LEANMD: A CHARM++ FRAMEWORK FOR HIGH PERFORMANCE MOLECULAR DYNAMICS SIMULATION ON LARGE PARALLEL MACHINES

BY

VIKAS MEHTA

B.Tech., Banaras Hindu University, 2001

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

Abstract

Molecular dynamics programs simulate the behavior of biomolecular systems, leading to insights and understanding of their functions. However, the computational complexity of such simulations is enormous. Parallel machines provide the potential to meet this computational challenge. To harness this potential, it is necessary to develop a program that can scale well on large parallel machines. Application domain programmers should be able to easily reuse the parallel program with minimal modifications to integrate their science routines and test their motivations.

This thesis presents LeanMD, a parallel molecular dynamics simulation framework written in Charm++ for PetaFLOP class supercomputers. LeanMD is designed to be scalable to large parallel machines (with tens of thousands of processors). LeanMD uses fine-grained spatial decomposition combined with force decomposition to enhance its scalability. The computation is modeled using a large number of virtual processors, which are mapped flexibly to available processors with assistance from the Charm++ runtime system. Charm++ allows the use of "parallel" libraries to facilitate common operations such as 3-D FFTs. Charm++ also provides libraries for communication optimizations and has built in support for automatic load-balancing.

To my parents and my brother.

Acknowledgments

I would first like to thank my adviser, Professor Laxmikant Kale, for his guidance.

I would like to thank my senior colleagues at the Parallel Programming Laboratory: Gengbin Zheng, Orion Lawlor, Sameer Kumar, Chee Wai. They have provided valuable inputs to this work. Gengbin particularly helped me a lot with design and implementation issues. This work would not have been possible without him. Thanks to Chee Wai, for helping me with projections (the performance analysis tool) and thanks to Sayantan for helping with many Charm++ related technical issues and keeping me motivated. Thanks to my colleagues in PPL, who have been great friends, and have made my stay memorable.

I would like to thank application researchers Glenn J. Martyna, Mark E. Tuckerman and Dawn A. Yarne for providing science routines for integration with LeanMD. They helped me a lot in understanding the Molecular dynamics simulation algorithm.

I would like to thank my parents and my brother for their support and guidance.

Table of Contents

List of Tables viii

List of Figures ix

Chapter 1 Introduction 1
- 1.1 Molecular Dynamics Simulation 1
- 1.2 Importance of Molecular Dynamics Simulation Software 1
- 1.3 Increase in System Size and Simulation Length 2
- 1.4 Importance of Parallel Computing to Simulations 2
- 1.5 Increasing Availability of Large Parallel Resources 3
- 1.6 Other Available Simulation Programs 3
- 1.7 Motivation for LeanMD Development 5
- 1.8 LeanMD Development History 5
- 1.9 My Contribution to LeanMD Project 5
- 1.10 Thesis Organization 6

Chapter 2 The Computation 7
- 2.1 Molecular Dynamics Simulation 7
  - 2.1.1 Intra–molecular Interactions 7
  - 2.1.2 Inter-molecular Interactions 8
  - 2.1.3 Molecular Dynamics Simulation Algorithm 9
- 2.2 Parallelization 11
  - 2.2.1 Data Decomposition 12
  - 2.2.2 Parallelization of Non-Bonded Force Computations 14
  - 2.2.3 Parallelization of Bonded Force Computations 17
  - 2.2.4 Parallelization of PME Method 19
  - 2.2.5 Integration 21
  - 2.2.6 Exclusions in Non-Bonded Force Computation 22

Chapter 3 LeanMD as Parallel Molecular Dynamics Simulation Framework 24
- 3.1 Simulation Runtime Options 24
- 3.2 Using LeanMD for Simulating Different Motivations 25
  - 3.2.1 Modifying/Replacing Science Routines 26
  - 3.2.2 Integrating Different Data Reader 26
  - 3.2.3 Adding New Force Fields and Interaction Types 27

3.3 An Example: LeanMD with PINY Science and Data Reader 29

Chapter 4 Performance and Optimizations 30
4.1 Machine and Dataset used for LeanMD Performance Testing 30
4.2 Simulation Configuration 30
4.3 Solving Load Imbalance Problem 31
4.4 A Communication Optimization 34
4.5 Usage Profile of a LeanMD Simulation 36

Chapter 5 Conclusion and Future Work 39

References 41

List of Tables

4.1 Execution times for 1-away LeanMD simulation before load balancing. These simulations do not use the section multicast library. Time per step reported in this table is average over five simulation steps. 32
4.2 Execution times for 1-away LeanMD simulation before load balancing. These simulations use section multicast library. Time per step reported in this table is average over five simulation steps. 32
4.3 Execution times for 1-away LeanMD simulation on 64 processors comparing the Load Balancing strategies available in Charm++. These simulations do not use section multicast library. Time per step reported in this table is average over five simulation steps. 33
4.4 Execution times for 1-away LeanMD simulation after load balancing (Greedy) without using section multicast library. Time per step reported in this table is average over fifteen simulation steps. 34
4.5 Execution times for 1-away LeanMD simulation after load balancing (Greedy) using section multicast library. Time per step reported in this table is average over fifteen simulation steps. 35

List of Figures

2.1 Schematic flowchart of a typical MD algorithm 9
2.2 A serial view of a typical MD algorithm, showing data dependencies 10
2.3 Parallel structure of our implementation 11
2.4 Data decomposition in LeanMD 13
2.5 Algorithm to cache bond info in cell-pair object 16
2.6 Algorithm to calculate bond interactions in cell-pair object 16
2.7 Algorithm used by cell to know the atoms that need to be sent to intramolecular compute objects 17
2.8 Algorithm used by intramolecular compute objects to calculate intramolecular interactions 18

3.1 LeanMD as parallel molecular dynamics simulation framework 25
3.2 Interaction between LeanMD and science routines 26
3.3 Interaction between LeanMD and data reader 27

4.1 This graph shows the overview of LeanMD 1-away simulation on 64 processors. In this simulation, RefineLB strategy was used for load balancing. 31
4.2 This graph shows the overview of a LeanMD 1-away simulation on 64 processors. In this simulation, GreedyLB strategy was used for load balancing. . . 36
4.3 This graph shows the average processor utilization of a LeanMD 1-away simulation on 64 processors over time. In this simulation, GreedyLB strategy was used for load balancing. 37
4.4 A 512 processor 1-away LeanMD simulation showing 70% time consumption by inter-molecular force computation routine. 38

Chapter 1

Introduction

1.1 Molecular Dynamics Simulation

In a molecular dynamics (MD) simulation, full atomic coordinates of proteins, nucleic acids, and/or lipids of interest, as well as explicit water and ions, are obtained from known crystallographic or other structures. An empirical energy function, which consists of both intermolecular and intramolecular interactions, is applied. Intramolecular interactions are modeled using harmonic approximations to describe the bonds and bends between atoms. The intermolecular interactions are modeled using Lennard-Jones 12-6 potential and coulombic interactions to describe the interaction of point charges. The resulting Newtonian equations of motion are typically integrated by symplectic and reversible methods. Modifications are made to the equations of motion to control temperature and pressure during the simulation.

1.2 Importance of Molecular Dynamics Simulation Software

The application of molecular dynamics simulation methods in biomedicine is directly dependent upon the capability, performance, usability and availability of the required simulation and analysis software. Simulations often require substantial computing resources, sometimes available only at supercomputing centers. Development of LeanMD provides the biomedical research community with a freely available software framework in which science routines can be modified/replaced, for performing high quality (with respect to accuracy and performance) MD simulations of large biomolecular systems using a variety of available and cost-effective hardware.

1.3 Increase in System Size and Simulation Length

With continuing increase in high performance computing technology, the domain of biomolecular simulation has rapidly expanded from isolated proteins in solvent to include complex aggregates, often in a lipid environment. Such simulations can easily exceed 100,000 atoms. Similarly, studying the function of even the simplest of biomolecular machines requires simulation of 10 ns or longer, even when techniques for accelerating processes of interest are employed.

However, getting good speedup for small systems (25000 - 30000 atoms) is a very challenging task. As the number of processors used for simulation increases, work load per processor decreases and performance is affected by communication latency.

1.4 Importance of Parallel Computing to Simulations

Despite the seemingly unending progress in microprocessor performance, the urgent nature and computational needs of biomedical research demand that we persue the additional factors of tens, hundreds, or thousands in total performance which may be achieved by harnessing a multitude of processors for a single calculation. While the MD algorithm is blessed with a large ratio of calculation to data, its parallelization to large number of processors is not straightforward.

1.5 Increasing Availability of Large Parallel Resources

The Accelerated Strategic Computing Initiative, a U.S. Department of Energy program, has provided an unprecedented impetus for the application of massively parallel teraflop computing to the problems of the physical sciences and engineering. The National Science Foundation has followed this lead, funding terascale facilities at the national centers with the intent of enabling research that can employ many or all of the processors on these new machines. The concept of grid computing promises to make this computational power readily available from any desktop. Huge computing resources (with tens of thousands of processors) will soon be available to the biomedical researcher who will be able to harness it using software such as LeanMD.

1.6 Other Available Simulation Programs

The biomolecular modeling community sustains a variety of software packages with overlapping core functionality but varying strengths and motivations. Examples of these are AMBER, CHARMM, GROMACS, NWChem, TINKER and NAMD.

AMBER [14] and CHARMM [1] are often considered the standard "community codes" of structural biology, having been developed over many years by a wide variety of researchers. Both AMBER and CHARMM support their own force field development effort, although the form of the energy functions themselves is quite similar. Both codes are implemented using FORTRAN 77, although AMBER takes the form of a large package of specialized programs while CHARMM is a single binary. The parallelization of these codes is limited and not uniform across features, e.g., the GIBBS module of AMBER is limited to costly shared memory machines. Neither program is freely available, although the academic versions are highly discounted in comparision to commercial licenses.

GROMACS 3.0 [8], claims the title of “fastest MD”. This can be attributed largely to the GROMOS force field, which neglects most hydrogen atoms and eliminates Van der Waals interactions for those that remain. In contrast, AMBER and CHARMM force fields represent all atoms and new development has centered on increasing accuracy via additional terms. Additional performance on Intel x86 processors comes from the implementation of inner loops in assembly code. GROMACS is implemented in C as a large package of programs and is released under the GNU General Public License (GPL). Distribution takes the form of source code and Linux binaries.

NWChem [3] is a comprehensive molecular simulation system developed by a large group of researchers at the PNNL EMSL, primarily to meet internal requirements. The code centers on quantum mechanical methods but includes an MD component. Implemented in C and FORTRAN, NWChem is parallelized using MPI and a Global Arrays library which automatically redistributes data on distributed memory machines. Parallel scaling is respectable given sufficient workload, although published benchmarks tend to use abnormally large cutoffs rather than 12 Å(or PME) typically used in biomolecular simulations. Access to NWChem source code is available with the submission of a signed license agreement, although support is explicitly unavailable outside of PNNL.

TINKER [11] is small FORTRAN code developed primarily for the testing of new methods. It incorporates a variety of force fields, in addition to its own, and includes many experimental methods. The code is freely available, but is not parallelized, and is therefore inappropriate for traditional large-scale biomolecular simulations. It does, however, provide the community with a simple code for experiments in method development.

NAMD [10] builds upon other available programs in the field by incorporating popular force fields and reading file formats from other codes. NAMD complements existing capabilities by providing a higher performance alternative for simulations on the full range of available parallel platforms. NAMD is a state-of-art molecular parallel molecular dynamics application that is written in Charm++ and has been proved to be able to scale to 3000 processors. However, NAMD is not ready for next generation parallel machines with tens of thousands of processors due to limited parallelization exploited in the application.

1.7 Motivation for LeanMD Development

LeanMD is being developed as a framework for parallel molecular dynamics simulation. In LeanMD, science routines can be easily modified/replaced to test different motivations. This framework allows the biomedical researchers to reuse the core parallelization technique, thus saving them a lot of time and energy.

Scaling to the massively parallel machines (PetaFLOP class supercomputers) of the future with tens or hundreds of thousands of processor was a requirement for the development of LeanMD. LeanMD is implemented using Charm++, which offers the benefit of virtualization, and enables us to demonstrate better scalability. The performance analysis/visualization tool "projections" (associated with Charm++) is used for analysis/optimization of LeanMD.

1.8 LeanMD Development History

LeanMD development started three years ago. Initial design and implementation of LeanMD was done by Gengbin Zheng and Joshua Mostkoff Unger.

When I started working on LeanMD, it had implementations for short range pair calculation and intra-molecular force calculation. It had a parallel data reader that could read 'pdb' (protein data bank) files. LeanMD was able to use the automatic load balancing support built in the Charm++ system.

1.9 My Contribution to LeanMD Project

My main contribution to the LeanMD project is completion of missing components in existing LeanMD structure and getting it to work correctly with PINY [13] science routines.

I separated data-reader from LeanMD and converted it into a library. I also worked towards making the interface with data-reader cleaner. I worked on optimizing LeanMD to improve performance for a relatively small benchmark (30652 atoms) for simulations up to 1024 processors.

LeanMD was integrated with PINY science routines and data-reader. This helped a lot in completing the parallel structure of LeanMD that is needed for MD simulations. It also tested the capability of LeanMD to act as a framework for MD simulations and provided us the opportunity to enhance the LeanMD interface. PINY was developed by application researchers Dr. Glenn J. Martyna[1] and Prof. Mark E. Tuckerman[2].

This thesis presents my work along with description of LeanMD structure.

1.10 Thesis Organization

In chapter 2, we describe the basic steps and data-structures used by the sequential algorithm, and expose the data-dependencies. We also describe our approach to parallelizing this computation, based on the idea of processor virtualization. Chapter 3 describes how LeanMD can be used as a framework for molecular dynamics simulation. Chapter 4 describes optimizations and performance results obtained by running LeanMD with PINY science to simulate Human Carbonic Anhydrase (with 30652 atoms) system on Pittsburg Lemieux[3] super-computer. Chapter 5 presents some concluding remarks and future research.

[1] Physical Science Division, IBM TJ Watson Research Center, Yorktown Hghts, NY

[2] Associate Professor, Department of Chemistry and Courant Institute of Mathematical Science, New York University

[3] more information on Lemieux can be found at http://www.psc.edu/machines/tcs/lemieux.html

Chapter 2

The Computation

2.1 Molecular Dynamics Simulation

The MD computation involves calculating forces on all atoms during each time-step, and "integration" – using these forces to update the positions and velocities of atoms. Integration requires a small fraction of the total computation time. The force computation can be broadly divided into two categories: bonded force computations and non-bonded force computations. Force fields (equations) in MD simulation (CHARMM and AMBER) are of the form:

$$\Phi_{total} = \Phi_{inter} + \Phi_{intra} \tag{2.1}$$

$$\Phi_{inter} = \Phi_{Coulomb} + \Phi_{vdw,rep} \tag{2.2}$$

$$\Phi_{intra} = \Phi_{bonds} + \Phi_{bends} + \Phi_{Urey-Bradley} + \Phi_{torsions} + \Phi_{1,4\ pairs} \tag{2.3}$$

2.1.1 Intra–molecular Interactions

There are several categories of bonded (Φ_{intra}) forces which involve between 2 and 4 nearby (bonded) atoms. Intramolecular interactions are defined between atoms connected by three or fewer bonds.

In molecular dynamics, a bond is defined as a link between a pair of atoms, a bend[1] is defined as a group of three atoms having two bonds which share a common atom and a torsion[2] is defined as a set of four atoms linked in a chain by three bonds. The bond, bend and torsion forces are represented by Φ_{bonds}, Φ_{bends} and $\Phi_{torsions}$ in equation 2.3 respectively.

A bend (1–2–3) interaction combined with a bond (1–3) interaction is termed Urey-Bradley interaction. $\Phi_{Urey-Bradley}$ in equation 2.3 represents the Urey-Bradley interaction force.

1-4 pair interaction is a modified Lennard-Jones interaction between atoms separated by three bonds. It is calculated as an intra-molecular interaction. $\Phi_{1,4\ pair}$ in equation 2.3 represents this interaction.

2.1.2 Inter-molecular Interactions

Intermolecular interaction (Φ_{inter}) calculation involves electrostatic (Coulomb) and Vander Waals and short-range repulsion (Lennard Jones) forces. Lennard Jones and Vander Waals interactions are short range in nature as they decay faster than $1/r^3$. These interactions are thus calculated using a spherical cutoff (R_C). However, Coulombic interactions are proportional to $1/r$ and are long range, and can not be accurately treated using a spherical cutoff (R_C). Coulomb interaction can be divided (as in equation 2.4) into a short range and a long range term as suggested by Ewald.

$$\Phi_{Coulomb} = \Phi_{Coulomb\ short\ range} + \Phi_{Coulomb\ long\ range} \tag{2.4}$$

The short-range term can be calculated using spherical cutoff without loss of accuracy and long-range term is calculated using Particle Mesh Ewald [2] (PME) method.

This inter-molecular interaction calculation needs to be modified or omitted for the pair of atoms that are involved in an intramolecular interaction. In most cases, such pairs of

[1] angles in NAMD are called bends in PINY
[2] dihedrals and impropers in NAMD are called torsions in PINY

atoms are excluded from inter-molecular force calculation.

2.1.3 Molecular Dynamics Simulation Algorithm

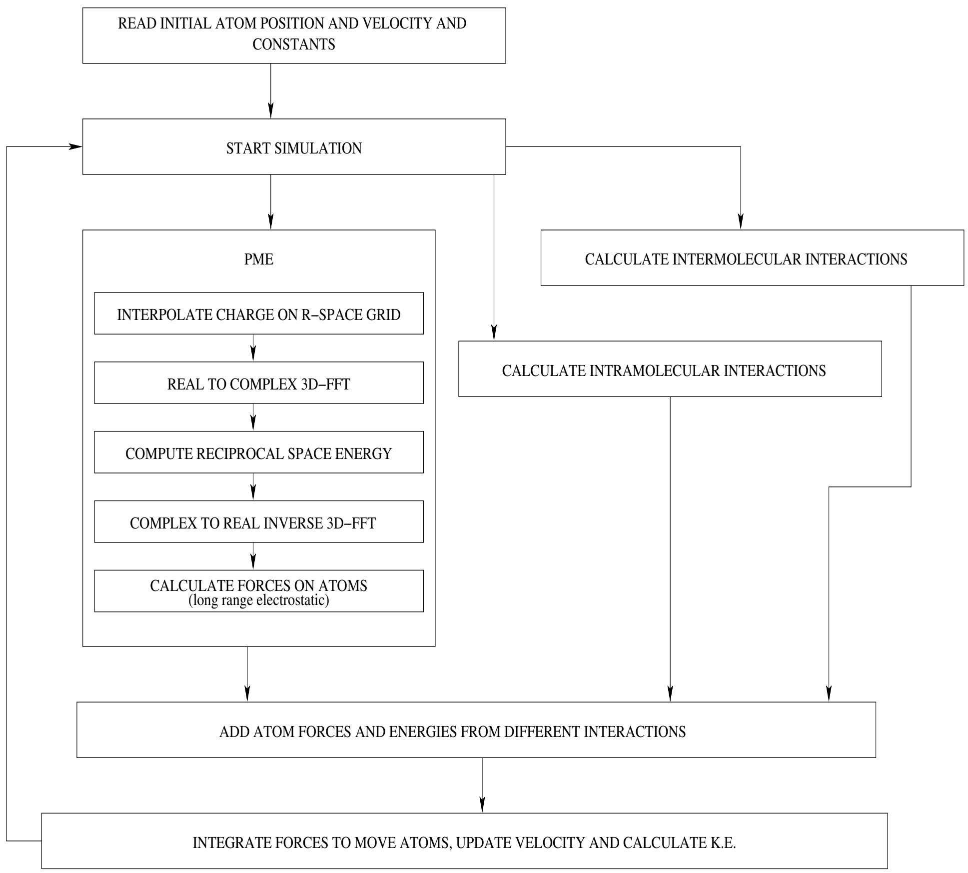

Figure 2.1: Schematic flowchart of a typical MD algorithm

The MD simulation algorithm is described broadly in the flowchart in figure 2.1. An overview of the data-structure and dependencies of the algorithm is shown in figure 2.2.

The basic object in a MD simulation is the simulation box, which is the bounding box for the atoms in the system being simulated. Simulation box uses the compute objects to calculate the interactions.

The intra-molecular compute object calculates all the intra-molecular forces on atoms. The inter-molecular compute object uses a pre-defined cutoff (R_C), to calculate the short-range inter-molecular interactions.

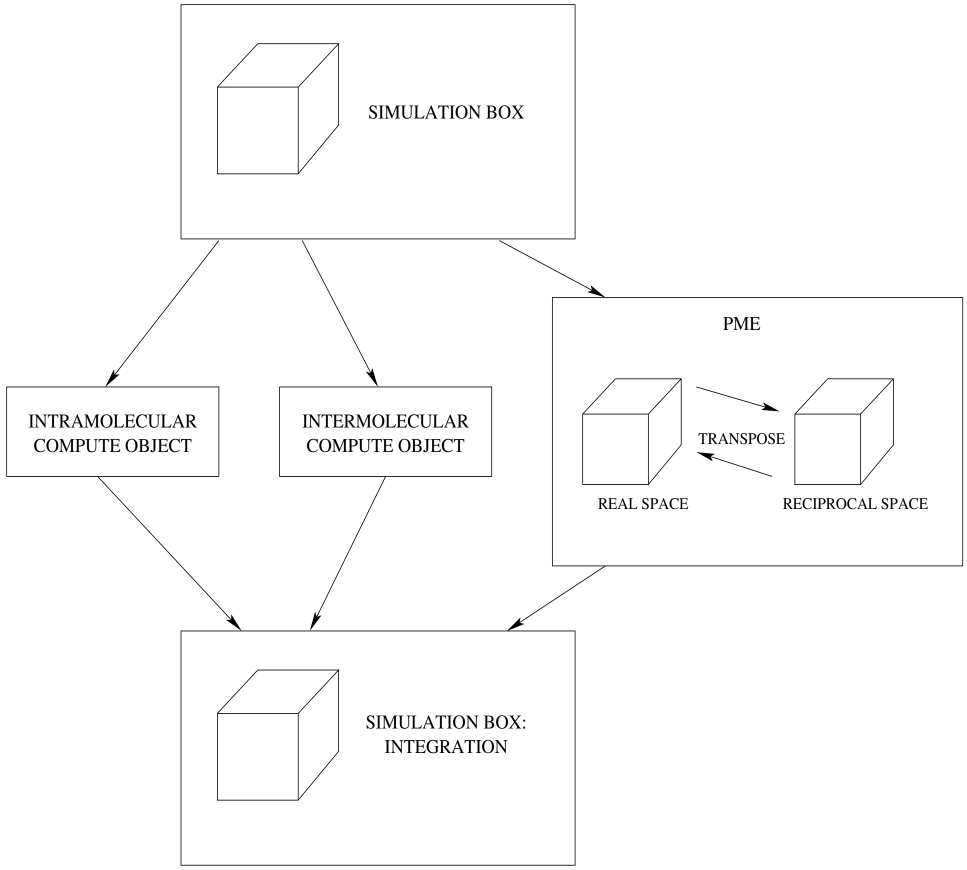

Figure 2.2: A serial view of a typical MD algorithm, showing data dependencies

PME method is used to calculate the long-range electrostatic energy. PME algorithm is briefly explained in figure 2.1. The first step in PME method is creation of a real-space PME grid. The real-space PME grid is obtained by interpolating charges on atoms in the simulation box. A 3D real-to-complex 3D FFT is done to get the reciprocal-space PME grid. The reciprocal-space PME grid is used to calculate the long-range electrostatic energy. After long-range energy calculation, a complex to real inverse 3D FFT is done to update the real-space PME grid. Once real-space inverse FFT completes, real-space PME grid is used to calculate the long-range electrostatic forces on atoms.

Forces and energies from the interaction objects are added to get the aggregate force on each atom. Simulation box integration module updates the atom position and velocity using the calculated force and the time quanta.

2.2 Parallelization

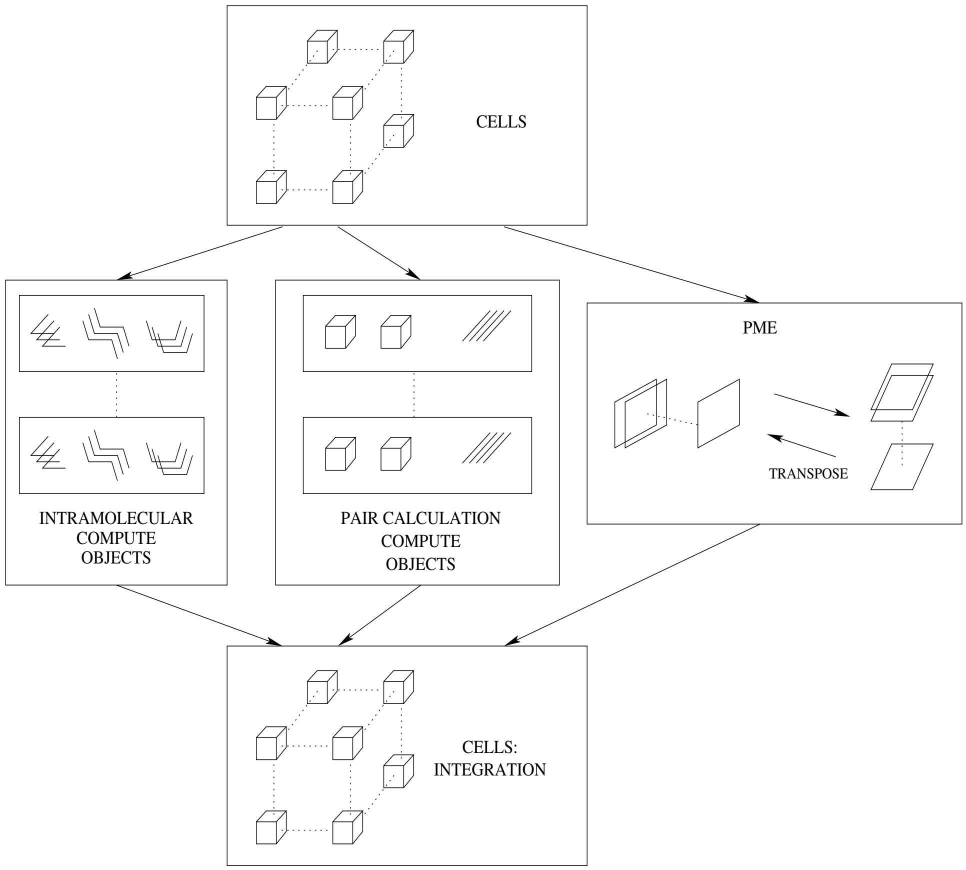

Figure 2.3: Parallel structure of our implementation

We parallelized this algorithm using the *processor virtualization* approach [5] supported by Charm++ [6], an advanced parallel programming system. In this approach, work is divided into a large number of objects or virtual processors (VPs), the computation is initially expressed in terms of these virtual processors, ignoring the issue of which physical processor they are mapped to. Then, either the runtime system or users can optimize their program by separately specifying or changing the mapping – either initially or even during the execution of program. In Charm++, the VPs are implemented as C++ objects [3] (called chares), which communicate with each other via asynchronous method invocation (also called mes-

[3]The Charm++ runtime system (RTS) supports multiple types of virtual processors. In *Adaptive MPI [4]* for example, each virtual processor is a user-level thread implementing an MPI "process". C++ objects are just one type of a virtual processor.

sages). Essentially, Charm++ allows programmers to separate the issue of decomposition and mapping.

2.2.1 Data Decomposition

The idea of assigning nearby atoms to the same processor is called spatial decomposition. There are three ways in which spatial decomposition can be done:

- Partition space into P boxes, one per processor,
- Partition space into fixed-size boxes, with dimension greater than or equal to the cutoff distance (R_C), requiring communication only between neighboring boxes ('one-away' interactions).
- Partition space into a large number of small boxes, requiring each box to communicate with a large number of boxes

A potential problem with the second decomposition strategy is that the number of processors one can utilize is limited to the number of boxes. The third decomposition strategy addresses the issue of fine-grained parallelism for cutoff interaction. LeanMD uses the third decomposition strategy. The 'one-away' approach of the second decomposition strategy is replaced with a 'k-away' approach [15] to divide the simulation space into large number of small boxes (called cells). With this strategy, 'k' neighboring cells represent the cutoff distance(R_C). To do the cutoff calculation, a cell must compute its interaction with every cell that is k-away or closer.

Defining formally, a 'cell' in a LeanMD simulation is a regular cubic region of simulation space. For a k-away simulation, a cubic region of simulation space formed by $k \times k \times k$ cells is called a 'patch'. Figure 2.4 pictorially shows the decomposition of simulation space. A cell is responsible for all the atoms within its boundary, their coordinates, and the forces exerted on them. Cells in LeanMD are represented by a 3-D array of chares.

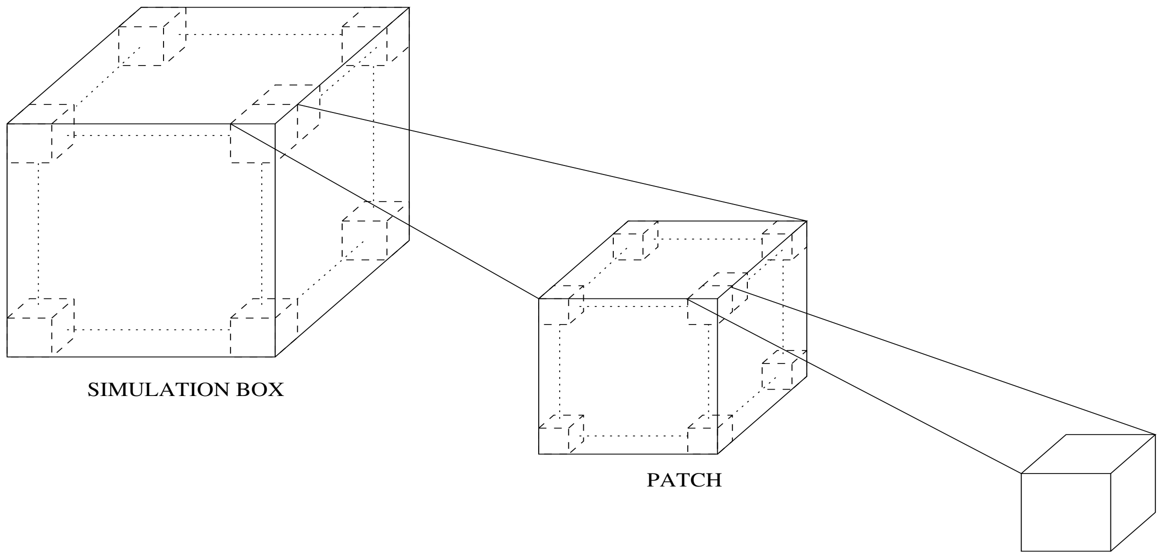

Figure 2.4: Data decomposition in LeanMD

Equation 2.6, shows the formula to calculate the dimension of an edge of a cell for a k-away simulation. While dividing the simulation space, 'patchdim' is chosen to be slightly larger than the cutoff radius (eqn 2.5). Thus one can make the assumption that a cell needs coordinates from at most k-away neighbors.

$$patchdim = R_C + margin \tag{2.5}$$

$$celldim = patchdim/k \tag{2.6}$$

To ensure that all the atoms in a cell indeed belong there, it is necessary to check all atoms at the end of every timestep, and migrate them to their proper home cells, if needed. This check for migration and migration itself can be prohibitively expensive. One can optimize this by choosing a 'margin' properly. Even if an atom strays outside the coordinate space of its patch, the assumption stays valid as long as the straying is less than the margin. Thus depending on the choice of 'margin', the number of time steps between migration can be varied.

2.2.2 Parallelization of Non-Bonded Force Computations

This section describes the parallelization of short-range pair interactions. The pair calculation compute block in figure 2.3 represents the 'cell-pair' objects. There is one cell-pair object for every pair of cells between whom cutoff (short-range pair interaction) calculation is required. There are some cell-pair objects which represent only one cell, these objects compute cutoff interactions between atoms present in the cell they represent. Cell-pair objects that receive atoms from two cells are responsible for calculating inter-cell cutoff interactions. Cell pair objects in LeanMD are represented by a very sparse 6-D array of chares, the first three and the last three components of a cell-pair object's index represents the index of cells that object interacts with.

Communication between cell-pair and cell object is straight-forward. Each cell maintains a neighbor list (list of cells-pair objects it interacts with). This neighbor list is precomputed before the simulation starts. A cell sends its atoms data to all the cell-pair objects in the neighbor list, i.e. all the cell-pair objects that a cell interacts with receive same data. For communication optimization, section-multicast (because each cell sends data to a section of cell-pair chare array) library is used in this case. This section-multicast[4] library is one of the many libraries built into Charm++ system. Atom data sent to cell-pair objects include atoms positions, ids (used to get the static information as mass, charge) and rootids (for the reason explained below).

When a cell-pair object receives the atoms from cell(s), it calculates the cutoff interactions and sends the forces and energy back to the cells. Cell-pair objects receiving atoms from two cells, send zero energy back to one of the two cell they interact with. This is done to avoid adding same energy twice.

Cell objects on receiving force messages from cell-pair objects, accumulates force on each atom. This communication between cell-pair and cell is optimized using the section-

[4]For details, see the array section multicast/reduction section in Charm++ manual available at http://charm.cs.uiuc.edu/manuals/html/charm++/manual.html.

reduction[5] support built in the section multicast library. Using section-reduction, the force messages from cell-pair objects are combined (accumulate is one of the combine operations supported) on the fly and the final force message is delivered to the designated cell object.

As an optimization, calculation of bond interactions (which involves a pair of atoms) is also divided among cell-pair objects. This optimization will not affect the correctness of the simulation if bond lengths are smaller than 'patchdim'.

If a pair of atoms in a cell-pair object are part of a bond, then the bond interaction can be calculated locally by that cell-pair object. Knowing whether a pair of atoms form a bond is not a trivial operation. To do this efficiently, each cell-pair object caches the bond information. This cache needs to be updated once after every atom-migration step, as between the atom-migration steps a cell-pair object always receives same atoms information from the cells it interacts with. A hash-table with bond-index (a unique bond identifier) as the key is used to cache the bond information. Using a hash-table makes the process of updating the cache more efficient. As an additional optimization, a per-atom bond list (list of bonds an atom is part of) is maintained on every processor to make cache updation efficient. The algorithm used to populate/update the cache with bonds is described in figure 2.5.

An additional check is needed in the algorithm described in figure 2.5 to avoid bond interactions that are calculated at cell-pair objects that receive atoms from one cell, from being added to the cache of cell-pairs that receive atoms from two cells. This check is omitted in the algorithm mentioned for presentation purposes.

Once the cache is populated, to calculate the bond-interaction, a simple iteration over the objects in the cache is needed to know the bonds for which force calculation can be done. The size of the cache is equal to the number of bonds that cell-pair object can calculate interaction for. Figure 2.6 describes the calculation of bond interaction at a cell-pair object.

Before this optimization was implemented, intramolecular force computation objects were

[5] Section-reduction reduces the communication cost as it is built on the multicast principles.

```
// add all the bonds for which any atom is locally available
foreach (atom ‘a’)

  get the bond list ‘l’.

  foreach (bond ‘b’ in ‘l’)

    if (‘b’ is not in the hash table) {

      create a bondinfo object with ‘a’.
      put the bondinfo object in hash table.

    } else {

      add ‘a’ to the bondinfo object in the hash table.

    }

// delete the bonds from the hash table for which both atoms are not available
foreach (bondinfo object ‘b’ in the hash table)

  if (‘b’ has one atom) {

    delete ‘b’ from hash table.

  }
```

Figure 2.5: Algorithm to cache bond info in cell-pair object

```
// loop over all the objects in the hash table and calculate bond interaction
foreach (bond in hash table) {

    calculate bond interaction.

}
```

Figure 2.6: Algorithm to calculate bond interactions in cell-pair object

responsible for bond force computation. Thus the atoms needed to compute bond forces were sent to the responsible intramolecular objects. Now as the bond forces are calculated at cell-pair objects there is a reduction in communication cost.

2.2.3 Parallelization of Bonded Force Computations

The Intramolecular compute objects block in figure 2.3 represents the 1-D array of chares responsible for calculating intramolecular interaction. Each intramolecular interaction computation object is responsible for computing a part of bends, torsions and 1-4 interactions[6] in the simulation. Each of these interactions (bends, torsion and 1-4 interactions) have a global unique intramolecular interaction identification number.

For communication efficiency and to reduce the memory required to store atoms at the intermolecular compute objects, a cell object sends the atoms to an intramolecular compute object that are needed by that compute object. To send the required atom to intramolecular compute objects, a cell caches the information that tells it which atoms should be sent to a intramolecular compute object. This information helps a cell to pack all the required atoms in one message per intramolecular compute object. This cached information needs to be updated once after every atom-migration step.

```
foreach (atom ‘a’ in the cell) {

  get the list of intramolecular interactions ‘a’ is part of.

  foreach (intramolecular interaction ’i’)

    add ‘a’ to the required atoms list for the intramolecular compute object
    responsible for calculating ‘i’.

}
```

Figure 2.7: Algorithm used by cell to know the atoms that need to be sent to intramolecular compute objects

The algorithm used to create the list of atoms to be sent to intramolecular compute objects is described in figure 2.7.

The algorithm used by a intramolecular compute object for calculating intramolecular interactions it is responsible for is described in figure 2.8.

[6] currently Urey-Bradley interaction calculation routine is not integrated with LeanMD

```
foreach (atom 'a' received from a cell)

  get the list of local intramolecular interactions 'a' is part of.

  foreach (intramolecular interaction 'i') {

    add 'a' to the list of available atoms.

    if (all atoms needed to compute 'i' are available) {

      compute 'i'.

      foreach (atom 'a1' involved in 'i') {

        increment the number of interactions calculated for 'a1'.

        if (number of interactions calculated for 'a1' is equal to the number of
            interactions 'a1' is involved in interactions to be calculated by
            this intra-molecular compute object) {

          increment the number of atoms for which all the interactions are
          calculated for the cell 'c' that sent 'a1'.

          if (number of atoms sent by 'c' is equal to the number of atoms for
              whom all the interactions have been computed) {

            send the forces and energy to 'c'.
            reset energy to zero.

          }
        }
      }
    }
  }
```

Figure 2.8: Algorithm used by intramolecular compute objects to calculate intramolecular interactions

When a cell object receives a force message from intra-molecular compute object, it adds the intra-molecular forces in the force message to the force object for the corresponding atom it houses.

2.2.4 Parallelization of PME Method

As explained earlier, the PME method involves creation of real-space and reciprocal-space grids. In LeanMD, both real-space and reciprocal-space grids are represented by separate 1-D chare arrays. Each element (chare) of the real-space slab array represents a slab (a set of y-z planes) of the real-space grid. Similarly, each element (chare) of the reciprocal-space slab array represents a set of x-y planes of the reciprocal-space grid. The real-space and the reciprocal-space grids are partitioned along different axises to allow the FFT library to do the transpose operation (required in a 3D FFT) efficiently. Communication and computation aspects of our parallel PME implementation are discussed next.

As the region of simulation space represented by each cell is fixed, the number of cells that need to communicate with a real-space slab is precomputed. This number is dependent on two variables, the thickness of the real-space slab and the interpolation order used. Interpolation order is one of the runtime input parameters to LeanMD.

Currently, each cell interpolates the charges on atoms within its boundaries to generate the portion of real-space grid that is affected by that cell. For PME calculation, each cell has a PMECellData object. The PMECellData class inherits from BasePMECellData and is defined specific to the science routine (currently available for PINY science) being used. PMECellData object of a cell holds the real-space charge grid for that cell. PMECellData object is mainly an abstraction to science specific data. BasePMECellData is an abstract class that defines the API's used by Cell to generate the real-space grid and calculate forces from real-space grid. Once the real-space grid is generated, Cell sends the sections of the grid to the real-space slabs responsible for it.

Charm++ provides a library for parallel 3-D FFT [7] which supports both plane based (currently used) and pencil based decomposition strategies. This library is used to do a real-to-complex 3D FFT from real-space slab to reciprocal-space slabs and complex-to-real

[7] For more details, see Charm++ libraries manual available at http://charm.cs.uiuc.edu/manuals/html/libraries/manual.html

inverse 3D FFT from reciprocal-space to real-space. Before start of simulation, FFT library is notified about source and destination chare arrays for real-to-complex 3D FFT and complex-to-real inverse 3D FFT.

When a real-space slab receives a grid message from a cell object, it adds the charges (for its section of real-space grid) in the grid message to corresponding points in its slab. Once a real-space slab receives all the expected grid messages, it calls the routine in 3-D FFT library to compute real-to-complex 3D FFT.

In the process of calculating FFT, FFT library uses the memory allocated by chares in destination chare array to store intermediate and final result of FFT computation. When real-to-complex 3D FFT completes, reciprocal-space slabs have their slabs initialized. Once reciprocal-space slab is notified that FFT is complete, it computes the long-range electrostatic energy for the part of reciprocal-space grid it represents. To compute the long-range electrostatic energy, each reciprocal-space slab uses its PMEGSpaceData object. The PMEGSpaceData class inherits from BasePMEGSpaceData and has properties similar to the PMECellData class.

After energy computation, FFT library is invoked by reciprocal-space slab to do the complex-to-real inverse 3D FFT. When inverse FFT finishes, real-space slab sends the relevant data to cells.

When a cell object receives a grid message from a real-space slab, it adds the charges in the grid message to corresponding points in its local grid. After receiving all the grid messages, a cell object uses PMECellData object to compute long-range electrostatic force on the atoms within its boundaries.

As long-range interactions change very slowly, this costly PME computation can be done less frequently (once every few steps). However, frequency of PME computation will affect the overall accuracy of the simulation.

Thickness of real-space and reciprocal-space slabs (number of planes represented by an object) limits the number of processors this parallel PME implementation can use. Another interesting parallelization technique to explore will be pencil-based decomposition, where each chare will represent few lines (pencils) of the grid.

2.2.5 Integration

When all the force messages are received by a cell object, force integration is preformed to update the position and velocity of atoms within its boundaries. Main parameters for integration routine are the forces on atoms and the time quanta.

If there are rigid bonds in the system being simulated, integration routine has to do some additional computation which is explained below.

Constraints Calculation for Rigid Bonds

Some molecules have bond structures which are rigid in nature, i.e. the length of bonds in such bond structures do not change as their atoms move. In other words, movement of atoms involved in rigid bond structures is constrained.

To keep these rigid structures rigid, integration module uses “shake” and “rattle” routines. PINY shake and rattle routines (currently used by LeanMD) implement Ryckaert [12] constraints.

Typically, rigid bond structures involve hydrogen (called “branch”) atoms bonded to other atom (called “root”). To implement shake and rattle efficiently, it is desirable to keep branches along with their corresponding roots, i.e. the branch atoms of a rigid bond structure always stay on the cell where its corresponding root atom is mapped. To model this correctly, patchdim is increased as in equation 2.7, where max_branch_root_bond_length is the maximum rigid-bond length between any two pairs of atoms in the simulation system. Increasing patchdim avoids missing any pair calculation (short-range inter molecular forces) due to grouping of branches with their roots.

$$patchdim = R_C + margin + (2 * max\_branch\_root\_bond\_length) \tag{2.7}$$

Each cell caches information about rigid bond structures that are formed by atoms in that cell. This information is updated once every atom-migration step.

Currently, three bond structures namely bond, bend and tetrahedral are considered rigid. A rigid bond involves two atoms and one interaction, a rigid bend involves three atoms and three interactions and a rigid tetrahedral involves four atoms and six interactions.

2.2.6 Exclusions in Non-Bonded Force Computation

The non-bonded force field used in molecular dynamics is complicated by the bonding structure and variety of atoms present in the molecule. Non-bonded interactions are generally excluded (set to zero) for pairs of atoms that are directly bonded (1-2 pairs) or are bonded to a common atom (1-3 pair). In addition, these interactions are modified for pairs of atoms separated in the molecule by three bonds (1-4 pairs). [8]

To check whether a pair of atoms are to be excluded from short-range pair calculation, a per atoms exclusion list is maintained. Each atom's exclusion list tells the atoms with whom its pair calculation must be omitted. Atoms in any exclusion list have atom indices less than the atom's index to which that exclusion belongs. This simplification reduces the average size of exclusion lists by a factor of two and thus the number of comparisions required to find whether a pair of atoms are in exclusion list or not.

As an additional optimization, only the pairs which have distance between them less than "exclusion_distance" are checked for exclusion in the exclusion list. Equation 2.8 explains exclusion_distance calculation.

$$exclusion\_distance = max\_distance\_between\_excluded\_pairs + constant \tag{2.8}$$

In equation 2.8, max_distance_between_excluded_pairs is calculated before simulation starts (i.e. with initial atom position), and the constant in the equation is chosen accord-

[8]The 1-4 interaction calculated by the intramolecular compute objects is the modified non-bonded interaction for 1-4 pairs.

ing to units used such that exclusion_distance represents the maximum possible separation between pair of atoms that can be excluded.

This algorithm is currently used in LeanMD and was taken from PINY. As the test for exclusion only uses the exclusion list, the definition of pair of atoms to be excluded can be modified easily.

Chapter 3

LeanMD as Parallel Molecular Dynamics Simulation Framework

The biomolecular community maintains a variety of software packages with overlapping "core" functionality but with varying strengths and motivations. One of the motivations for developing LeanMD is to provide the "core" functionality required for parallel molecular dynamics simulation. Using LeanMD, application researchers will be able to focus more on their area of expertise. The block diagram in figure 3.1 shows LeanMD catering to different sets of science routines. In this chapter we discuss how LeanMD can be used to simulate different motivations.

3.1 Simulation Runtime Options

Different molecular dynamics simulation systems (Argon, Butane, Water, proteins, etc) have different properties and require calculation of different set of interactions to simulate their behavior. Moreover the force fields used in molecular dynamics simulations can vary. LeanMD provides run-time configuration options to turn on/off calculations for these interactions. Currently, LeanMD provides broad options like turning on/off intramolecular, PME, constraints and Van der Waals calculation. LeanMD can also be configured at runtime to simulate periodic or non-periodic boundary conditions.

Run-time configuration options of LeanMD include many parameters for performance op-

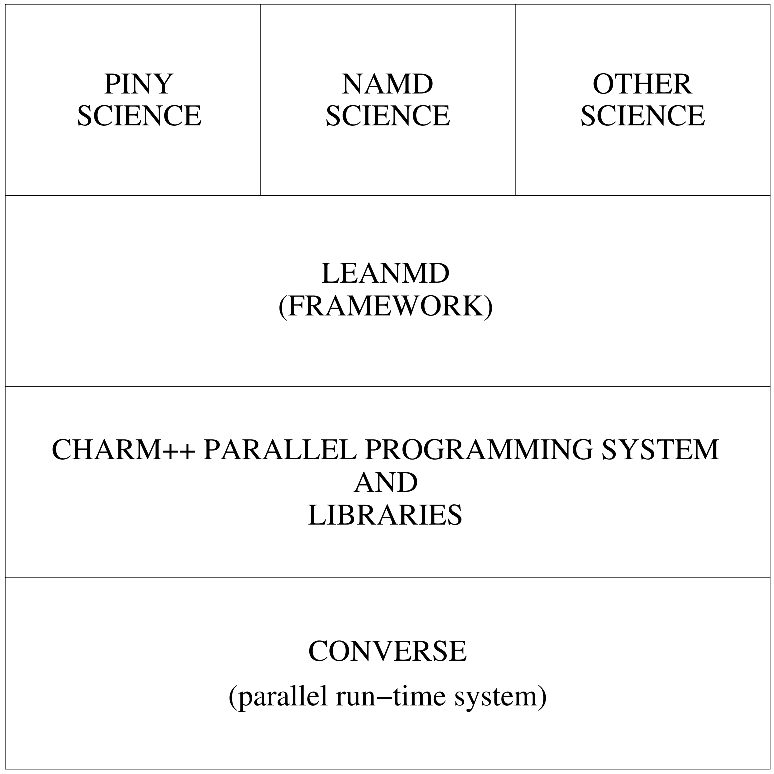

Figure 3.1: LeanMD as parallel molecular dynamics simulation framework

timization. PME calculation frequency, atom-migration frequency, load balancing frequency are some of these optimization parameters.

3.2 Using LeanMD for Simulating Different Motivations

As shown in figure 3.1, LeanMD can be used with different sets of science routines. To integrate a new set of science routines, the interface between LeanMD and science routines must be updated to call the new science routines with the input in the appropriate format. LeanMD can also be extended easily to implement new force fields as needed.

3.2.1 Modifying/Replacing Science Routines

All the science routines interact with LeanMD via an interface class named "Physics". Figure 3.2 shows the interaction between LeanMD and science routines. The Physics class provides interfaces for all the force calculation routines and integration routine. All the science routines are invoked by LeanMD and not vice versa. This relieves the application programmer of the burden of understanding how LeanMD exactly works.

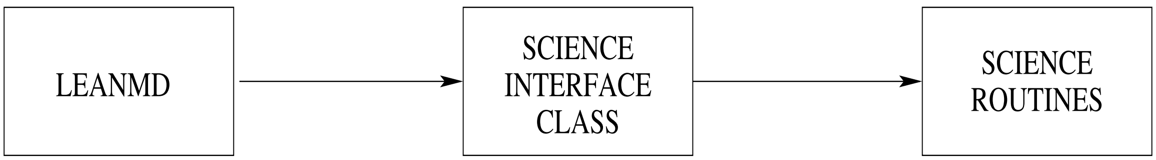

Figure 3.2: Interaction between LeanMD and science routines

An application programmer can integrate new science routines by modifying the API definitions in the Physics class to call new routines. In case of data-structure incompatibility, routines to convert from LeanMD data structures will be needed.

Modifying existing science routines is recommended where suitable to avoid data structure incompatibility issues. Also the Physics class needs no modification in this case.

3.2.2 Integrating Different Data Reader

There is a clean interface between data-reader and LeanMD. Figure 3.3 shows interaction between LeanMD and data reader. When the simulation starts, LeanMD creates the data-reader object and passes a call-back object to it. After the data reader object finishes reading input files and all the input data objects have been initialized, data-reader uses the call-back object to notify LeanMD. Input data objects are the objects where data read from input files are stored by the data reader. Task of "InputInterface" object in LeanMD is to initialize LeanMD objects. The InputInterface object uses the input converter object to transfer data to LeanMD defined data-structures. The input converter object gets data from the input data objects initialized by the data-reader, or reads directly from input files, and copies it to LeanMD data-structures.

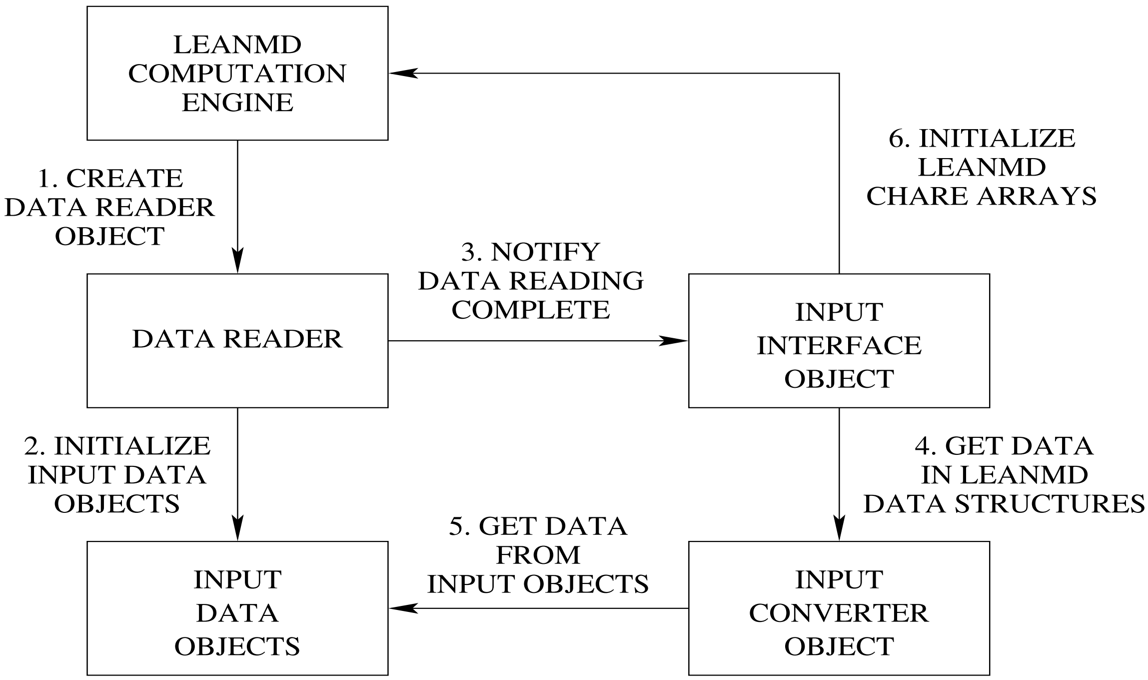

Figure 3.3: Interaction between LeanMD and data reader

To integrate a new data-reader, a new reader class with a constructor that takes a callback object as a parameter is needed. This reader object, once created should read the input files and initialize its input data objects. For a new data-reader, a converter class is also needed for the reasons explained above. This converter class must be derived from "BaseInputInterface" which is an abstract class. All the virtual functions declared in BaseInputInterface must be defined in the new converter class.

3.2.3 Adding New Force Fields and Interaction Types

In parallel programs it is generally very difficult to trace the control flow. Life cycle of each object in LeanMD is written using structured-dagger [9]. Using structured-dagger makes it much easier to understand and modify the control flow of entire parallel program. Adding a new force field to LeanMD requires some understanding of the working of LeanMD at the code level. In this section, we explain how a new force field object can be integrated into the life-cycle of a LeanMD simulation. [1]

Adding New Force Fields

Going over the implementation of PME will make it easier to understand, how a new force field can be added to LeanMD. As explained earlier, PME is implemented using two 1-D chare arrays representing PME real-space and reciprocal-space. These chare arrays are created and initialized in the main chare. In a molecular dynamics simulation, all the communication is between the cells (simulation box in case of sequential simulation) and the compute objects. Thus it is easy to see the life cycle of LeanMD in the control loop (written in structured-dagger) of cell class. To process the messages received from PME real-space slabs, cell's control loop was slightly modified to call the appropriate function for message processing.

Life cycle of each newly added object (example real-space slab and reciprocal-space slab) should also be written using structured-dagger.

Adding New Intramolecular Interaction Type

Adding a new intramolecular interaction type involves much simpler changes. All the intramolecular interaction objects in LeanMD are derived from an abstract class. Intramolecular compute objects in LeanMD do not distinguish between different intramolecular interaction types. This makes it easier to add or remove interactions types.

Once the new intramolecular interaction type is defined, only the InputInterface and intramolecular compute object need to be modified for initialization.

[1] Syntax and semantics of Charm++ and structured-dagger constructs are explained in their manuals available at http://charm.cs.uiuc.edu.

3.3 An Example: LeanMD with PINY Science and Data Reader

Currently, LeanMD is available with PINY science routines and data reader. This section describes the integration of PINY science routines and data reader with LeanMD.

Most of the simulation input data remains constant through out the simulation. PINY data reader reads the constant data into read-only[2] objects. A piny converter class derived from BaseInputInterface is used to populate LeanMD data-structures. Only atom position, velocity, mass, charge and other atom specific data along with bond, bend, torsions and 1-4 pairs information is copied to LeanMD data-structure. Read-only objects makes the rest of constant data available to science routines on all processors.

PINY science routines were slightly modified to use some simplistic LeanMD objects as input parameters. PINY science routines interact with LeanMD via the Physics interface class. For integration, Physics API definitions were modified to include and invoke proper PINY science routines.

[2] read-only objects in Charm++ are defined as objects which once initialized are available on all the processors for reading.

Chapter 4

Performance and Optimizations

4.1 Machine and Dataset used for LeanMD Performance Testing

We ran our code on PSC Lemieux, a 750 node, 3000 processor cluster. Each node in Lemieux is a Quad 1Ghz Alpha server connected by Quadrics Elan, a high speed interconnect with $4.5\mu s$ latency.

The Human Carbonic Anhydrase (HCA) dataset was used for LeanMD performance testing. This HCA system has 30652 atoms, 2135 bonds, 3833 angles, 11769 torsions and 10798 1-4 interactions.

4.2 Simulation Configuration

All the simulation results reported do not include PME calculation cost. Cutoff used for simulations is 10 Å. Timing results reported do not include the load balancing time. However, these timing results do include atom migration cost. Reported simulation time per step is the average time per step. All the speedup values reported are with respect to time taken by same simulation on one processor.

4.3 Solving Load Imbalance Problem

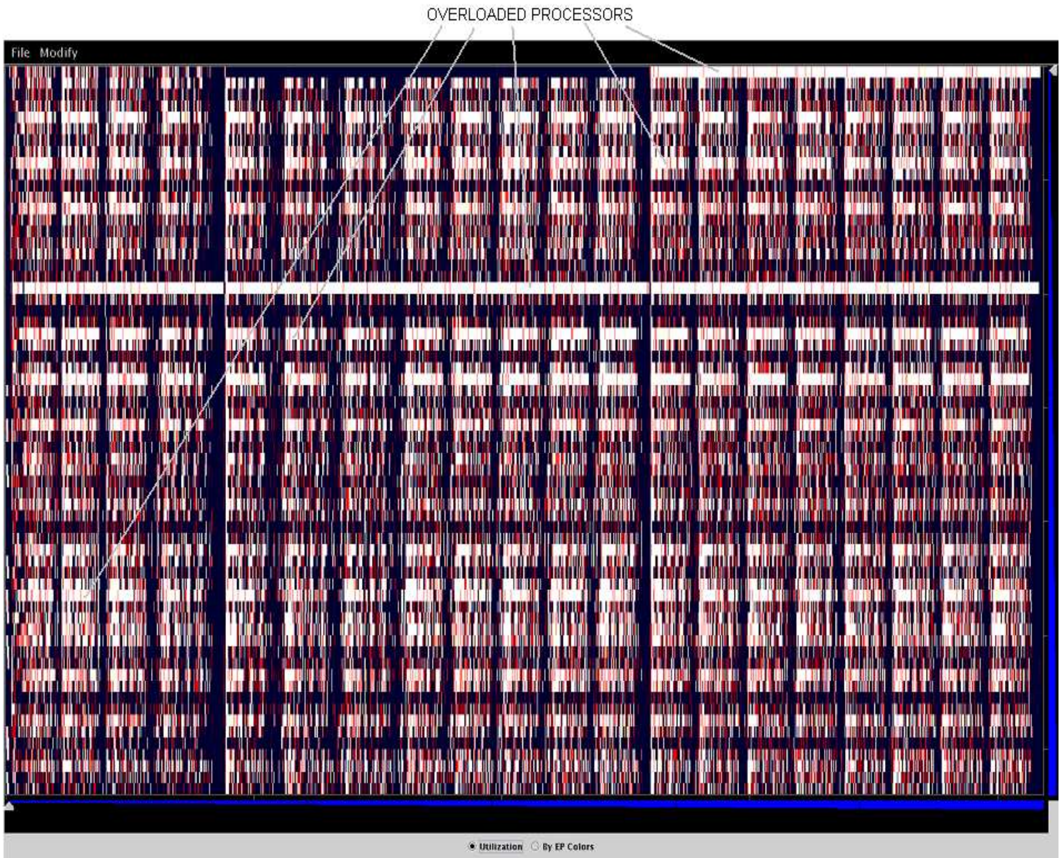

Figure 4.1: This graph shows the overview of LeanMD 1-away simulation on 64 processors. In this simulation, RefineLB strategy was used for load balancing.

Initial runs of LeanMD on Lemieux showed bad scalability beyond 32 processors. Table 4.1 and 4.2 shows the result of 1-away LeanMD simulation before load balancing. The load balancing strategy used in these runs was RefineLB. RefineLB is good when load is almost balanced, but it shows no improvement when there is severe load imbalance. Using projections [7] for the analysis of a 64 processor run showed that load imbalance was the cause of poor scalability beyond 32 processors. Figure 4.1 shows the overview (one of analysis views in projections) graph of a 64 processor run (without section multicast optimization) with RefineLB strategy used for load balancing. This graph shows the overview of load distribution on processors. In an overview graph, x-axis represents time and y-axis represents processors.

White color in overview graph represents 100% utilization and black color represents 0% utilization, intermediate colors represent utilization between 0% and 100%. In figure 4.1 some of overloaded processors are marked. It can be easily seen that most of the work is done by few processors. Load distribution in this figure is shown from the time simulation starts to the time simulation ends (omitting the initialization time).

Nodes	Processors	Average time per step(sec)	Speedup
1	1	10.804893	1.00
1	2	5.660082	1.91
2	4	2.933771	3.68
4	8	1.479690	7.30
8	16	0.846485	12.76
16	32	0.466989	23.14
32	64	0.432417	24.99
64	128	0.198827	54.34
128	256	0.098633	109.55

Table 4.1: Execution times for 1-away LeanMD simulation before load balancing. These simulations do not use the section multicast library. Time per step reported in this table is average over five simulation steps.

Nodes	Processors	Average time per step(sec)	Speedup
1	1	11.934467	1.00
1	2	5.500512	2.17
2	4	2.781817	4.29
4	8	1.550974	7.70
8	16	0.833586	14.32
16	32	0.435545	27.40
32	64	0.369723	32.28
64	128	0.196094	60.86
128	256	0.099805	119.58

Table 4.2: Execution times for 1-away LeanMD simulation before load balancing. These simulations use section multicast library. Time per step reported in this table is average over five simulation steps.

To identify a better load balancing strategy for LeanMD, 64 processor runs were made with different load balancing strategies available in Charm++. Table 4.3 summarizes the time taken per step after load balancing was done. This comparision showed that LeanMD performed better after load balancing with all of the Greedy strategies and also with Metis and RecBisectBf strategy.

LB Strategy	Average time per step(sec)
GreedyCommLB	0.187109
GreedyLB	0.183593
GreedyRefineLB	0.190820
MetisLB	0.183203
RandRefLB	0.249608
RecBisectBfLB	0.181249
RefineCommLB	0.328516
RefineLB	0.432417
RandCentLB	0.258789

Table 4.3: Execution times for 1-away LeanMD simulation on 64 processors comparing the Load Balancing strategies available in Charm++. These simulations do not use section multicast library. Time per step reported in this table is average over five simulation steps.

For next set of runs, Greedy load balancing strategy was chosen above other strategies because of low load balancing overhead shown by Greedy with respect to other strategies.

Results after load balancing (with Greedy strategy) are summarized in tables 4.4 and 4.5. Figure 4.2 shows the overview graph for a 64 processor 1-away (without section multicast optimization) simulation. This figure shows that after load balancing, computation load was almost equally distributed between processors. Figure 4.3, is another graph showing the average processor utilization.

In figure 4.2, it can be noted that after the first load balancing step, processor 0 is completely idle. In this simulation, the load balancer was collecting statistics from the very begining of the program, i.e. including initialization (which happens mainly on processor 0). Due to this, the load balancer observed that there was too much load on processor 0 and moved compute objects from it. This problem can be fixed by switching the statistics collection on/off dynamically.

Once load imbalance problem was solved, LeanMD (table 4.4) showed good speedup till

Nodes	Processors	Average time per step(sec)	Speedup
1	1	11.85	1.00
1	2	6.36	1.86
2	4	3.52	3.37
4	8	1.49	7.95
8	16	0.78	15.19
16	32	0.38	31.18
32	64	0.19	62.37
64	128	0.10	118.5
128	256	0.05	237
256	512	0.03	395
512	1024	0.019	623.68

Table 4.4: Execution times for 1-away LeanMD simulation after load balancing (Greedy) without using section multicast library. Time per step reported in this table is average over fifteen simulation steps.

1024 processors. However, speedup achieved for simulations (on more than 64 processors) with section multicast communication optimization (table 4.5) was worse compared to runs without this communication optimization. Multicast optimization will show better performance when number of processors to which multicast message has to be delivered is small. The reason for this being the fact that number of hops to deliver the message increases as number of processors in multicast tree increases. Also the benefit of only one message being sent to a processor irrespective of the number of objects expecting the multicast message on that processor decreases with number of recipients per processor. However a better mapping of objects to processor (considering communication pattern along with computational load for mapping of objects) might show benefit with section multicast optimization.

4.4 A Communication Optimization

As explained in chapter 2, cell-pair objects that receive atoms from two cells, compute inter-cell short-range cutoff interactions. Such a cell-pair object will do no useful work if it receives zero atoms from one of the cell (because, at this instant this cell has no atoms) object it

Nodes	Processors	Average time per step(sec)	Speedup
1	1	12.12	1.00
1	2	5.56	2.18
2	4	2.78	4.36
4	8	1.44	8.42
8	16	0.77	15.74
16	32	0.38	31.89
32	64	0.20	60.6
64	128	0.11	110.18
128	256	0.06	202
256	512	0.045	269.33

Table 4.5: Execution times for 1-away LeanMD simulation after load balancing (Greedy) using section multicast library. Time per step reported in this table is average over fifteen simulation steps.

interacts with. An optimization was done for this scenario to reduce the number of messages that need to be sent and hence reduce the communication cost. In this optimization, a cell-pair object notifies the cell object that it is not doing any useful work with the atom data it is receiving. Knowing this, cell object updates its active neighbor list, and sends atom data to cell-pair objects in the active list only from the next step onwards. However, after migration step, cell objects need to send their atoms to all the cell-pair objects they are supposed to interact with and update their active list on receiving a response from the cell-pair objects. This optimization cannot be used when section multicast optimization is used, because the syntax of section multicast/reduction library. This is another reason why simulations without section multicast optimizations perform better.

This optimization will show increased performance boost for simulations with fine-grained spacial decomposition because the probability of a cell object having no atoms increases as its size decreases. This claim was confirmed with 2-away LeanMD simulations.

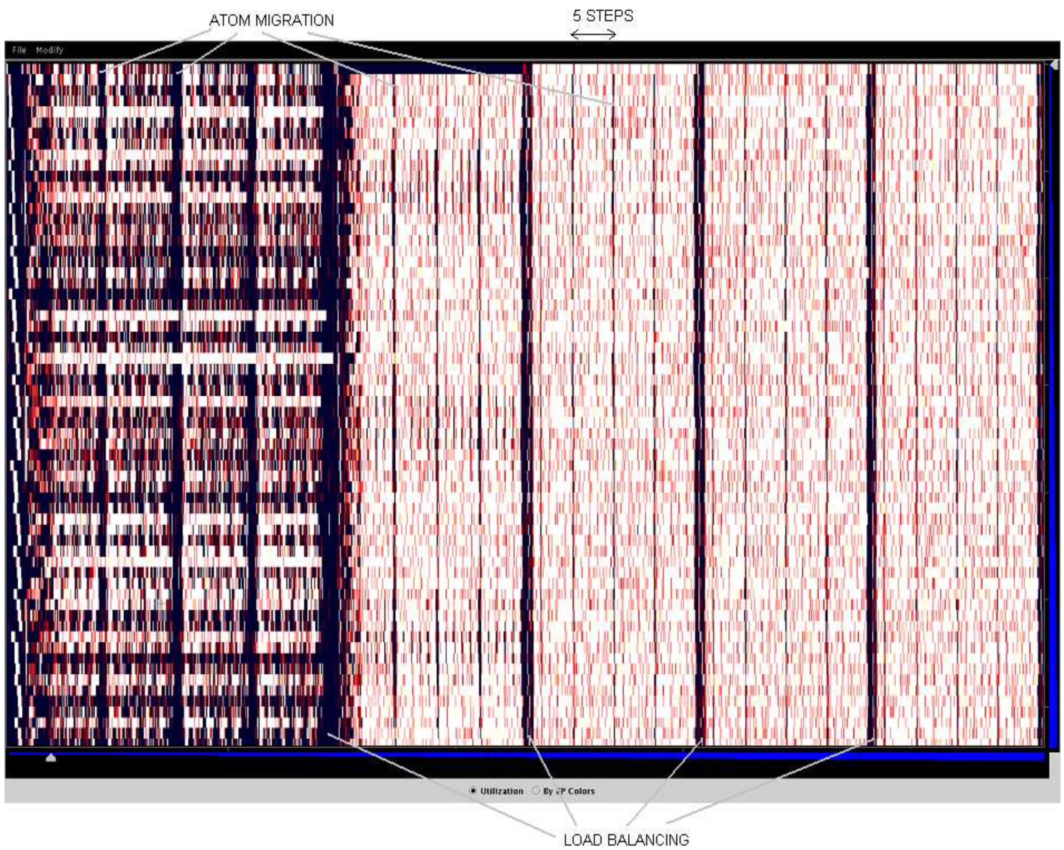

Figure 4.2: This graph shows the overview of a LeanMD 1-away simulation on 64 processors. In this simulation, GreedyLB strategy was used for load balancing.

4.5 Usage Profile of a LeanMD Simulation

Figure 4.4, is a usage profile graph of a 1-away LeanMD simulation on 512 processors. In a usage profile graph, x-axis represents processors and y-axis represents percentage usage. Different colored bars on a processor represent the functions invoked on that processor. The length of a colored bar tells the percentage time consumed by the routine on the processor over which that bar appears. Figure 4.4 is drawn for a period of time between two load balancing steps in that simulation.

White band in this graph represents idle time on processors. The thick grey patch in this graph represents the inter-molecular force computation routine. In this simulation, average time consumed by inter-molecular force computation routine is 70% on any processor.

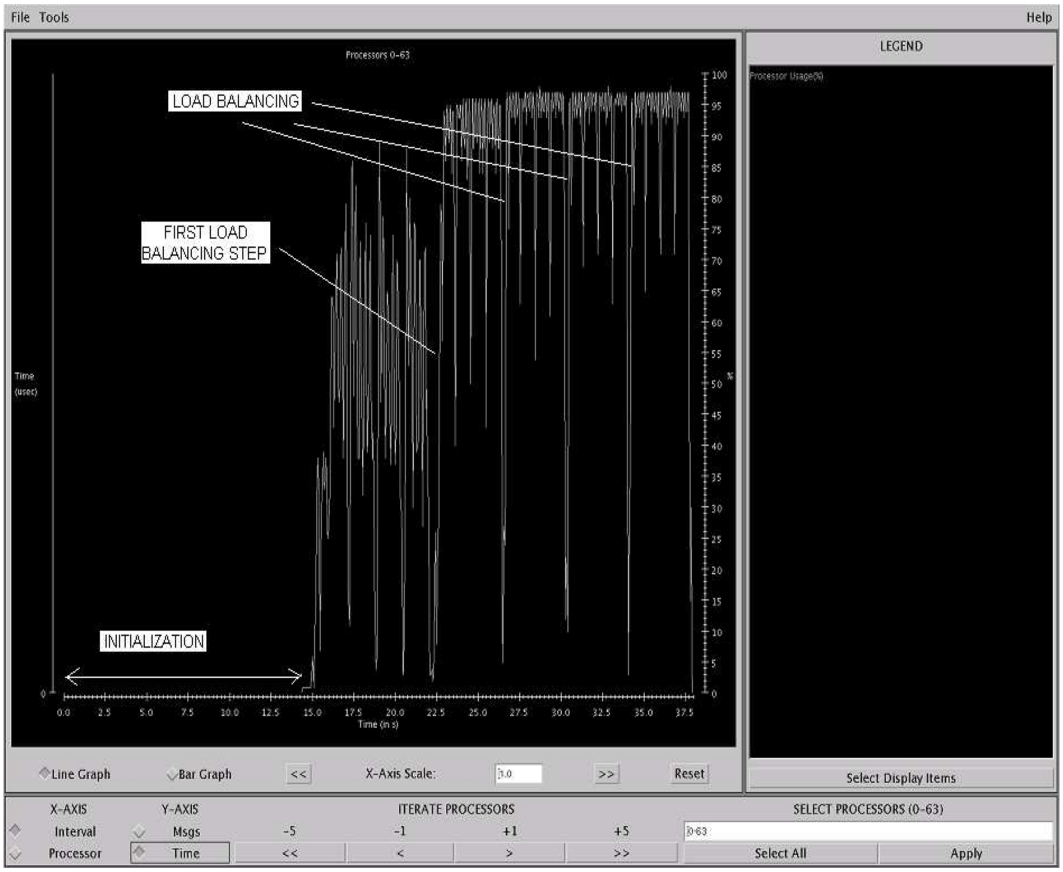

Figure 4.3: This graph shows the average processor utilization of a LeanMD 1-away simulation on 64 processors over time. In this simulation, GreedyLB strategy was used for load balancing.

Currently, the PINY inter-molecular force calculation routine computes each interaction based on the Lennard Jones and electrostatic force equations. PINY also provides a science routine which uses a table lookup to approximate inter-molecular forces on a pair of atoms based on the distance between them. Integration of this routine in to LeanMD should significantly reduce the time spent in inter-molecular force computation.

Reduction in computation will expose communication inefficiencies (as there is enough computation currently to completely hide the communication latency) and bring up new challenges in scaling LeanMD.

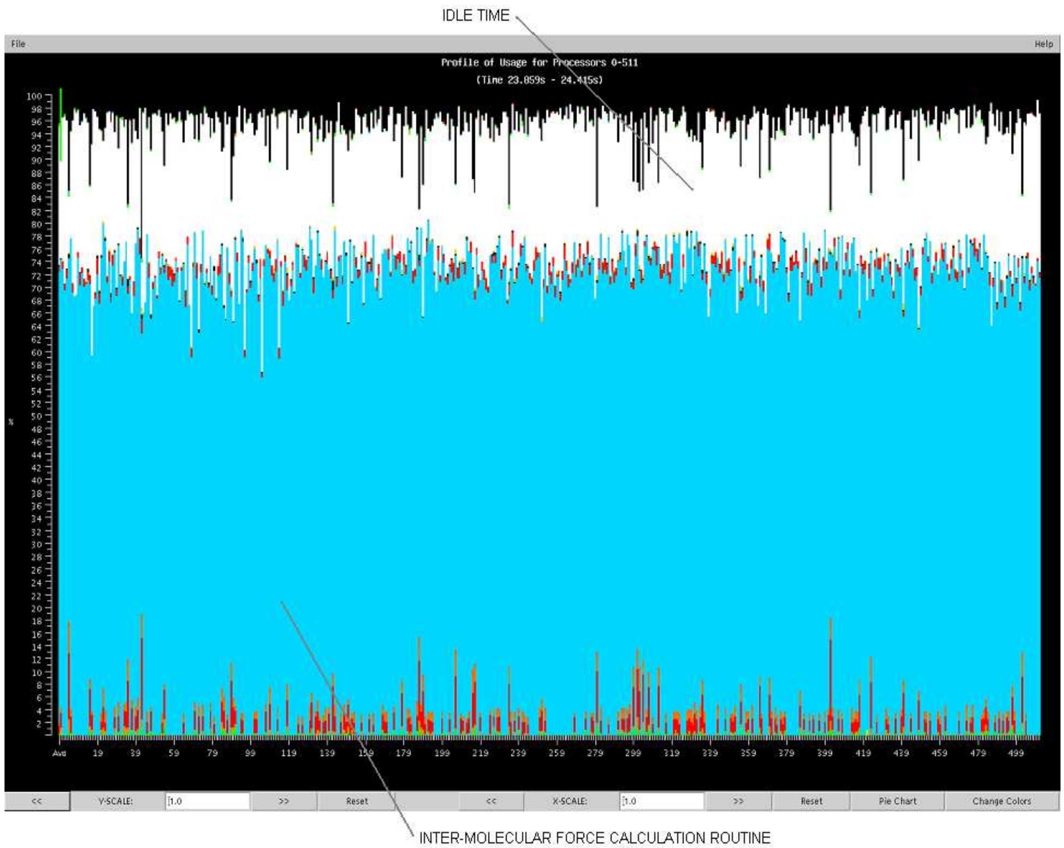

Figure 4.4: A 512 processor 1-away LeanMD simulation showing 70% time consumption by inter-molecular force computation routine.

Chapter 5

Conclusion and Future Work

In this thesis, we described the parallelization of the molecular dynamics algorithm using Charm++. We also described how LeanMD code can be reused with different science routines. This parallel implementation with PINY science routines scales well till 1024 processors for a relatively small benchmark system with 30652 atoms. Virtualization and adaptive overlap of communication, automatically engendered by Charm++, are clearly helpful for this application.

There is lot of room for both sequential and parallel optimizations in LeanMD. A detailed study and optimizations for 2-away (or even finer spatial decomposition) simulations should show better scalability of LeanMD for simulations with 1024 or more processors. LeanMD 2-away simulations showed that there are a lot of tiny messages, communication optimization is one important work to be done in future as it is necessary to achieve good performance results in this case. Charm++ provides a library with many different strategies for communication optimizations. This library can be used to reduce the communication cost in LeanMD simulations.

In current LeanMD simulations, PME calculation was turned off because currently used PINY PME routines cannot be used to calculate PME on more than one processor. Integrating science routines for parallel PME computation will show more avenues for performance optimizations. Integrating science routines from NAMD or other molecular dynamics simulation software will bring up the hidden challenges involved in reusing LeanMD. It will also help in making LeanMD interfaces better for easy reuse of LeanMD parallel structure.

Overall, current LeanMD scalability results are good. LeanMD's clear interface with data-reader and science routines should allow application researchers to easily reuse the parallelization techniques used in LeanMD for MD simulations.

References

[1] Bernard R. Brooks, Robert E. Bruccoleri, Barry D. Olafson, David J. States, S. Swaminathan, and Martin Karplus. CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4(2):187–217, 1983.

[2] Ulrich Essmann, Lalith Perera, Max L. Berkowitz, Tom Darden, Hsing Lee, and Lee G. Pedersen. A smooth particle mesh Ewald method. *Journal of Chemical Physics*, 103(19):8577–8593, 1995.

[3] High Performance Computational Chemistry Group. NWChem, a computational chemistry package for parallel computers, version 4.0.1. http://www.emsl.pnl.gov:2080/docs/nwchem.

[4] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003.

[5] Laxmikant V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[6] Laxmikant V. Kalé and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[7] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS)*, Melbourne, Australia, June 2003.

[8] E. Lindahl, B. Hess, and D. van der Spoel. GROMACS 3.0: a package for molecular simulation and trajectory analysis. *J. Mol. Mod.*, 2001. http://link.springer.de/link/service/jornals/00894/contents/01/00045.

[9] Theckla Louchios. Structured Dagger constructs and their implementation. Master's thesis, Dept. of Computer Science, University of Illinois, 2003.

[10] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[11] J. W. Ponder and F. M. Richards. An efficient Newton-like method for molecular mechanics energy minimization of large molecules. *Journal of Computational Chemistry*, 8:1016–1024, 1987.

[12] J.P. Ryckaert, G. Ciccotti, and Berendsen. Numerical integration of the cartesian equations of motion of a system with constrains: molecular dynamics of n-alkanes. *J. Comp. Phys.*, 23:327–341, 1977.

[13] M. E. Tuckerman, D. A. Yarne, S. O. Samuelson, A. L. Hughes, and G. J. Martyna. Exploiting multiple levels of parallelism in Molecular Dynamics based calculations via modern techniques and software paradigms. *Comp. Phys. Comm.*, 128:333–376, 2000.

[14] P. K. Weiner and P. A. Kollman. AMBER: Assisted model building with energy refinement. A general program for modeling molecules and their interactions. *Journal of Computational Chemistry*, 2(3):287–303, 1981.

[15] Gengbin Zheng, Arun Kumar Singla, Joshua Mostkoff Unger, and Laxmikant V. Kalé. A parallel-object programming model for petaflops machines and blue gene/cyclops. In *NSF Next Generation Systems Program Workshop, 16th International Parallel and Distributed Processing Symposium(IPDPS)*, Fort Lauderdale, FL, April 2002.