A SOFTWARE PIPELINE FOR ENSEMBLE MOLECULAR DYNAMICS

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by
Badi’ Abdul-Wahid

Jesús Izaguirre, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
April 2015
A SOFTWARE PIPELINE FOR ENSEMBLE MOLECULAR DYNAMICS

Abstract
by
Badi’ Abdul-Wahid

Proteins are the "machines of life" and understanding their motions is crucial to understanding the nature of various diseases, for instance Alzheimer's and Huntington's. Molecular Dynamics simulations are useful since they provide a atom-level resolution of a protein's motion. Of particular interest are measuring the rates at which different molecular conformations interchange. A significant bottleneck in the use of molecular dynamics is the difference in timescales. As a result, simulations require billions of steps to access slow motions such as protein folding. This work describes the development of a software pipeline to address this issue. The result is a set of software packages aimed at facilitating specific points is the study of proteins: exploratory simulations, conformational sampling, and rate calculations. Due to the large number of simulations that need to be run, each step supports the use of distributed systems and supports long-running applications and fault-tolerance. The problem addressed is that current approaches suffer from systematic bias and my contribution is an implementation of a method that does not suffer this bias, as well as a distributed computing pipeline.
To Mei and Zora Saurus Rex. Thank you.
CONTENTS

FIGURES ........................................................................................................ v

TABLES ........................................................................................................ viii

ACKNOWLEDGMENTS ................................................................................... ix

CHAPTER 1: INTRODUCTION ........................................................................ 1

CHAPTER 2: BACKGROUND ........................................................................... 3
  2.1 Chemistry of Biomolecules ..................................................................... 3
  2.2 Molecular Dynamics ............................................................................. 10
  2.3 High Performance Computing ................................................................. 14
  2.4 Ensemble Simulations ............................................................................ 17

CHAPTER 3: PRELIMINARY WORK ................................................................. 19
  3.1 The WW Domain .................................................................................. 19
  3.2 Molecular Dynamics ............................................................................. 23
  3.3 Folding@home ..................................................................................... 25
  3.4 Markov State Models ........................................................................... 26
  3.5 Accelerated Weighted Ensemble ............................................................. 30
  3.6 Comparison of MSM and AWE methods ............................................... 32
  3.7 Summary ............................................................................................. 33
  3.8 Folding@home data ............................................................................. 34
  3.9 HPC Simulations .................................................................................. 34
  3.10 Initial analyses .................................................................................... 35
  3.11 Summary ............................................................................................ 39

CHAPTER 4: PERFORMANCE OF NAMD AND GROMACS ....................... 41
  4.1 Introduction ........................................................................................ 41
  4.2 Materials and Methods ......................................................................... 46
  4.3 Results ................................................................................................ 52
  4.4 Discussion and Conclusion .................................................................. 65
<table>
<thead>
<tr>
<th>FIGURES</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Molecular Structure of Alanine</td>
<td>4</td>
</tr>
<tr>
<td>2.2 3D Structure of Alanine</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Protein Primary Structure</td>
<td>5</td>
</tr>
<tr>
<td>2.4 Protein Folding Schematic</td>
<td>6</td>
</tr>
<tr>
<td>2.5 Antiparallel Beta-sheets</td>
<td>7</td>
</tr>
<tr>
<td>2.6 Parallel Beta-sheets</td>
<td>7</td>
</tr>
<tr>
<td>2.7 Ubiquitin Protein Secondary Structure</td>
<td>7</td>
</tr>
<tr>
<td>2.8 Kinesin Protein</td>
<td>8</td>
</tr>
<tr>
<td>2.9 Quaternary Structure of a Proteosome</td>
<td>9</td>
</tr>
<tr>
<td>2.10 Chaperonin Protein</td>
<td>9</td>
</tr>
<tr>
<td>2.11 Illustration of Mycoplasma mycoides</td>
<td>11</td>
</tr>
<tr>
<td>2.12 Simplified Molecular Dynamics Algorithm</td>
<td>12</td>
</tr>
<tr>
<td>2.13 ASIC Chip for Anton</td>
<td>17</td>
</tr>
<tr>
<td>3.1 The WW Domain, Sequence, and Mutation Positions</td>
<td>21</td>
</tr>
<tr>
<td>3.2 Two-Well Free-Energy Surface</td>
<td>23</td>
</tr>
<tr>
<td>3.3 Building a Markov State Model</td>
<td>27</td>
</tr>
<tr>
<td>3.4 MSM Network Analysis</td>
<td>29</td>
</tr>
<tr>
<td>3.5 WE Method</td>
<td>30</td>
</tr>
<tr>
<td>3.6 Comparison of RMSD Distributions</td>
<td>35</td>
</tr>
<tr>
<td>3.7 Mean First Passage Time Correlation to Experiment</td>
<td>36</td>
</tr>
<tr>
<td>3.8 Free-Energy for Radius of Gyration and RMSD</td>
<td>37</td>
</tr>
<tr>
<td>3.9 MSM Implied Timescales</td>
<td>37</td>
</tr>
<tr>
<td>3.10 Network Models from MSMs</td>
<td>38</td>
</tr>
<tr>
<td>3.11 Folding &amp; Unfolding of MD Trajectory</td>
<td>39</td>
</tr>
<tr>
<td>4.1 Timescales of Molecular Dynamics</td>
<td>42</td>
</tr>
<tr>
<td>4.2 Scaling for Vacuum Systems</td>
<td>52</td>
</tr>
<tr>
<td>4.3 Scaling for Explicit Solvent</td>
<td>55</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>4.4</td>
<td>Scaling for Implicit Solvent</td>
</tr>
<tr>
<td>4.5</td>
<td>Single-sided Branching Speedup</td>
</tr>
<tr>
<td>4.6</td>
<td>Double-sided Branching Slowdown</td>
</tr>
<tr>
<td>4.7</td>
<td>GPU Write-back Slowdown</td>
</tr>
<tr>
<td>5.1</td>
<td>Folding@Work Data Model</td>
</tr>
<tr>
<td>5.2</td>
<td>Folding@Work Workflow</td>
</tr>
<tr>
<td>5.3</td>
<td>WorkQueue Architecture and Resource Usage</td>
</tr>
<tr>
<td>5.4</td>
<td>Folding@Work performance model</td>
</tr>
<tr>
<td>5.5</td>
<td>Folding@Work Weak Scaling</td>
</tr>
<tr>
<td>5.6</td>
<td>Folding@Work Workunit Wait Times</td>
</tr>
<tr>
<td>5.7</td>
<td>Folding@Work Transfer Times</td>
</tr>
<tr>
<td>5.8</td>
<td>Folding@Work Runtime Distribution</td>
</tr>
<tr>
<td>5.9</td>
<td>Folding@Work Timeline</td>
</tr>
<tr>
<td>5.10</td>
<td>Folding@Work Scaling</td>
</tr>
<tr>
<td>5.11</td>
<td>Folding@Work Straggler Wait Times</td>
</tr>
<tr>
<td>5.12</td>
<td>Folding@Work Tasks Waiting</td>
</tr>
<tr>
<td>6.1</td>
<td>Comparison of Uniform Random Sampling with Poisson Disc Sampling</td>
</tr>
<tr>
<td>6.2</td>
<td>Potential Workflow using Poisson Subsampling</td>
</tr>
<tr>
<td>6.3</td>
<td>Work Queue on Multiple Resources</td>
</tr>
<tr>
<td>6.4</td>
<td>Free Energy Surface</td>
</tr>
<tr>
<td>6.5</td>
<td>Coverage and Clustering with Poisson Subsampling</td>
</tr>
<tr>
<td>6.6</td>
<td>Decomposition of Circles Dataset</td>
</tr>
<tr>
<td>6.7</td>
<td>Incremental Online Poisson Subsampling</td>
</tr>
<tr>
<td>6.8</td>
<td>Workflow for Adaptive Sampling Molecular Dynamics</td>
</tr>
<tr>
<td>6.9</td>
<td>Adaptive Sampling Work Queue Monitoring</td>
</tr>
<tr>
<td>6.10</td>
<td>New States discovered by Adaptive Sampling</td>
</tr>
<tr>
<td>7.1</td>
<td>Model of Free-Energy Landscape Exploration</td>
</tr>
<tr>
<td>7.2</td>
<td>WE Algorithm</td>
</tr>
<tr>
<td>7.3</td>
<td>AWE-WQ Flowchart</td>
</tr>
<tr>
<td>7.4</td>
<td>AWE-WQ Workflow</td>
</tr>
<tr>
<td>7.5</td>
<td>Cell Weights Before and After AWE-WQ</td>
</tr>
<tr>
<td>7.6</td>
<td>Distribution of Execution Time</td>
</tr>
<tr>
<td>7.7</td>
<td>AWE-WQ Throughput with Task Replication</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>7.8  The WW Domain</td>
<td>143</td>
</tr>
<tr>
<td>7.9  Forward and Backwards Fluxes for WW from AWE</td>
<td>146</td>
</tr>
<tr>
<td>7.10 WW Major Pathways Computed from AWE-WQ</td>
<td>147</td>
</tr>
<tr>
<td>8.1  WorkQueue Architecture</td>
<td>150</td>
</tr>
<tr>
<td>8.2  Software Architecture for AWE-WQ</td>
<td>154</td>
</tr>
</tbody>
</table>
TABLES

3.1 WW DOMAIN MUTANTS ........................................ 20
3.2 FOLDING@HOME CLIENT STATISTICS ......................... 26
3.3 COMPARISON OF MSM AND WE METHODS ................. 33
4.1 ARCHITECTURES AND SOFTWARE BENCHMARKED .......... 47
4.2 ATOMS IN BENCHMARKED SYSTEMS ...................... 48
4.3 INTEGRATOR PERFORMANCE WITH GROMACS-OPENMM .. 59
4.4 CONSTRAINT PERFORMANCE WITH GROMACS-OPENMM .. 59
4.5 WRITE-BACK PERFORMANCE OF GROMACS-OPENMM ...... 60
5.1 FOLDING@WORK WEAK SCALING ......................... 84
5.2 FOLDING@WORK TURNAROUND TIME .................... 93
6.1 SUBSAMPLING DATASETS USING POISSON SUBSAMPLING .. 111
6.2 METRICS COMPARING CLUSTERING RESULTS OF SUBSAMPLED DATA ........................................ 112
7.1 FORWARD, BACKWARD RATES VIA AWE AND BRUTE-FORCE CALCULATIONS ........................................ 145
ACKNOWLEDGMENTS

Many thanks are due to my family for their boundless support and encouragement: Melissa, Zora, Sarah, Faried, Sami, Liz, Brently, Sarah. My advisor Jesús Izaguirre for guidance and advice. Collaborators Prof. Eric Darve, Prof. Martin Gruebele, Prof. Vijay Pande, Prof. Douglas Thain, Dr. Ronan Costaouec, Haoyun Feng, Dinesh Rajan, and Li Yu. My committee members Drs. Colllin McMillan, Dr. Christopher Sweet, and Dr. Douglas Thain for their time and feedback. Lab-mates and friends Kevin Kastner, Yong Hwan Kim, Ronald Nowling, James Sweet. Funding has been provided by NSF CCF-1018570, NIH 1R01 GM101935-01, NIH 7R01 AI039071. The Notre Dame Center for Research Computing, institute for Computational and Mathematical Engineering at Stanford, and the Folding@home donors for use of computing resources.
Chapter 1

Introduction

Proteins are complex molecules of fundamental importance in biological processes. Numerical simulation using Molecular dynamics (MD) has proven to be a powerful tool to predict many important properties. For instance, several diseases such as Alzheimer’s, Parkinson’s, Huntington’s or Sickle cell disease are known to result from protein misfolding. Additionally, the interaction between drugs and proteins is dependent in part on the protein’s conformation and dynamics. The previous decades has seen significant growth in the development of MD methods and their applications to the study of biomolecules. However, a significant limitation is that of time scales: many of the biologically interesting motions such as folding occur in the microsecond to millisecond timescales. MD simulations are typically limited to low microseconds or nanoseconds.

In order to study the dynamics of proteins and other interesting biomolecules, molecular dynamics is a powerful tool. Like many tools, however, a number of challenges present themselves when attempting to push the limits of the tool. I started studying the effect several mutations have on the dynamics of a protein, the WW domain. The initial attempts using the commonly accepted approaches failed to capture the differences in an acceptable way. Thus a more targeted set of methods is needed, address specific problems encountered. Since the ultimate goal is to capture populations and rates of change the individual steps to capture these properties was optimized.

The problem addressed is the following: studying molecular dynamics
requires applying enormous computing resources (on the order of thousands of aggregate compute years) to capture rare transitions between conformations. Current approaches to address the challenges that arise are susceptible to systematic bias that cannot be easily corrected. As a result, different methods have been proposed which do not suffer this bias.

My contribution is the implementation of such a method and a software pipeline to support the studying of protein kinetics.

In short, the problem is divided into sub-problems:

- conformational sampling
- thermodynamics calculations
- kinetic calculations

The solution uses a distributed computing approach to solve each problem. By dividing these problems, each solution can focus solely on its specific goal and be combined as part of a pipeline.

This document is organized as follows: Chapter 2 presents the background information: a brief introduction to molecular dynamics methods, a description of the context in which this work resides with molecular dynamics as well as the use of computing resources. Chapter 3 then describes the goal and initial attempts which initiated this work. Chapter 4 investigates the scalability of GROMACS and NAMD. Chapter 5 describes and initial tool developed which can be used as a preliminary step within the pipeline. Chapter 6 describes the development of the adaptive sampling method within a distributed computing context. Chapter 7 then describes the implementation of use of the Accelerated Weighed Ensemble method for computing kinetic rates. Chapter 8 discusses the software artifacts generated by this work. Finally, Chapter 9 summarizes the work, presents the current state, and describes possible future work.
CHAPTER 2

BACKGROUND

2.1 Chemistry of Biomolecules

The structure of a molecule can be thought of as a graph in three-dimensional space: each node represents an atom with an associated type, coordinate, and charge, and each connection represents a type of inter-atomic interaction. The types of atom are those from the periodic table, but in the study of biomolecules, these atoms are predominantly one of

- carbon
- nitrogen
- oxygen
- hydrogen

although atoms such as iron, sulfur, or sodium are sometimes present.

These few atoms are most of the building blocks for amino acids, which are small molecules whose name identifies the atomic composition and connectivity. For example, alanine consists of carbon, oxygen, nitrogen, and hydrogen and is typically shown as in Figure 2.1. In the figure, if an atom label is not present it is understood to be carbon. The chemical formula for alanine is $C_3H_7NO_2$ meaning: three carbons, seven hydrogens, one nitrogen, and two oxygens. The figure also provides some structural and connectivity information: the double bond from carbon to singular
Figure 2.1. Molecular structure of Alanine.

Figure 2.2. The three-dimensional structure of alanine. Carbon is black, hydrogen white, nitrogen blue, and oxygen red.

oxygen in the carboxylic acid (O=C=OH) forms a plane and allows the alpha carbon, which is connected to the methyl (H₃C) and amine (NH₂) groups, to rotate about the bond to the carboxylic carbon. A three-dimensional representation of alanine is shown in Figure 2.2.

Each atom has different properties of mass and charge. For instance, oxygen is heavier nitrogen, which is heavier than carbon. Hydrogen is the lightest of all. Additionally, there are several different types of atomic interactions: single, double, and triple bonds, ionic forces, electrostatics, van der Waals’. Single, double, and triple bonds when different categories of electrons are shared between two atoms. Electrostatics is the interaction between charges of atoms due in part to their protonation state. Ionic forces are a type of electrostatic interaction in which two atoms of op-
posing charge associate. For instance table salt is interaction between the positive sodium (Na\(^+\)) and negative chloride (Cl\(^-\)) ions. Additionally, biomolecules are solvated in water, which is a small, polar molecule (H\(_2\)O): the oxygen is electronegative and tends to draw the electrons from the hydrogens closer. This results in a partial negative charge around the oxygen and partial positive around the hydrogens.

In addition to alanine, there are 19 other amino acids which form the building blocks for proteins. Each amino acid has a slightly different chemical makeup: some are small and flexible, others large and bulky, some are charged while others are not. Since water is partially charged molecules which form ionic associations with water molecules are called hydrophillic. Other molecules, such as phenylalanine, which are non-polar will not interact with water and are called hydrophobic.

A protein is a sequence of these amino acids, as shown in Figure 2.3. An amino acid in this chain is often referred to as a residue. These residues form a ”string of beads” folds around itself and gets ”knotted” due to the chemical properties of neighboring residues. This process is commonly called the ”folding process”, and is illustrated in Figure 2.4.

The sequence of residues in a molecule is known as the primary structure and can be represented as a string of the codes for each residue. For instance: **Ala-Ser-His-Trp**
Figure 2.4. The conceptual schematic of protein folding. Black dots represent hydrophobic residues, while is hydrophillic, grey are neutral.

is a sequence of alanine, serine, histidine, and tryptophan using the three-letter codes. Often a single-letter code is used: \textbf{ASHW} is the same sequence. The primary structure has two endpoints: since amino acids bind together carboxylic acid to amine, the end with an unbound amine is the N-terminal and the unbound carboxylic acid is the C-terminal. As the primary structure folds additional, consistent and repeating structural patterns form. Common patterns are the beta-sheet and alpha helix. The beta-sheet structure is shown in Figures 2.5 and 2.6 and is formed by hydrogen bonds between the amines and oxygens of adjacent backbone residues. If the strands run such that N-terminal is next to N-terminal of adjacent strands then the sheet is called "parallel", and "antiparallel" if running in the opposite direction.

Another common element is the alpha-helix: the backbone of a strand looks as if wrapped around a pole. These structures, beta-sheet, alpha-helix, form the secondary structure. As the sequence of amino acid forms secondary structure, these elements combine to form a tertiary structure, as shown in Figure 2.7. As Figure 2.7 demonstrates, the secondary structure elements can be visualized using simplified
Proteins can interact with one another. Due to a combination of the chemistry of certain residues and the shape of the structure, these interactions have various biological functions. In some cases, multiple tertiary-structure elements will join to form the quaternary structure of a protein. These functions include binding to small molecules such as signal recognition (of neural transmitters or hormones), identify-
ing pathogens (antibodies), carrying oxygen (Hemoglobin), assisting other sequences to fold (chaperonins), degradation of other sequences, or motors (ATP Synthase, Kinesins and transport proteins).

For instance, Ubiquitin, shown in Figure 2.7, is a protein whose function is to mark other proteins for degradation by the proteosome (Figure 2.9). A chaperonin, modeled in Figure 2.10, is a large protein whose quaternary structure is composed of several subunits. Larger and more complex proteins often cannot fold to their active conformation on their own: misfolded structures may have reduced to no activity, undesirable activity that may result in disease. Chaperonins act as guides along the folding pathway for these proteins[51]. As part of the cell’s processes of protein synthesis, the ubiquitin is can be added to the structure. Ubiquitin can affect the interactions of the host protein with other proteins, and has functions associated with a number of cellular processes, including marking the host for degradation by the proteosome. Additionally, proteins, such as ATP Synthase – which stores energy in the chemical bonds of Adenosine Tri-Phosphate and operates in the mitochondria – and kinesin – shown in Figure 2.8 a transport protein which moves a cargo across the cell by walking on microtubules – form an integral part of the machinery of life.
Figure 2.9. Quaternary structure of a proteosome (PDB: 1FNT) A proteosome is a protein that decomposes other proteins by breaking their primary sequence. This is a complex protein which plays a vital role in the regulation of cellular lifecycle. The structure is that of a tube (blue) with caps (red).

Figure 2.10. A chaperonin protein is composed of many subunits and whose role is to assist the folding of other proteins. Left: top view, right: side view.
2.2 Molecular Dynamics

Since proteins form the backbone of life’s machinery, understanding the factors that effect protein behavior and formation is crucial to understanding the how life works. To illustrate some of the complexity of cells, David Goodsell, whose lab works on the Protein Data Bank’s Molecule of the Month series, has created figures based on observations. In Figure 2.11, Goodsell illustrates a Mycoplasma cell with a diameter or 300 nm, with molecules as appropriate size, location, and concentration. Included are DNA, DNA polymerase, RNA polymerase, mRNA, Ribosome, tRNA, chaperonine GroEL, proteosome ClpA, glycolytic enzymes, ATP synthase, sodium pump, and others. Additionally, small molecules like water, sugars, salts, ATP are not shown, but pervasive.

Since proteins exist in a crowded space with many possible interactions one important question is: how do proteins interact with each other? An old model in molecular biology is known as the lock-and-key theory: different molecules have different shapes possibly with complementary motifs and in order for them to interact the motifs need to ”fit” together like a log in a keyhole. As knowledge of protein dynamics has grown this theory is yielding to a better, more dynamic, understanding: everything is in motion, and proteins can adjust their structure based on stimuli. The chemical and physical properties of the partners drive the interactions.

As such, the field of Molecular Dynamics as grown significantly over the last several decades and has contributed to this improved understanding of proteins. Put simply, Molecular Dynamics attempts to answer the question: how do proteins move?

The fundamental concept of Molecular Dynamics is the propagation of the motion of atoms over time. In order to do so, the molecules need to be represented accurately. Atomic properties such as charge, shape, mass, need to be close to the observable values. The properties of atomic interactions: single, double, and triple bonds, van der Waals forces, electrostatic interactions need to be measured and mod-
Figure 2.11. This cell is 300 nm in diameter and the molecules are represented to scale. Illustration by David S. Goodsell, the Scripps Research Institute
Figure 2.12. Simplified Molecular Dynamics algorithm: Given positions at time $t_0$ and an appropriate timestep $\Delta t$, compute the forces, update the positions and time, and loop.

An important phase is observing the atomic coordinates of proteins through X-Ray Crystallography.

Initial methods of computer-simulated molecular dynamics were developed in the 1950s by Alder and Wainright \[7\]. Their model, while extended and elaborated on since, forms the basis for contemporary molecular dynamics. Simply put, atoms can be modeled as masses on a spring. Once the atomic coordinates are known, numerical integration of Newton’s equations of motion are used to project the coordinates forward in time. The force on each atom at time $t$ is computed. Then the positions and time are updated. It is crucial to choose an appropriate timestep $\Delta t$ otherwise numerical instability will cause the system to degenerate. See Figure 2.12 for an illustration of the MD algorithm.

Alder and Wainright’s initial work was limited but Michael Levitt a decade later pioneered the application of MD to biological molecules in the 1970s \[42, 69, 70, 121\]. Since, MD has been applied to a wide range of systems: simulations of the bovine pancreatic trypsin inhibitor (BPTI) \[84\], identification of integral motions such as hinge bending modes \[38\], tRNA flexibility \[32\], and study of E. coli chaperone GroEL \[126\] (Figure 2.10).

In order to facilitate this wide-spread use of MD, various programs have be-
come established for this use. The Groningen Machine for Chemical Simulations (GROMACS) [54, 99] is a very popular program. Others include the Assisted Model Building with Energy Refinement program (AMBER) [32], the Chemistry at Harvard Macromolecular Mechanics program (CHARMM) [30], the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [95] program, the Nanoscale Molecular Dynamics program (NAMD) [91], Tinker [89], and Desmond [19].

Before running the simulations, each program needs to parameterize the atoms and interactions of the chemical system. These parameters include the mass of each atom and its electronic state. These parameters are known as the "Force field" parameters and different programs can use them differently. Popular force fields are the AMBER [120] and CHARMM [30] family of forcefields. Each forcefield is tuned with experimental parameters to represent as closely as actual values. Certain forcefields are known to perform better for certain systems. For instance CHARMM forcefields are usually used to simulate systems with DNA, RNA, and lipids [82].

One of the major constraints on using MD is the force-calculation step. Since each atom can exert an effect on all others, this requires a $O(n^2)$ time calculation, where $n$ is the number of atoms. By making certain assumptions this limit can be lowered. For instance, on large enough systems, the interaction between atoms that are far enough apart can be negligible so the atoms are "cut off" from each other, reducing this to a linear algorithm. This can be a very crude assumption however, and so production uses of the cut-off method will periodically recompute the neighbors-list for each atom.

Another constraint is that of time-scale. In order for the numerical integration to be stable an appropriate timestep needs to be used. This timestep needs to be faster than the fastest motion of the system. Since certain bond vibrations occur in the femtosecond (fs) [13] timescales ($10^{-15}$ seconds) the usual choice is 1 fs. This creates a significant overhead, since most of the biologically-relevant motions are
much slower. One of the fastest known folding proteins, the Villin headpiece, folds in a few microseconds ($10^{-6}$s), a difference of nine orders of magnitude. Many others fold in milliseconds ($10^{-3}$s), seconds, and even hours for larger systems. As a result a simulation requires billions of MD steps to begin to reach the timescales of interest.

One approach to improving performance is to model waters implicitly. Since there are a large number of water molecules, but their motion is not of central interest, the effect of the water can be approximated using a potential. This results in higher performance (measured as the ratio of simulated time to real time, eg nanoseconds per day). Additionally, the lack of water molecules provides less drag on the protein, which speeds up some of the molecule’s slower timescale motions. In some cases, however, the error introduced by implicit solvent cannot be easily accounted for, as various proteins require the presence of water to coordinate certain motions and activity.

2.3 High Performance Computing

In order for MD simulations to reach the necessary timescales to observe the behavior of interest, specialized systems can be used. These High Performance Computing (HPC) environments can consist of protocols such as the Message Passing Interface (MPI), communication links like InfiniBand, specialized hardware like Graphics Processing Units (GPUs).

By using the Message Passing Interface (MPI), most MD programs (e.g. AMBER, CHARMM, GROMACS, NAMD) can scale can scale across multiple nodes in a compute cluster. However, certain programs are better suited for different systems. For instance, in explicit solvent GROMACS outperforms NAMD for small to medium systems whereas NAMD is better suited for large systems running in HPC. On the other hand, GROMACS outperforms NAMD in implicit solvent systems.

The Message Passing Interface (MPI) is closely associated with High Performance
Computing. MPI is a language-independent protocol that allows instances of a program running on different nodes to communicate with each other. A program using MPI can broadcast to all nodes, communicate with a single node, and accept communication from other nodes. The data communicated are typically primitive data types such as arrays, integers, floating point numbers, though MPI datatypes can be defined for complex types.

Important MPI concepts include Communicator Groups, Point-to-Point communication, and collective communication. MPI processes can be organized into groups connected by a communicator, which manages each process in its group. Point-to-point communication allows a process to send and receive data from one other process in the MPI session. This allows, for instance, an array to be shared and synchronized between the processes. Additionally, processes can broadcast and receive data to and from all the processes in the session.

Programs written using MPI account for being remote or local using branch statements. This is because each instance of an MPI program is running the same instructions until a branch occurs. This switch is done based on the rank of a process: the rank is an integer uniquely identifying a process within a group. For example, implementing a "ping-pong" example with two processes where process ranked 0 sends "ping" to a waiting process ranked 1. The second process then responds "pong" to the waiting first process. This program is identical until the first branch: if rank is zero then send "ping", else wait.

Another important aspect of MPI is the requirement that all processes exist for the entire runtime of the program. As a result, if a process fails, for instance, the node it is running on has a power failure, then the entire program will halt. As such, considerable effort is put into checkpointing the running state of the program as well as increasing the fault tolerance of the machines.

InfiniBand is a communications connection in the sense the commonly known eth-
ernet connection is but supports higher transfer rates (into the hundreds of Gigabits per second). InfiniBand is commonly used in supercomputing clusters.

Graphical Processing Units (GPUs) are specialized for processing three-dimensional graphical data for visualization. Originally used in the Gaming industry, the last decade has seen the application of GPUs to scientific applications. Programs like Mathematica and MATLAB, support using GPUs to drive certain computations. The MD community now has GPU support for several commonly used programs such as AMBER, CHARMM, GROMACS, and NAMD. GPUs supports single instruction multiple data (SIMD) instructions, which is a different architecture than the single instruction single data (SISD) on which most current processors were initially based. As a result, GPUs can have hundreds of cores where each core applies the same instruction to a different data point. Since the force calculation in a MD step can be done independently for each atom proper use of GPUs can result in substantial speedup over the same calculation on a CPU. However, there are certain costs to running a calculation on the GPU: GPUs have independent memory so data needs to be transferred from main memory to the GPU and back, and all paths of a branch statement are executed.

An alternative approach, taken by D. E. Shaw Research, has been the development of hardware specialized for running MD simulations. This machine, called Anton, uses applications-specific integrated circuits (ASIC) for the MD engine. Specialized control logic, programmable processors, specialized memory layout were used to increase computational performance. Increased network performance was accomplished by creating a high-performance network within and between ASICs, push-based communication from producers to consumers, autonomous direct memory access engines, prioritization of packets based on algorithm-specific data types. As a result, Anton was able to access the millisecond timescales with several orders of magnitude higher performance than MPI-based approaches.
Some notable achievements of the use of HPC for MD include the use of Anton to simulate between 100 $\mu$s and 1 ms for 12 proteins [74]. Additionally, the entire Satellite Tobacco Mosaic Virus has been simulated using NAMD [47].

2.4 Ensemble Simulations

One approach to accessing the longer timescales of MD is by statistically coupling independent simulations [110]. The key observation that allows is that simulations are sampling and underlying stochastic process. Given infinite time, a simulation should eventually sample every possible point in configuration space. Since infinite time is impractical, running simulations from multiple initial configurations increases the likelihood that a transition is observed.

The Folding@home (F@H) project [109] is an extreme example of the use of ensemble simulations by harnessing massive distributed computing resources. A key difference between the crowd-sourced F@H and the use of HPC infrastructure is the use of commodity hardware, instead of specialized, and high communication latency rather than optimized.

This approach has yielded further advances in understanding of folding pathways and topologies, statistical methods for analysis of MD data, understanding misfolding
and diseases such as Alzheimer’s and Huntington’s, and others.
3.1 The WW Domain

The WW domain, so named for two conserved tryptophan residues (W), is a 34 amino-acid sequence that forms three anti-parallel $\beta$-sheets separated by two flexible loop regions, as shown in Fig 3.1. This domain is an independently folding part of the PIN1 protein. PIN1 has been associated with protein regulation, the cell life-cycle regulation, and various other disorders [124, 78, 108, 94]. The WW domain itself is part of the binding region of the PIN1 protein [78].

Along with other $\beta$-sheet domains the WW domain is used as a model to study the formation of $\beta$-sheets. In this particular case, Liu et al. studied the changes in folding due to induced mutations [76]. Liu et al. primarily target the mutations to the Loop 1 (L1) sequence (insertions, deletions, substitutions), and the Threonine and Tryptophane in L2 and Sheet 3 respectively (substitutions). These mutations are shown in Table 3.1.

While Liu et al. report data demonstrating a marked difference is folding rates due to these mutations, as of now we may only theorize the underlying reason for these differences. For instance, mutant 14 has a folding time of 758 $\mu$s, compared to the WT’s 69 $\mu$s, due simply to the removal of the Glycine in S1. This may reduce the flexibility of the sequence forming S1, inhibiting its proper formation. However, deletion of the first Arginine in S1 (mutant 13) reduces the folding time by from 69 to 65 $\mu$s.
# TABLE 3.1

## WW DOMAIN MUTANTS

<table>
<thead>
<tr>
<th>MutantId</th>
<th>L1 Residues</th>
<th>Residue 23</th>
<th>Residue 28</th>
<th>Exp. Folding Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (WT)</td>
<td>SRSSGR</td>
<td>T</td>
<td>W</td>
<td>69</td>
</tr>
<tr>
<td>1</td>
<td>SRSSGR</td>
<td>T</td>
<td>F</td>
<td>56</td>
</tr>
<tr>
<td>2</td>
<td>SRSSGR</td>
<td>A</td>
<td>F</td>
<td>93</td>
</tr>
<tr>
<td>3</td>
<td>SRSSGR</td>
<td>D</td>
<td>W</td>
<td>96</td>
</tr>
<tr>
<td>4</td>
<td>SRSSGR</td>
<td>A</td>
<td>W</td>
<td>124</td>
</tr>
<tr>
<td>5</td>
<td>GRSSGR</td>
<td>T</td>
<td>W</td>
<td>236</td>
</tr>
<tr>
<td>6</td>
<td>ARSSGR</td>
<td>T</td>
<td>W</td>
<td>185</td>
</tr>
<tr>
<td>7</td>
<td>TRSSGR</td>
<td>T</td>
<td>W</td>
<td>333</td>
</tr>
<tr>
<td>8</td>
<td>SGSSGR</td>
<td>T</td>
<td>W</td>
<td>135</td>
</tr>
<tr>
<td>9</td>
<td>SRSEGR</td>
<td>T</td>
<td>W</td>
<td>200</td>
</tr>
<tr>
<td>10</td>
<td>SRSSGGR</td>
<td>T</td>
<td>W</td>
<td>325</td>
</tr>
<tr>
<td>11</td>
<td>SRSSGGR</td>
<td>T</td>
<td>W</td>
<td>193</td>
</tr>
<tr>
<td>12</td>
<td>CRSSGRC</td>
<td>A</td>
<td>F</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>S-SSGR</td>
<td>T</td>
<td>W</td>
<td>65</td>
</tr>
<tr>
<td>14</td>
<td>SRSS-R</td>
<td>T</td>
<td>W</td>
<td>758</td>
</tr>
<tr>
<td>15</td>
<td>S-SSGR</td>
<td>T</td>
<td>W</td>
<td>25</td>
</tr>
<tr>
<td>16</td>
<td>S-SSGR</td>
<td>A</td>
<td>W</td>
<td>19</td>
</tr>
<tr>
<td>17</td>
<td>S-ARGR</td>
<td>T</td>
<td>W</td>
<td>55</td>
</tr>
<tr>
<td>18</td>
<td>S-ADGR</td>
<td>T</td>
<td>W</td>
<td>12</td>
</tr>
<tr>
<td>19</td>
<td>S-ADGR</td>
<td>A</td>
<td>F</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>S-ADGR</td>
<td>T</td>
<td>F</td>
<td>14</td>
</tr>
<tr>
<td>21</td>
<td>S-RDGR</td>
<td>T</td>
<td>W</td>
<td>13</td>
</tr>
<tr>
<td>22</td>
<td>S-RDGR</td>
<td>T</td>
<td>F</td>
<td>13.9</td>
</tr>
<tr>
<td>23</td>
<td>SSGR</td>
<td>T</td>
<td>W</td>
<td>45</td>
</tr>
<tr>
<td>24</td>
<td>SRGR</td>
<td>T</td>
<td>W</td>
<td>51</td>
</tr>
<tr>
<td>25</td>
<td>SNGR</td>
<td>T</td>
<td>W</td>
<td>20</td>
</tr>
<tr>
<td>26</td>
<td>SNGR</td>
<td>T</td>
<td>F</td>
<td>14.5</td>
</tr>
<tr>
<td>27</td>
<td>SNGR</td>
<td>A</td>
<td>F</td>
<td>35.9</td>
</tr>
</tbody>
</table>

Mutations to the WW Domain with rates from Liu et al. [76].
Consider the following:

- Sheet 1 (S1) WEKRM: 2 out of 5 residues are hydrophobic (W,M) the rest (E,K,R) are electrically charged
- Loop 1 (L1) SRSSGR: all residues have a high pKa
- Sheet 2 (S2) VYYFN: 4 out of 5 residues (V,Y,F) are hydrophobic while the other (N) is basic
- Loop 2 (L2) HITN: consists of 3 basic residues (H,T,N) and a hydrophobic residue (N).
- Sheet 3 (S3) ASQW: has 2 hydrophobic residues (A,W) and 2 basic residues (S,Q).

As the mutations are directed towards L1, L2, and S3, changes in the bulk, flexibility, and electronegativity alter the free energy landscape. Previous studies of the WW domain have proposed that L1 and L2 form by different methods. L1 may form S1 and S2 via a “zipper”-like mechanism initiated at the loop residues, while L2 may for S2 and S3 via a collapse [125]. Examining the chemical properties of the amino acids in the loop and sheet regions supports this conclusion.
In order to understand how these mutations affect the landscape, first MD must be run, followed by further analysis. In addition to traditional MD, the AWE algorithm will be used to refine the results.

Liu et al.’s work \[76\] measure the experimental folding times (Table \[3.1\]) of the mutants by measuring the fluorescence over time. The T-Jump experiment uses laser-excitation to raise the energy of the systems. By measuring the change in tryptophan fluorescence, the folding rates can be determined. Since the Trp is hydrophobic and the folded structure thus shields the residue away from the solvent, a T-Jump experiment can be simulated by measuring the solvent accessibility of the residue.

The Mean First Passage Time (MFPT) gives an initial approximation of the folding rate as well. The MFPT can be determined given $\theta$ (the total time simulated), $n$ reactive trajectories (simulations that fold), $N$ total trajectories, the mean rate $\langle k \rangle$ and variance are given by \[61\]:

$$\langle k \rangle = \frac{n + 1}{\theta}, \quad \text{var}(k) = \frac{n + 1}{\theta^2} \quad (3.1)$$

However, understanding the effects of the mutations requires an approximation of the free-energy landscape each system. The free-energy landscape associates an energy level with each point of a system’s conformational space. A free energy surface, such as the two-well surface in Fig. \[3.2\] consists of wells inhabited by the stable conformations which are separated by energy barriers.

In order for the system to cross the barriers, the conditions have to be favorable to do so. Adding energy to the system in the form of kinetic energy is one way to do so by increasing the temperature.
Figure 3.2. A simple two well free energy surface. The color gradient from warm/red to cold/white indicates an increase in energy. Circles represent conformations on the energy surface. Wells are more populated since they have lower energy, with few conformations on the transitions regions.

3.2 Molecular Dynamics

Molecular Dynamics (MD) is the study of molecules using computers to generate simulations of molecular motion. MD allows probing of biomolecular motions at an atomistic resolution, which is difficult to achieve experimentally. The study of MD is not without it’s challenges, however.

A first major challenge is that of timescales. Essentially, MD is the propagation of Newton’s equations of motion over time. This integration is fundamentally limited to the scale of one or two femtoseconds. However, biologically relevant motions occur several order of magnitude time slower. For instance, one of the fastest known folding proteins is the Villin Headpiece folds at 1 microsecond with 35 amino acids. Other systems, such as the WW domain, NTL9, and the $\lambda$-repressor range is size from 35 to 80 residues, with folding times in low to mid microsecond range. While the study of these systems is accessible using MD, albeit computationally expensive, the size of other systems either prohibits their study, or constrains the study to fast motions. In addition to folding, other structural motions such as allosteric interactions, multimer formation, aggregation, and misfolding, are crucial.
A second challenge is that of sampling. The chemical milieu is not a solitary one: observable behavior is the aggregation of many individual events. As such, a few MD trajectories is typically not enough to capture in sufficient quantity events of interest. There are two solutions to this: run sufficiently long simulations, or compute an ensemble simulation by running many trajectories in parallel. Both these approaches have their trade-offs of resource requirements and time requirements.

A third challenge is also related to the unbiased nature of MD. Molecular motions occur in high-dimensional space with an energy potential associated with each coordinate. Since MD is unbiased, energy wells are sampled much more frequently than those along the slopes or mountains of the free-energy landscape.

Running molecular dynamics consists of integrating Newton’s equations of motion over time from an initial state. In the study of biomolecules, the initial state typically consists of the atom positions in three-dimensional space. Each MD step then consists of computing the inter-atomic interactions and updating each atom’s velocity vector, and moving to the next discretized time point.

MD has been an active field of research for several decades. As artefacts of this research, several software packages exists. Initially designed for custom hardware GROMACS (the GROningen MAchine for Chemical Simulation) [15], the latest major release, GROMACS 4 [54], runs on a variety of platforms. GROMACS is heavily optimized for each architecture on which it runs using hand-written assembly and SIMD instructions. The NAMD package is developed at the University of Illinois Urbana-Champaign [91]. Designed to simulate systems with a large number of atoms, NAMD employs the above techniques to scale up to many processors on a cluster or supercomputer. For example, in 2006, NAMD was used to perform a full atom 50-nanosecond simulation of the complete Satellite Tobacco Mosaic Virus, a much larger system than those normally approached by MD [47]. The AMBER software suite consists of a set of force fields and software for running MD [96]. CHARMM
has been under active development since the early 1980s, and supports the CHARMM set of force fields and several execution backends \[29, 30\].

Additionally, an emerging trend in MD is the movement towards general-purpose GPU computing. Since the atomic interaction computed at each time step are independent, the parallel architectures of GPUs are attractive to increasing performance. Friedrichs \textit{et al.} showed that under certain conditions GPUs can perform simulations several hundred times faster than 8-core CPUs \[48\].

These works have expanded on two main paradigms of MD data collection: the “heroic,” and “ensemble approaches.

The Anton system is a set of hardware components engineered to increase MD performance \[106\]. By using these application-specific integrated circuits, Anton allows simulations to probe the millisecond time scales. However, while these simulations can achieve biologically relevant time scales, each simulation must be scheduled to the system. The “heroic” paradigm thus is scoped to a small number of very long simulations.

On the other hand, the “ensemble” approach runs many short simulations exploring the free energy landscape. By finding overlapping regions in the trajectories the results can be “stitched” together to form a holistic view of the system. The Folding@home project exemplifies this approach on a global scale, allowing people from around the world to donate their unused computer cycles to MD \[109\].

3.3 Folding@home

The Folding@home (F@H) project is an endeavor that fits in the “ensemble” paradigm of molecular dynamics. F@H is a massively distributed platform for running MD using a crowd sourced model. This allows interested clients from all over the world to donate the free cycles of their personal computers to the study of protein dynamics. Since it’s inception \[109\] F@H has seen steady growth in the active
TABLE 3.2

FOLDING@HOME CLIENT STATISTICS

<table>
<thead>
<tr>
<th>OS Type</th>
<th>Total CPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>4,568,451</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>23,911</td>
</tr>
<tr>
<td>Linux</td>
<td>38,806</td>
</tr>
<tr>
<td>ATI GPU</td>
<td>358,566</td>
</tr>
<tr>
<td>NVIDIA GPU</td>
<td>583,475</td>
</tr>
<tr>
<td>Total</td>
<td>5,573,209</td>
</tr>
</tbody>
</table>

Folding@home client statistics as retrieved May 10 2013.

With 5.5 million CPUs, F@H thus present a large resource for gathering molecular dynamics data. Indeed, F@H has been used to access the millisecond timescales of protein dynamics [68], mutations associated with Alzheimer’s disease [123], and evaluating the ability of force fields and solvent models options to experimental observables [13].

The challenge that thus emerges is the analysis of the plethora of generated data.

3.4 Markov State Models

3.4.1 Motivation

Markov State Models (MSMs) have emerged in recent years as a tool for analyzing large ensemble of MD simulations. As a result of methodology development [112, 35, 23, 25], software tools have been release [88, 11, 40] and systems studied [119, 22, 26, 67, 122, 34, 12, 123].

Figure 3.3. A Markov State Model is built by first clustering conformations as sampled by molecular dynamics into “microstates”. Then transitions between microstates are determined. Finally, microstates are lumped together in order to form “macrostates.”

The consistent challenge arising from the use of MD is that of sufficient sampling. By running number of simulations in parallel with varying initial conditions but in the same generalized ensemble, sampling of state space can be accelerated. The problem remains, however, of extracting meaningful rates from these data. One key assumption underlying the use of MSMs is that biological systems exhibit multiple timescales. For instance, bond vibrations and rotations occur in the fast timescales – in the picosecond range – while global motions such as folding occur slowly – order of microseconds to milliseconds. By exploiting this characteristic, an MSM can be constructed as following (see Fig. 3.3).

First, the conformations sampled by the MD simulations are clustered according to some similarity metric – in cartesian or dihedral space, for instance. The resultant clusters are termed “microstates.” This clustering is based solely on the similarity between two conformations, disregarding and kinetic information. The microstates are intended to capture conformations within local minima on the free-energy surface, and a small enough threshold for the metric should be used. Once the microstates have been determined, the simulation data is reexamined to extract changes between conformations. By counting the transitions between microstates a network model
capturing the fast timescales is created. However, the goal is presumably to study the slow motions, so further work is needed. By specify a lagtime $\tau$ microstates can be lumped together, forming “macrostates.”

A further assumption is that behavior is Markovian. That is, a system’s state $x(t)$ is only dependent on $x(t - 1)$. This assumption also allows the system to be propagated forward in time. The probability distribution at time $t + \tau$ can be computed directly from the transition probability matrix $T$:

$$p(t + \tau) = p(t)T(\tau)$$  \hspace{1cm} (3.2)

### 3.4.2 Usage

One desirable property is that MSMs allow the underlying energy landscape to be modeled as a network. In this case, nodes are assumed to be wells and edges transitions over the energy barriers. By associated probability with these edges, pathway analyses can be applied in order to understand folding dynamics. This has lead to several studies positing on the fundamental natures of folding dynamics. For instance, an alternative theory to the traditional “folding funnel” model for protein folding has emerged from the use of MSMs. In this alternate model the folded conformations form kinetic hubs that form intermediate states in the folding and unfolding pathways [22]. Additionally, work suggests that folding is a robust process even in the presence of noise as exemplified by the actual biological milieu [123]. The long “heroic”-type simulations can also be studied using the MSM model. An exemplar of the ultralong trajectory paradigm, the Anton system [106] allows probing the millisecond timescales of large systems [43, 107, 93, 102, 74]. By building MSM models of these long trajectories deeper understand of the underlying dynamics can be obtained [67]. MSMs have also been applied to the study of several different protein systems such as NTL9 [119] which folds in the milliseconds timescale, Protein
Figure 3.4. MSMs allows the folding pathways to be studied as a network model. Starting from the extended structure (left), multiple pathways exist to arrive at the folded structure (right). This figure appears in “To milliseconds and beyond: challenges in the simulation of protein folding” [68].

\(\lambda_{6-85} [26]\), MSMs have also been used in an attempt to understand general properties of protein folding by analysis of the network topologies [122, 34].

Fig. 3.4 shows an example result from the application of MSMs to folding dynamics as appearing in [68]. The network of states allows intermediate state between the unfolded and folded structures to be found. Additionally, potential bottlenecks and high-flux pathways are found.

MSM models are however still susceptible to uncertainties due to poor sampling. In order to overcome this, recent work has been done to use MSM models to guide the sampling mechanism [25]. In this scheme, the MSM can be used to both guide state discovery and minimize uncertainties: first, simulations are run out of an initial state definition. By building an MSM new simulations can be started after determining each state’s contribution to the error of slow timescales [55].
3.5 Accelerated Weighted Ensemble

The Accelerated Weighted Ensemble (AWE) is an emerging method for overcoming some of the sampling challenges of MD \[41, 4\]. Specifically, Weighted Ensemble (WE) methods, of which AWE is a variation, allow unbiased statistics to be collected from a biased ensemble of simulations \[41\]. A problem with the simple ensemble approach is that the MD simulations preferentially sample the wells of the free energy landscape over the transition regions. In order to overcome this, the WE ensemble forces undersampled regions to be sample equally as the stable regions by weighting the trajectories. Regions of interest on the landscape are labeled as Product or Reactant, for instance “unfolded” or “folded” regions.

Fig. 3.5 illustrates the WE ensemble algorithm. State space is first partitioned into cells out of which walkers (MD trajectories) are run. Periodically, simulation is halted in order to accomplish three things: 1) maintenance of cell population is done by splitting/merging walkers, 2) update the walker weights, 3) update color if a product or reactant region is entered. By tracking the number of walkers entering
the product and reactant regions (a color update) the flux of the process can be computed. By monitoring the flux over time we can stop the simulations once convergence is achieved. By providing an initial approximation for the walker weights this convergence can be accelerated (hence the “A” of AWE).

Several properties of the AWE algorithm make it desirable. First, it is robust in the face of the state partitioning [46]. Second, forward and backward fluxes are readily computed using the coloring method [39]. Third, storage requirements are minimal, compared with traditional ensemble MD data collection [4]. As a result of running AWE, we obtain a transition matrix of the cells and can apply further analysis to uncover different pathways. Similar protein systems can then be compared by examining these networks for common and divergent paths and states.

As described by Darve and Ryu [41], the computation cost of WE methods is quite high. Given $C$ cells, $N$ walkers in each cell, each iteration requires $N \times C$ MD simulations run, which themselves are costly. A realistic application may require tens of thousands of simulations run in each iteration [4, 5]. As this is not feasible, even running in parallel, on a single computer a scalable distributed application is developed [4, 5].

Conceptually, AWE fits into a fork/join programming model: at the beginning of each iteration, a number of MD simulations are spawned. There is a synchronization barrier joining the conclusion of these simulations. After the barrier completes and the resampling procedure is run a new set of simulations must be spawned. This process proceeds iteratively until convergence is achieved.

In order to finish within a reasonable time period the application needs to scale to as many resources as available, and this scaling must be graceful. That is it must adapt to shared resources, minimize network traffic, and be fault-tolerant.
3.6 Comparison of MSM and AWE methods

A partitioning of state space is crucial to both the MSM- and WE-based methods. While other methods require a reaction coordinate, which is difficult in practice to obtain due to the high dimensionality of the problem, MSMSs and WE do not have this limitation. Additionally, both methods are very general; while WE methods are only now beginning to emerge as applicable tools, MSMSs have been applied successfully to examine MD data.

However, there are cases where MSM-bases analyses are biased. In particular, an MSM is very sensitive to the location of the barriers of the underlying partitioning. By adjusting the position of the partition to represent an entire transition regions as one cell compared to the partition more naturally representing the energy barrier, recent work shows that significantly increases the bias in the rate calculations [46]. This work shows that the MSMSs can be very sensitive to the choice of partitioning.

Table 3.3 Both methods are highly general and produce a network model for further analysis, flux can be computed for different processes. However, AWE is not as sensitive to the choice of partition as MSM, is guaranteed to converge to steady state, does not require a Markovian assumption. Additionally, AWE is only now beginning to be applied to studying large protein system while MSMSs have been used extensively.

Since the ultimate goal is to have MD-based result match experimental data, further refinement of MSM results is needed. By using an MSM to provide an initial approximation which can be rapidly computed, WE can thus be seeded. This will accelerate the convergence of the WE method. The complementary nature of the two methods implies the following workflow: 1) run unbiased ensemble MD, 2) develop an initial partition of the energy landscape using MSM, 3) refine the MSM results to overcome the biases.
TABLE 3.3

COMPARISON OF MSM AND WE METHODS

<table>
<thead>
<tr>
<th>Property</th>
<th>MSM</th>
<th>AWE</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Network model</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Flux computation</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Adaptive Sampling</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Partition robustness</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Guaranteed convergence</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Doesn’t need Markovian assumption</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Computational cost</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Extensively applied to protein dynamics</td>
<td>✓</td>
<td>X</td>
</tr>
</tbody>
</table>

An overview comparing MSM and WE methods.

3.7 Summary

The study of protein dynamics encompasses a wide range of problems. In order to understand how mutations affect dynamics several issues need to be overcome. First, due to the number of mutants and the computational cost of MD a large ensemble of simulations need to be run; The F@H platform provides resources to gather the data. Next, aggregate analysis of the can be done using Markov State Models. However, MSMs are susceptible to the partition definition such that poor partitions lead to biases subsequent analyses. WE methods share many commonalities with MSMs, but are guaranteed to reach the steady state and are not as impacted by the partition as MSMs. This is desirable since most data will likely be undersampled due to the complexity of running MD. MSMs can thus be used to complement the use of WE by providing an initial approximation that WE can refine.
3.8 Folding@home data

Several projects have been started on Folding@home investigating different simulation parameters and have produced over 16 TB of data. One group of projects (10009 - 10039) was run using a Normal Mode Langevin integrator\cite{62, 116} with a timestep of 75 fs, with SCPISM implicit solvent model, and at a temperature of 300K starting from the extended structures. In addition to studying the dynamics of the WW mutants, these data are being applied by collaborators to the development and validation of accelerated molecular dynamics methods. The second group of projects (10062 - 10082) uses the Langevin integrator with the Amber96 force field and the Generalized Born implicit solvent model at a temperature of 300K from the extended conformation. Additionally, these simulations take advantage of the virtual sites approximation, allowing the use of a 5 fs timestep. In a further attempt to accelerate the dynamics the $\gamma$ parameter was set to $1 \text{ ps}^{-1}$. The third group of projects (7000 - 7028) extends the previous set by raising the temperature to 395K, starting simulations from both extended and folded conformations, and with $\gamma$ values of 1 and $91 \text{ ps}^{-1}$. Each of these projects thus has four ensembles running: 0) extended, $\gamma = 91 \text{ ps}^{-1}$, 1) extended, $\gamma = 1 \text{ ps}^{-1}$, 2) folded, $\gamma = 91 \text{ ps}^{-1}$, 3) folded, $\gamma = 1 \text{ ps}^{-1}$. Using a higher temperature was done to mimic the melting temperature of the protein as reported in \cite{107}.

3.9 HPC Simulations

Several simulations have been run on GPUs using the Langevin integrator with a timestep of 2 fs, the Amber99sb-ILDN-NMR force field \cite{13}, the Generalized Born implicit solvent model, $\gamma = 1 \text{ ps}^{-1}$, and a temperature of 370K. Additionally, we have a long ($200+\mu$s) simulation of WW22 with the Langevin integrator, 2fs timestep, Amber96 force field, Generalized Born implicit solvent, $\gamma = 1 \text{ ps}^{-1}$, at 395K.
3.10 Initial analyses

3.10.1 RMSD Distributions

Fig. 3.6 shows the distribution of the RMSD values for a slow and fast folding mutant. Each distribution has two peaks at 3 Å and 8 Å. This indicates that the simulations are folding the proteins.

3.10.2 Mean First Passage Time correlation

A first approximation for the aggregate folding time of the simulation is the mean first passage time (MFPT). Fig. 3.7 plots the mean first passage times for each mutant against the experimental values. The correlation which is expected is not present. This measure is unable to capture the differences between folding rates for these mutants, so further analyses are needed.

3.10.3 Approximate Free-energy Surfaces

The free energy surfaces based on the radius of gyration and RMSD to folded structure are shown in Fig. 3.8. Fig. 3.8a and 3.8b compare the slowest mutant folding in 758µs with one of the fastest at 13µs. This comparison is unable to differentiate between the two. Fig. 3.8c and 3.8d compares the dynamics of the same proteins but
over a smoother energy surface ($\gamma = 1\text{ps}^{-1}$) from the folded conformation: showing a greater tendency of the faster folder to unfolded over the slower. While these results are promising, this analysis in unable to scale to all the mutants: the free energy plots for the majority of the mutants are very similar and do not differentiate between their folding rates. Thus, while this analysis is initially interesting, it is not enough to explain the differences in folding times.

3.10.4 MSM results

Calculation of the implied timescales of an MSM should be indicative of the quality of the model. Two properties in particular are looked for of the slowest timescales: 1) separation between timescales and 2) convergence to a timescale. In practice is has been impossible to consistently use the implied timescales across the mutant datasets. An example is given in Fig. 3.9 for two mutants (units in nanoseconds). The MSM for the slow folder is able to separate the two slowest timescales but convergence is not achieved, while the fast folder is unable to separate the timescales.
Figure 3.8. Free energy surface for 758µs (left) and 12µs (right) mutants. The x- and y-axes are the radius of gyration and RMSD in Å. The color bar is the free energy: smaller/blue is lower energy while larger/red is higher energy.

Figure 3.9. Implied timescales from building MSM models for two mutants. The x-axis is the lagtime and the y-axis the implied timescale. Both axes are in units of nanoseconds.
3.10.5 MSM Network models

Regardless of the implied timescale results above, network models were nonetheless built from the MSMs. With the caveat that the partitioning is most likely poor, these models nonetheless present promising results, although further work remains to be done. Fig. 3.10 shows two network models from the MSMs built of a slow and fast folder. Nodes are the states and edges indicate transitions between states. Nodes are sized relative according to their betweenness centrality measure: a large node indicates that a greater fraction of the shortest paths pass through that node compared to smaller nodes. The node color indicates population where lighter indicate higher population. The differing topologies imply several interesting properties of the proteins. The first is that the slow mutant has a small number of nodes that are crucial for connecting different parts of the network, while the fast folder has many. The second is the location of these high-betweenness nodes: the slow folding protein’s few hubs are far away from the major groups, the fast folder has several of these hubs throughout the network.

Figure 3.10. Network models from the MSMs of two mutants. Node size represents the betweenness centrality measure (larger = higher betweenness). Node color represents population (lower is darker).
Figure 3.11. The long trajectory simulation displays multiple folding and unfolding events.

3.10.6 Ultralong simulation

By this point an ultralong simulation of WW22 had been obtained reaching over 200\(\mu s\). In order to obtain this data, a simulation was run for over 9 months on a GPU. It should be emphasized that two simulations were actually run with \(\gamma = 1ps^{-1}\) and \(\gamma = 91ps^{-1}\). While the \(\gamma 91ps^{-1}\) simulations folded, very few folding/unfolded events were found. On the other hand, \(\gamma = 1ps^{-1}\) resulted in several events as shown in Fig. 3.11.

3.11 Summary

In summary, several different types of analyses have been applied in an attempt to extricate the effect of mutations on the folding rates of several WW domain systems. Multiple different analyses were applied, but the underlying differentiating features of the free-energy landscape remain obscured. An ultralong simulation has indicated that the MD parameter choices induce a significant bias. Additionally, in order to obtain sufficient folding events high temperature and reduced viscosity had to be used, further removing the systems from biological relevance. Therefore, in order to build a deeper understand of the underlying causes of different folding rates, improved
methods are needed.
CHAPTER 4

PERFORMANCE OF NAMD AND GROMACS

4.1 Introduction

Molecular Dynamics (MD) is a physics-based technique for simulating the movement of molecules at the atomic level by approximating solutions to Newton’s equations of motions. MD finds uses in studies of protein folding, virtual drug screening, design of polymers, and sampling of molecular configurations.

Unfortunately, MD is very computationally intensive, often requiring months of computer time on a large distributed cluster to simulate milliseconds of dynamics for a medium-sized protein. Simulating a large protein (e.g. the $\beta_2$ Adrenergic Receptor) on biologically-relevant time scales (milliseconds through hours) using a standard desktop computer would take billions of years. Figure 4.1 shows the relationship between the amount of time that can be simulated and the amount of real time required to run that simulation on an average desktop computer. The figure shows the amount of simulated time in a single iteration of a MD simulation, the upper bound on the total amount of time we can currently simulate, and the time scales we’d like to be able simulated. Lastly, the figure relates these three amounts of simulated time with the amount of time it would take to execute on an average desktop computer.

Multiple approaches exist for the parallelization of MD simulations. Some of these techniques approach the problem from a technological perspective. The GROMACS package has been under development for the last 20 years. Initially designed for
Figure 4.1. For most MD simulations, the time-step is around 1 femtosecond. For full atom systems, increasing the timestep results in unstable mathematical solutions. However, biologically interesting behaviors usually occur in the millisecond to seconds timescales, thus requiring many very small time steps to reach such timescales. On a single fast modern computer, accessing the desired timescale is unfeasible [50].

custom hardware (the GROningen MACHine for Chemical Simulation) [15], the latest major release, GROMACS 4 [54], runs on a variety of platforms. GROMACS is heavily optimized for each architecture on which it runs using hand-written assembly and SIMD instructions.

For example, the adaptation of MD to clusters and supercomputers employs algorithms and constraints that allow the work load for groups of atoms to be assigned to separate processors and limits the need for communication to time scales allowable by the latencies of such machines. The NAMD package is developed at the University of Illinois Urbana-Champaign [91]. Designed to simulate systems with a large number of atoms, NAMD employs the above techniques to scale up to many processors on a cluster or supercomputer. For example, in 2006, NAMD was used to perform a full atom 50-nanosecond simulation of the complete Satellite Tobacco Mosaic Virus, a much larger system than those normally approached by MD [47].
Recently, D.E. Shaw et al. developed Anton, a massively parallel machine that makes use of Application-Specific Integrated Circuits (ASICs) for running long simulations\cite{105}. Anton allows simulations to probe behavior at millisecond time scales, which is several orders of magnitude greater than those approached by other contemporary runs. The purpose of Anton is to generate few but very long simulations and does so by providing a system architecture optimized for MD.

An emerging trend in MD is the movement towards general-purpose GPU computing. MD solves a separate equation for each atom in the system that depends on the positions of every other atom. The input and output data for each time step are independent such that the positions of the atoms in the former time step will not be modified in the computation of the positions in the next time step. In addition, each atom’s position is stored in a separate location. Thus, there is data dependency from one step to another but not between atoms in a single time step. This sort of model is “embarrassingly parallel” on architectures that support shared memory and SIMD (single instruction, multiple data) instructions up to the number of data which a single instruction can be executed simultaneously. Friedrichs et al. showed that under certain conditions GPUs can perform simulations several hundred times faster than 8-core CPUs \cite{48}.

There has been great success in porting MD simulations to the GPU, however it has not been without its problems. For example, the Particle Mesh Ewald (PME) allows long-range electrostatic forces to be calculated in $O(n \log n)$ time instead of $O(n^2)$ time when used with explicitly solvated systems and periodic boundary conditions. However, the GROMACS developers reported difficulty parallelizing and implementing the algorithm on the GPU, resulting in an almost no speed up over an 8-core CPU despite the theoretical improvement in run-time complexity \cite{3}. There are several factors likely influence this outcome, among them are the memory access, flow control, and data parallelism and dependencies. Parallelizing simulations across
multiple GPUs is also difficult due to the latency of copying data between one GPU, the host, and another GPU.

Other approaches solve the problem at a higher, methodological level, effectively changing how the experiment is performed. The Pande group at Stanford uses a statistical approach where multiple simulations are run in parallel on the Folding@Home distributed computing environment and then analyzed as an “ensemble” to look at the statistical behavior of the system. Folding@Home utilizes donated idle time on participants’ average desktop computers to build a massive, distributed cluster. Multiple MD simulation packages can be used with Folding@Home as long as the developers compile their software against the Folding@Home library to produce a “core.” Task assignment and collection is handled by servers maintained by the researchers using of Folding@Home.

All of these approaches have trade offs and implications that affect the types of experiments that can be run. For example, virtual drug screening performs many small simulations in parallel to filter libraries containing millions of compounds in order to find few (around one hundred) drug candidates. As these simulations are computationally cheap and independent they can be run on a cluster or distributed system. After several candidates have been identified, a scientist will run much more extensive simulations to see if those candidates cause desired conformational changes in the protein’s structure upon binding. As these proteins are often membrane bound or explicitly solvated, the systems being simulated are very big and require substantial computing power if they are to be simulated over biologically-relevant timescales. Such simulations may require more computational power and lower latency than is available on the systems used in distributed computing platforms (e.g. Folding@Home), requiring a cluster or supercomputer.

Although some differences (like the above) do not effect the relevance of the simulation, other trade offs do. The GPU speed ups depend on the usage of an
“implicit solvent” model which solves mathematical equations approximating the
effect of water molecules on the protein rather than including water molecules among
the atoms being simulated. Unfortunately, this trade off can lead to results that
do not match \textit{in situ} experimental results or are physically inaccurate. Thus, the
choice of architecture ultimately has an effect on the scientific questions that can be
answered through MD simulations.

We investigated the effect of computer architecture on the different sorts of scientific experiments that can be done with MD. We determined how different simulation methods perform on each architecture. We also investigated how the design of MD simulation packages affect performance on different architectures. And lastly, we used the results of these investigations to show how the choice of architecture ultimately affects the type of science inquiries that can be performed.

We used three main approaches to investigate these questions: First, we ran several types of simulations (vacuum, implicitly solvated, and explicitly solvated) with molecular systems of different sizes on different CPU architectures (single core, multicore, cluster) and a GPU architecture using different MD packages (NAMD, GROMACS) to determine the conditions under which different methods perform best. Then, we used microbenchmarks to benchmark the differences in performances of common operations such as branches and memory access between the CPU and GPU. Lastly, we used the results of the microbenchmarks to explain why certain methods perform better on one architecture over another.
4.2 Materials and Methods

4.2.1 Benchmark Hardware Configurations

4.2.1.1 Hardware

We ran all benchmarks on machines maintained by the Center for Research Computing (CRC) at the University of Notre Dame. The CRC provided a cluster of nodes to which jobs were submitted through the Sun Grid Engine (SGE) system. In order to use SGE, one defines a task as a script specifying the resources needed and the program to execute. For example, one might specify the minimum amount of available memory required, the number of processors on the node, the number of nodes to use with MPI, and so forth. Since one of our lab’s machines is a head node, we also used it for some of the benchmarks. There were no problems with multiple jobs on this node interfering with each other since we had control over the jobs being run on the machine. This node had four 6-core Intel Xeon E7540 processors operating at 2.00 GHz and 128GB of RAM. Additionally, the CRC provided a node with several GPUs for development. It had two NVIDIA Tesla M2050 graphics cards, each with 2.5GB of RAM, 448 cores, and a clock rate of 1.15 GHz, two 6-core Intel Xeon X5650 processors running at 2.67 GHz, and 48GB of RAM.

4.2.2 Software

We used three different MD simulation packages for performing the benchmarks. GROMACS is heavily optimized for speed on single and multi-core architectures. GROMACS has experimental MPI support for distributed computation across multiple nodes, however we were unable to get it working. GROMACS-OpenMM allows GROMACS to use the GPU for MD simulations via the OpenMM GPU MD simulation library. NAMD is designed for running simulations of large molecular systems in a distributed manner. Using MPI, NAMD can scale to thousands of processors on
TABLE 4.1

ARCHITECTURES AND SOFTWARE BENCHMARKED

<table>
<thead>
<tr>
<th>Architecture</th>
<th>NAMD</th>
<th>NAMD&lt;sub&gt;mpi&lt;/sub&gt;</th>
<th>GMX</th>
<th>GMX&lt;sub&gt;GPU&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 node, 1 CPU core</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>1 node, 12 CPU cores</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>2 nodes, 24 CPU cores</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 node, 1 GPU</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Combinations of architectures and software used in benchmarks.

GMX = GROMACS, GMX<sub>GPU</sub> = GROMACS<sub>OpenMM</sub>

a cluster or supercomputer.

Table 4.1 shows the combination of architectures and software packages used in the benchmarks.

4.2.3 Benchmark Simulations

To compare the relative performance of MD on a single CPU, multiple CPUs, distributed with MPI, and on the GPU, we prepared common molecular systems and determined common simulation parameters to be used on all of the software / platform combinations. Each system was prepared in vacuum, with an implicit solvent model, and with an explicit solvent model. Table 4.2 indicates the number of atoms in each benchmark simulation\(^1\).

There are several basic parameters for configuring MD simulations. The first parameter is the integrator. The GROMACS simulations were run using the stochastic Langevin Leapfrog integrator while the NAMD simulations used the Velocity-Verlet

\(^1\)The PDB code for the Tumor Suppressor Complex is 2H1L
TABLE 4.2

ATOMS IN BENCHMARKED SYSTEMS

<table>
<thead>
<tr>
<th>Molecule</th>
<th>Vac &amp; Imp</th>
<th>Exp (GMX)</th>
<th>Exp (NAMD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alanine Dipeptide</td>
<td>20</td>
<td>1100</td>
<td>1444</td>
</tr>
<tr>
<td>Fip35</td>
<td>542</td>
<td>6232</td>
<td>8257</td>
</tr>
<tr>
<td>Plasmepsin 4</td>
<td>5155</td>
<td>30639</td>
<td>41845</td>
</tr>
<tr>
<td>DHFRTS</td>
<td>9094</td>
<td>52950</td>
<td>68793</td>
</tr>
<tr>
<td>Tumor Suppressor Complex</td>
<td>107984</td>
<td>1850042</td>
<td>1110689</td>
</tr>
</tbody>
</table>

Number of atoms in benchmark simulations.
Vac = Vacuum, Imp = Implicit, Exp = Explicit, GMX = GROMACS

integrator. Another parameter, the length of the time step, determines the amount
of time simulated in each iteration of the simulation. We used a time step of 1 fs
for all of the simulations. Unfortunately, due numerical instabilities cause by very
high resonating frequencies of the bonds, the time step is limited to 2-5fs for most
simulation methods. This parameter has no effect on the run time of the simulations
as long as they are all run for the same number of iterations.

The number of time steps for which the simulation is run one of the most essential
parameters as we stated in the introduction. For all the simulations except the
GROMACS explicit solvent models, we conducted 100 picosecond simulations. Since
1 femtosecond of protein movement is simulated in each iteration, 100 picoseconds
is equivalent to 100,000 steps. For the GROMACS explicit solvent models, we only
ran the simulations for 0.1 picoseconds (1000 steps) due to their high computational
complexity. For the large Tumor-Suppressor Complex the NAMD simulations were
run for 0.1 ps as well. However, the GROMACS simulations of the complex were run
for less than 50 steps due to the amount of time each step took. The run time of the
simulations scale linearly with the number of iterations.

For explicit solvent models, we set the dimensions of the box of water molecules around a protein system to extend 10 angstroms from the edge of the protein in each direction. As the size of the the water box increases, the number of atoms the model contains and the computation time increases. This explicit solvent set up causes the molecular system to contain 5 to 70 times more atoms than the vacuum or implicit solvent models. This increase in the number of atoms significantly increases run time as the run time of the simulations scale quadratically with the number of atoms in the model.

Another crucial parameter is the cutoff distance which determines the maximum distance from an atom at which other atoms are used in the calculation of the long-range electrostatic forces acting on that atom. By setting an appropriate cutoff distance, the time complexity can be reduced from $O(N^2)$ to $O(N \log(N))$ or $O(N)$ in some cases, but will sacrifice accuracy to some degree. For NAMD simulations, we chose a cutoff distance of 12 nanometers. This set up is commonly used for NAMD because it usually leads to significant speed ups and still maintains the stability and accuracy of simulation. For GROMACS simulations, since the cutoff made the simulations very unstable, we disabled this method, which is equivalent to setting the distance to infinite.

The run times for the simulations run on the CPUs range from 10 seconds to several days. We ran each simulation with the same set up 3 times and calculated the average run time to eliminate random errors.

The MD community often use a metric of the amount of elapsed simulated time over the amount of elapsed real time normalized to nanoseconds per day. This metric is used by Friedrichs, et al. when comparing the performance of MD on the GPU and CPU. This is the metric that we used.
4.2.4 GPU Parameter Sweep

In addition to the common systems, we performed a parameter sweep of the configuration parameters to determine the effect different algorithms had on GPU performance. We used the molecular systems from the CPU vs GPU benchmarks as baselines. The first test compared the affect of using the deterministic Velocity Verlet integrator instead of the stochastic Langevin Leapfrog integrator. The second test compared the effect of using constraints on the bonds against using no constraints on the bonds (baseline). The third test compared the effect of using cutoffs with different parameterizations against using no cutoffs (baseline). (In GROMACS-OpenMM, cutoffs are only supported for implicitly solvated systems.) The fourth test compared how the frequency at which data is written out to permanent storage affects the performance of the simulation.

4.2.5 Microbenchmarks

In an attempt to understand how the architectural differences between hardware configurations affected the simulations we decided to define some microbenchmarks. Due to time and complexity constraints we focused on two of the hardware configurations only, namely using a single core of a CPU and a GPU (using one to all cores). With these two configurations we devised a set of tests to cover some basic architectural features based on our performance expectations.

One of the architectural features of the GPU that we had knowledge of and believed would cause issues with performance was dynamic branching. From our knowledge, GPUs always calculate both sides of the branch are required to be executed.

The tests we decided on were the following:

**Single Sided Branching** This test covered the case of branching where computation was either executed or not. We chose to test this as we knew that long
range forces within simulations rely on cutoffs to reduce computational load. By having this test we could see if the fact that both sides of the branches must be executed might affect the entire warp even if there is only a single side.

We will test this by looping over all atoms and then cull data based off of a cutoff condition.

**Double Sided Branching** This test covered the case of branching where different execution of different computations are predicated on some condition. We chose this test to measure the full extent of performance degradation due to both sides of the branches requiring execution.

We will test this by looping over all atoms and then doing different calculations based on a pre-specified condition.

**GPU Write back** This test covered writing data back from the GPU to the main memory as it is known to be a bottleneck. This is important because as this is the only way that the user could receive any feedback on how the simulation is running. It is also important if there are pieces of code that must be executed on data that are not able to be developed for the GPU.

We will test this by running \( n \) iterations of a function writing data back at a frequency less than \( n \).

**GPU Single Thread** This test covers forcing a GPU to execute code solely in a single threaded manner. The goal was to show that the parallelism that a problem exposes can be exploited by running on a GPU, even though a CPU core has greater performance than a single GPU core.
4.3 Results

4.3.1 Scalability of Architecture/Software Combinations with System Size

Figures 4.2 - 4.4 detail the scaling in performance in ns/day of the different architecture/software combinations with respect to the size of the molecular systems in atoms. There is a separate figure for each type of solvation. The horizontal axes are the number of atoms in the systems being simulated, while the vertical axes are the amount of simulated time (in nanoseconds) obtained per day of run time. Due to the multiple orders of magnitude differences in the performance and the number of atoms in the systems, the plots use logarithmic scales on both axes. Figure 4.2 shows the results for the systems in a vacuum, Figure 4.4 shows the results for the implicitly-solvated systems, and Figure 4.3 shows the performance of the explicitly-solvated systems.

Figure 4.2 shows the results of simulating systems in vacuum using NAMD on
a single core (blue line), 12 cores (green line), and a 24-core cluster (pink line) and GROMACS on a single core (red line), 12 cores (teal line), and a GPU (cyan).

We were unable to run simulations of the tumor suppressor using NAMD on the 24-core cluster set up and with GROMACS on the GPU. In the case of NAMD, our job was not scheduled to run on the cluster in the period of time we had before the project was due. In terms of GROMACS on the GPU, the memory usage (4GB) required to simulate the system exceeded the amount of memory on the GPU (2.5GB). Therefore, we were not able to test the scaling of the 24-core cluster and GROMACS on the GPU on a system beyond about 50,000 atoms. The absence of these data points do not affect the conclusions that we reached.

For simulations of models with up to about 5,000 atoms, architecture is the dominant factor of performance for the systems in vacuum. The single-core versions of NAMD and GROMACS show similar performance and scaling up to about 5,000 atoms, although the single-core version of GROMACS outperforms the single-core version of NAMD by a constant amount. Going from the single core architecture to the 12-core architecture, there is an order of magnitude increase in performance.

Of the 12-core versions, NAMD performs worse than GROMACS by a constant amount, like with the single-core versions. There is less of an increase in performance going from the 12-core architecture to the 24-core cluster. The 12-core version of GROMACS and 24-core cluster version of NAMD show nearly identical performance up to about 5,000 atoms. The 12-core versions of NAMD and GROMACS and 24-core cluster version of NAMD also show similar performance and scaling trends up to about 5,000 atoms.

The GPU shows about an order of magnitude increase in performance over the 12-core version of NAMD for a system of about 500 atoms but this seems to be somewhat of an outlier since the GPU version of GROMACS performs similarly to and follows the same scaling trends as the 12-core versions of NAMD and GROMACS.
and 24-core cluster version of NAMD for other systems with 5,000 or fewer atoms. It should be noted that the GPU version of GROMACS outperforms all other architecture/software combinations for systems below about 5,000 atoms in size. As there seems to be little difference between NAMD and GROMACS on the same architectures for systems with fewer than 5,000 atoms but significant differences between the performance when only considering the architectures, we can conclude that the architecture is the most dominant factor of performance when simulating systems under 5,000 atoms.

However, when we consider the performance trends when simulating systems with 5,000 to 50,000 atoms, the software starts to have an effect on performance. Between systems of 5,000 and 10,000 atoms, the single-core version of NAMD starts to outperform the single-core version of GROMACS. Likewise, the 12-core version of NAMD begins to outperform the the 12-core version of GROMACS. The 24-core version of NAMD and GPU version of GROMACS show similar performance for systems up to 10,000 atoms in size, but we were unable to test a larger them with a larger system, so we are unable to determine how they scale past 10,000 atoms. Although all of the single and 12 core versions of NAMD and GROMACS show decreasing performance as the system size increases, it is apparent that NAMD scales better than GROMACS on both the single and 12 core architectures. It seems that GROMACS is able to achieve better performance than NAMD on systems of 5,000 or fewer atoms, but its performance is not able to scale as well as NAMD’s for larger systems.

Figure 4.3 details the results for simulating explicitly-solvated systems using NAMD on a single core (blue line), 12 cores (green line), and a 24-core cluster (pink line) and GROMACS on a single core (red line), 12 cores (teal line), and a GPU (cyan).

For the same reasons described for the vacuum simulations, we were unable to simulate the Tumor Suppressor Complex using NAMD on the 24-core cluster and
As GROMACS and NAMD have different methods for preparing the systems, the number of atoms in explicitly-solvated versions of the molecules differs between the systems prepared for GROMACS and NAMD. Nonetheless, the number of atoms is similar enough that a meaningful comparison is possible.

Within a software family (NAMD or GROMACS), architecture seems to be the dominant factor like with the vacuum systems. The 12-core instances outperform their single-core counterpart by an order of magnitude, which correlates with the increase in cores. Although GROMACS GPU does not show an order of magnitude improvement in performance over 12-core GROMACS, the improvement is still significant. NAMD, however, does not show the same significant increase that occurs when going to a single core to 12 cores when going from 12 cores to 24-core cluster.

When simulating explicit solvent systems, NAMD outperforms GROMACS on similar hardware, unlike the results we obtained for the systems in vacuum. Single-
core NAMD outperforms single-core GROMACS, and 12-core NAMD outperforms 12-core GROMACS. As the system size increases, 12-core and 12-core cluster NAMD outperform GROMACS GPU. In terms of differences between the software packages, the most significant result is that single core NAMD outperforms 12-core GROMACS on larger system sizes. For the explicitly solvated systems, the choice of implementation of the software shows almost as much influence on the performance as the architecture, unlike in the vacuum systems.

It is interesting to note the scaling of the software packages with respect to system size. NAMD’s performance seems to decrease more slowly than GROMACS’ performance as the system size increases meaning that NAMD has better scaling behavior with respect to system size than GROMACS. The scaling trends of NAMD and GROMACS are irrespective of the architecture. The general scaling trends for NAMD and GROMACS on explicitly-solvated systems agree with the general scaling trends for NAMD and GROMACS observed on vacuum systems.
Figure 4.4 details the results of running implicitly-solvated systems with GROMACS on a single core (blue line), 12 cores (green line), and a GPU (red line). We were unable to simulate the Tumor Suppressor Complex using implicit solvent as the software for preparing the systems with implicit solvent was unable to handle a system with so many atoms. Also, we were only able to perform implicit solvent simulations with GROMACS. NAMD only has experimental support for implicit solvent at the moment, which we had difficulty getting to run.

With implicitly solvated systems, GROMACS on the GPU outperforms single and 12 core versions of GROMACS. When going from the smallest system (22 atoms) to the next biggest system (542 atoms), GROMACS on the GPU shows the best scaling, the 12-core version of GROMACS shows the second-best scaling, and the single-core version of GROMACS shows the worst scaling. When going from the 542-atom system to larger systems, the 12-core and single-core versions of GROMACS show the same scaling with a constant difference in their performance. The scaling of the GPU version of GROMACS gets worse with every increase in system size. Between the system with about 5,000 atoms and the system with about 10,000 atoms, the performance is scaling is worst. The GPU version of GROMACS seems to scale much worse than sub-linearly.

4.3.2 GROMACS-OpenMM Parameter Sweep

We did a parameter sweep over the configuration parameters for GROMACS-OpenMM to determine how various parameters affected performance.

The first parameter sweep compared the effect of using cutoffs. As GROMACS-OpenMM only supports cutoffs for implicitly-solvated systems, we used implicitly-solvated versions of the Fip35 and DHFRTS systems. We tested combinations of the following parameters:

- neighbor-list update frequency
- once
- every timestep
- every 10 timesteps

- update method
  - simple
  - grid

- cutoff
  - none
  - 0.2 nm
  - 2 nm
  - 999 nm

Our results show that enabling cutoffs hurt performance of the Fip35 simulation, but improved performance of the DHFRTS simulation. For DHFRTS, we saw a performance increase from 3.5 ns/day to 13.8 ns/day with a speedup of 3.9, while Fip35 showed a decrease in performance from 312 ns/day to 297 ns/day with a slowdown of 1.1. We saw no difference in performance beyond enabling cutoffs for different combinations of cutoff distances, update methods, and neighbor-list update frequencies.

We also invested the effect of using a different integrator and enabling constraints on the distance of bonds. The results from running experiments with different integrators are shown in Table 4.3.

For Fip35 in implicit solvent, the performance increase was less substantial going from 312 ns/day to 334 ns/day (speedup of 1.1). Implicitly solvated DHFRTS showed no change in performance with the Velocity Verlet integrator.

Enabling constraints decreased performance. The results are in Table 4.4.

Fip35 with implicit solvent went from 312 ns/day to 173 ns/day (slowdown of 1.8) and Fip35 with vacuum went from 747 ns/day to 255 ns/day (slowdown of 2.3). The
### TABLE 4.3

**INTEGRATOR PERFORMANCE WITH GROMACS-OPENMM**

<table>
<thead>
<tr>
<th>System</th>
<th>Solvent</th>
<th>Integrator</th>
<th>ns/day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fip35</td>
<td>Implicit</td>
<td>Langevin</td>
<td>312</td>
</tr>
<tr>
<td>Fip35</td>
<td>Implicit</td>
<td>Velocity</td>
<td>334</td>
</tr>
<tr>
<td>Fip35</td>
<td>Vacuum</td>
<td>Langevin</td>
<td>747</td>
</tr>
<tr>
<td>Fip35</td>
<td>Vacuum</td>
<td>Velocity</td>
<td>894</td>
</tr>
<tr>
<td>DHFRTS</td>
<td>Implicit</td>
<td>Langevin</td>
<td>3.5</td>
</tr>
<tr>
<td>DHFRTS</td>
<td>Implicit</td>
<td>Velocity</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Results of integrator on performance with GROMACS-OpenMM on the GPU.

### TABLE 4.4

**CONSTRAINT PERFORMANCE WITH GROMACS-OPENMM**

<table>
<thead>
<tr>
<th>System</th>
<th>Solvent</th>
<th>w/ Constr. (ns/day)</th>
<th>w/o Constr. (ns/day)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fip35</td>
<td>Implicit</td>
<td>173</td>
<td>312</td>
</tr>
<tr>
<td>Fip35</td>
<td>Vacuum</td>
<td>255</td>
<td>747</td>
</tr>
<tr>
<td>DHFRTS</td>
<td>Implicit</td>
<td>3.4</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Results of constraints on performance with GROMACS-OpenMM on the GPU.
TABLE 4.5
WRITE-BACK PERFORMANCE OF GROMACS-OPENMM

<table>
<thead>
<tr>
<th>System</th>
<th>Solvent</th>
<th>Write Out Frequency (steps)</th>
<th>ns/day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fip35</td>
<td>Implicit</td>
<td>1</td>
<td>39.8</td>
</tr>
<tr>
<td>Fip35</td>
<td>Implicit</td>
<td>10</td>
<td>196.7</td>
</tr>
<tr>
<td>Fip35</td>
<td>Implicit</td>
<td>100</td>
<td>291.6</td>
</tr>
<tr>
<td>Fip35</td>
<td>Implicit</td>
<td>1000</td>
<td>307.3</td>
</tr>
<tr>
<td>Fip35</td>
<td>Implicit</td>
<td>10000</td>
<td>312.0</td>
</tr>
<tr>
<td>DHFRTS</td>
<td>Implicit</td>
<td>1</td>
<td>3.5</td>
</tr>
<tr>
<td>DHFRTS</td>
<td>Implicit</td>
<td>10</td>
<td>3.5</td>
</tr>
<tr>
<td>DHFRTS</td>
<td>Implicit</td>
<td>100</td>
<td>3.5</td>
</tr>
<tr>
<td>DHFRTS</td>
<td>Implicit</td>
<td>1000</td>
<td>3.5</td>
</tr>
<tr>
<td>DHFRTS</td>
<td>Implicit</td>
<td>10000</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Results of the frequency at which data is written out on performance with GROMACS-OpenMM on the GPU change in performance for DHFRTS with implicit solvent when enabling constraints was negligible: going from 3.5 ns/day to 3.4 ns/day.

Lastly, we compared the effect of how often data is written to disk on the performance of Fip35 and DHFRTS simulations. The results are in Table 4.5 in the Appendix.

Changing the frequency at which data was written out had no effect on the performance of the DHFRTS simulations. However, the more often data was written out for the Fip35 simulation, the lower the performance.

The overarching pattern of the parameter sweep experiments was that the effect resulting from using different parameters affected smaller systems more than bigger
systems. This is most likely because the percentage of time spent on force calculations increases with the number of atoms, so performance degradation in other parts of the simulation represent a smaller portion of the overall time, leading to less of an effect on overall performance.

4.3.3 Microbenchmarks

To help test the architectural differences between a CPU and a GPU we wrote and executed a set of microbenchmarks intended to test specific architectural features between the two systems. We used these results to help understand some of the performance discrepancies between GPU and CPU results.

We first tested the impact that branching has on performance. The rational for this is that using cutoffs – which can cause branching in the inner loops of MD code – may negatively impact performance. GROMACS implements cutoffs by partitioning the system into bins whose atoms belong to the same neighborhood – a set of “neighbor-lists”. By setting cutoffs to update these neighbor-lists (since atoms move in the simulation) every step we expected to see worse performance when compared to updating every 10 steps or only once. This was our initial hypothesis due to the way GPUs handle branching: all sides are evaluated because the threads within a warp need to execute the same instruction.

Another factor that would impact the performance of simulations is the write-back frequency. Molecular dynamics would be useless if we could not get a view of the system as it changes in time. This is accomplished by saving the positions of each atom every X steps corresponding to an intrinsic timescale (eg 50 ps of 500 ps). The result is a sequential ordering of positions corresponding to snapshots taken according to this timescale, such as every 50 ps. For systems running on the GPU, this requires sending the data back to the system memory Since accessing main memory from a GPU may be a bottle neck and since MD is used on a range of systems sizes,
we investigated the impact of write-back frequency against system size (number of atoms).

A third factor we considered is the effect of the parallel architecture of a GPU. A GPU is comprised of many simple processors. Therefore, a GPU should be optimally suited for problems that expose a high degree of inherent parallelism. In order to get a sense of the performance enhancement attained due to exploiting this parallelism we forced the GPU to run in a single-threaded manner.

4.3.3.1 Branching: Single-sided

The single sided branch was written in an attempt to determine if performance could be improved by reducing the workload on the systems. The results are shown in Figure 4.5 where the CPU data is shown in red and the GPU data is shown in green. The horizontal axis represents the size of the system as the number of atoms and the vertical axis is the speedup obtained by single-sided branching. The CPU speedup dominates that of the GPU, which showed no speedup. This microbenchmark indicates that using branching to cull that amount of data and computational time can provide a significant speedup for the CPU. However, there is no performance gain for the GPU.

4.3.3.2 Branching: Double-sided

The double sided branch test was written to help determine how performance is affected on a GPU due to the branch calculation issues. The results are shown in Figure 4.6 where the CPU data is shown in red and the GPU data is shown in green. The horizontal axis represents the size of the system as the number of atoms and the vertical axis is the slowdown as a result of double-sided branching. These data show that double-sided branching has only a negligible affect on the CPU version of the benchmark while there is a significant slowdown on the GPU as the system size
Figure 4.5. Speedup of single-sided branching microbenchmark. The GPU showed no speedup.

increases.

4.3.3.3 Write-back frequency

One of the big factors affecting the results of our GPU simulations was the write back performance. The results are shown in Figure 4.7 where we compare how different write-back frequencies can slowdown the computation for various system sizes. The horizontal axis represents the size of the system as the number of atoms and the vertical axis is the slowdown due to a given write-back frequency for a given system size. The data show that for small systems the write-back frequency can have a large negative impact on the performance due to the overhead of transferring data between off the GPU. However, as the number of atoms increases, the write-back frequency becomes negligible. These data are related to the results from the parameter sweep: changing the write-back frequency for the DHFR-TS simulations has smaller impact
than changing the frequency for Fip35 (a smaller system).

4.3.3.4 GPU Single Thread

In order to see how poorly a GPU would run if restricted to the same model as a CPU we forced it to execute a task on a single core. This was accomplished by using one warp for which a single thread was allocated. By comparing the same task executing on a single GPU core an a CPU we are able to compare the effects of pipelining, branch prediction, and other extensive enhancements of modern CPUs to the simpler GPU core. The results show that the GPU executes instruction on the order of $10^5$ times slower than the CPU.
4.4 Discussion and Conclusion

From our experiments, we have been able to make several conclusions. We have identified several key scenarios and which combinations of architecture and MD package perform best for those scenarios. We have also identified several key factors that could influence performance of code running on the GPU and the choice of algorithms.

Through our experiments, we have a better understanding of how the GPU can benefit MD simulations and under which situations. Our results show that MD simulations run on the GPU can perform better than or equal to simulations run on 12 CPU cores, regardless of the solvation methods used. However, we found that the GPU had scalability problems with systems that have more than 10,000 atoms. In simulations of more than 10,000 atoms, our results show that NAMD running on 12 or more cores will perform and scale better.

Our results for implicit solvent agree with benchmarks done by the GROMACS
developers, but our results for explicit solvent disagree to some extent [3]. The developers initially directly implemented GPU support in an early version of GROMACS as a proof of concept. The decision was later made to utilize the OpenMM library rather than directly supporting GPUs. The GROMACS developers ran benchmarks using DHFR, a protein with about 2,500 atoms, using implicit solvent (with and without cutoffs) and explicit solvent. The GROMACS developers showed a significant speed up when using implicit solvent but that the GPU performance was on par with 8-core CPU performance for an explicitly-solvated system of about 23,500 atoms. We showed a 5 to 10 times speed up between GROMACS on the GPU and GROMACS on a 12-core architecture where their results show parity between the GPU and 8-core CPU.

A key difference between their work and ours stems from the choice of parameters. The GROMACS developers used periodic boundary conditions and Particle Mesh Ewald (PME) for the calculation of long-range electrostatic forces. Periodic boundary conditions make it seem like the system is “mirrored” infinitely in every direction so that if an atom moves out of the region on the right side, it will re-appear on the left side of the simulation box. This better mimics the physics of a system in nature where there is significantly more solvent than what is used in a simulation. PME uses Fourier Analysis and a Fast Fourier Transform to calculate the effect of the forces from an “infinitely repeating” set of atoms by exploiting the periodic placement of the atoms and the convergence of the sums due to diminishing forces as the distance increases. A side effect of using PME on the CPU is that it allows the electrostatic forces to be calculated in $O(N \log N)$ time instead of $O(N^2)$ time. However, the GROMACS developers have stated that they have had difficulty parallelizing and adapting the method to GPUs in a way that would allow for significant speed ups. It is likely that if we had time to run simulations with explicit solvent, periodic boundary conditions, and PME we would have gotten similar results.
Our results do agree with a paper published by the OpenMM developers [48]. The OpenMM developers performed benchmarks indicating speed ups of up to 735 times over a single-core CPU when simulating an implicitly-solvated system of about 5,000 atoms when using a GPU. Our results show a speed up of a little less than 1,000 times over a CPU when using a GPU on implicitly-solvated systems about 5,000 atoms.

The microbenchmarks shed insight into two areas: MD algorithm development and MD usage. Since branching can significantly reduce performance on the GPU fast algorithms that have been developed and refined for the CPU may not be immediately transferred to the GPU. Since GPU warps are a collection of threads executing the same task, a warp must execute all sides of a branch if one is encountered. In order to maintain high performance, MD code may need to remove layers of algorithmic complexity in order to expose a simpler solution that has a higher degree of parallelism that the GPU can take advantage of. This concept mirrors the architecture of the GPU itself when compared to modern CPUs: many small, simple processors versus few, but highly complex processors. Secondly, researchers using MD on the GPU need to consider the size of their system when setting up their simulations. The microbenchmarks explain the result seen with the parameter sweep: the write-back frequency has a greater impact for smaller systems than larger ones. For a small system the computational cost is low, therefore the bandwidth transfer data back becomes the bottleneck. However, on a larger system, as the computational cost increases, the cost of the data transfer is no longer the bottleneck. The size of the system is still limited by the memory available on the GPU. This constrains the size of systems that can be run on GPU.

It was also interesting to note the trade off between scalability and performance between GROMACS and NAMD. On the same architecture, GROMACS had higher performance than NAMD for systems with fewer than 10,000 atoms. As the number of atoms increased beyond 10,000 atoms, the performance of GROMACS decreased
significantly, showing that GROMACS’ higher performance comes at the cost of scalability. NAMD, on the other hand, did not perform as well as GROMACS for systems smaller than 10,000 atoms, but NAMD did scale significantly better for larger systems and outperformed GROMACS in those cases. Thus, it is clear how GROMACS and NAMD have been tuned for their particular intended usages and how that affects their performance.

Due to time and resource restrictions, we ended up having a few inconsistencies in some of our simulations and were not able to explore all combinations of interest. For example, we used 12nm cutoffs and the Velocity Verlet integrator for the NAMD simulations, but we used not cutoffs and the Langevin Leapfrog integrator for the GROMACS simulations. The inconsistencies with the cutoffs resulted because we found that we were not able to disable cutoffs in NAMD and were unable to resolve issues with simulation stability in GROMACS with the cutoffs in the allotted time. The difference in integrator was a result of miss-communication between group members and not having enough time to re-run the simulations. In addition, we had problems with the parameters for the cutoffs in GROMACS on GPU in that beyond enabling cutoffs, none of the parameters seemed to affect the performance. We believe there might have been a problem such that the parameters weren’t being corrected passed to the OpenMM library. We would like to re-run these simulations so that they use the same integrators and cutoff settings and resolve the issues with the cutoff settings on the GPU.

As noted above, the configuration of our explicitly-solvated benchmarks differed from the GPU benchmarks done by the GROMACS developers. The GROMACS developers used periodic boundary conditions and PME to compute the long-range electrostatics, where we did not. This is another problem we realized when it was too late and were unable to re-run the simulations. As the set up used by the GROMACS developers is a more commonly used and more accurate type of simulation,
the results would be of greater interest to the MD community. In addition, the GROMACS developers indicated difficulties parallelizing PME on the GPU and obtaining significant speed up for explicitly-solvated systems that matched the speeds up found for implicitly-solvated and vacuum systems.

Our set up for the cluster version of NAMD involved two 12-core nodes. This set up is unrealistically small and the small number of nodes severely limits the potential impact in performance caused by communication latencies. We were unable to run other cluster configurations due to limited availability of machine configurations on the CRC SGE system due to high load. We were also unable to run the Tumor Suppressor Complex on the cluster configuration since we added the system at the last minute to clarify scaling trends for NAMD and GROMACS and were not able to get a job in our queue running. We would like to run more simulations that explore the effect of the cluster configurations, especially with large molecular systems.

Lastly, we were not able to delve as deep into the implementation of common algorithms as we had liked. Our group members have experience with the implementation of methods in ProtoMol (our lab’s MD software) and OpenMM (our collaborator’s software), but ProtoMol is not highly parallelized nor does it support clusters. It became clear from our studies that the branching on the GPU did not affect the performance of cutoffs as much as we had anticipated since it only affected the much smaller Fip35 system but not the larger DHFRTS system. The same with the write back tests. We found later that the main algorithm that is difficult to parallelize is PME and would like to further explore why this algorithm is difficult to parallelize on the GPU.

In conclusion, we found that a combination of the architecture, MD software package, size of the molecular system, and the type of solvation produced clear trends for performance. When simulating systems with fewer than 10,000 atoms, GROMACS on the GPU seems to be a clear choice. If a GPU is not available or when simulating
systems of more than 10,000 atoms, NAMD seems to be the best choice, as long as it supports your solvation method. For the CPU architectures, there were a significant speed up when going from a single core to 12 cores but less of a speed up when going from 12 cores to the 24-core cluster configuration, at least with the systems we were able to test on all of the architectures. Our parameter sweeping showed how cutoffs, choice of integrators, constraints, and frequency of writing data out could all affect performance, especially on smaller systems. Our microbenchmarks agreed with the results of the parameter sweeping. And lastly, we identified areas of future work.
5.1 Introduction

Molecular dynamics (MD) simulations solve Newton’s equations of motion at the atomic scale for molecules and proteins. These simulations are used by biologists and chemists to study the dynamical behavior of their system(s) of interest. Applications of MD include “pure” science topics such as investigation of the effects of mutations on protein folding as well as “applied” work such as drug development. A significant challenge, however, is the large gap between the time that can be simulated and the real time required: “fast” behavior occurs in microsecond ($10^{-6}$ s) timescales, whereas more typical processes occur in milliseconds ($10^{-3}$ s), and slower motions are in the range of seconds. The size of the biological systems being simulated also adversely impacts the time required to access relevant timescales. Because MD usually only capture nanoseconds ($10^{-9}$ s) of simulated time per day it is infeasible for a single simulation to gather enough data reflecting the aggregate behavior of the system.

One solution is to use an ensemble approach: many short simulations are run in parallel to capture aggregate behavior \[109, 110\]. Multiple independent simulations are started from the initial conditions (a folded or extended conformation, temperature, pressure, solvent viscosity, etc); Due to the stochastic nature of the simulations, the same initial conditions produce different outcomes – thus allowing greater sampling of dynamical space. These replicated simulations (called “trajectories”) are independent, therefore the task of parallelizing the simulations becomes an “embarrassingly parallel” one. The simulation time of trajectories – typically microseconds
Figure 5.1. Folding@Work data model. Experimental conditions have multiple independent clones (to facilitate adequate sampling). The clones’ trajectories (which run for microseconds) are divided into serially dependent (within a clone) generations that are independent across different clones.

- are too long to run at once and so are broken into shorter sequential pieces called “generations” that only require hours or days of real time computation. The payload of required parameters to run a generation is called a “workunit.” In order to generate a coherent model from all these data, statistics can be computed over the timescales of interest by “stitching” the individual trajectories together through common states [35, 23].

In this paper, we describe, benchmark, and mathematically model the scaling behavior of Folding@Work, software developed by our lab to simplify running multiple simulations on distributed computing environments. In summary, our results are:

- Folding@Work shows linear weak scaling
- completion times exhibit a tri-modal distribution
- wait times are indicative of server load
- development of a performance model
- the model accurately predicts poor performance
Figure 5.2. Folding@Work workflow. The server runs in an event loop waiting for results from workers. The server processes the results and submits them. Workers receive workunits, execute them, return the results, and repeat.

- Folding@Work exhibits “self-correcting” behavior
- predictably small task around and transfer times
- the resolution of a scaling issue limiting workers
- identifications of “lazy” results

The paper is organized as follows: In Section 5.2, we describe Folding@Work, its data model and compare Folding@Work to similar endeavors. Section 5.3 describes the benchmarking procedures, the metrics that were used, and how the scaling behavior model was derived. Our results are described in Section 5.4. We discuss potential improvements to Folding@Work in Section 5.5. Lastly, we conclude the paper in Section 5.6.

5.2 Related Work

Folding@Work is built on top of WorkQueue, a library developed by the Cooperative Computing Lab at the University of Notre Dame. The goal of WorkQueue is to facilitate development of distributed computing applications [127]. WorkQueue
Figure 5.3. WorkQueue architecture and resource usage. An application using the WorkQueue library is able to view a potentially heterogeneous collection of resources as a unified pool of workers [31].

provides a Master/Worker paradigm for distributed jobs using a fork/join model. The Master manages workers, programs that receive and execute workunits and then return results. Workers are submitted to various systems, such as Condor, the Sun Grid Engine (SGE), and/or cloud services such as Amazon EC2 or Windows Azure. Rajan, et al. use WorkQueue to convert an MPI-based program to WorkQueue [100]. Bui, et al. explore the use of Python bindings for WorkQueue for running high temperature sampling using MD as well as implicit solvent simulations [31]. Lanc, et al. provide a case study for use WorkQueue for Bioinformatics applications [66]. Folding@Work builds on top of the Python bindings to the WorkQueue API to provide a mechanism for running stochastic MD simulations as illustrated in Figure 5.1.

Folding@Work organizes the ensemble of related simulations into a “project” which is managed by an instance of the Folding@Work process (also called a WorkQueue “server” or “master”). To configure a project the researcher needs to provide details for generating the initial conditions, a simulation software executable, and a mechanism for preparing subsequent generations previous ones.

Figure 5.1 illustrates the Folding@Work data model. A “run” consists of simulations (called “clones”) with the same experimental conditions (e.g., structure, temperature, viscosity, etc.) but different initial conditions (e.g., randomly-generated
velocities). Due to the stochastic nature of simulations, a higher number of clones implies greater sampling of the dynamical space at the cost of greater computational resources. Clones are divided into sequential units called “generations” which make up a single “trajectory.” Clones (and thus runs) are independent of each other while subsequent generations for an individual clone depend on the completion of previous generations.

The Folding@Work process assigns workunits to workers, continually collects the results, and generates further workunits. This process ends when the researcher shuts down the server or all simulations are completed. Figure 5.2 illustrates the Folding@Work workflow between the Master process and a connected Worker. The server initially creates then queues a task. Once a worker becomes available, this task, along with required files, are transferred to the worker. The worker executes the task then notifies the server. Once the server is available, it retrieves then processes the produced files. This processing typically creates new tasks, and the cycle starts over. While a project may generate $T_{total} \leq runs \times clones \times generations$ tasks during it’s lifetime, at most $T_{running} \leq runs \times clones$ tasks will be running at a given time.

As mentioned earlier, the abstractions provided by WorkQueue allow Folding@Work to utilize workers running on heterogeneous resources, such as SGE, Condor, or cloud services (see Figure 5.3).

Much of the initial work in adapting MD simulations to distributed computing environments has been done by the Pande group at Stanford University. Pande et al. have developed Folding@Home [109], a software and crowd-sourcing platform for MD simulations. Folding@Home has been used to investigate many properties of protein folding [113, 114, 36, 101, 48, 119, 22, 26]. Volunteers download and allow the Folding@Home clients to run MD simulations on their computers during idle periods. When the core client is run, it downloads a workunit and program for executing the simulation (a “core”). Once the simulation is complete, the results are returned to
the assigning server. To encourage participation, volunteers are awarded points for successfully completing workunits. The value of these points is determined by the computational resources required: large systems are worth more than smaller ones. To prevent volunteers from biasing the projects they run, the assignment of workunits cannot be controlled.

In order to take advantage of the resources provided by Folding@Home, researchers need to accomplish three goals. First, the researchers must adapt their MD simulation software to function as cores integrated with the Folding@Home client software. In order to prevent problems, these cores must be very stable on a broad number of platforms. Likewise, to ensure data integrity, and protect the volunteers' computers, cores must be adapted to pass all I/O and system calls through Folding@Home. Secondly, researchers must set up and manage servers for Folding@Home, one for assigning new workunits and one for collecting them. Thirdly, all these resources need to be continuously monitored and upgraded.

Although Folding@Home provides extensive computational resources to users (thanks to the volunteers) for little financial cost, it has significant cost in terms of time and management overhead for the researchers. Due to the stringent requirements for cores, Folding@Home does not lend itself to rapid prototyping and investigation of new methods. Routine management of the system requires a nontrivial amount of time on the part of the maintainers. Starting new projects a time-intensive task, making it hard to adapt to changes in research. Finally, the researchers need to invest time in order to provide customer service to the volunteers lest they become discouraged and irate.

Our software, Folding@Work, offers several advantages over Folding@Home. First, our software requires less procedural overhead with greater flexibility. Secondly, the Folding@Work software is simple and easily modified, allowing it to adapt research goals. Third, Folding@Work is intended to be run in mostly trusted environments
(in contrast to crowd sourcing). Fourth, Folding@Work is designed to use resources dedicated to research rather than relying on volunteers’ computers. Finally, the use of WorkQueue allows the use of heterogeneous resources.

Similar work includes Copernicus [98], platform for running MD and dynamically re-allocating simulations to improve sampling. Copernicus uses a master-worker model where the master process continually performs analyses on the incoming data in order to dynamically assign tasks to explore undersampled regions and skip oversampled regions of dynamical space. WorkQueue has also been used for performing replica exchange MD (unpublished, accepted at CloudCom2011). Replica exchange is method for achieving greater sampling by intermittently swapping configurations between multiple simulations running at different temperatures. As replica exchange can adhere to a simple fork/join model, Rajan, et al. take advantage of the abilities provided by WorkQueue.

5.3 Methods

Folding@Work was instrumented to enable the recording of statistics for evaluating performance and identifying bottlenecks. A performance model which relates the number of concurrent workunits to the recorded statistics was developed. Lastly, two weak-scaling benchmarks, one resembling a realistic workload and one designed to illicit poor performance, were run.

5.3.1 Instrumentation of Folding@Work

5.3.1.1 Amount of Time Simulated Per Day of Run Time (ns/day)

The number of nanoseconds simulated per day of run time is a commonly used performance metric in the field of Molecular Dynamics. We measured the run time as the elapsed time from when Folding@Work began execution to when the last
workunit was received before Folding@Work was terminated. The simulated time is determined as the sum of time simulated by all of the workunits. The metric is then defined as the simulated time divided by the run time.

5.3.1.2 Wait Time

After a worker completes a workunit, the master may not yet be able to process the result. While the master is dealing with other tasks, the result will sit in a queue. The time a worker spends in this queue until it is processed is the “wait time.” This is used to indirectly measure the load on the Folding@Work master.

When a worker finishes a workunit, the worker is added to a results queue to wait until the master is able to retrieve the results. The Folding@Work master operates on an event-based model where each event is processed serially. Handling each result incurs an overhead. Thus, if the master cannot process results as quickly as they are received, the queue will start to fill up and workers may need to wait a significant amount of time before their results are retrieved.

The worker life cycle consists of the following stages:

1. Wait to be assigned workunit
2. Receive workunit files from master
3. Execute workunit
4. Wait to be acknowledged by master
5. Send results to master

The WorkQueue library reports the elapsed time between the time that the worker disconnects from the master to begin executing the task until the time the master pops the worker from the results queue and begins retrieving the results as the worker time \( t_{\text{worker,time}} \). We define the task execution time \( t_{\text{exec}} \) as the amount of time that
the worker spends executing the workunit. This does not include the times to transfer files to and from the worker $t_{\text{trans}}$. We can compute wait time as the worker time minus the execution time of the workunit (see Equation 5.1). (As the workunit files are transferred before the worker is commanded to execute the workunit and result files are downloaded after acknowledging the worker has results, transfer times are not included in the elapsed time.) The workunit execution time is computed from the workunit logs as the sum of the elapsed time to download the simulation software and execution time reported in the simulation software’s logs.

$$t_{\text{wait time}} = t_{\text{worker time}}$$

$$- t_{\text{download ProtoMol}} - t_{\text{work unit execution}}$$

(5.1)

5.3.1.3 Number of Workers

The workload managed by Folding@Work is embarrassingly parallel with respect to the simulations in that each simulation is independent of the rest. Thus, in ideal conditions, the workload should have perfect weak scaling such that Folding@Work can utilize as many workers as there are simulations with very little overhead. However, we may not be able to get as many workers as would be needed to fully parallelize the work load. First, there may not be enough resources available. Second, even if a user is able to submit enough jobs to start the desired number of workers, not all nodes may have the required hardware or software to support the simulation software. This may result in the number of usable workers being less than the number of actual workers. Third, when using shared distributed systems such as Condor that construct a computational grid from both dedicated and idle machines, jobs tend to be rescheduled to redistribute resources among users or when the owners of the machines begin actively using the machines, causing the number of workers to
vary over time. Finally, nodes on distributed systems such as Condor are not always reliable – tasks may be assigned but never complete. Therefore, knowing the actual number of workers available (versus the number of workers requested) over time is important for computing scaling and other statistics as well as determining whether performance issues are due to bottlenecks in Folding@Work or not having enough workers. To capture this, Folding@Work stores the number of connected workers every 10 seconds when the master is waiting.

5.3.1.4 Completion Time

The resources available through systems such as Condor are heterogeneous. Differences in hardware can effect execution time of the workunits and thus effect the efficiency of the Folding@Work workload. In order to measure this effect we recorded the execution time of each each workunit.

5.3.2 Performance Model

We developed a mathematical model to relate the maximum number of tasks that can be executed, the average amount of time spent executing the workunits, the average amount of time the master spends generating each new task, the average amount of time workers spends waiting to return results, and the average amount of time the master spends waiting for new results to process. To understand how the model was developed, we need to understand the life cycle of the master and workers (see Figure 5.4). We are only interested in the life cycle of the master after it has submitted the initial tasks since this is simplifies the model and consumes a very small portion of the overall execution time. Since the master is serial and processes results on a first-come first-serve basis, the master will receive results, process the results to generate a new workunit, send the new workunit to a worker, and wait to receive more results. (We assume that there are enough workers such that the
Figure 5.4. Example of time spent by workers and master. The life cycles of the workers and master are broken up into receiving workunit files (S), executing workunits (Execute), waiting for results to be acknowledged by the master (Wait), sending result files back to the master (R), and processing results and generating a new workunit (P). The workers are labeled by color, and the master’s current stage is labeled by the color of the associated worker. When both the worker and master are involved in a task such as transferring files, both have the same box.

outgoing queue contains at most one task for a very short amount of time.)

We refer to the time required to receive results and send the new workunit as the transfer time $t_{trans}$, time to generate a new workunit as the turn around time $t_{new\_task}$, and the time the master waits in between receiving results as the master wait time, or $t_{m\_wait}$. We partition the life cycle of the worker as the time to execute the workunit $t_{exec}$, and wait time $t_{w\_wait}$ as the time spent waiting to return results. By determining how many tasks can be turned around by the master before the first task finishes executing, we can determine the maximum number of parallel tasks the master can efficiently support.

When more tasks are running than the master can turn around before the first task finishes, each worker will incur a wait time while waiting for the master to acknowledge and process its results. If fewer tasks are running than the master can turn around before the first task finishes, the master will incur a wait time while waiting for results to process. It should be noted that the wait time for the master and workers are mutually exclusive – if one has a wait time, then the others’ wait
time must be zero. Furthermore, the aggregate amount of time simulated per day is not affected by the master wait time. Therefore it is preferable to incur a wait time on the master that on the worker (which reduces the simulated time per day metric). We divide the average time the workers spend working independently of the master on a task by the average time per task spent on the master and workers transferring files and generating new workunits to get the maximum number of supported tasks (see Equation 5.2).

\[
N_{\text{work\_units}} = \frac{\langle t_{\text{exec}} \rangle + \langle t_{\text{w\_wait}} \rangle}{\langle t_{\text{new\_task}} \rangle + \langle t_{\text{trans}} \rangle + \langle t_{\text{m\_wait}} \rangle}
\]

subject to

\[
\langle t_{\text{w\_wait}} \rangle \langle t_{\text{m\_wait}} \rangle = 0 \]

(5.2)

5.3.3 Benchmarks

We ran two primary sets of benchmarks. In all of the benchmarks, we used the ProtoMol Molecular Dynamics package to simulate WW Fip35, a 35-residue, 544-atom molecular system with the Amber force field and Generalized Born implicit solvent model. Fip35 was chosen because it is small enough that it is tractable while possessing biological characteristics that are of interest in simulation (Fip35 is used as a model to understand formation of β-sheets).

The first set of benchmarks reflect realistic experimental conditions varying only the number of tasks. We benchmarked Folding@Work with 10, 100, and 1,000 trajectories of 10 generations each. The length of generations were set such that a generation took 2.5 hours to run on a single core on stats.crc.nd.edu. For each experiment we submitted an equal number of workers as the tasks available to run concurrently.
The second set of benchmarks were designed specifically to cause scalability issues with Folding@Work. Based the average task transfer and turn around times (about 0.5 s total) computed from the realistic benchmarks, an average workunit execution time of 5 minutes, and no wait times, the performance model in Equation 5.2 predicted an upper bound of 600 concurrent work units could be supported by Folding@Work. A workunit execution time of 5 minutes was chosen to limit the upper bound on the number of concurrent workunits (and hence workers) to an amount that we were reasonably sure we could allocate on Condor. We ran a weak scaling benchmark with 250, 500, 600, and 750 workers and workunits.

5.4 Results

Two weak-scaling benchmarks were performed with Folding@Work. The first benchmark modeled a realistic workload, while the second benchmark was designed to illicit poor performance. Statistics related to overall performance and how time was spent were recorded and analyzed.

5.4.1 Folding@Work Has Perfectly-Linear Weak Scaling Under Realistic Conditions

Our realistic benchmarks of Folding@Work demonstrated perfectly-linear weak scaling up to 1,000 workers, the maximum number of workers which we tested. Folding@Work was able to simulate 23.7, 266.8, and 2569.8 ns per day of run time with 10, 100, and 1,000 workers, respectively (see Table 5.1). The squared correlation coefficient $r^2$ for the number of workers and $ns/day$ is 1.000 (when using four significant figures), indicating perfectly-linear weak scaling (see Figure 5.5).
TABLE 5.1

FOLDING@WORK WEAK SCALING

<table>
<thead>
<tr>
<th>Number of Workers</th>
<th>Average ns/day</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>23.7</td>
</tr>
<tr>
<td>100</td>
<td>266.8</td>
</tr>
<tr>
<td>1,000</td>
<td>2,569.8</td>
</tr>
</tbody>
</table>

Aggregate ns simulated per day of run time ns/day for each run of realistic benchmarks. The aggregate ns/day scales linearly with the number of workers.

Figure 5.5. Weak scaling of Folding@Work in ns/day for the number of workers and clones. Folding@Work demonstrates linear scaling up to 1,000 workers. Three runs were performed with 10, 100, and 1,000 workers and only one trial per run due to time constraints.
5.4.2 Small Wait Times Indicate Low Server Load

The recorded wait times showed higher wait times for a larger number of clones / workers (see Figure 5.6). Generally, higher wait times indicate that the master is experiencing a higher load, however we did not initially account for the time to download ProtoMol, the simulation software. The workers download ProtoMol from an external web server, so the higher wait times seen at the beginning of the run time more likely result from longer download times as the web server is under pressure from workers rather than load on the master.

5.4.3 File Transfers Consume Very Little Time

We observed that the file transfers consume very little time (less than 0.5 s), as shown in Figure 5.7. Transfer times peak past 3 seconds with 1000 tasks due to increases in the number of running workers (Figure 5.9). As the projects progress, the fluctuations in the environment cause some workers to spend more time transferring
5.4.4 Heterogeneous Hardware Affects Completion of Simulations

The reported completion times appear to follow a tri-modal Gaussian distribution with peaks around 2 hours, 2.5 hours, and 3 hours which is skewed towards long completion times (see Figure 5.8). The distribution seems to converge towards a bi-modal Gaussian distribution with modes of approximately 2.5 and 3 hours that is also skewed towards long completion times as the number of workers is increased. Such multi-modal distributions are the result of mixing independent Gaussian distributions.

The presence of multiple modes is indicative of three different classes of hardware available through Condor, the assignment of workers to these machines essentially
Figure 5.8. Histogram of workunit completion times. Completion times for runs with 10 and 100 workers appear to follow a tri-modal Gaussian distribution skewed towards long completion times. The distribution appears to converge to a bi-modal Gaussian distribution with 1,000 workers. The number of modes in the distributions reflect the availability of multiple classes hardware available through Condor.

performs sampling of the available hardware classes, allowing the relative weighting of the three hardware classes to converge towards the actual distribution. The data indicate that a small number of the fastest machines are available while most machines are of the middle type with some of the hardware being of the slowest category.

The skew towards long completion times indicates that workunits are not being completed. The workers to which these workunits were assigned may be running on bad hardware or the Condor software may be misconfigured on these machines.

5.4.5 Limited Availability of Resources (Number of Workers)

We attempted to provide each running instance of Folding@Work with the same number of workers as available tasks to ensure maximum parallelization. We recorded the number of available workers for each running instance as a function of the amount of time that had elapsed since the instance was started.

We had difficulty keeping enough workers running (see Figure 5.9). For both the
Figure 5.9. Number of workers available through Condor as a function of run time. The small dip in the number of connected workers around 15 hours into the run with 1,000 workers was due to Condor’s reassignment of hardware as other users requested computer time. As the number of workunits was increased, the variance in completion time of the trajectories increased as the longer amount of time required for the run with 1,000 workers to kill its workers.

runs of 100 and 1,000 workers, we had a significant number (up to 20%) of the workers die for various reasons and had to submit new workers to replace the dead workers.

As individual simulations finished, fewer workers were needed and Folding@Work started disconnecting workers (see Figure 5.9). We noticed that the effect of the differences in completion time or workunits that were never completed had the most effect in the run with 1,000 workers. Where as the other runs quickly killed off their workers at the ends of the runs, the 1,000 worker run killed off 90% of the workers but was left waiting on the remaining 10% of simulations to finish. The observed behavior of the 1,000-worker run correlates with the observed behavior of the completion times in indicating that excessively long completion times may need to be handled by reassigning workunits.
Figure 5.10. The amount of time simulated per day of run time $ns/day$ (red) from a weak-scaling benchmark designed to cause performance issues with the Folding@Work master plotted against the number of concurrent workers and workunits. Each workunit executed in 5 min. on stats.crc.nd.edu. Folding@Work demonstrates linear weak scaling until reaching peak performance with 600 workers/workunits. Between 600 and 750 workers, the aggregate $ns/day$ achieved decreases significantly from 4162.7 $ns/day$ to 2106.7 $ns/day$. The aggregate $ns/day$ achieved in the realistic benchmarks (described above) were plotted in blue for comparison.

5.4.6 Performance Model Accurately Predicts Poorly Performing Work Loads

The benchmarks designed to illicit poor weak-scaling from Folding@Work demonstrated approximately linear weak scaling in terms of $ns/day$ up to 600 workers with a significant decrease in aggregate $ns/day$ between 600 and 750 workers (see Figure 5.10). Peak performance was observed to occur with 600 clones and workers at 4162.7 $ns/day$.

Wait times for the four runs are shown in Figure 5.11. Longer wait times indicate more load on the Folding@Work server which affect aggregate performance. The runs with 250, 500, and 600 workunits have maximum wait times of 10 seconds with an average wait time of 1 second. The run with 750 workunits initially has wait time similar to the other runs but after running for about 75 minutes, the wait times increase significantly with a maximum wait time closer to 100 seconds and an average wait time of 10 seconds. The high wait times for the run with 750 workunits
Figure 5.11. Wait times for results with 250, 500, 600, and 750 short (5 min.) workunits. The runs with 250, 500, and 600 workunits demonstrate average wait times of around 1 s, while the run with 750 workunits has an average wait time of 10 seconds after about 750 minutes into the run.

are evidence that the Folding@Work server is overloaded and not able to adequately support that many workunits with such a small workunit execution time (5 minutes), which agrees with the observed decrease in \( \text{ns/day} \) (compare Figures 5.10 and 5.11).

The performance model given in Equation 5.2 was used to predict that Folding@Work could support up to 600 concurrent workunits before incurring performance penalties, which agrees with the peak in aggregate \( \text{ns/day} \) achieved with 600 workunits and decreased performance and increased wait times seen with 750 work units. Thus, the results validated the accuracy of the performance model.

5.4.7 Folding@Work Automatically Adjusts Staggering of Workunits via Implicit Feedback Loop

Figure 5.12 shows the number of workunits with results waiting to be processed for the two medium sized projects (with 400 and 600 workers) over the first 90 minutes of run time. Folding@Work attempts to submit initial workunits as quickly as possible rather than staggering submissions. As observed from the periodic increases in the
Figure 5.12. The number of workunits with results waiting to be processed from the first 90 minutes of run time for projects with 400 and 600 concurrent tasks are plotted against time. Each workunit took 5 minutes to execute on stats.crc.nd.edu. As the workunits were submitted at uniformly distributed intervals, workunits tended to return at similar times as evidenced by the increases in the number of queued results. The peak number of waiting results decreases over time as the workunits become increasingly uniformly distributed as a consequence of the single-threaded, first-come, first-serve WorkQueue event loop.
number of queued results with a period congruent with the run time (5 min) of the workunits, the submission policy results in most tasks returning around the same times while fewer tasks are returned in between those times. In contrast, an ideal submission policy would stagger the submission of workunits such that the return times would follow a uniform distribution and thus distribute the load on the master evenly over time to prevent overloading the master.

As the simulation progresses, the number of tasks returning at once decreases, indicating a self-correcting behavior (see Figure 5.12). Since the master can only process a maximum number of results per second and the amount of time to process each result is relatively constant, the creation of new work units are staggered and submitted at equal times apart. As a consequence, workunits return times become more evenly distributed as the simulation progresses. Thus, Folding@Work and the workers form a feedback loop which tends towards stable behavior and could be modeled using control theory. The observed self-correcting behavior may also be considered “emergent” behavior.

5.4.8 Task Turn Around Time is Small and Very Predictable

Task turn around times were observed as being very small on average with very small variances, indicating that the turn around time is very deterministic. The average time required to generate a new workunit from a result after the result has been downloaded is less than 1 second (see Table 5.2). As a corollary, the Folding@Work masters should be able to process at least 60 workunits per second on similar loads.

5.5 Discussion

As we have demonstrated, Folding@Work is able to scale perfectly linearly up to the number of workers we are able to run due to limited resources. Nonetheless, we
TABLE 5.2

FOLDING@WORK TURNAROUND TIME

<table>
<thead>
<tr>
<th>Number of Workers</th>
<th>Average processing time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0.72</td>
</tr>
<tr>
<td>500</td>
<td>0.28</td>
</tr>
<tr>
<td>600</td>
<td>0.28</td>
</tr>
<tr>
<td>700</td>
<td>0.50</td>
</tr>
</tbody>
</table>

Average time to create the next generation.

have identified several issues that warrant future work.

5.5.1 Availability of Resources

In our tests we had difficulty running enough workers to cause scalability problems in Folding@Work under realistic conditions. As noted in Section 5.4.5 nearly 20% of our workers were killed for various reasons while the Folding@Work instances were running. We are working with the WorkQueue maintainers to investigate the use of a new WorkQueue feature called WorkQueue Pools which will automatically start new workers when they are killed. We are looking forward to the future possibility of over-loading the Condor system.

5.5.2 Hard Limit on Number of Workers

Our tests found that Folding@Work has a limit for the number of workers after which it either crashes or ceases to respond to events. We tracked this problem to two causes: (1) the number of file descriptors available per process and (2) the maximum number of connections the data structures associated with the `select()` system call support. We were able to exacerbate the issues by decreasing the file descriptor limit.
When we increased the file descriptor limit, Folding@Work was able to support 2,500 workers (up from 1,000) before experiencing issues.

Further investigation traced the problem to the usage of the `select()` system call. According to the `select()` system call manual page, the maximum number of file descriptors `select()` supports is defined in the system constant `FD_SETSIZE` and using more file descriptors than supported results in undefined behavior [2].

We are working with the WorkQueue developers to investigate the usage of `epoll()` system call, which is designed to efficiently support a large number of file descriptors [1].

5.5.3 Improved Handling of Work Units that are Not Returned

Our results indicate the presence of a phenomenon where workunits are not completing which prevents simulations from finishing. WorkQueue has a feature called fast abort which will reassign workunits that are taking significantly longer than the running average completion time. Based on our results, we will investigate the use of the fast abort option to handle workunits that are not being completed.

5.6 Conclusions

Folding@Work is software used for decomposing Molecular Dynamics trajectories and allowing them to be run on heterogeneous resources by using the master/worker paradigm provided by WorkQueue. This paper describes the instrumentation of Folding@Work in order to measure the scalability of Folding@Work, develop a performance model, and explore issues encountered when running large simulations.

We find that Folding@Work displays linear weak scaling. The performance model provides guidelines for decomposing simulations in order to effectively take advantage of many disparate resources. Progress has been made in finding and resolving limitations with the WorkQueue library that affect Folding@Work. However, we found
that the workload and software architecture of Folding@Work does not introduce significant scaling bottlenecks beyond those present in WorkQueue.
6.1 Introduction

Sampling has a variety of uses. With applications in image processing, computer graphics, to machine learning and analysis of computer-generated simulations, there has been a great deal of work studying various sampling methods, analyzing their properties, and implementations.

Clustering of data according to common features is a ubiquitous problem in data analysis. This problem is exacerbated in scenarios with large datasets.

The specific problem we examine is the development of an online algorithm suitable that can be efficiently incrementally updated with models generated remotely. This consists of examining data that is continuously generated as a stream of points with goal of discovering similarities between groups of points. Since there are no known bounds to the data, the stream may be infinite, it would be impractical to apply a method that expects the full dataset to be present.

An algorithm for clustering streaming data needs to satisfy the following requirements \cite{10,27}:

- Compact representation
- Incremental processing of new data
- Identification of outliers
- Use only single-pass over input
6.2 Background

6.2.1 Motivation

One well-known, fast sampling method is Uniform Random Sampling (URS), where each point is generated from a uniform probability distribution. This is desirable since it is easily implemented, fast, and efficient. However, the choice of points results in regions that are over- and under-sampled, as shown in Fig. 6.1a.

One approach to overcome this limitation is to enforce an invariant on the distance between points when selecting them. A solution of this is known as the Poisson Disc Sampling algorithm (PDS)[28]. Given a given radius $R$, each point is selected such that the distance to its closest neighbor is in the range $[R, 2R]$. The result of running PDS is shown in Fig. 6.1b (using the same number of points as URS in Fig. 6.1a).

This improvement can also be shown by the distribution of distance to the nearest
neighbor. Figures 6.1c and 6.1d show the histograms of distance between each point and its nearest neighbor. The distribution for PDS (Fig. 6.1d) is narrower than that for URS (Fig. 6.1c) indicating a more even spread.

Additionally, PDS can be executed in $O(n)$ time: Bridson has provided a grid-based algorithm, that is scalable to arbitrary dimensionality, for running PDS\cite{28}. In brief, the method proceeds as follows:

1. Define a grid $G$ of $d$ dimensions with cell size $R/\sqrt{d}$
2. Select initial point $x$, insert $x$ into $G$, and add it to an active list $A$ and output $O$.
3. While $A$ is not empty, remove $x_i$ from $A$ where $i$ is random. Choose up to $K$ points $p_j$ s.t. $j \in \{1, \ldots, k\}$ within the annulus between $R$ and $2R$. If $\text{distance}(x_i, p_j) \geq R$ then add $p_j$ to $A$ and $O$, mark the corresponding point in $G$, and add $x_i$ back to $A$.

6.2.2 Poisson Subsampling

While Poisson Disc Sampling can be used for generating points in space, it is not directly applicable to a scenario where points have already been generated according to some arbitrary distribution. Here we discuss the modifications to PDS to accomplish this, and term the result Poisson Subsampling (PS).

Algorithm\cite{1} provides the pseudo-code for the PS method. Given a set of points $S$, a radius $r \in \mathbb{R}$, and threshold $k \in \mathbb{Z}^+$ of the maximum neighbors of each point to search for – where $\text{NEIGHBOR}(p, \mathcal{P}, r)$ returns the set of points in $\mathcal{P}$ that are within an annulus $[r, 2r]$ from $p$ – return a subset $O$ of $S$ such that no two points in $O$ are closer than $r$. The algorithm maintains three sets of points: $A$: “active” or those to consider next, $X$: “available” or those that “may” be considered, and $O$: “output” or those that have been selected and which satisfy the invariants.
Figure 6.1. Figures 6.1a and 6.1b show points sampled using the Uniform Random Sampling (URS) and Poisson Disc (PDS) Sampling methods. URS shows regions of under- and over-sampling while PDS points are more evenly distributed through space. Figures 6.1c and 6.1d show the distribution of the distance between each point and its nearest neighbor. The URS method 6.1c shows a larger range than the PDS 6.1d method.
First the “active” and “available” lists are initialized to a point selected at random from \( \mathcal{X} \), and \( \mathcal{S} \), respectively. The loop consists of considering a point \( x \) by removing it from the “active” list and discarding it from the “available” list. If \( x \) is too close to the “output” list the choose the next point to consider. Otherwise add \( x \) to the “output” list and add up to \( k \) neighbors of \( x \) within an annulus \([r, 2r]\) to the “active” list. If any of these neighbors is too close the those in \( \mathcal{O} \), remove it from the “available” list. Once there are no more points to be considered (\( \mathcal{A} \) and \( \mathcal{X} \) are empty) return the “output” list. Each point is either in the “active”, “available”, or “output” lists and so is considered at most \( kN \) times. Since \( k \) is a constant and neighborhood searching can be done using branching data structures the algorithm can run in \( \mathcal{O}(n \log n) \) time.

6.2.3 Online Poisson Subsampling

The PS algorithm can easily be extended to be an incremental, online method. Algorithm 2 shows the procedure. Given a set of new points \( \mathcal{S} \) and a set of initial points \( \mathcal{C} \), the first step is to assign \( \mathcal{S} \) to \( \mathcal{C} \) and determine which points \( s \in \mathcal{S} \) are further than the radius \( r \) from the points in \( \mathcal{C} \). These points form \( \mathcal{X} \). The purpose of this step is to reduce the search space, since \( \text{POISSON SUBSAMPLE}(\mathcal{C}, r, k) = \mathcal{C} \). Therefore \( \text{POISSON SUBSAMPLE}(\mathcal{X}, r, k) \) will only sample from the ”new” points. The final step merges the subsample new points with the initialized points, ensuring that the invariant on distance between selected points holds.

6.2.4 Application to Distributed Streaming Data

Recall that the requisites for a method application to streaming data are that is posses a compact representation, incrementally incorporate new data without needing to recompute previous result, easily detect outliers, operating on a single pass of the data, provide a “best answer” at any given point during the progress, and suspendable
Algorithm 1 The Poisson Subsampling Algorithm

function POISSON_SUBSAMPLE(\(S, r, k\))
   \(\triangleright \) \(S\): set of points
   \(\triangleright r \in \mathbb{R}\): radius
   \(\triangleright k \in \mathbb{Z}^+:\) number of neighbors to consider

\(x\leftarrow\) random point in \(S\)
\(A\leftarrow\{x\}\) \(\triangleright\) Active list
\(\mathcal{X}\leftarrow S\) \(\triangleright\) Available points
\(\mathcal{O}\leftarrow\emptyset\) \(\triangleright\) Chosen points

while \(A\) or \(\mathcal{X}\) are not empty do
   \(x\leftarrow\) pop next point from \(A\) or \(\mathcal{X}\) \(\triangleright\) Whichever is not empty
   discard \(x\) from \(\mathcal{X}\)
   if \(\exists p \in \mathcal{O} \mid distance(x, p) < r\) then
      continue \(\triangleright\) Ignore \(x\) in the future
   end if
   add \(x\) to \(\mathcal{O}\)
   for \(ki \in \{1, \ldots, k\}\) do
      \(n\leftarrow\) NEIGHBORS(\(\mathcal{X}, x, r, 2r\))
      if \(n\) is NULL then break
      else if \(\exists p \in \mathcal{O} \mid distance(n, p) < r\) then
         discard \(n\) from \(\mathcal{X}\)
      else add \(n\) to \(A\)
      end if
   end for
end while
return \(\mathcal{O}\)
end function

Algorithm 2 Online Poisson Subsampling

function ONLINE_POISSON_SUBSAMPLE(\(S, r, k, \mathcal{C}\))
   \(\triangleright \) \(S\): set of points
   \(\triangleright r \in \mathbb{R}\): radius
   \(\triangleright k \in \mathbb{Z}^+:\) number of neighbors to consider
   \(\triangleright \mathcal{C}\): set of points

\(\mathcal{X}\leftarrow\) points in \(S\)
   s.t \(\forall(x, c) \in \text{CROSS}(\mathcal{X}, \mathcal{C}): distance(x, c) > r\)
   \(\triangleright \mathcal{X}\) is the set of points that have not yet been visited

\(A\leftarrow\) POISSON_SUBSAMPLE(\(\mathcal{X}, r, k\))
\(\mathcal{B}\leftarrow\) append \(A\) to \(S\)
\(\mathcal{O}\leftarrow\) POISSON_SUBSAMPLE(\(\mathcal{B}, r, k\))
return \(\mathcal{O}\)
end function
and thus resumable.

In this context Poisson Subsampling satisfies the requirements for compactness, incremental operation, requiring a single pass, and suspendable/resumable computation. Compactness is achieved by only needing to store a subset of the points. Satisfying the “single pass” over a (potentially infinite) stream of data can be done by applying Online PS to the data, possibly in chunks of at least one element. Considering the online version of PS (Algorithm 2), incremental operation is achieved by only adding points that have not yet been “discovered”. Finally, suspendable computation can be done by storing the current “definitions” (the output of PS) to disk, and resuming by reading the data into memory.

On the other hand, PS does not directly address the question of determining “outliers” or providing a “best answer” as these are more specific to the domain of study. Poisson Subsampling does ensure that potential outliers are not discarded before a more appropriate method can be used to determine their status. Ultimately, PS is intended to be used as part of a data-processing pipeline.

Fig. 6.2 shows a hypothetical workflow for data processing using PS. The stream of data passed through the online PS algorithm which stores the result in persistent form, such as on disk. After each update, the clustering method is triggered which then yields the current “best” result.

6.2.5 Work Queue

Work Queue [86] is a framework for writing scalable distributed applications in the Master/Worker paradigm. An application generates Tasks with are executable units with input and output files. Tasks are then submitted to Work Queue (WQ) which schedules the Tasks on available Workers and materializes the necessary data on the worker. While fault tolerance of the master node needs to be handled by the application, worker failures are managed transparently by rescheduling the failed
Figure 6.2. Potential workflow using Poisson Sampling: Data streams and is passed to Online PS in chunks. The output is stored to disk for suspension/resumption of the process and a signal is sent downstream to Cluster. The cluster then access the model, perhaps from shared disk, and filters and outliers and provides the “answer” to result.

A WQ application can use multiple, otherwise disparate and with different operational semantics, computing systems to increase application performance, as shown in Fig. 6.3. For instance Condor\textsuperscript{[75]} is a cycle-scavenging system that allows underutilized commodity computers, such as those pervasive in an office or university campus, to run jobs. It is important to note that these jobs may be terminated at any point when the Condor system detects that the owner of the node is using it. On the other hand, High Performance Computing (HPC) Grids have the reverse semantics: once a job begins execution it is expected to run indefinitely. Finally, use of cloud infrastructure such as Amazon EC2 may be dependent on budgetary requirements. By using WQ one may run a large distributed application by combining the Condor and HPC resources, adding EC2 once funding is acquired and stopping the EC2 machines once the budget is exhausted.

6.2.6 Adaptive Sampling Molecular Dynamics

Molecular Dynamics (MD) simulations allows molecular motions to be viewed and understood at the atom-level. The general procedure is as follows: given a set of
Figure 6.3. Work Queue distributed applications can use resources running in different environments, such as Condor, Amazon EC2, or HPC grid. Applications are written with a Master submitting tasks to Work Queue (WQ), which materializes them on workers (W). WQ manages data transfer and automatic rescheduling on worker failure.
atomic coordinates and parameters describing the interactions between each atom, compute the force on each atom, increment the timestep, and integrate to determine the new atomic positions. Problems of particular interest are those concerning rates of change between different molecular conformations. These kinetic rates can be used to understand protein folding (the formation of biologically active proteins from inactive ones), the effect of drugs and their efficacy, and the understanding of diseases caused by protein misfolding.

A significant challenge when running MD is the difference between timescales: due to numerical instabilities the timestep is typically one to two femtoseconds ($10^{-15}$) while molecular motions occur in the range of femtosecond to seconds (or even hours for large complex proteins), a range of many orders of magnitude. As a result it may take several days to simulate several nanoseconds of a medium-size protein. Much work has been done recently to overcome this limitation by running multiple simulations in parallel and combining the results to form holistic statistical models of the system.\[109, 110, 88, 111, 117\].

Another way to view the MD process is that of an exploration (or sampling) of state-space (possible molecular conformations). Since each conformations has an associated energy, MD predominantly samples low-energy regions of space. However, since calculation of transition rates are dependent on crossing energy barriers, these transitions are infrequently observed even when running a large number of parallel simulations: the vast number of samples are low-energy conformations. Fig. 6.4 show the sampling of a simple one-dimensional energy surface.

One way to overcome this is to decompose state-space and maintain an equivalent number of (potentially concurrent) simulations in each cell. This method biases the system to reduce the over- and under-sampling of low- and high-energy regions yet allows calculation of accurate kinetic rates and is known as the Weighted Ensemble (WE) approach \[58, 18, 17, 41\]. However, a significant source of error in running
Figure 6.4. Molecular dynamics sampling of a one-dimensional energy surface. In this scenario the rates of conversion between reactant (R) and product (P) would be of interest.

WE is the presence of unaccounted-for regions of molecular state-space. One way to overcome this issue is known as Adaptive Sampling (AS), which biases sampling to find only find “new” conformations.

6.3 Previous Work

One very-well known clustering method is the \( k \)-means algorithm. The operation of \( k \)-means is as follows: Given an initial set of points \( S \), choose a subset \( X \subseteq S \) of \( k \) points at random from \( S \). Next, assign each point in \( S \) to its nearest centroid in \( X \). Define \( X' \) as the new set of points determined by taking the mean of each points in a given cluster. If \( X \) and \( X' \) are different, then update \( X \leftarrow X' \) and repeat, otherwise
return $X'$ as the set of representative for $S$. Since $k$-means needs to visit each point in $S$ multiple times it is not appropriate for streaming data.

The DBSCAN algorithm identifies clusters by determining areas of density [45]. This allows the method to identify clusters that have arbitrary shapes, unlike $k$-means where the clusters must be regular shapes. An advantage of DBSCAN is that clusters are identified automatically and is only dependent on the threshold $\tau$ for determining neighbors and the minimum number of points to consider such a group a “cluster”. On the other hand it is neither online nor incremental. The method proceeds as follows: for each point $p$ in a dataset $S$, let $N$ be the neighborhood of $p$. If $N$ is larger than $\tau$, then expand the cluster, otherwise mark $p$ as “noise”.

The Weighted Ensemble (WE) method was introduce by Huber and Kim [58] to address the problem of capturing infrequent transitions of Brownian dynamics. WE has developed as a way to maintain a fixed number of simulations in a specific decomposition of molecular state-space [58, 18, 17]. While WE has a higher computational cost that traditional molecular dynamics, there have recent studies demonstrating satisfactory scaling in parallel and distributed computing environments. Recently, Abdul-Wahid et al. demonstrated an implementation that scaled to thousands of nodes in a heterogeneous distributed environment [4, 5].

The goal of Adaptive Sampling (AS) in MD is to guide the simulation to explore physically feasible but heretofore – in the span of the running simulation – unseen molecular conformation. Recently, Bowman et al. and described a method using Markov State Models to guide the sampling procedure [25]. Markov State Models have recently gained popularity as a method for developing a holistic statistical model of protein dynamics that are run in parallel to accelerate sampling of state-space [110, 88, 111, 117].
6.4 Experimental

This section describes the experimental analysis of the properties of the Poisson Subsampling (PS) algorithm. First, section 6.4.1 shows that PS captures the shape of data. Second, section 6.4.3 shows the ability of PS to incrementally add new observations to an existing model without having to “relearn” previous data. This also demonstrates the “online” ability when used with a post-processing method and the detection of possible outliers. Finally, section 6.4.4 applies Online PS to a distribution application: finding possible conformations of a biomolecule using molecular dynamics.

6.4.1 Coverage

Our aim here is to show that PS sufficiently recovers the “shape” of an arbitrary dataset. To do so, points were generated to create general shapes using the scikit-learn library for data processing [90]:

- Blobs: Gaussian distribution of points around a given number of centers,
- Circles: two clusters of points forming two circles such that the smaller fits within the larger,
- Moons: two clusters of interleaving semicircles.

We define the “coverage” of a subsampling as the normalized maximum distance of any point in the original dataset to its nearest neighbor in the subsample. Given $\mathcal{X}$ and $\mathcal{S}$, a set of data and its subsampled subset respectively, let $d$ be the distance between each point in $\mathcal{X}$ and its nearest neighbor in $\mathcal{S}$ and $D$ be the distance between each pair of points in $\mathcal{X}$, we can compute the coverage score $C_s(\mathcal{X}, \mathcal{S})$ as the
normalized maximum of $d$:

$$\begin{align*}
  &\text{Given : } \mathcal{X} \text{ and } \mathcal{S} \\
  &\text{let : } d = \text{dist}(\mathcal{X}, \mathcal{S}) \\
  &D = \text{dist}(\mathcal{X}, \mathcal{X}) \\
  &\text{in } Cs(\mathcal{X}, \mathcal{S}) = \frac{\max(d) - \min(D)}{\max(D) - \min(D)}
\end{align*}$$

Note that smaller values indicate a more evenly distributed subsample of the data.

Our initial test compared the improved coverage of PS on the Blobs, Circles, and Moons datasets shown in Fig. 6.5a. The reference dataset is shown in the left column while the middle and right columns show the subsampled points along with the respective coverage scores. In each case the coverage score is improve by nearly a factor of two.

Next we used the Boston, Diabetes, Digits, Iris, and Linnrud datasets from the UCI Machine Learning Repository [9]. Each dataset was subsampled with PS and the coverage score computed. Table 6.1 summarizes the properties of the datasets, number of features and size, the size of the subsample and the percent reduction. Additionally, the radius parameter to PS is given, the maximum distance between a point and its assigned centroid, the largest population of a centroid (the minimum is no smaller than 1), and the cover score.

The number of features ranged from 3 to 64 while the smallest and largest datasets had 20 and 1797 points – linnerud and digits – respectively. PS reduced the size of the data from 30% to 60%. Additionally, note that the maximum distance is smaller than the radius, indicating the enforcement of the Poisson Subsampling invariant.
(a) The columns show uniform random subsampling (middle) and poisson subsampling (right) applied to artificial datasets (left).

(b) $\text{k-means}$.

(c) $\text{DBSCAN}$.

(d) $\text{DBSCAN}$.

Figure 6.5. Coverage and clustering of data with Poisson Subsampling.
TABLE 6.1

SUBSAMPLING DATASETS USING POISSON SUBSAMPLING

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nf</th>
<th>Nr</th>
<th>Ns</th>
<th>%r</th>
<th>R</th>
<th>Dist</th>
<th>Px</th>
<th>Cs</th>
</tr>
</thead>
<tbody>
<tr>
<td>boston</td>
<td>13</td>
<td>506</td>
<td>294</td>
<td>58</td>
<td>10.00</td>
<td>6.93</td>
<td>16</td>
<td>0.01</td>
</tr>
<tr>
<td>diabetes</td>
<td>10</td>
<td>442</td>
<td>267</td>
<td>60</td>
<td>0.07</td>
<td>0.06</td>
<td>7</td>
<td>0.06</td>
</tr>
<tr>
<td>digits</td>
<td>64</td>
<td>1797</td>
<td>807</td>
<td>45</td>
<td>20.00</td>
<td>15.81</td>
<td>17</td>
<td>0.15</td>
</tr>
<tr>
<td>iris</td>
<td>4</td>
<td>150</td>
<td>41</td>
<td>27</td>
<td>0.50</td>
<td>0.36</td>
<td>15</td>
<td>0.05</td>
</tr>
<tr>
<td>linnerud</td>
<td>3</td>
<td>20</td>
<td>6</td>
<td>30</td>
<td>45.00</td>
<td>23.03</td>
<td>8</td>
<td>0.06</td>
</tr>
</tbody>
</table>

Subsampling datasets using PS. Nf: number of features; Nr: size of the dataset; Ns: size of the subsampled data; %red.: percent reduction; R: radius; Dist: maximum distance of points from assigned centroid; Px: maximum population of centroids; Cs: coverage score.

6.4.2 Clustering

Next we considered the effect that subsampling has on clustering the datasets. We use the k-means and DBSCAN clustering algorithms.

While k-means is appropriate for the Blobs dataset it is unable to correctly distinguish between the regions of the Circles and Moons datasets. This is due to k-means’ operation of finding the k centroids by iteratively assigning points the nearest center and updating the centers by averaging their assigned points. Ultimately k-means is better suited for identifying even clusters where k is known apriori. As such, a more appropriate method for Circles and Moons is the DBSCAN algorithm which is able to identify an unknown number of clusters which may not be even. Fig. 6.5b, 6.5c, and 6.5d show the result of clustering on the Blobs, Circles, and Moons datasets after they were subsampled using PS. In each case the clustering method was able to accurately identify the clusters.

Additionally, metrics were calculated on the results of clustering the reference
TABLE 6.2

METRICS COMPARING CLUSTERING RESULTS OF SUBSAMPLED DATA

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>Metric</th>
<th>Ref</th>
<th>PS</th>
<th>Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blobs</td>
<td>KMeans</td>
<td>ARi</td>
<td>1.00</td>
<td>0.97</td>
<td>0.03</td>
</tr>
<tr>
<td>Blobs</td>
<td>KMeans</td>
<td>MutInf</td>
<td>1.00</td>
<td>0.97</td>
<td>0.03</td>
</tr>
<tr>
<td>Circles</td>
<td>DBSCAN</td>
<td>ARi</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Circles</td>
<td>DBSCAN</td>
<td>MutInf</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Moons</td>
<td>DBSCAN</td>
<td>ARi</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Moons</td>
<td>DBSCAN</td>
<td>MutInf</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Metrics comparing the results of clustering on the reference (Ref) and subsampled (PS) data. The absolute difference (Diff) between Ref and PS is also shown. **ARi**: Adjusted Rand index; **MutInf**: Mutual Information.

dataset and the subsample and then compared in Table 6.2. The metrics used are the following: The Adjusted Rand index (ARi) measures the similarity between two clusters by comparing pairs of sample in the actual and predicted clusters [60], values near zero indicate random labeling while values near one indicate identical labeling. The (normalized) Mutual Information score (MutInfo) is a measure of similarity between probability distributions normalized between zero and one where zero means no relation while one is identical.

6.4.3 Incremental Addition

Incremental operation of PS was tested by decomposing the Circles dataset as shown in Fig. 6.6. Given the original dataset, each group of points was presented to Online PS. The intermediate model was then used as input to the DBSCAN clustering
Figure 6.6. The Circles dataset decomposed according to the dotted lines. All the points in each partition is presented to Online PS.

the progress after five iterations. At this point clustering finds three clusters and marks a singular point as an outlier. In Fig. 6.7b the final result is shown: after 10 iterations the original dataset is completely subsampled and the clustering algorithm correctly identifies the two clusters.

6.4.4 Adaptive Sampling of Molecular Conformations

In this section, we use the Poisson Subsampling method to run distributed Adaptive Sampling MD simulations and show that the AS finds states that MD did not in an equivalent amount of simulated time.
Figure 6.7. Different stages in running Online PS. In each subfigures, the left shows the points and the iteration at which they were “learned” and the right show the output of clustering.
Figure 6.8. The Workflow for running adaptive sampling molecular dynamics using Poisson Subsampling. The master starts by running PS on a set of input coordinates and broadcasts them to the workers. The workers then run molecular dynamics simulations from these starting conditions. Once the simulations complete, PS is applied to update the local model. These models are then sent back to the master which merges these models using Online PS. Finally, “underpopulated” regions are identified in the model and their coordinates sent to the workers along with the updated model.

The distributed workflow is shown in Fig. 6.8, which was implemented using Work Queue, a framework for writing fault-tolerant distributed applications in the Master/Worker paradigm.

The method works as follows: given an initial set of points, such as the folded conformation, apply PS to determine an initial model $M$. Each worker is assigned a cell in $M$ and receives $M$ along with the molecules coordinates for that cell and runs MD from these positions. After MD, workers use Online PS to generate a local model $m$, which is then sent back to the master. The master then merges, using Online PS, models $(m_0, m_1, \ldots, m_w)$ from each worker to obtain an updated model $M'$. The difference between $M$ and $M'$, $\Delta M$, is the set of new conformations, and only $\Delta M$ is sent back to the workers. The master then analysis $M'$ to determine the states that need further sampling and these conformations are sent to the workers to continue the MD. Additionally, each $\Delta M$ is stored to disk to allow resumption, as well as the analysis of the model by other programs.
In our experiment we applied the PS-enabled AS method to Alanine Dipeptide. The simulation used the Amber03 forcefields with the Tip3p water model at 273K and ran for 191 ns. Using four machines we ran 60 workers as resources and monitored task execution time, Fig. 6.9b, data transfer time, Fig. 6.9c, and the number of workers, Fig. 6.9a. After starting the AS program on the master node we added and removed resources to evaluate the ability of the application to handle these failures. The application completed 1,1046 tasks in slightly over 4 hours. The machines running the simulations were

- 2.5 Ghz Intel Xeon E5-2640
- 2.0 Ghz Intel Xeon E7540
- 2X 2.4 Ghz Intel Xeon E5620

In Fig. 6.9a is shown the number of workers busy over the lifetime of the application. The master was started along with 12 workers. After 15 minutes another 24 were started and then another 24 after and additional 15. After a total runtime of one hour 24 workers were removed for 45 minutes before being started again.

Since the machines were not identical task execution ranged from 40 to 90 seconds per task, with a few taking longer as part of the expected long tail effect. Additionally these machines were not entirely dedicated to running AS and as other users accessed the machines and automated processes started and stopped the performance of each task varied. This is reflected in the multinomial distribution of task execution times in Fig. 6.9b.

Next, the data transfer times in Fig. 6.9c show the effect of the incremental and online nature of Poisson Subsampling. There are two types of data being transferred: “common” and “unique”. “Common” files are those that are shared by all processes and these include the programs necessary to run MD and state definitions. “Unique” files are those for a specific simulation, namely the coordinates to run MD from.
Since “common” files can be cached, the communication overhead becomes the cost of transferring the “unique” files. In the case of AS, there is a third situation: “unique” files that changes. Since PS is incremental, after each AS iteration the new model is written to disk in a way that the newly “learned” data is saved in a file for each iteration. This file then needs to be sent to the workers and any variation is the amount of data transferred is due to this. If AS is slowly exploring state-space then these files are small. However when a new energy well is discovered many new states may be found in a single iteration. As shown in Fig. 6.9c the majority of the time needed to transfer these files is small – less than 50 milliseconds. However, there are several tasks that took longer to transfer their data. This is due to the discovery of new areas as well the addition of workers late in the runtime (hour 1.75). In Fig. 6.9d is shown the actual amount of new data that is transferred to each workers after each iteration. Larger sizes indicate that more cells were discovered during that iteration than during a previous. Thus, the two large increases before iteration 30 are most likely when a path between two energy wells was completed and AS rapidly explored the new low-energy region.
Finally, Fig. 6.10 show the improvement of Adaptive Sampling over plain MD. Since the protein being simulated is small, the possible conformations can be visualized as a function of its two dihedral angles, $\Phi$ and $\Psi$. The cell region of each center can be shown using a Voronoi plot. For comparison, after the AS completed, the total time simulated was calculated and a MD trajectory run for the same amount. The conformations sampled by MD were then subsampled by PS to obtain the first diagram. Next the the cell definitions accumulated by AS were plotted in the second diagram. Then the new states found by AS were determined by using Online PS: using the definitions found with MD as the base, the AS cells were added. The resulting model combined both MD and AS and those that were new were found by AS and not MD, which are highlighted in the third figure.

6.5 Conclusions

We have extended the Poisson Disc Sampling technique from computer graphics for generating points satisfying a distance invariant to operate on a pre-selected set
of points extracted from some arbitrary distribution. This method, Poisson Sampling (PS), is online and incremental, and requires a single pass of the input data, making it ideal for processing streams of potentially infinite data. PS is shown to satisfactorily cover the original dataset, and can be used as a preprocessing step of a data analysis pipeline. Additionally the online and incremental properties of PS are desirable features for efficiently communicating a common data model to nodes in a distributed algorithms. Finally we have applied Poisson Sampling to a real problem in molecular dynamics by developing a distributed Adaptive Sampling program based on PS for efficiently finding new conformations of a protein.

Further work can be done to improve Poisson Subsampling by labeling the points so that subsampling can accumulate additional information than just choosing a point. This also has implications in use. For example, for Adaptive Sampling, this could be used to provide an initial estimation of the population of each cell, a value that is used in the estimation of the relative probabilities of each cell.
7.1 Introduction

Proteins are complex molecules of fundamental importance in biological processes. Numerical simulation using molecular dynamics (MD) has proven to be a powerful tool to predict many important properties such as the native state of the protein or its free energy\cite{71, 72}. In this paper, we will focus on methods, based on MD, to calculate reaction rates, which are defined as transition rate between metastable states or conformations of the protein. As a byproduct of our analysis, we will also calculate the main mechanisms of the reaction, i.e., the transition pathways. For example, a cartoon model of the free energy for a bio-molecule is shown in Figure 7.1. It illustrates schematically how MD trajectories explore the conformational space. The left region represents the reactant states (R), and the right the product states (P). Trajectories spend most of their time in R or P with infrequent transitions due to the energy barrier that separates the two states.

In 1977 McCammon et al. applied MD to the bovine pancreatic trypsin inhibitor (BPTI).\cite{84} While the system was simple (vacuum with a crude force field), the simulation nonetheless contributed to shifting the view of proteins as rigid “brass models.” Since this initial simulation, MD has been used to study a wide variety of topics, such as identification of integral motions such as hinge bending modes\cite{38}, tRNA flexibility\cite{52}, and the study of \textit{E. coli} chaperone GroEL\cite{126}. The application of MD to larger and more complicated molecules impacts development of force-field
parameters such as the CHARMM\cite{81, 57} and AMBER families\cite{56}, and solvent models (generally categorized as implicit\cite{33} or explicitly defined\cite{64, 63, 16}). Recently, MD has been used to study an HIV capsid\cite{129}, a complete satellite tobacco mosaic virus\cite{47}, as well as screening designed proteins\cite{65}.

MD simulation of protein is a notoriously difficult computational task. In fact, a large portion of the major supercomputers’ time is currently dedicated to this type of simulation. The main difficulty is that the timescales of interest are typically in the milli-second ($10^{-3}$ sec) while a typical time step in MD is on the order of the femto-second ($10^{-15}$ sec). Therefore a brute force simulation would require on the order of 1,000 billion time steps, which is impractical given the current hardware.

High-performance computing infrastructures allow development of efficient parallel algorithms to speed up simulations\cite{54, 91, 29}. Specialized hardware, such as MDGRAPE\cite{49, 115} and Anton\cite{106, 104}, provide orders-of-magnitude speedup over traditional HPC simulations. Similarly, work with Graphical Processing Units (GPUs) have shown that GPUs achieved greater performance over a single CPU\cite{77, 92, 79, 48}, allowing a single GPU to simulate biological systems with comparable performance to a cluster\cite{48, 8}.
Most approaches to date attempted to accelerate a single (or a few) very long simulations. Due to the sequential nature of MD simulations, this is clearly a challenging task, which limited scalability. Force-calculations are a major bottleneck for MD. As such, much work has gone into the development of fast and efficient algorithms to dedicate large resources to long simulations of molecules. For instance, NAMD and AMBER were among the first to achieve scaling to hundreds of nodes and microsecond-timescales using parallel implementations [87, 44]. Improvements to constraint algorithms [103, 53], particle interactions [73, 20, 21], and memory access patterns [85, 54] have resulted in significant performance improvements over the decades.

At the other end, a recent class of methods is attempting to predict equilibrium properties, such as free energy, reaction rate and reaction pathways, from a large number of short trajectories. Since the trajectories are largely independent, this approach is naturally scalable. In fact, in this paper, we will demonstrate large-scale simulations using parallel computer grids, showing that scalability and performance can be achieved even on slow networks or when using geographically remote compute nodes.

This class of approaches have been applied with great success by several groups, using a method called Markov State Model (MSM) [110, 88, 111, 117]. This method partitions the conformational space into cells or states. Then a stochastic transition matrix is computed. The entry \((i,j)\) in the matrix stores the probability of a trajectory to go from cell \(i\) to cell \(j\) after some lag time \(\tau\). From the eigenvalues, eigenvectors, and other properties of the matrix, we can extract all relevant time scales and kinetics of interest.

This type of approach requires running a large number of trajectories of length \(\tau\) and as such is easily parallelizable. The accuracy of this method is independent of the presence of energy barriers or of the metastability of regions in conformational
space. Instead it relies on efficient local sampling to predict time scales that can be 
opters of magnitude longer than the aggregate simulation time.

One limitation of MSM is that its accuracy depends on the choice of \( \tau \). In fact, 
getting converged rates is very difficult. At short lag times \( \tau \), a significant bias is 
present in the prediction, that is even in the presence of infinite sampling the predicted 
rate is incorrect. This can be addressed by increasing the lag time but this has two 
limitations. First, this leads to an increase in required simulation time. Second, this 
typically leads to a larger variance. In practice, it is difficult to obtain converged 
results with respect to the lag time and sampling size.

Once a MSM has been computed, one can obtain the implicit time scales from 
the diagonalization of the stochastic transition matrix. These implicit time scales 
are highly sensitive to the time lag, or the time between samples used to construct 
the transition matrix: at short time lags implicit time scales have a bias that makes 
them appear much faster than they really are. At large time lags the statistical error 
dominates, and it is hard to find a compromise that balances these two errors.

In this paper, we considered a variant method, called the Weighted Ensemble 
(WE) approach which has a higher computational cost but convergence is more ro-
ut and easier to monitor\[58, 13, 17\]. WE also avoids the systematic biases found 
in MSM\[11\]. The WE method was introduced by Huber and Kim\[58\] to address the 
problem of capturing infrequent reaction events under Brownian dynamics. Further 
studies have applied it to simulated annealing (SI) obtaining higher success rate (or-
der of magnitude) for finding the global optimum as compared to traditional SI\[59\], 
as well as in the investigation of Super Oxide Dismutase and Monoclonal Antibody 
NC6.8. Zuckerman et al. have extended WE and describe several key properties\[128\]: 
(i) WE produces unbiased results for both Markovian and non-Markovian dynamics, 
indicating that the method is much more general than previously understood, and 
(ii) the method depends on the number of cells, which may be updated dynamically,
allowing the system to discover unknown cells without loss of accuracy in the calculation of reaction rates. Applications of WE to toy models and alanine dipeptide,[18], coarse-grained models of Adenylate Cyclase,[17], and the Sodium Symport Mhp2 Transporter protein[6] have yielded promising results.

In WE a large number of parallel walkers (MD trajectories) are used. Later, we will explain in more details the algorithm, but the basic process is to start a large number of trajectories from regions in a partitioning of the free-energy surface. After a small number of MD integration steps we consider the population of walkers in each cell. Then, using a statistically unbiased procedure, we are able to either remove walkers from over-populated cells or, on the contrary, repopulate cells that are becoming depleted. This is done through an appropriate sub-selection (killing walkers) or duplication process. This ensures that when the simulation restarts the number of walkers in each cell is the same, and is equal to some target number. Then, we resume the simulation and integrate forward in time each walker using MD. This process is repeated until the quantities of interest, such as the reaction rates, which are computed through an appropriate post-processing procedure, are converged.

This method avoids the slow time scales found in MD by ensuring that all cells are equally populated. This means for example that regions near the transition barrier between two metastable regions are sampled as often as minimum free energy basins. This method can be interpreted as a way of enhancing the sampling of conformational space. We note that its efficiency still depends on an appropriate choice of macro-states or cells, although the fact that these cells form a simple partition and can be constructed in a variety of ways (e.g., they can be simple Voronoi cells) gives a lot of flexibility to the method. Some of the advantages of WE is that it is easy to setup, is inherently massively parallel and scalable, and is unbiased. For example, unlike MSM, WE is guaranteed to converge to the exact result with enough samples.

Notwithstanding these properties, AWE, like MSM and other related methods,
remains computationally expensive due to the large number of walkers that need to be run. This paper will present a software infrastructure we developed to facilitate running such calculations with accelerated convergence. This is achieved by an appropriate selection of the initial conditions as described in [41, 17]; hence the method is known as Accelerated Weighted Ensemble (AWE), an extension of WE where the initial weights have been approximated to the steady state weights to accelerate the convergence. We compute them from a transition matrix obtained from the simulation, based on cells as they come from clustering, perhaps using MSMBuilder. AWE also uses a simpler and more accurate method for resampling and generating new walker populations through splitting and merging of walkers [41].

AWE allows one to eliminate the bias introduced in MSM by generating the statistically exact distribution of walkers inside each cell. MSM uses the equilibrium distribution for the walkers in each cell, which causes a bias. Instead, AWE uses an out-of-equilibrium distribution corresponding to walkers flowing from reactants to product states (e.g., folding/unfolding) and vice versa, thereby producing exact statistics [41].

We focused on the following main goals when designing the software:

1. Allow using any MD code in a black-box manner, without having to make intrusive changes inside the code. This is important since many MD codes are available and people are often required to use a particular MD code because of some special required capability. For instance, the following MD codes are usable with our framework: AMBER [32], CHARMM [30], GROMACS [54, 99], LAMMPS [95], NAMD [91], MDynaMix [80], Orac [97, 83], GROMOS [37], Tinker [89], Desmond [19], DL_POLY [118].

2. Dynamically scale to available resources as they are added and removed as availability or funding allows and robustly handle resource failure. Details are discussed later, but the main idea is to support, for example, starting AWE us-
ing a dedicated cluster locally then adding cloud infrastructure such as Amazon EC2.

3. Use a scripting language for quick prototyping, interactive simulations, and user-friendly codes. We based our software on Python for this purpose. The use of this simple scripting interface also means that it is not difficult to reuse our framework for other methods that require running a large number of short trajectories with some appropriate post-processing, e.g., MSM and other related methods.

In this paper, we present a distributed computing implementation of AWE, based on Work Queue, that is capable of using computing resources with different architectural properties (e.g., GPU, CPU), as well as different job execution semantics such as dedicated (HPC grid) or cycle-scavenging (Condor) resources. At peak performance, for our validation, AWE-WQ simulated an aggregate 1.5 milliseconds, with a peak performance of 1000 nanosecond/hour, showing that a large amount of resources can be efficiently managed by the system.

Similar software packages include the Copernicus framework developed by Pronk et al. [98], Adaptive Markov State Models (adaptive MSMs) by Bowman et al. [25], and the Weighted Ensemble Simulation Toolkit with Parallelization and Analysis (WESTPA) developed by Zwier et al. [130]. Copernicus is a framework for running ensemble molecular dynamics that allows multiple resources to be used, supports multiple project types, and adaptive sampling. Adaptive MSMs iteratively build MSM models to determine the contribution of each state to the slowest kinetic process. Based on this information further simulations are run from states based proportionally to their contribution to the uncertainty. WESTPA is a software framework for running WE simulations. Currently under development by the Lillian Chong and Dan Zuckermann and not yet released, WESTPA has been used to study several systems, such as molecular association [131] and the sodium symporter Mhp1 [6].
The fault-tolerance model, intelligent task scheduling, and caching distinguish AWE-WQ. These features allow the program to dynamically handle resource addition and removal, automatically handle worker failures, recover from machine failures, support clusters of heterogeneous computers, and minimize data transfer.

We first describe the algorithm, design challenges, and implementation in the Design and Implementation section, and provide a brief overview of usage. Secondly, in the Results section, we demonstrate that AWE-WQ meets the criteria described in the Design and Implementation section, as well as provide a validation of the implementation on a non-trivial protein. Finally, we provide public access to the software, the datasets used for the results, as well as describing the current limitations and future directions in Availability and Future Directions section.

7.2 Design and Implementation

7.2.1 The Accelerated Weighted Ensemble Algorithm

The WE approach proceeds as follows and illustrated in Figure 7.2.

1. Partition state-space into $C$ cells.

2. Determine $W$ number of simulations (walkers) to maintain in each cell.

3. Parallel step (walk): run the $N = W \times C$ short simulations, assign each of the $N$ final conformations to a cell.

4. Synchronization barrier (resample): once all the $N$ simulations are assigned, merge and split the walkers to maintain $W$ walkers in each cell and update the associated weights.

5. Go to step 3 unless weights converged.

The splitting and merging technique used in AWE generally works as follows. Given a population of walkers with weights $w_i$, we first calculate a mean weight
\[ W_m = \sum_i w_i n_{\text{target walkers}}, \]  
where \( n_{\text{target walkers}} \) is the desired number of walkers that we are trying to achieve. Then, if a walker has a weight greater than \( W_m \), it is split into an integer number of walkers of weight \( W_m \), with a remainder that is reinserted into the list of walkers to process. Walkers that have a weight less than \( W_m \) are merged in a statistically exact way to create walkers with a weight greater or equal to \( W_m \).

By repeating this process of splitting and merging walkers, we are able to generate a population of walkers with weights exactly equal to \( W_m \), in a statistically exact manner. This is different from the procedure in Huber and Kim\cite{58} that generates walkers with weights between \( W_m/2 \) and \( 2W_m \) but not exactly equal to the target weight \( W_m \).

The choice of method for defining cells is orthogonal to WE. As long as a cell definition exists, in cells.dat currently, AWE-WQ is able to use it. This allows one to define cells arbitrarily to examine the effect of a cell definition method. We based the cells on the MSM construction in order to accelerate the convergence of cell weights. As explained previously, the main reason to use WE is to remove the bias present in MSM models due to the lag time and the (incorrect) distribution of walkers inside each cell.

Upon reaching steady-state the weight of each cell converges to its probability, allowing this procedure to calculate free energy. The resampling procedure allows even low-probability cells to be accurately sampled.

Calculation of transition rates requires a further modification: Define two sets of states, \( R \) and \( P \), associated with the reactant and product regions (Figure 7.2). Each walker is assigned a color, such as blue for \( R \) and red for \( P \), corresponding to the previously visited region. At each step in the simulation whenever a blue walker enters \( P \), its color changes to red, and vice versa. The rate from \( R \) to \( P \) is then directly obtained by computing the flux of blue particles that change color, and similarly for the \( P \) to \( R \) rate. If extended to multiple colors and sets multiple kinetic
Figure 7.2. Illustration of WE algorithm. Each region represents a cell on the energy surface, the circles the walking simulations, the colors the association with the reactant (R) or product (P) regions.

rates can be determined. In practical terms, the $R$ and $P$ regions may correspond to the unfolded and folding regions and the fluxes to the folding and unfolding rates. By providing an initial approximation of the free energy the convergence to steady state can be accelerated.

From an implementation standpoint, the method requires the following three steps (as illustrated in Figure 7.3): (1) Most of the work is done in parallel as a large number of independent short MD trajectories. (2) The parallel barrier: wait for all MD steps to complete then collect walker assignments and final positions, split and merge walkers, and update weights. (3) Go to step (1) if needed.

7.2.2 Usage

In order to use AWE-WQ one must define the cells from which to run the walkers. The general protocol, illustrated in Figure 7.4, is as follows: (1) sample the search space, (2) determine cell definitions, (3) prepare input files, (4) run AWE-WQ, (5) monitor progress.
Figure 7.3. The AWE-WQ flowchart of the major steps. On startup molecule conformations are loaded into the Walker datastructures. Each walker is then, in parallel, executed and assigned to a cell. The join waits for all walkers to finish, before the resample procedure is applied. If the system has converged then the program halts, others walkers are rerun.

7.2.2.1 Sampling

First some sampling must be done to explore state space. This can be accomplished with programs such as GROMACS, CHARMM, AMBER, NAMD, or using infrastructure such as Folding@home. For this instance, sampling of Alanine Dipeptide has been run previously, which can be extracted:

$ tar xf XTC.tar.bz2

7.2.2.2 Determine Cell Definitions

To start, we have collected MD data and stored them in the XTC directory. The directory structure is XTC/RUN#/frame0.xtc, where # is the trajectory number. The sampling data is imported into MSMBuilder using the ConvertDataToHDF.py command, paramterized with a PDB file containing the system coordinates and the path to the MD data:
Sampling
Run molecular dynamics using some MD simulation software to do an initial sampling of conformational space. This can be accomplished through arbitrary simulation software.

Determine Cell Definitions
- Programs: Use MSMBuilder to cluster the sampling data. The number of clusters should be tractable (100's-1000's) with cells in the two regions under study (e.g. "folded" or "unfolded") and intermediate states.
- Programs: ConvertToHDF.py, Cluster.py, BuildMSM.py

AWE-WQ Input Preparation
- Programs: SaveStructures.py, awe-import-gens, awe-prepare, [awe-calc-gens-rmsd], [awe-define-regions]
- Programs: Extract a given number of conformations from each of the cells defined above (SaveStructures.py) and import the cell definitions from MSMBuilder format to AWE-WQ format (awe-import-gens) and setup the initial files for the AWE-WQ run (awe-prepare).

Regions of metastable states need to then be determined, which are ultimately given as a list of the cells belonging to each region (e.g. "folded" and "unfolded") and scripts are given to do so using RMSD to a reference structure (awe-calc-gens-rmsd, awe-define-regions).

Run AWE-WQ
- Master: Start the AWE-WQ process on a machine. This process loads the initial conformations (walkers) for each cell, and is in charge of scheduling each task, processing the result, and the resampling procedure.
- Workers: Resources can be allocated in either a hierarchical fashion through work_queue_worker to start intermediate nodes, or through the work_queue_pool to automatically maintain workers for the master.

Monitor AWE Progress
- Programs: Use work_queue_status to get the current resources runtime status (number of workers, number of tasks waiting/completed, etc). By using awe-plot-wq-stats the plot of the resource usage over the runtime of the program can be obtained.
- Programs: Finally, the awe-flux command allows the convergence of the calculated flux to be monitored. Once convergence within a determined threshold is obtained the program may be halted.

Figure 7.4. Diagram of the workflow to run AWE-WQ with associated programs and descriptions. First a sampling must be run with some MD software such as GROMACS, CHARMM, AMBER, etc. Next, define the cells using some clustering procedure. In this case, the MSMBuilder package provides some infrastructure. Prepare the input to AWE-WQ by converting the files and defining metastable states. Run AWE-WQ by starting the master process and allocating workers with various resources. Finally, continuously monitor the status and progress of AWE-WQ.
$ ConvertDataToHDF.py -s native.pdb -i XTC

The next step defines the cells. Conformations are clustered with a hybrid k-centers/k-medoids algorithm using the RMSD between atoms as the distance metric. The *AtomIndices.dat* defines the atoms to consider when computing the distance between conformations. Using a subset (such as all non-hydrogens) prevents too fine a granularity from overfitting the data. Finally, we will cluster the data into 100 groups.

$ Cluster.py rmsd -a AtomIndices.dat hybrid -k 100

By inspecting the implied timescales (not shown) we build a Markov State Model at lagtime 10.

$ BuildMSM.py -l 10

7.2.2.3 AWE-WQ Input Preparation

Extract a given number of conformations from each of the cells defined above (SaveStructures.py) and import the cell definitions from MSMBuilder format to AWE-WQ format (awe-import-gens) and setup the initial files for the AWE-WQ run (awe-prepare). Regions of metastable states need to then be determined, which are ultimately given as a list of the cells belonging to each region (e.g. ”folded” and ”unfolded”) and scripts are given to do so using RMSD to a reference structure (awe-calc-gens-rmsd, awe-define-regions). We plan to maintain 10 simulations in the cells, so we need to extract conformations from the states using MSMBuilder.

$ SaveStructures.py -c 10 -f pdb -S sep -o Walkers

In order to run AWE-WQ we must then import the cell definitions which were written by MSMBuilder to *Gens.lh5*. 
$ awe-import-gens -g Data/Gens.lh5 -o cells.dat -m Data/Mapping.dat

In order to compute the fluxes we need specify regions of metastable states. AWE-WQ provides some commands to assist with this process: `awe-calc-gens-rmsd` and `awe-define-regions`. We use `awe-calc-gens-rmsd` to compute the RMSD of each cell to some reference conformation such as the native state.

$ awe-calc-gens-rmsd -r native.pdb -n AtomIndices.dat \
   -g Data/Gens.lh5 -o native-rmsd.dat

By plotting the distribution of values we can classify conformations with RMSD $\leq 2.3$ Å as *folded* and those with RMSD $\geq 2.5$ Å as *unfolded*. The two output files `folded.dat` and `unfolded.dat` now contain the integer indices of the states belonging to these regions.

$ awe-define-regions -i native-rmsd.dat -c 0.23 -O '<=' -o folded.dat
$ awe-define-regions -i native-rmsd.dat -c 0.25 -O '>=' -o unfolded.dat

We can now prepare for AWE-WQ by checking dependencies and populating the directory with other necessary files by running `awe-prepare`.

7.2.2.4 Running AWE-WQ

There are two components to consider when running AWE-WQ: the master process and the resources. The master is the driver of the algorithm, managing task definitions, scheduling, processing, and the resampling procedure. In order to run the walkers, resources must be allocated.

Master  Start the AWE-WQ process on a machine. This process loads the initial conformations (walkers) for each cell, and is in charge of scheduling each task, processing the result, and the resampling procedure. This runs AWE-WQ maintaining
10 walkers in 100 cells, whose definition is provided in cells.dat with initial weights in Data/Populations.dat. The coordinates for the walkers are found in the Walkers directory. The metastable regions are provided in folded.dat and unfolded.dat as a list of cell id numbers belonging to each region. Finally, we give a name to the master ("awe-wq") to that workers can easily locate the host and port.

```
$ awe-wq -N 10 -C 100 -c cells.dat -w Data/Populations.dat -W Walkers \ 
   -r folded.dat unfolded.dat -n awe-wq
```

Workers Resources can be allocated either directly using work_queue_worker or managed automatically using work_queue_pool. Using work_queue_worker also allows the worker to operate as a "Foreman", enabling the hierarchical distribution of tasks. The work_queue_pool program maintains workers for the master based on need, and is in charge of submission to various backends such as the local machine, Condor, SGE, PBS, etc. Since the master has started we can start some workers locally.

```
$ work_queue_pool -a -N awe-wq 24
```

7.2.2.5 Monitoring AWE Progress

Use work_queue_status to get the current resources runtime status (number of workers, number of tasks waiting/completed, etc). By using awe-plot-wq-stats the plot of the resource usage over the runtime of the program can be obtained.

Additionally, the current status of the master, such as workers busy and tasks complete can be viewed using the work_queue_status command.

```
$ work_queue_status
```

```
PROJECT HOST PORT WAITING BUSY COMPLETE WORKERS
awe-wq fah.crc.nd.edu 1024 133 36 57547 36
```
The \texttt{awe-flux} command allows the convergence of the calculated flux to be monitored. Once convergence within a determined threshold is obtained the program may be halted.

Additionally, other analyses are appropriate. For instance, the energy surface for Alanine Dipeptide can be visualized as a function of its dihedral angles. As such, we can plot, as shown in Figure 7.5, the cell coordinates and the initial estimation of the weights as well as computed weights after several iterations of AWE-WQ.

7.2.3 Design and Implementation

Since the AWE method follows a pattern of fan-out followed by fan-in we implement using a Master/Worker paradigm. AWE-WQ is designed to 1) support off-the-
shelf MD backend software, 2) dynamically scale according to resource availability, 3) support heterogeneous runtime environments, and 4) be robust in the face of system failure. Application of AWE to a non-trivial biological system may have on the order of 10,000 simulations per iteration. On one CPU, if each walker requires 30 minutes to simulate, seven months would be required to run a single iteration of AWE. Since walkers are short, assuming 30 minutes per walker, seven months would be required to run 10,000 walkers for each AWE iteration using a single processor. With 1000 CPUs, the expected time drops to 5 hours, assuming uniform performance. Additionally, the synchronization barrier allows straggling workers to slow down the entire application. Since we expect to need hundreds of iterations to converge, the challenge is to implement AWE such that it scales to 1000+ processors and can be sustained for months with no major slowdown due to stragglers.

7.2.3.1 Support of à la carte MD software

Rather than reimplement the core MD algorithms which are present in multiple software packages (i.e., GROMACS, AMBER, CHARMM, etc) we decided to provide flexibility of MD backend. This flexibility presented several challenges by imposing a requirement of specifying the steps needed to run the backend, the transfer of files, the execution environment, and the interaction with the other steps of the procedure. The approach taken in the following: a Walker is described as a Task which consists of a program to run and the input and output files. Once specified, this Task is scheduled to run on an available Worker, which is a process that accepts any input files, executes the Task’s command, and returns the results upon request.

We used GROMACS 4.5 for the MD backend. The `awe-prepare` script sets up files and scripts in the local directory necessary to run `awe-wq`. One of these scripts, `execute-task.sh`, executes the commands on the worker. In order to use a different MD backend `execute-task.sh` needs to define how to run the MD, and `awe-wq` needs
to specify the files to send to the worker.

7.2.3.2 Dynamically using available resources

Our solution to the worker fault-tolerance problem implies the elasticity – the ability to dynamically scale to available resources – of the program. If a task fails during execution it is rescheduled to run on the next available worker. As resources are added or removed tasks are started or reschedule, respectively. This allows a project to be started with the campus cluster, then EC2 machines run until budget exhaustion, then continued with only the cluster. Managing the resource pools are the interactions between the master, the Catalog Server (CS), and WorkQueue Pool (WQP) processes. The CS allows workers to dynamically determine the location of the master by mapping a project name (e.g. awe-wq) to a hostname and port (e.g. fah.crc.nd.edu:1024). The master registers this information when started, which allows it to be restarted in the event of machine failure without requiring the resources to be reallocated. In addition, the CS store information about the current resource requirements of a master, such as the number of tasks waiting to be scheduled. A WQP can thus use this information to automatically start workers based on the runtime needs. This is further enabled by the asynchronous model, which treats tasks as independent, arbitrarily executable entities. This would not have been possible using an MPI-based implementation.

7.2.3.3 Heterogeneous runtime environments

Additionally, AWE-WQ can use heterogeneous resources, allowing systems with different properties and (potentially) operational semantics to be used. For instance, this supports using GPU-based as well as CPU-based MD backend codes. This is accomplished by extending the Task abstraction to dynamically resolve file location upon worker connection rather than statically. For example, an input file can be spec-
ified for a Task as `binaries/$OS/$ARCH/mdrun`. When a worker connects to the master it reports the operating system and architecture of the remote node in the `$OS` and `$ARCH` variables, which may then be expanded to send `binaries/Linux/i686/mdrun` and `binaries/Darwin/x86_64/mdrun` to workers running on a 32-bit Linux and 64-bit Mac OS machine, respectively. Finally, these variables can be overridden by the user to provide additional granularity.

### 7.2.3.4 Fault-tolerance

AWE-WQ is implemented with the assumption that machine failures are not only possible but in fact common, which is the opposite of the paradigm underlying MPI infrastructures. As such two broad categories of failures need to be accounted for: failure of the worker or master processes. In the first case, worker failure are handled transparently and Tasks rescheduled to the next available worker.

Failure of the master node would normally result in the termination with an inability to resume. This is overcome by using a transactional logging mechanism: a combination of persistent and volatile logs. Each result from a walker is added to a log and after each AWE iteration the state of the system is checkpointed to disk and the log cleared. If the process fails, then the state is initialized from the checkpoint and the log used to replay and missed results from that iteration. This way only uncompleted walkers need to be restarted.

### 7.2.3.5 Intelligent Scheduling

Several components of task scheduling affect the scalability. The first is knowledge of a worker’s past task execution times. This allows tasks to be scheduled preferentially to the most performant machines. The second involves task replication to remove the effect of straggling workers. By monitoring the state of the workers – either busy or idle ($w_i$) – and the number of tasks waiting to be assigned a worker, task
replication can be engaged when \( w_i > 0 \). Once a task returns, all its replicates are cancelled, results are accepted in first-come-first-serve fashion. By combining replication with preferential scheduling, workers are continuously used and slow workers allowed to contribute results until the fast machines are available, which then run the replicated tasks.

7.2.3.6 Event-model

The inner loop of AWE-WQ is an event-loop with asynchronous task execution. Each iteration begins by translating the internal data model of the walkers into Task, which are submitted to the WQ scheduler. While tasks are executing, the program is idle until a result is returned. Each result creates an event that must be handled, either by translation or resubmission. Handlers are lightweight: read the final coordinates and cell assignment into the data models. Since Task executions are asynchronous and results are processed serially, the event-model enforces load balancing by spreading out the calls to the event handlers.

7.2.3.7 Caching allows scaling

Increasing scalability is further accomplished by minimizing the communication overhead and maximizing the parallel work. Ideally, the master process will spend most of its time waiting. By categorizing files as those common to all tasks (common) or specific to the current one (unique), the common files can be cached on the workers. Since we expect the number of walkers to be larger than resource availability the caching mechanism shifts overhead to depend on resources rather than problem size. For instance, using a GROMACS backend each Task requires 34 MB in input files, binaries, and miscellaneous files.

In the case of the WW domain simulation, each task required 34 MB of GROMACS-related files as overhead and 100 KB of task specific data. Assuming 10,000 tasks/it-
eration and 500 iterations, the total amount of data transferred is $34 \text{(MB/Task)} \times 10,000 \text{ (Tasks/Iteration)} \times 500 \text{ Iterations} \times 2^{-20} \text{ (TB/MB)} = 162 \text{ TB}$. By caching input files, data transferred depends only on the number of new workers seen, rather than total number of tasks, which may be orders of magnitude greater. In this case, assuming 1000 workers are connected the total data transferred becomes $34 \text{ MB} \times 1000 \times 2^{-10} \text{ (GB/MB)} = 33\text{GB}$.

7.3 Results

We have run AWE-WQ for several months where 3.5 million tasks have executed over 600 compute years in a heterogeneous environment simulating over 1.5 milliseconds of time. In this section we describe the application the AWE algorithm to a non-trivial biological system.

7.3.1 Heterogeneous Resource Usage:

Available resources consisted of the following: Notre Dame HPC grid with 6000 cores shared among campus users; Notre Dame Condor pool with variable (usually around 1200) cores and flocking capabilities with Purdue University and the University of Wisconsin-Madison; Stanford University’s Institute for Computational and Mathematical Engineering (ICME) cluster with 200 CPUs and 100 NVIDIA GPUs, and cloud virtual machines via Amazon EC2 and Microsoft Azure.

Figure 7.6 displays the distribution of task execution times. The multimodal distribution reflects the differences of performance within each pool. The GPU cluster executes tasks within minutes, while the ND HPC grid and EC2 machines are the best performant CPU-based pools. The range of performance of Condor machines is unsurprising and illustrative of the cycle-scavenging nature of the platform. The Azure machines were the least performant resources, partly due to the necessity of running the tasks via Cygwin.
Execution time distribution indicative of different environments. The distribution of task execution times indicate several peaks associated with the different underlying resources being used.
7.3.2 Elasticity and Scalability:

Figure 7.7 demonstrates the fault-tolerance elasticity and scalability of the application as an expanded view of the number of busy resources over the runtime – the inset shows the entire run. Unexpected failures of the master (months 4, 6), initiated development and integration of features, bug fixes, and monitoring capabilities. Due to power outages and infrastructure upgrades the master process restarted multiple times. In spite of these failures over 250 iterations were completed without data loss due to the transactional logging mechanism. Periodically persisting state additionally allowed us to integrate new features, such as task replication, and bug fixes over the course of the project’s execution.

Focusing on the region between days 865 and 900 shows a steady decrease in the number of busy resources until a new pool of workers became available and AWE-WQ automatically started using them. It must be emphasized as well that minimal human input was needed to take advantage of the additional resources: due to the use of the Catalog Server all that was required was to start a WQ Pool process for the additional resources.

The periodic dips, every 5 days in the first half and every $\approx 20$ hours for the second, correspond to the global synchronization barrier, each period requiring 12,000 tasks to complete. By scheduling the Tasks based on resource availability and progression through the AWE algorithm resources are fully utilized and the long tail effect from straggling workers is eliminated. Due to the use of the caching mechanism communication overhead is reduced: while the entire task payload (common and unique files) is 34 MB it is only for the first task a worker executes that his is transferred. Each subsequent task only requires 100 KB of data to be transferred (both in- and outbound traffic).
Figure 7.7. Expanded view of the entire run (inset) demonstrating elasticity, scalability, and fault tolerance. In the inset long gaps (such as the sudden failures at month 4 and 6) in activity indicate times when major development was done on the code (features added, bugs fixed). Despite the troubles experienced by the master node calculation of the results resumed once the code was deployed. Plotting the number of busy workers over many days showing the ability of the software to adapt to a changing environment. The periodic dips indicate the global synchronization barrier, each requiring 12,000 tasks to complete. The large increase after day 885 occurred when many nodes were suddenly available and were automatically incorporated.

Figure 7.8. The WW domain forms three anti-parallel $\beta$-sheets separated by two flexible loop regions.
7.3.3 The WW Domain protein

The WW Domain is a 34 residue protein domain with two conserved tryptophan (W) residues and comprised of two antiparallel β-sheets as shown in Figure 7.8. We studied a mutant which is known to folding 13 $\mu$s [76]. Simulation parameters were the Amber96 force field with Generalized Born implicit solvent, a viscosity parameter ($\gamma$) of 1ps$^{-1}$, temperature of 395K (close to the optimal folding temperature [107]), saving conformations every 1 ns. The simulation was run for 200 $\mu$s during which multiple folding and unfolding events were observed. The conformations sampled by the simulation were then clustered using the $\alpha$ and $\beta$ carbons using a cutoff of 3 Å. In order to reduce the time spent sampling the unfolded space we established that conformations with an RMSD to the folded structure less than 3 Å, 6 Å, and greater as folded, intermediate, and unfolded, respectively. Using these definitions the number of unfolded states was reduced, for a total of 601 cells. The AWE-WQ parameters were thus 601 cells, 20 walkers per cell, and walker length of 500 ps using the same MD parameters as the source data. GROMACS 4.5 was used for the MD backend.

7.3.4 WW Results:

Over the course of many months AWE-WQ completed 345 iterations, with an aggregate 2.3 milliseconds of time simulated using 3.6 million tasks, and a peak performance of 1000 ns/hour. Analysis indicates that the AWE-WQ results converge within acceptable error tolerance. Table 7.1 displays the forward and backwards rate estimations computed using AWE-WQ and the 200 $\mu$s simulation. The AWE estimation lies within the confidence interval – with smaller statistical error – than the brute force simulation.

Figure 7.9 illustrates that forward and backward fluxes took 30 iterations to converge to the confidence interval built from brute force simulation and are stabilized.
TABLE 7.1

FORWARD, BACKWARD RATES VIA AWE AND BRUTE-FORCE CALCULATIONS

<table>
<thead>
<tr>
<th>Method</th>
<th>Forward ($\mu s^{-1}$)</th>
<th>Backward ($\mu s^{-1}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWE-WQ</td>
<td>1.5 ± 0.3</td>
<td>2.5 ± 0.5</td>
</tr>
<tr>
<td>Brute-force</td>
<td>1.8 ± 1.0</td>
<td>2.0 ± 0.9</td>
</tr>
</tbody>
</table>

The forward and backward reaction rates as computed from AWE-WQ and the brute-force simulation.

within the interval afterwards. The resampling procedure of AWE introduces no bias in the estimation of transition rates while exhibiting smaller statistical error, showing an improvement over the brute force method.

Transitions among the states are examined with a time lag of $\tau = \Delta t = 0.5 ns$. Three significant pathways from extended to folded state are extracted from the network and shown in Figure 7.10. Starting from an unfolded conformation two pathways involve multiple helical intermediate structures before $\beta$-sheet formation occurs via rearrangement. The third pathway occurs as a rapid collapse before the formation of Loop 2. Loop 1 is then able to form, after which the molecule undergoes further refinement to arrive at the native state.

7.3.5 Conclusions:

AWE-WQ meets our criteria: allowing the use of off-the-shelf components for MD backends, pooling of heterogeneous resources, scalability to over 1000 workers, and is able to cope with failures of both worker and master processes. The WW domain was used to validate AWE-WQ by applying the method to a long MD simulation in which multiple folding events are observed. The forward and backward reaction rates
Figure 7.9. Average forward (top) and backward (bottom) flux from AWE (blue curve) with error bar of one standard deviation estimated by block averaging method, compared with confidence interval built from brute force simulation. Red solid line represents the mean flux from brute force simulation and dashed lines are mean flux plus/minus one standard deviation.
Figure 7.10. First three significant pathways from extended to folded conformations. Two pathways (top, middle) involve helix formation as intermediary while the third pathway (bottom) consists of a collapse then refinement.
computed by AWE-WQ are within the range, and have smaller error, of the rates calculated from the reference simulation.
CHAPTER 8

SOFTWARE ARTIFACTS

Several software artifacts have been produced as a result of this work. This chapter will describe them.

The major software packages are:

- Folding@Work
- AWE-WQ
- GUAMPS
- pxul
- mdq
- pwq
- WASQ

Software was predominantly written in Python, C, and shell.

Since Work Queue has a central role in this work, it is appropriate to briefly describe it here. Work Queue is a framework for writing fault-tolerant scalable distributed application. The API exposes the concept of a “Task” and a “Queue”. A Task is a description of a self-contained unit of work consisting of the set of input and output files and an executable which is the entry point into the task. Tasks are submitted to a WorkQueue data structure which is a queue that assigns Tasks to
Figure 8.1. WorkQueue Architecture allows merging of heterogenous resource pools.

Workers. A Worker is a process executing on a node. When a worker is started, it connects to a specified Master process where the tasks are being generated and submitted into the queue. Once a Task is assigned to a Worker, all the necessary files are transferred. If a file has been marked as “cached” in the Task, it is only transferred if not already present on the worker. Once all the required files are present, the Worker executes the task. Upon completion, the Worker notifies the master, which then transfers the output files to the master’s node.

Figure 8.1 shows the general features of Work Queue. By using the Task and Queue abstractions, distributed applications can be written to a generic interface while the framework supports the details of running in specific environments. This allows multiple resources to be combined.

Applications written using Work Queue are in the form of a synchronous event loop that dispatches to asynchronous tasks. The following example code demonstrates the use of the Work Queue framework. Ten tasks are created which execute the command `echo "hello <id>"`, where `<id>` is the task id. Then, while there are still tasks in the queue, wait a maximum of 5 seconds for a task to return. If a task
from work_queue import WorkQueue, Task

Q = WorkQueue(port=9123)

for i in range(10):
    task = Task('echo "hello {}"'.format(i))
    task.specify_tag('{}'.format(i))
    Q.submit(task)

while not Q.empty():
    task = Q.wait(5)
    if task:
        print "Received task {}".format(task.tag)
        print task.output

Work Queue is developed by Professor Douglas Thain and the Cooperative qComputing Lab at the University of Notre Dame.

8.1 Folding@Work

Folding@Work (F@W) is a program inspired by the popular Folding@home project. The focus of F@W is to enable running large distributed molecular dynamics ensemble simulations in High Performance Computing (HPC) environments. This is desirable because, due to the high-profile nature of Folding@home, it is usually used once preliminary data has been gathered about a system. F@W is intended to support preliminary exploratory simulations to narrow down experimental parameters before a system like Folding@home is used.
8.2 AWE-WQ

The Accelerated Weighted Ensemble using Work Queue (AWE-WQ) is set of commands to enable running the Accelerated Weighted Ensemble algorithm for Molecular Dynamics using the Work Queue framework for distributed computing. Details of AWE and WQ are presented elsewhere.

AWE-WQ is implemented using Python, C, and shell script. The logic of the code running on the master is written in Python. This component is responsible for reading input files, replicating tasks, checkpointing results, recovering from failures.

The input to AWE-WQ is a decomposition of the search space into cells, representative conformations from each cell, and an initial estimation of the weight of each cell. The major steps in running AWE-WQ are: preparation of input files, running the master process, submitting workers, and monitoring progress.

To assist with input preparation, several commands are provided. These allow datafiles to be imported from a previous step in the pipeline such as MSMBuilder. The main entry point into AWE-WQ is the `awe-wq` command, which is parameterized by the port to listen on, location of input files, and number of times to replicate tasks when resources are available. Workers can be started using Work Queue commands `work_queue_worker` and `work_queue_pool`.

An AWE-WQ run is measured in weeks to months for reasonably sized proteins. As such, monitoring the progress of the calculations, as well as recovering from system failures, is important. By plotting the flux between groups of cells an estimation of the convergence can be obtained and the user can stop once satisfied. For instance, a cron job can be created to create a figure of the flux to-date and email it to the users.

Since long-running jobs are susceptible to machine failure, the AWE-WQ master process keeps a transactional log of it’s progress. Should the machine on which the master process is running fail, barring failure of the filesystem, simply rerunning the
The `awe-wq` command will load the most recent checkpoint and replay the log of any results since.

One feature that arises out of the checkpointing process is the ability to update a running AWE-WQ instance with new features during an experiment. By stopping the application and restarting with a new version, users can take advantage of new features without needing to restart the entire experiment.

8.2.1 Architecture

As AWE-WQ is the most complex of the software artifacts produced, here is described the software architecture, illustrated in Figure 8.2. The master application consists of the high-level data models representing the state of the AWE algorithm: atomic coordinates, walkers, cells, and walker colors. This rests on top of a WorkQueue stack, a transactional logging interface, as well as the I/O Python API.

The purpose of the transactional logger (Trax) is to allow the master program to resume upon downtime of the head node. Conceptually there are two parts: checkpoints and logs. As simulation results arrive from the workers, they are added to the AWE system data structure and appended to the log file. A checkpoint, at the end of every AWE iteration, saves the entire state of the system to disk and clears the log file. This is done is such a way (using intermediate files) that if the master node goes down during a checkpoint no data is lost or corrupted.

Passing a simulation specification from the top level to the worker nodes requires several layers to handle certain performance- and translation-related aspects. First, a marshalling layer translates a high-level specification of a simulation (the molecule’s coordinates) into a Task. Next in the path is the Replicator, which manages task replication as resources allow: if there are more resources (workers) than tasks available, tasks are duplicated until there are an equal number of each. The first result to return will be accepted and its duplicates canceled. Finally the tasks are scheduled
using WorkQueue, preferentially choosing workers with historically lower time-to-completion.

The master and worker processes can be coordinated using the Catalog Server (CS). As tasks enter the WorkQueue queue, information about the master is published to the CS. This allows workers, as they are started, to look up an appropriate master to connect to based on a name. In this case the CS maps a name, such as awe-wq-ww-run42, to the correct host and port that the master is running on.

The WorkQueuePool (WQP) takes on the responsibility of allocating resources for a given set of names registered with the CS. As the WorkQueue master publishes statistics about the runtime state to the CS, each WQP can look up the number of tasks and workers currently available. Each WQP can then determine a number of workers to start based on some policy. For instance, a WQP managing workers on a HPC grid may be managing workers for three projects A, B, C with a policy of
70%, 20%, and 10% respective allocation and a maximum of 500 total submissions. Assuming all workers can be started, this would result in 350, 100, and 50 works running for projects A, B, and C respectively. If projects B and C are not visible in the CS, and A only requires 200 tasks, then only 200 tasks would be started. This allows the user to use WQP for each set of resources available to them e.g. HPC and Condor which ensure that their projects have workers available based on the runtime requirements of their program.

Once the workers are started, by the user of WQP, they connect to the master to accept a task. This transfers all the necessary input files (cached if possible) to the worker, which then triggers the entry point to the task. With WorkQueue, this entry point is a shell command to run. For example, `main.sh` would call the `main.sh` script which would have to be transferred as an input file.

8.3 GUAMPS

The GUAMPS (Gromacs Utilities Are a Pain in the Shins) application provides a “get” and “set” API for extracting and modifying GROMACS simulation parameters. Some explanation of process for running simulations with GROMACS is warranted.

All molecular dynamics simulations require: atomic coordinates, parameters describing the properties of each atom and their interactions with each other, and a description of the simulation parameters (length, output frequency, solvent type, etc). Different MD packages manage these in different ways. GROMACS is used in this case because it is appropriate for small to medium systems. GROMACS requires a series of steps in order to prepare simulations. First, the `pdb2gmx` command takes the provided description of a system (atom names and coordinates) and converts them into an internal format. This format can then be further manipulated with various commands. Finally the `grompp` command takes a description of the simulation parameters and creates a binary file (.tpr) with the system ready for simulation. At
this point, calling the `mdrun` command will run the simulation. This pipeline requires significant overhead when GROMACS is being used as a backend in a program like AWE-WQ or Folding@Work. Additionally, it can be difficult to debug when something fails in a distributed system. Finally, the random seed of a simulation is set in the tpr file. This ostensibly allows simulation runs from a tpr to be identical. And while this is normally desirable, the overhead of creating and transferring thousands of tprs is in the gigabytes when only a few bytes (seed) or kilobytes (positions/velocities) need to be changed.

By providing commands (`guamps_get`, `guamps_set`) to retrieve and extract values from the tpr and other output files, the MD backend becomes easier to decouple, less data needs to be transferred, and error can be traced to the data linked with the error.

GUAMPS is written in C and currently allows retrieval and update of

- simulation seed
- simulation timestep
- number of simulation steps
- output frequency of forces
- output frequency of logging
- output frequency of velocities
- output frequency of positions
- coordinates
- velocities
- temperature
8.3.1 PXUL

Since there are many common features being used amongst the Python programs developed, PXUL provides a simple library with these as extensions to the standard Python library. These are extensions to directory changes, executiong programs, logging facilities, efficient string building, and environmental variables. In order to minimize the footprint of installed applications, care was taken to ensure that PXUL has no dependencies other than a standard python interpreter.

8.4 MDQ

MDQ is a Python library and program for running molecular dynamics in a distributed environment in a robust manner, supported task replication and multiple MD backends. MDQ provides the facilities to run long MD tasks. Currently supporting only GROMACS, the API is designed such that arbitrary backends can be used. An internal representation of simulation state (positions, velocities, seed) is used that is common to all MD engines. Thus, in order to support e.g. NAMD, converters to and from the internal representation is needed.

When running an MD simulation one parameter of interest is the simulation time. In practical terms, this time may need to be divided into smaller chunks due to resource availability. For instance, instead of running for one microsecond a simulation may be split into 1000 one nanosecond simulations that are executed sequentially. This would allow programs using Work Queue to merge resources with different runtime properties i.e. HPC where a task can be assumed to run to completion and Condor where short tasks are ideal (we found around 30 minutes to be ideal).
8.5 PWQ

The Python Work Queue Implementation (PWQ) is born out of the need for common features that are enable by but not present in the Work Queue implementation. The prime example is support for task replication. Work Queue exposes an API to replicate and cancel tasks by a tag. This allows one to write a Work Queue application that automatically replicates tasks depending on the ratio of submitted tasks to available resources. Since Work Queue was being used for both standalone applications like Folding@Work and AWE-WQ, as well as small scripts that spawned hundreds of tasks, having a relocations code for accessing the intelligent task replication features was desirable.

Therefore, the PWQ library mirrors the upstream Work Queue API, and is intended to be used as a drop-in wrapper around Work Queue. While PWQ uses Work Queue, it provides a layer between the user’s application and Work Queue, where features like task replications can be managed.

8.6 WASQ

The Working Adaptive Sampling Queue (WASQ) provides an implementation of the Poisson Subsampling (PS) algorithm (an extension of the Poisson Disc Sampling method) and a prototype adaptive sampling method using PS. There are several components to the package.

The PoissonCover module contains the implementation of poisson subsampling algorithm, as well as the incremental (recursive) version The Cell module provides the datastructure representing the points sampled that can be used for efficient representation, incremental addition, and efficient synchronization. The API provides a learn method, accessors cells and their labels, and methods supporting serialization. The learn method accepts an array of the points to add and their associated labels.
The AdaptiveSampling module provides the logic and datastructures for using Poisson Subsampling as the kernel to guide running an ensemble of molecular dynamics simulations. Simulation state is represented as the current positions, velocities of the atoms and the time simulated so far. An Molecular Dynamics Engine then translates the simulation state to specification describing a simulation, which is then executed. A Walker represents a unit of execution: a simulation state, parameters to pass to the MD engine, and a metric function.

An MD Sampler then iterates over a list of Walkers, submitting them to WorkQueue, collecting their results, and selecting a subset of cells to continue to the next iteration.
CHAPTER 9

CONCLUSION

9.1 Summary

This work was initiated by a desire to explore the effect that mutations have on the folding pathways of a protein. During the course of investigating this question several limitations of current state-of-the-art methods were encountered. In order to overcome these issues, software addressing the specific components was developed. Since each step requires significant computing resources, the programs written were done to take advantage of distributed computing to be feasible. Naive conformational sampling with Folding@Work, leads to an adaptive sampling approach with WASQ. Implementing an adaptive sampling method, suitable for distributed systems, requires decomposition of search space that is incremental, online, and minimized communication overhead during synchronization. This was achieved by extending the Poisson Disc Subsampling method to operate on streams of discrete points. Finally, with the use of the Weighted Ensemble implementation in AWE-WQ, a pipeline can be formed from few (at least one) initial structure, to measuring rates of changes between different conformations.

9.2 Future Work

There are several areas in which this work can be extended. On the software side, implementation of task interfaces will allow the pipeline to be used with other MD programs such as NAMD, improving robustness, and integrating with other
analysis tools to further streamline the path from initial structure to kinetic rates. Additionally, application of the pipeline to larger systems, such as penta-alanine, various Trpzip mutants, and the WW mutants and the questions that sparked this work.


V. P. group at Stanford. Timescales accessible by molecular dynamics.


