feedback from the major advisor, the committee as a whole also reviewed the progress of the project and gave valuable inputs during presentations, which made this project learning experience and helped to improve the quality of the product.

5. Results

As described in the overview, various parallel algorithms based on 1) Synchronization mechanism, 2) the pattern of thread creation and 3) Granularity, are implemented to determine the best suitable design for the Molecular Dynamics Simulation. Its is determined that an implementation based on the Vertical pipeline thread creation pattern with the mixed use of a Barrier and a Bounded Buffer for synchronization and a large granularity, yielded the highest speed-up. The source code for this design is submitted with the Final report. The results for all the designs were documented in the Performance Testing section of the Assessment Evaluation document.

The maximum speed-up achieved in the project is 2.85 four threads running on a four processor machine. So, the efficiency is (2.85/4)*100 = 71.25%. The reason attributed to this is that all the four threads are not really running independent of each other. They will have to stop at different points to synchronize with each other. In this design, all the threads will have to stop twice and the neighboring threads have to stop four times to synchronize during a single time step. So, delays occur and the efficiency is substantially decreased.

CHAPTER 12: FORMAL TECHNICAL INSPECTION

1. Introduction

The purpose of this document is to provide a formal checklist to inspect the architecture design document of the Molecular Dynamics Simulation project. The purpose of the formal technical inspection process is to ensure the quality and feasibility of the architecture design. Two independent MSE students will perform the inspection and their report on the result of the inspection will be documented.

2. Items to be inspected

The Architecture design document is the item to be inspected. The inspectors will be provided with the vision document for reference purpose.

3. Organization

Graduate Committee

Dr. Virgil Wallentine – Major Professor

Dr. Paul Smith

Dr. Mitch Neilsen

Developer

Lakshmikanth Ganti

Formal Technical Inspection Performed by

Srinivas Kolluri

Laxminarayan M

4. Formal Technical Inspection Checklist

Questions	Yes/No/Partial	Comments
Are design decisions for the current release documented as completely and as thoroughly as is known at the present time?	Yes	The document is iteratively written for each design.
Are the design views presented as per UML standards?	Yes	Class diagrams and sequence diagrams are used to illustrate the program
Is a class diagram present in the design document?	Yes	
Does the design document talk about software architecture and how the threads are created and how they communicate?	Yes	Different thread creation patterns and the communication between them are explained.
Single Interpretation: Does every design decision documented in SDD have only a single interpretation that is the same for both those who produce it and those who read it?	Yes	
Does the SDD document all significant unit design decisions?	Yes	
Is the SDD consistent with higher-level documents (e.g., System Requirements Specification, Project Glossary, Domain Object Model, and Software Architecture Document)?	Yes	

Does the SDD have a coherent, easy-to-use organization?	Yes	
Are the design decisions neither redundantly stated nor intermingled?	Yes	
Is consistent level of detail provided with design statements?	Yes	

Table 15: Formal Technical Inspection Checklist

REFERENCES

[1] D.C.Rapaport, “The Art of Molecular Dynamics Simulation”, Cambridge University Press.

[2] http://polymer.bu.edu/Wasser/robert/work/node8.html

[3] http://www.umass.edu/microbio/rasmol/pdb.htm

[4] http://www.cis.ksu.edu/classes/625/lectures/intro.htm

[5] Charles Blilie, Patterns in Scientific Software: An Introduction, Computing in Science and Engineering, May/June 2002.

[6] R.W.Hockney and J.W.Eastwood, Computer Simulation Using Particles, Adam Hilger, Philadelphia, 1998, pp. 6-23.

[7] Software Engineering: A Practitioner’s Approach, Roger S. Pressman, 5th Ed, McGraw- Hill.

[8] Java Path Finder User Guide, Klaus Havelund, NASA Ames Research Centre.

[9] Gregory R. Andrews, “Multithreaded Parallel and Distributed Programming”, Addison-Wesley

[10] IEEE STD 830-1998, “IEEE Recommended Practice for Software Requirements Specifications”, IEEE, New York

[11] IEEE STD 730 – 1998, “IEEE Standard for Software Quality Assurance Plans”, IEEE, New York

[12] http://www.cis.ksu.edu/~sdeloach/748/protected/home.html, Course Slides by Dr. Scott Deloach

[13] http://satc.gsfc.nasa.gov/fi/, Formal Technical Inspection Checklist

APPENDIX A

Source Code for verification with JPF

```
import gov.nasa.arc.ase.jpf.jvm.Verify;
import java.lang.Math;

class ObjBuf
{
	private int[] array;
	private boolean valueSet = false;
	public ObjBuf() {}

	public synchronized void put(int[] x)
	{
		if(valueSet)
			try{wait();}
			catch(InterruptedException e) {Verify.print("Exception Caught"
+e.toString());}
		array = new int[x.length];
		for(int i=0; i<x.length;i++)
			array[i] = x[i];
		valueSet=true;
		notify();
	}
	public synchronized int[] get()
	{
		if(!valueSet)
			try{wait();}
			catch(InterruptedException e) {Verify.print("Exception Caught"
+e.toString());}
		int[] x = new int[array.length];
		for(int i=0; i<array.length;i++)
			x[i] = array[i];
		valueSet=false;
		notify();
		return x;
	}
}


class MD_Thread  extends Thread
{
	private ObjBuf put_buffer1;
	private ObjBuf get_buffer1;
	private ObjBuf put_buffer2;
```

```
        private ObjBuf get_buffer2;
        private int num_iterations = 5;
        private int iter_step = 0;
        private int length = 1;
        private int[] putArray1;
        private int[] getArray1;
        private int[] putArray2;
        private int[] getArray2;
        private int xid;
        private int yid;
        private boolean has_two_neighbors;

        public MD_Thread(ObjBuf put_buffer1, ObjBuf get_buffer1, ObjBuf put_buffer2,
ObjBuf get_buffer2,int xid,int yid,boolean has_two_neighbors)
        {
                this.put_buffer1 = put_buffer1;
                this.get_buffer1 = get_buffer1;
                this.put_buffer2 = put_buffer2;
                this.get_buffer2 = get_buffer2;
                this.xid = xid;
                this.yid = yid;
                this.has_two_neighbors = has_two_neighbors;
                putArray1 = new int[length];
                putArray2 = new int[length];
                this.start();
        }

        public void run()
        {
                for(int i = 0; i< num_iterations; i++)
                {
                        putArray1[0] = iter_step;
                        putArray2[0] = iter_step;
                        try
                        {
                                put_buffer1.put(putArray1);
                                Verify.print("put array1, thread " + xid + " " + yid + " " +
iter_step);
                                getArray1 = get_buffer1.get();
                                Verify.print("Got array1, thread " + xid + " " + yid + " " +
iter_step);
                                if (has_two_neighbors)
                                {
                                        put_buffer2.put(putArray2);
                                        Verify.print("put array2, thread " + xid + " " + yid
+ " " + iter_step);
```

```
                                        getArray2 = get_buffer2.get();
                                        Verify.print("Got array2, thread " + xid + " " + yid
+ " " + iter_step);
                                }
                        }
                        catch (Exception e)
                        {
                                Verify.print("Exception Caught" +e.toString());
                        }

                        if(has_two_neighbors)
                                Verify.assertTrue("neighbor out of
step",(Math.abs(getArray1[0]- iter_step)<=1 ) && (Math.abs(getArray2[0]-iter_step)
<=1));
                        iter_step++;
                }
                Verify.print("End of iterations");
        }

}


class Parameters
{
        static final int ObjBuf_size = 4;
        static final int MD_Thread_size = 3;
}

class mse_check2
{
        public static void main(String[] args)
        {
                ObjBuf   b1 = new ObjBuf();
                ObjBuf   b2 = new ObjBuf();
                ObjBuf   b3 = new ObjBuf();
                ObjBuf   b4 = new ObjBuf();
                MD_Thread  md1 = new MD_Thread(b1,b2,b3,b4,0,0,true);
                MD_Thread  md2 = new MD_Thread(b2,b1,b3,b4,0,1,false);
                MD_Thread  md3 = new MD_Thread(b4,b3,b3,b4,1,0,false);
                /*try {
                                md1.join();
                                md2.join();
                                md3.join();
                        }
                catch(Exception e)
                {
```

```
                Verify.print(e.toString());
            }
            Verify.print("executing of threads complete, going to quit");*/
        }
}
```