Bigger, Better BigJob - Pilot-Job frameworks for large scale simulations

Antons Treikalis

August 23, 2013

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2013
Contents

1 Introduction 1
1.1 Objectives ......................................................... 2
1.2 Structure of the dissertation .......................... 3

2 Background 4
2.1 Nanoscale Molecular Dynamics - NAMD ..................... 4
   2.1.1 NAMD parallelization strategy ......................... 4
2.2 Replica Exchange ................................................ 6
   2.2.1 Theory of Replica Exchange Markov Chain Monte Carlo (MCMC) sampling ........................................... 8
   2.2.2 Theory of Molecular Dynamics Parallel Tempering ............. 9
   2.2.3 Optimal Choice of Temperatures .......................... 9
   2.2.4 Mathematical model for total simulation runtime ........... 10
   2.2.5 Synchronous replica exchange ....................... 11
   2.2.6 Asynchronous replica exchange .......................... 12
2.3 Pilot-Jobs ....................................................... 13
2.4 SAGA-Python ..................................................... 14
2.5 SAGA-BigJob ....................................................... 17
2.6 ASyncRE package .............................................. 19

3 Implementation 23
3.1 ASyncRE NAMD module ........................................ 23
   3.1.1 RE simulations with NAMD .......................... 24
   3.1.2 High-level design of the ASyncRE NAMD module .......... 24
   3.1.3 Adapting Temperature Exchange RE NAMD simulation setup for ASyncRE NAMD module .......................... 25
   3.1.4 Implementation of Temperature Exchange RE schema in ASyncRE NAMD module ...................................... 29
   3.1.5 Adapting RE with Umbrella Sampling NAMD simulation setup for ASyncRE NAMD module .......................... 33
   3.1.6 Implementation of RE with Umbrella Sampling schema in ASyncRE NAMD module ...................................... 35

4 Evaluation 37
4.1 ASyncRE package on Kraken .............................. 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.1</td>
<td>RE with Temperature Exchange simulations using NAMD</td>
<td>37</td>
</tr>
<tr>
<td>4.1.2</td>
<td>RE with Temperature Exchange simulations using ASyncRE NAMD module</td>
<td>38</td>
</tr>
<tr>
<td>4.1.3</td>
<td>RE with Umbrella Sampling simulations using NAMD</td>
<td>40</td>
</tr>
<tr>
<td>4.1.4</td>
<td>RE with Umbrella Sampling simulations using ASyncRE</td>
<td>40</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Analysis of NAMD RE simulations</td>
<td>41</td>
</tr>
<tr>
<td>4.1.6</td>
<td>Comparison with ASyncRE Amber module for RE with Umbrella Sampling simulations</td>
<td>44</td>
</tr>
<tr>
<td>4.1.7</td>
<td>Performance analysis of BigJob on Kraken</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>BigJob on Cray machines</td>
<td>49</td>
</tr>
<tr>
<td>4.2.1</td>
<td>MPI with Python</td>
<td>49</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Multiple <code>aprun</code> technique</td>
<td>50</td>
</tr>
<tr>
<td>4.2.3</td>
<td>MPMD technique</td>
<td>51</td>
</tr>
<tr>
<td>4.2.4</td>
<td>TaskFarmer utility</td>
<td>51</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Multiple single-processor program technique</td>
<td>51</td>
</tr>
<tr>
<td>5</td>
<td>Conclusions</td>
<td>55</td>
</tr>
<tr>
<td>5.1</td>
<td>NAMD ASyncRE module</td>
<td>55</td>
</tr>
<tr>
<td>5.2</td>
<td>BigJob on Cray machines</td>
<td>55</td>
</tr>
<tr>
<td>5.3</td>
<td>Future work</td>
<td>56</td>
</tr>
<tr>
<td>5.3.1</td>
<td>NAMD ASyncRE module</td>
<td>56</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Gromacs ASyncRE module</td>
<td>56</td>
</tr>
<tr>
<td>A</td>
<td>Stuff which is too detailed</td>
<td>57</td>
</tr>
<tr>
<td>A.0.3</td>
<td>BigJob example script (Kraken)</td>
<td>57</td>
</tr>
<tr>
<td>A.0.4</td>
<td>NAMD RE with Temperature Exchange example script</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td><code>-replica.namd</code></td>
<td>58</td>
</tr>
<tr>
<td>A.0.5</td>
<td>ASyncRE RE with Temperature Exchange template script</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td><code>-alanin_base.namd</code></td>
<td>63</td>
</tr>
<tr>
<td>A.0.6</td>
<td>NAMD RE with Umbrella Sampling example script</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td><code>-umbrella.namd</code></td>
<td>65</td>
</tr>
<tr>
<td>A.0.7</td>
<td>ASyncRE RE with Umbrella Sampling template script</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td><code>-alanin_base_umbrella.namd</code></td>
<td>70</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Schematic representation of NAMD parallelization mechanism [2]</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Schematic representation of exchanges between neighbouring replicas at different temperatures [11]</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Energy histograms for a system at five different temperatures [11]</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Schematic representation of the SAGA-Python architecture [25]</td>
<td>17</td>
</tr>
<tr>
<td>2.5</td>
<td>Schematic representation of the BigJob architecture [29]</td>
<td>20</td>
</tr>
<tr>
<td>2.6</td>
<td>Schema of the system of software layers used for MD simulations</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>ASyncRE NAMD module class inheritance diagram</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Code snippet from the <code>replica.namd</code> script demonstrating how temperatures for replicas are set</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>Code snippet from the <code>alanin_base.namd</code> ASyncRE script demonstrating how input files for the simulation are set</td>
<td>27</td>
</tr>
<tr>
<td>3.4</td>
<td>Code snippet from the <code>replica.namd</code> script demonstrating implementation of the mechanism for exchanges between replicas</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td>Code snippet from the <code>replica.namd</code> script demonstrating how exchanges of the temperatures are performed</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Code snippet from the <code>doExchanges()</code> function of the <code>pj_async_re.py</code> file, demonstrating how to switch between Metropolis and Gibbs sampler based implementations. In the snippet the code is setup to use Metropolis based method</td>
<td>31</td>
</tr>
<tr>
<td>3.7</td>
<td>Code snippet from the <code>umbrella.namd</code> script demonstrating how collective variable biases are set.</td>
<td>32</td>
</tr>
<tr>
<td>3.8</td>
<td>Code snippet demonstrating how collective variable biases are set in the <code>alanin_base_umbrella.namd</code> ASyncRE script</td>
<td>32</td>
</tr>
<tr>
<td>3.9</td>
<td>Code snippet from the <code>umbrella.namd</code> NAMD script demonstrating how input files are set</td>
<td>32</td>
</tr>
<tr>
<td>3.10</td>
<td>Code snippet demonstrating how input files are set in the <code>alanin_base_umbrella.namd</code> ASyncRE script</td>
<td>33</td>
</tr>
<tr>
<td>3.11</td>
<td>Code snippet from the <code>umbrella.namd</code> script demonstrating implementation of the mechanism for exchanges between replicas</td>
<td>34</td>
</tr>
<tr>
<td>3.12</td>
<td>Code snippet demonstrating how resulting energy differences are calculated in the <code>alanin_base_umbrella.namd</code> ASyncRE script</td>
<td>35</td>
</tr>
</tbody>
</table>
3.13 Code snippet from the `umbrella.namd` script demonstrating how collective variable biases are re-evaluated after the exchange between replicas was accepted .................................................. 35

4.1 Comparison of NAMD and ASyncRE simulation speeds for RE with Temperature-Exchange simulations; varying numbers of replicas and CPU cores; fixed number of simulation time-steps (for NAMD); fixed simulation time (for ASyncRE) .................................................. 38

4.2 Comparison of NAMD and ASyncRE simulation speeds for RE with Umbrella Sampling simulations; varying numbers of replicas and CPU cores; fixed number of simulation time-steps (for NAMD); fixed simulation time (for ASyncRE) .................................................. 39

4.3 Simulation speeds for RE with TE and RE with US schemes using ASyncRE NAMD module; varying numbers of replicas and nodes; fixed simulation time .................................................. 41

4.4 Comparison of simulation run-times (in minutes) for Temperature Exchange and Umbrella Sampling NAMD RE schemes; varying simulation length; number of replicas is fixed at 12; number of CPU cores is fixed at 12; all runs are performed on one node .................................................. 42

4.5 Comparison of speeds for Umbrella Sampling NAMD simulations; varying numbers of replicas and CPU cores; fixed simulation length; for a "from start" graph each of the simulation runs was performed using initial input files; for a "continuing" graph first run was performed using initial input files, but subsequent runs were performed using output of the previous run .................................................. 43

4.6 Scaling results for ASyncRE/BigJob with the AMBER MD engine employing a serial quantum mechanical/molecular mechanical potential. The total number of cores allocated for the full REMD run was varied, while each simulation was run on a single core. As should be expected, performance (as measured in ns/day) increases linearly (red fit line) with the number of simulations running at any given point in time. It is perhaps relevant to note that the performance of individual simulations is also quite consistent at different core counts (green fit line), verifying that the linear scaling is in fact representative of an increase in the number of simulations running and not just an increased core count [33] ........ 45

4.7 Scaling results for ASyncRE with the NAMD MD engine using RE-US scheme; varying numbers of replicas and nodes; simulation speed calculated by dividing total number of simulation time-steps performed by all replicas, by total simulation run-time .................................................. 46

4.8 Average Compute Unit (sub-job) throughput using BigJob with Saga-Python on Kraken; throughput in Compute Units per second; number of CUs fixed at 192; Varying Pilot size (number of BigJob allocated cores) 47

4.9 Average Compute Unit (sub-job) queuing times using BigJob with Saga-Python on Kraken; number of CUs fixed at 192; Varying Pilot size (number of BigJob allocated cores) .................................................. 48
Chapter 1

Introduction

In recent years, a significant scientific progress has been made in understanding the inner workings of biological systems. This lead to an unprecedented increase in the amount of data required to capture atomic resolution structures and new sequences. Due to this phenomenon the role of the biomolecular simulations is of much greater significance now. For example, at the point when this dissertation was written, the protein data bank [1] had 92104 structures stored. The number of available structures doubled in the last five years. Availability of this data provides a solid experimental base for the MD simulations and enables scientists to research the relationships between structure and function. Unfortunately for the majority of proteins only amino-acid sequences are available [2]. This means that in order to obtain a detailed 3D structure of proteins, protein folding MD simulations must be performed.

A number of computational challenges are associated with protein folding Molecular Dynamics (MD) simulations. To perform an accurate and scalable integration, simulation timestep must be around 1 femtosecond (1 fs = $10^{-15}$ of a second). This requirement is motivated by the fact, that hydrogen atoms in a biomolecule vibrate with a period of approximately 10 fs [2]. At the same time, simulation runtime necessary to observe the desired changes would be at the range of microseconds (1 ms = $10^{-6}$ of a second). For example, just to allow a protein to relax into the nearest low-energy state an MD simulation must run for at least a couple of nanoseconds (1 ns = $10^{-9}$ of a second) [2]. Limitations described above mean that required runtime for protein folding MD simulations is approximately in the range from $1 \times 10^6$ to $1 \times 10^9$ timesteps.

In MD simulations a problem size is typically fixed - a number of atoms is constant and is at the range of several hundred thousands of atoms. Of course, the size of molecular assemblies is variable, but is nevertheless limited. This places another challenge - how to efficiently exploit increasingly faster and larger HPC systems in order to minimise the runtime of the MD simulation requiring millions of timesteps.

A replica-exchange method [3] - may be used for certain problems in order to increase sampling of molecular configurations. Another popular technique is to use an artificial steering forces [4] in order to steer the molecule through a series of transitions. In order to achieve accuracy of simulation results, steering forces used must be of a smallest
possible value, which still allows to produce the results in a timely manner. This means that increase in simulation times is required to achieve the desired accuracy. In addition to that, systems composed of larger molecules typically are more complex and consequently require an even longer simulation runs. Thus scaling MD simulations of such systems to thousands of CPU cores with every timestep requiring only a millisecond of wall clock time is a challenging task. This dissertation involves investigation of the job submission frameworks and Pilot Job Frameworks in particular to overcome this challenge.

1.1 Objectives

While working on this dissertation the following objectives were identified. The major objective of this dissertation is to implement a NAMD based Replica Exchange module in ASyncRE Python package. This module should support two RE schemes:

- RE with Temperature Exchange
- RE with Umbrella Sampling

Next objective is to analyse performance and scalability characteristics of the ASyncRE Python package. Another objective of this dissertation is to investigate BigJob performance on Cray machines - Hector [5] and Kraken [6]. In order to achieve these objectives a series of tasks were performed:

- Obtaining access to Kraken
- Installing and configuring Globus tools in order to use Kraken
- Correctly installing BigJob on Hector and Kraken
- Correctly installing ASyncRE Python package and related libraries on Kraken
- Learning how to use ASyncRE Python package to perform Replica Exchange with Umbrella Sampling simulations using Amber MD engine
- Identifying an application that will be used as an MD engine for the implementation of the extension module
- Understanding how to perform conventional Temperature Exchange and Umbrella Sampling RE simulations using NAMD
- Learning NAMD Tcl scripting
- Adapting NAMD configuration scripts to asynchronous RE simulations
- Implementing NAMD ASyncRE module for Temperature Exchange and Umbrella Sampling RE schemes
- Analyse performance and scalability characteristics of the implemented module and ASyncRE package in general
Investigating how limitations imposed by `aprun` can be mitigated on Cray machines

1.2 Structure of the dissertation

There are five chapters in this dissertation. First chapter is an introductory chapter. This chapter is designed to motivate this dissertation. In this chapter are briefly introduced Molecular Dynamics simulations and motivation for this dissertation is provided. In addition to that in first chapter are stated objectives of this dissertation. The next chapter is a background chapter. This chapter is aimed to provide enough information to understand the work what was done during the implementation phase. In this chapter is given an overview of the NAMD molecular dynamics code. In addition to that here are briefly described Replica Exchange simulations including the theory behind them. Chapter two also introduces the main software tools used in this dissertation - SAGA-Python, SAGA-BigJob and ASyncRE package. In chapter three are presented implementation details of the ASyncRE NAMD module. This implementation supports two Replica Exchange schemes - RE with temperature exchange and RE with Umbrella Sampling. Next chapter - chapter four, provides evaluation details of the ASyncRE NAMD module on Kraken HPC system. Performance and scalability evaluation in comparison with NAMD and ASyncRE Amber module is provided. In addition to that analysis of the BigJob on Cray machines is presented. The final chapter of the dissertation summarises accomplished work and provides the main conclusions. In chapter five are also included suggestions for the future work.
Chapter 2

Background

2.1 Nanoscale Molecular Dynamics - NAMD

NAMD [7] is a parallel molecular dynamics application which is aimed at simulations of biomolecular systems. NAMD can be used on various systems, starting with regular desktops and ending with the most powerful HPC systems available today. Often problem size of the biomolecular simulations is fixed and systems are analysed by performing a large number of iterations. This means that MD applications must be highly scalable. A parallelization strategy used in NAMD is a hybrid of force decomposition and spatial decomposition. In addition to that, NAMD uses dynamic load-balancing capabilities of the Charm++ parallel programming system. [2]

2.1.1 NAMD parallelization strategy

Parallelization in NAMD uses a hybrid strategy, consisting of spatial decomposition and force decomposition. This is complemented by the dynamic load-balancing framework of the Charm++ system. In Molecular Dynamics, calculation of the non-bonded forces between all pairs of atoms is the most computationally demanding. The algorithm for this calculation has a time complexity of $O(n^2)$. In order to reduce time complexity of this algorithm to $O(n \log n)$, the terms cutoff radius $r_c$, separation of computation of short-range forces and separation of computation of long-range forces were introduced [2]. For the atoms within the cutoff radius $r_c$ non-bonded forces are calculated on per atom basis. For the atoms outside the $r_c$, long-range forces are calculated using particle-mesh Ewald algorithm, which has a $O(n \log n)$ time complexity. Decomposition of MD computation in this way, results in calculation of non-bonded forces for atoms within the $r_c$ being responsible for 90% of total computational effort.

A well known problem, arising from parallelising MD simulations using as a base sequential codes, is poor scalability. Scalability is often measured using isoefficiency,
Figure 2.1: Schematic representation of NAMD parallelization mechanism [2]

which is defined as a rate of change of problem size required to maintain parallel efficiency of the increasingly parallel system. Isoefficiency is defined by:

\[ W = \frac{1}{t_c} \left( \frac{E}{1-E} \right) T_o \]  

(2.1)

or, if \( K = E/(t_c(1 - E)) \) is constant depending on efficiency:

\[ W = KT_o \]  

(2.2)

where:
- \( W \) - is problem size
- \( t_c \) - is cost of executing each operation
- \( E \) - is parallel efficiency; \( E = \frac{S}{p} \) where \( p \) is number of processors and \( S \) is speedup; \( S = \frac{T_1}{T_P} \) where \( T_1 \) is sequential runtime and \( T_P \) is total parallel runtime
- \( T_o \) - is total overhead

Small rate of change of problem size \( W \) means that in order to efficiently utilise increasing number of processors, relatively small increments in problem size are required. This means that parallel program is highly scalable. On the other hand, large rate of change of \( W \), indicates that parallel program scales poorly.

For MD simulations scalability is often limited by the rate of change of communication-to-computation ratio, which increases too rapidly while increasing the number of processors. This limitation leads to "large" isoefficiency function, meaning that weak scaling is poor for the particular code.

In order to avoid issues specified above, NAMD uses a hybrid parallelization strategy,
which is a combination of spatial decomposition and force decomposition. This strategy is aimed to allow increase of parallelism without proportional increase of communication cost. This strategy involves dividing a simulation space into patches - cubic boxes, size of which is calculated based on the cutoff radius $r_c$. The size of a patch is determined based on parameter $B$. The length of a patch in each dimension is $b = \frac{B}{k}$ and $B = r_c + r_H + m$, where $r_H$ is the maximum length of a bond to a hydrogen atom times two, $m$ is the margin equal double distance that atoms may move without being required to migrate between the patches and $k$ is a constant [2]. The value of constant $k$ is from the set \{1, 2, 3\}. A one-way decomposition can be observed when $k = 1$, which typically results in having from 400 to 700 atoms per patch. With $k = 2$ number of atoms per patch decreases to approximately 50 atoms.

With NAMD 2.0, to the spatial decomposition described above was added a force decomposition, forming a hybrid parallelization. NAMD force decomposition can be described as follows. A force-computation object or compute object is initialised for each pair of interacting patches. For example, if $k = 1$ the number of compute objects is fourteen time greater than the number of patches. Compute objects are mapped to processors, which in turn are managed by the dynamic load balancer of Charm++. This strategy also exploits Newton’s third law in order to avoid duplicate computation of forces [2]. A schematic representation of this hybrid parallelization strategy utilising patches and compute objects can be found in Figure 2.1.

NAMD is based on Charm++ - a parallel object-oriented programming system written in C++. Charm++ is designed to enable decomposition of computational kernels of the program into cooperating objects called chares. Chares communicate with each other using asynchronous messages. The communication model of these messages is single sided - a corresponding method is invoked on a compute object only then a message is received and no resources are spent on waiting for incoming messages (e.g. posting a receive call). This programming model facilitates latency hiding, while demonstrating good resiliency to system noise. A processor virtualization feature of Charm++ allows the code to utilise a required number of compute objects in order to meet it’s requirements. Typically several objects are assigned to a single processor and execution of objects on processors is managed by the Charm++ scheduler. Scheduler performs a variety of operations, including getting messages, finding destination chare of each message, passing messages to appropriate chares and so on. Charm++ can be implemented on top of MPI without using its functionality, which makes NAMD a highly portable program.

### 2.2 Replica Exchange

While undertaking simulations of complex systems such as proteins, using Molecular Dynamics (MD) or Monte Carlo (MC) methods, it is problematic to obtain accurate canonical distributions at low temperatures [8]. Replica Exchange Markov chain Monte Carlo (MCMC) sampling or Parallel Tempering is a widely used simulation technique,
aimed to improve efficiency of MCMC method, while simulating physical systems. This technique was devised as early as 1986 by Swendsen and Wang [9], but Molecular Dynamics version of the method, which is known as Replica-Exchange Molecular Dynamics (REMD) was first formulated by Sugita and Okamoto [10] in 1999. Today parallel tempering is applied in many scientific fields including chemistry, physics, biology, materials science and other.

In Parallel Tempering simulations $N$ replicas of the original system are used to model phenomenon of interest. Typically, each replica can be treated as an independent system and would be initialised at a different temperature. While systems with high temperatures are very good at sampling large portions of phase space, low temperature systems often become trapped in local energy minima during the simulation [11]. Replica Exchange method is very effective in addressing this issue and generally demonstrates a very good sampling. In RE simulations, system replicas of both higher and lower temperature sub-sets are present. During the simulation they exchange full configurations at different temperatures, allowing lower temperature systems to sample a representative portion of phase space.

It is obvious that running a simulation of $N$ replicas of the system would require $N$ times more compute power. Despite that fact, RE simulations are proven to be at least $1/N$ times more efficient than single temperature simulations. This is achieved by enabling replicas with lower temperatures to sample phase space regions, not accessible for them in case of regular Monte Carlo simulation, even if it would run $N$ times longer than RE simulation involving $N$ replicas. In addition to that, RE method can be very efficiently mapped to distributed-memory architectures of parallel HPC machines like HECTOR. Implementation details of the mechanism for performing swaps of configurations between replicas significantly influence efficiency of the method. Issues to consider are: how often should exchanges take place, what is the optimal number of replicas, what is the range of temperatures, how much of a compute power should be dedicated to each replica. In addition to that, in order to prevent the growth as $\sqrt{N}$ of a system with $N$ replicas, solutions on how to swap only a part of the system are required [11].

It is worth noting that RE simulations may involve tempering of parameters other than temperature. For example, in some cases simulations where chemical potentials are swapped may demonstrate a better efficiency. A special case of RE method is defined by configuration swaps in a multi-dimensional space of order parameters. This is typically referred as multi-dimensional replica exchange. Thanks to better sampling of phase space enabled by the use of RE method, inconsistencies in some widely used force fields were discovered [11]. It is clear now, that RE method has established its important role in testing of new force fields for atomic simulations.
2.2.1 Theory of Replica Exchange Markov Chain Monte Carlo (MCMC) sampling

In standard RE simulation each of $N$ replicas is assigned a different temperature $T_i$ and $T_1 < T_2 < \ldots < T_N$. Here $T_1$ is a temperature of the simulated system. The partition function of the ensemble is:

$$Q = \prod_{i=1}^{N} \frac{q_i}{M!} \int dr_i^M \exp \left[ -\beta_i U(r_i^M) \right]$$  \hspace{1cm} (2.3)

where:

$q_i = \prod_{j=1}^{M} (2\pi m_j k_B T_i)^{3/2}$ is from integrating out the momenta

$m_j$ is the mass of atom $j$

$r_i^M$ defines the positions of the M particles in the system

$\beta_i = 1/(k_B T_i)$ is the reciprocal temperature

$U$ is the potential energy or the part of the Hamiltonian that does not involve the momenta

In case if for all conditions a probability of an exchange is equal, the probability what an exchange between replica $i$ and $j$ would be accepted is:

$$A = \min \left\{ 1, \exp \left[ + (\beta_i - \beta_j) \left( U(r_i^M) - U(r_j^M) \right) \right] \right\}$$  \hspace{1cm} (2.4)

where:

$j = i + 1$, meaning that exchanges are attempted between replicas with "neighbouring" temperatures

Replica Exchange method facilitates any set of Monte Carlo moves for a single temperature system, where single-system moves take place between exchanges [11]. An example of a typical series of exchanges and single-temperature Monte Carlo moves is depicted in Figure 2.2.

If we assume that system is having Gaussian energy distributions, average acceptance rate $\langle A \rangle$ can be defined as:

$$A = \text{erfc} \left[ \left( \frac{1}{2} C_v \right)^{1/2} \frac{1 - \beta_j / \beta_i}{(1 + (\beta_j / \beta_i)^2)^{1/2}} \right]$$  \hspace{1cm} (2.5)

where:

$C_v$ is the heat capacity having a constant volume in a temperature range from $\beta_i$ to $\beta_j$

In the equation above the average acceptance rate varies based on the probability that replica having a higher temperature, arrives to the phase space region relevant to replicas at the lower temperatures. The above formulation of the average acceptance rate plays the important role in decision making process regarding the choice of the replica temperatures in the Parallel Tempering simulation.
2.2.2 Theory of Molecular Dynamics Parallel Tempering

In contrast to the Monte Carlo RE simulations, where only positions of the particles are taken into account, Molecular Dynamics RE simulations must also consider the momenta of all the particles defining the system of interest. According to the REMD method devised by the Sugita and Okamoto the momenta of the replica \( i \) after the successful exchange can be calculated by:

\[
p^{(i)'} = \sqrt{\frac{T_{\text{new}}}{T_{\text{old}}}} p^{(i)}
\]  

(2.6)

where:
- \( p^{(i)} \) is the old momenta for replica \( i \)
- \( T_{\text{old}} \) is replica temperature before the exchange
- \( T_{\text{new}} \) is replica temperature after the exchange

The above equation ensures that the average kinetic energy can be calculated as: \( \frac{2}{3} N k_B T \).

The probability that the exchange will be accepted is the same as for the Monte Carlo RE method and can be calculated using equation (1.2).

2.2.3 Optimal Choice of Temperatures

In order to maximize sampling quality and to minimize required computational effort, it is vital to correctly choose the temperatures for the replicas as well as the number of
replicas involved in the RE simulation. The choice of temperatures is motivated by the need to avoid the situation when some of the replicas are "trapped" in local energy minima. At the same time, it is also important to have enough replicas, for the exchanges to be performed between all "neighbouring" replicas. From the equation (1.2) and Figure 2.3 it is obvious that in order for exchanges to be accepted, energy histograms of adjacent replicas must have some overlapping parts. It was demonstrated that using a geometric progression for temperatures, so that \( \frac{T_i}{T_j} = \text{constant} \) for the systems having a constant volume heat capacity \( C_v \) for the whole temperature range, allows to achieve equal exchange acceptance probability for all participating replicas [11]. Various studies [12] [13] showed that exchange acceptance probability around 20% results in best possible performance of the RE simulation.

A great care should be taken then analysing the results of the REMD simulations. It is important always to remember that REMD simulations simply allow us to perform a variant of phase space sampling, meaning that we cannot make any assumptions about the dynamics of the analysed system, since performed exchanges are not physical processes.

### 2.2.4 Mathematical model for total simulation runtime

In order to determine the total runtime of the RE simulations A. Thota et. al. [14] defined a mathematical model which is described in this subsection.

First, the total time to completion of the simulation is formulated for the idealistic model. In this case, it is assumed that the total time to completion is equal to the parallel runtime of the array of replicas. Coordination and synchronization time is not
taken into account. Another assumption is the pure homogeneity of the used hardware components, such as interconnect, memory and compute. For this model the total time to completion is:

$$T = \frac{1}{p} \times \left( T_{MD} \times \frac{N_X}{N_R/2} \right)$$  \hspace{1cm} (2.7)

where:

- $N_R$ - is the total number of replicas
- $N_X$ - is the total number of pairwise exchanges
- $T_{MD}$ - is replica runtime required to complete a pre-defined number of simulation time-steps
- $p$ - is the probability of successful exchange between replicas

An established way of deciding on the proposed change of state - Metropolis scheme [15] is used to make a decision if the exchange will be accepted or not.

- $N_R/2$ - is the number of independent exchange events that can occur concurrently for $N_R$ replicas
- $N_X/(N_R/2)$ - is the number of simulation runs needed to complete $N_X$ exchanges

For example, if we have the total number of replicas $N_R = 4$ and $N_X = 16$, then:

- $N_R/2 = 2$ - only two exchanges are possible after each simulation run
- $N_X/(N_R/2) = 16/2 = 8$ - in total eight simulation runs would be required in order to perform 16 pairwise exchanges

After an idealistic model have been defined, we provide a more realistic formulation of total time to completion, involving $N_X$ replica exchanges:

$$T = \frac{1}{p} \times \left[ \left( T_{MD} \times \frac{N_X}{N_R/2} \right) + (T_{EX} + T_W) \times \frac{N_X}{\eta} \right]$$  \hspace{1cm} (2.8)

where:

- $T_{EX}$ - is the time to perform a pairwise exchange: $T_{EX} = T_{find} + T_{file} + T_{state}$
- $T_{find}$ - is the time to find a partner
- $T_{file}$ - is the time to manipulate files (write to file and/or exchange files, transfer files)
- $T_{state}$ - is the time required to perform state updates in a central database (or by some other means) and to perform book-keeping operations associated with replica pairing/exchanging; this parameter is highly implementation dependent
- $T_W$ - is the time spent by replica waiting for other replicas to perform predetermined number of time-steps, so the synchronization can take place
- $\eta$ - specifies how many exchanges can be performed concurrently; $\eta_{max} = N_R/2$

From equation (1.2) we can see, that increase of replica runtime $T_{MD}$ results in time for coordination being responsible for a smaller proportion of the total time to completion.

### 2.2.5 Synchronous replica exchange

The conventional model of exchanges between replicas is a synchronous model. The majority of the well known RE algorithms are based on this model. Synchronous model
can be characterised as follows. All participating replicas are required to reach a certain state before exchanges can occur. For the Molecular Dynamics simulations this often is defined as a number of simulation steps, but in principle can be specified by some other conditions, such as potential energy. The number of replicas participating in exchanges is fixed and is equal to \( N_R / 2 \), where \( N_R \) is total number of replicas. This means that coordination in the Synchronous RE model occurs periodically at defined intervals. It is important to note that pairs of replicas participating in exchanges are determined before all replicas are ready for exchanges. When all replicas are ready for exchanges, e.g. required number of MD simulation steps is performed, an attempt to exchange thermodynamic and/or state parameters is made according to the Metropolis scheme. If the attempt is successful then exchange of parameters takes place.

There are two major limitations of this model. First, number of replicas participating in exchanges at each coordination step is fixed. Second, replicas are paired together for exchanges before they reach a qualifying state. While time for finding an exchange partner \( T_{\text{find}} = 0 \), the number of possible exchange partners is severely limited. The result is that the number of exchanges between replicas with highly varying values of thermodynamic or state parameters is confined. The effect of this is reduced probability of crosswalks. The crosswalk is the event, which is defined by the replicas with low starting temperature, first reaching the upper temperature range and then returning to a lower temperature range [16].

Rigid replica pairing mechanism of the synchronous RE model not only imposes limitations from the modelling perspective, but also from the performance perspective. This RE model demonstrates good performance only on homogeneous systems. Performance of synchronous RE model on heterogeneous systems suffers due to differences in performance of individual components and fluctuations in availability of compute resources.

### 2.2.6 Asynchronous replica exchange

In contrast to the synchronous models, in asynchronous RE models [16], [17] replicas are not required to wait for all other replicas to reach a certain state for exchanges to occur. Each replica qualifies for exchange as soon as it reaches a certain state. Exchange attempt is performed with any other replica that has reached that state as well.

In the context of this dissertation synchronous RE is associated with static pairing, but asynchronous RE with dynamic pairing. It is important to point out that this is simply an assumption and not a requirement, since it is possible to have synchronous RE with dynamic pairing - exchanges occur periodically at defined intervals, but with different replica each time, or to have asynchronous RE with static pairing - replica pairs are fixed and predetermined, but there is no synchronization of exchanges.

This means that for asynchronous RE models \( T_{\text{find}} \neq 0 \), since replicas are paired on the fly, but \( T_W = 0 \), since there is no wait time associated with synchronization.

The effectiveness of Asynchronous RE is not absolute. There may exist models of the physical systems for which synchronous RE is more effective and delivers a faster
convergence of the solution.

2.3 Pilot-Jobs

Pilot-Job is an established term in Distributed Computing for multilevel scheduling constructs. A Pilot-Job construct encloses in itself an array of smaller (possibly coupled) jobs. These smaller jobs are often referred to as sub-jobs. Pilot-Job is treated by the scheduling system of the target resource as a single batch job. Pilot-Jobs are aimed to:
- reduce the total time to completion for a series of smaller jobs. Here time to completion consists of two components: queue waiting time and total execution time of enclosed sub-jobs
- improve utilisation of the computational resources on target systems
- simplify the process of setting up and running complicated scientific simulations

In past decade the concept of Pilot-Job became increasingly popular, leading to emergence of numerous implementations. Some of the most widely known Pilot-Job frameworks are: BigJob [18], DIANE [19], Condor-G [20], Falkon [21] and DIRAC [22]. Typically acquisition of required compute resources is performed by the Pilot-Job framework itself, followed by the submission of the sub-jobs to the target system according to some sort of scheduling algorithm. This approach allows to significantly minimise total time to completion of the sub-jobs, since queue waiting time is neglectable and execution occurs in parallel.

The need for Pilot-Jobs arises from the fact, that often scheduling systems assign a very low priority to the jobs requiring a small number of processors. There are many scientific simulations that require submission of very large numbers of such small jobs, meaning that time to completion of such simulations using conventional job submission techniques would be immensely long. Situation then number of such small jobs is much greater than number of allocatable processors is quite common. In addition to that Pilot-Jobs allow to effectively manage complex job workflows, including chained and/or coupled ensembles of sub-jobs. Flexibility of Pilot-Jobs makes them a good candidate for applications designed to perform Replica Exchange simulations. In particular, support of complicated workflows provided by Pilot-Jobs, is an important feature, required for implementations of asynchronous RE.

One of the very important features of the Pilot-Job frameworks is support for dynamic resource addition. Since resource allocation occurs at the application level, it is possible to tune the framework for the specific application. Execution process can not only be monitored, but also adjusted in order to achieve a better time to completion. It is important to note that submission of the sub-jobs through user provided script does not guarantee immediate execution on the target system. On the other hand it is possible to configure the framework to facilitate execution of sub-jobs even before user-specified amount of resources is allocated, or to allocate additional resources for a better load balancing between sub-jobs.

Despite the fact that the Pilot-Job concept is fairly established, the industry standard model for Pilot-Job frameworks doesn’t exist yet. Design models for Pilot-Job frame-
works often have substantial differences due to the fact that these frameworks are designed for a particular underlying architecture. As the result of that, portability and extensibility of such frameworks is limited. In addition to the differences in the design models, many Pilot-Job implementations introduce new definitions and concepts, making it almost impossible to compare them in a meaningful way. In order to address this issue, Luckow et al. proposed a P* Model of Pilot Abstractions, aimed to be a unified standard model suitable for various target architectures and frameworks [23]. The main goal of P* Model is to provide a convenient means to qualitatively compare and analyse various Pilot-Job frameworks. The P* Model consists of the following elements:

- Pilot (Pilot-Compute) - is the actual job that is submitted to scheduling system of the target resource; Through the Pilot-Compute is provided application level control of target resources
- Compute Unit (CU) - is a compute task defined by the application that is executed by the framework; there is no notion of target system defined at the Compute Unit
- Scheduling Unit (SU) - is an internal framework unit; SUs are not accessible at the application level; First Compute Unit is acquired by the framework, then it is assigned to Scheduling Unit
- Pilot Manager (PM) - is the most sophisticated part of the P* Model; Pilot Manager is responsible for management of interactions between all other units of the P* Model (Pilots, CUs and SUs); once resources are acquired by the Pilot-Job, it is responsibility of the Pilot Manager to perform assignment of the resources internally; This may include decisions of how many CUs are assigned to SUs; then SUs are submitted to the target architecture, Pilot Manager decides how many cores are assigned to each SUs or how SUs are ordered or grouped

2.4 SAGA-Python

Simple API for Grid Applications (SAGA) is a high-level programming interface, designed to simplify access to the most commonly used functionality of the distributed computing resources. SAGA forms an infrastructure independent access layer to various functions of distributed systems such as file transfer, job scheduling, job monitoring, resource allocation and other. SAGA interface can be used on a variety of computing resources such as Clouds, Grids and HPC clusters. A POSIX-style API provided by SAGA can be seen as an easy to use interoperability layer for heterogeneous distributed architectures and middle-ware, that efficiently abstracts the user from the system specific details. SAGA programming interface is mainly focused on the developers of tools, applications and frameworks for various distributed infrastructures. SAGA has a strong focus on becoming an industry standard, which is facilitated through the Open Grid Forum (OGF) - an international standardization body focused primarily on the steering evolution and adoption of the best practises in the field of distributed computing. Currently there are four SAGA implementations:
From the above four SAGA implementations, main focus of this dissertation is on SAGA Python, since this API implementation is used as an interoperability layer by BigJob. Key packages of SAGA interface are:

- File package - provides means for accessing local and remote file-systems; performing basic file operations such as delete, copy and move; setting file access permissions; file package also provides support for various i/o patterns; classes of this package are syntactically and semantically POSIX oriented; this may lead to performance issues in case if many remote access operations are performed; performance issues are addressed by implementing a number of functions in a similar way as in GridFTP specification

- Replica package - provides functions for the management of replicas; this includes access to the logical file-systems; means for basic operations on the logical files such as delete, copy and move; means for modifying logical entries; functionality to search for logical files using attributes; a logical file is simply an entry in the name space described by the associated meta data; logical file (entry) has a number of physical files (replicas) associated with it

- Job package - provides functionality for submission of the jobs to a target resource; means for job control (e.g. cancel(), suspend(), or signal() a running job); for status information retrieval for both running and completed jobs; jobs can be submitted in batch mode or interactive mode; functions for local and remote job submission are provided as well; jobs package is heavily based on the DRMAA-WG [24] specification with many parts being directly borrowed from it

- Stream package - provides functionality to launch remote components of a distributed application; this requires establishing a remote socket connection between participating components; stream package allows to establish TCP/IP socket connections using simple authentication methods with support for application level authorisation; streams package is not performance-oriented, but API design allows implementing user-defined performance-oriented functions; while significantly reducing challenges associated with establishment of authenticated socket connections, functionality of streams package is very limited, making it suitable mostly for moderately sophisticated applications

- Remote Procedure Call (RPC) package - implements a GridRPC API defined by the Global Grid Forum (GGF); GridRPC specification was directly made available as a part of the RPC package with only slight modifications required to ensure conformance to the SAGA API specification; initialisation of the remote function handle typically involves connection setup, service discovery and other
procedures; remote procedure calls can return data values; it is important to note
that asynchronous call functionality of the GridRPC interface is not provided in
this package, but is specified in the SAGA Task Model

SAGA-Python is a light-weight Python implementation of SAGA, which fully conforms
to the OGF GFD.90 SAGA specification. SAGA-Python is fully written in Python.
Schematic representation of the SAGA-Python architecture can be found in Figure 2.4.
The main design goals of SAGA-Python are usability, flexibility and reduced deploy-
ment time. A very simple scripting techniques allow direct use of SAGA-Python by
users in order to facilitate execution of tasks on target resource. One of the most im-
portant features of SAGA-Python is dynamically loadable adaptors, which provide an
interface to an array of scheduling systems and services. SAGA-Python architecture is
very flexible in a sense that it is relatively easy for application developers to implement
their own adaptors, in case if target middle-ware is not supported. SAGA-Python pro-
vides support for various application use-cases starting with ensembles of uncoupled
tasks and ending with complex task workflows. SAGA-Python has been used in con-
junction with various scientific applications on well known distributed infrastructures
such as XSEDE, FutureGrid, LONI and OSG.

SAGA-Python adaptors can be used to access target resources locally or remotely. For
remote job submission can be used either ssh or gsissh, while local jobs must be
submitted using fork scheme. The following adaptors are currently supported by
SAGA-Python:

- Shell job adaptor - provides functionality to submit jobs to local and remote re-
sources using shell command line tools; there is a requirement for login shell of
the resource to be POSIX compliant; adaptor provides means to specify custom
POSIX shell by:

  \[ js = saga.job.Service ("ssh://remote.host.net/bin/sh") \]

  which creates a job service object representing a target resource; remote job sub-
  mission is supported through ssh:// and gsissh:// url prefixes; for local
  job submission using bin/sh.fork:// must be specified

- Condor adaptor - is designed for job execution and management on a Condor
gateway; job service object supports fork:// and local:// url prefixes for
local jobs; ssh:// and gsissh:// prefixes are supported for submission of
remote jobs (e.g. condor+gsissh://); adaptor’s runtime behaviour can be
controlled through configuration options

- PBS adaptor - provides functionality to execute and manage jobs using Portable
Batch System (PBS) and TORQUE Resource Manager; for local job submission
pbs:// prefix should be passed as a part of the url scheme to the job service
object; addition of +ssh or +gsissh to the existing prefix allows for remote
job submission (e.g. pbs+ssh://)

- SGE adaptor - is designed for submission of the jobs to Sun Grid Engine (SGE)
controlled environments; in order to submit a job locally to SGE, sge://
prefix must be specified in the url scheme of the job service object; remote job submis-

16
Figure 2.4: Schematic representation of the SAGA-Python architecture [25]

- SLURM adaptor - allows to run and manage jobs through a SLURM [26] open-source resource manager; url prefix for this adaptor is slurm://; local and remote jobs are supported similarly as for PBS and SGE adaptors.

In addition to the above adaptors allowing to access various resource management systems, SAGA-Python provides two adaptors for file and data management:

- Shell file adaptor - allows to access local and remote file-systems and to transfer files using shell command line tools; for access of remote file-systems the following schemes are supported: sftp using url prefixes ssh:// or sftp://, gsiftp using url prefixes gsiftp:// or gsissh://; local file-systems can be accessed using bin/sh with prefixes file:// or local://

- HTTP file adaptor - provides data transfer capability from remote file-systems using HTTP/HTTPS protocols; to use adaptor, url prefixes http:// or https:// must be specified, e.g.:  
  remote_file = saga.filesystem.File('http://site.com/file')

## 2.5 SAGA-BigJob

BigJob is a general purpose Pilot-job framework aimed at simplifying job scheduling and data management on the target resource. BigJob can be used on various heterogeneous systems including clouds, grids and HPC clusters. BigJob is fully written in
Python in order to enable it to use SAGA-Python as an interoperability layer for underlying architectures and resource managers. In addition to that BigJob natively supports parallel MPI applications. Through simple scripting techniques BigJob provides convenient user-level control of Pilot-Jobs.

Three main design goals of BigJob are: flexibility, extensibility and usability. BigJob is used in tandem with various scientific and engineering applications for solving of real world problems. Examples of BigJob use-cases include replica-exchange simulations, protein folding simulations, bioinformatics simulations and other.

As depicted in Figure 2.5, at a high-level BigJob architecture is composed of three main components: BigJob Manager, BigJob Agent and Advert Service.

- **BigJob Manager** - is a central part of the framework; BigJob Manager serves as an entry point into BigJob, providing abstraction layer for Pilots; as the name suggests, this component is responsible for creation, monitoring and scheduling of pilots; BigJob Manager ensures that sub-jobs are correctly launched on the resource, using provided job-id and user-specified number of cores

- **BigJob Agent** - is a framework component representing the actual pilot-job, running on the respective resource; BigJob Agent can be seen as the application-level resource manager, allowing job submission to target system; BigJob Agent is responsible for allocation of the required resources and for submission of sub-jobs

- **Advert Service** - is designed to provide means of communication between BigJob Manager and BigJob Agent; in current version of BigJob (0.4.130-76-gd7322desaga-python) Advert Service uses a Redis server [27] - an open-source key-value data store; this data store is used by BigJob Manager to create entries for individual sub-jobs, which are then retrieved by the BigJob Agent; Advert Service is also sometimes referred to as Distributed Coordination Service

The overall process of job submission by BigJob consists of the following seven steps (Figure 2.5):

1. **BigJob (Pilot-Job) object is described (using *compute_unit_description* in a user-defined script) and initialised at BigJob Manager**
2. **BigJob Manager submits Pilot-Job to the scheduling system of the respective resource**
3. **Pilot-Job, submitted by BigJob Manager, runs a BigJob Agent on that resource and specified number of compute nodes is requested**
4. **Sub-Jobs, specified in a user-defined Python script are submitted to BigJob Manager**
5. **BigJob Manager creates an entry for each sub-job at Advert Service (Redis server)**
6. **If required resources are available, BigJob Agent retrieves sub-job entries from the Advert Service**
7. BigJob Agent submits sub-jobs to scheduling system of the respective resource

The above steps describe only one "cycle" of the Pilot-Job execution process. Typically the number of sub-jobs is much greater than allocated resources. The BigJob Agent is querying Redis server for sub-jobs until there are no entries left. After a new sub-job has been retrieved, BigJob Agent checks if previous job has finished executing and submits new sub-job to scheduling system. If previous job haven’t finished executing, BigJob Agent queues sub-job.

Current version of BigJob (0.4.130-76-gd7322de-saga-python) supports most widely used of SAGA-Python adaptors, including Shell job adaptor, PBS adaptor and SGE adaptor. In addition to that BigJob provides a GRAM adaptor, allowing to use Globus tools to submit jobs and Torque+GSISSH adaptor allowing to submit jobs using gsish. Internally Pilot-Jobs are represented by the PilotCompute objects, returned by the PilotComputeService. The PilotComputeService object creates a Pilot-Job based on the set of parameters supplied through use of the Python 'dictionary' PilotComputeDescription, which contains the following parameters:

- **service_url** - specifies adaptor to use on the respective resource
- **number_of_processes** - the number of processes required for pilot to execute all sub-jobs
- **processes_per_node** - how many cores of each node to use
- **allocation** - project allocation number or assigned budget
- **queue** - name of the job queue to be used
- **working_directory** - specifies path to the directory to which all output files will be redirected
- **walltime** - specifies time (in minutes) required for pilot to execute all enclosed sub-jobs

These are the most important parameters, required for all BigJob adaptors. In some cases, specifying additional parameters in the PilotComputeDescription may be required, depending on the system configuration, chosen adaptor and other factors. Basic example script can be found in Appendix A.

2.6 ASyncRE package

AsyncRE [28] is a software tool which allows scientists to undertake asynchronous parallel replica-exchange molecular dynamics simulations. AsyncRE is fully written in Python. Data exchange during the simulation is file based. AsyncRE package can be used on various HPC systems where job submissions are managed by the schedulers, such as Sun Grid Engine (SGE), Portable Batch System (PBS) and other. AsyncRE package is developed as an extension of the SAGA-BigJob. This means that job submission, scheduling and monitoring is performed using BigJob routines. The main
Figure 2.5: Schematic representation of the BigJob architecture [29]

functionality of the package, such as job scheduling using BigJob routines, parsing of
the input parameters and implementation of the algorithms for data exchange between
replicas is defined in a core module of the package - *pj_async_re.py*. Package is
highly extensible - developers are provided with the means to write extension modules
for arbitrary MD engines and RE schemes, such as temperature-exchange, Hamiltonian
and other. In addition to that package supports development of the modules based on the
multidimensional RE schemes. At the time of this writing *ASyncRE* package included
modules for two MD engines - Amber [30] and Impact [31]. Supported schemes are -
multidimensional RE Umbrella Sampling with Amber and Binding Energy Distribution
Analysis Method (BEDAM) alchemical binding free energy calculations with Impact.
In general terms RE simulation using *ASyncRE* package involves the following steps.
First, input files and executables are configured, according to the requirements of the
MD engine and RE scheme of the module in use. This involves parsing template in-
put files which include thermodynamic and potential energy parameters for a set of
replicas. In addition to that, for each replica is set up a separate folder in a working
directory, where intermediate and final results of the simulation for each replica are
placed. *ASyncRE* uses a very simple naming convention for these directories - *r0, r1, ..., r<N>*
where *N* is the total number of replicas.

After simulation set up is done, a sub-set of replicas is submitted to BigJob and state
of these replicas is changed to running state 'R’. After a replica finishes execution on
a target resource, its state is changed to waiting state 'W’, which means that replica
qualifies for parameter exchange with other replicas and for the next execution run. Ex-
ecution of the replica on the target resource is called a 'cycle’, the length of which is
specified in the input file supplied to the MD engine. Submission of sub-sets of replicas
to BigJob occurs periodically, until the simulation is done. Replicas in the waiting state exchange thermodynamic and/or other parameters according to the rules specified in the currently used ASyncRE module. The exchange rules may be based on the temperature values or potential energy settings, complemented by the structural and energetic information from the MD engine output files [32]. One of the central data structures of the ASyncRE package is status Python dictionary. The main aim of this data structure is to provide an easy access to a number of parameters associated with each replica, such as thermodynamic state, current cycle number, replica status and other. For each replica this data structure is periodically check-pointed to a pickle file called <basename>.stat. During replica restarting process status dictionary is restored using this pickle file. This allows to proceed with calculation from the point where it was stopped during previous run. In addition to that this mechanism allows to continue the simulation after it has finished. Simulation time is defined by the number of wall-clock time minutes specified by the user. It is worth mentioning that ASyncRE package demonstrated respectable level of robustness. Simulations involving 100’s of replicas and running for several days are regularly performed by the user communities on XSEDE [33].

In order to hide latencies associated with orchestration of data exchanges between replicas, in ASyncRE package is used a ’run buffer’. It is important to mention that replicas (sub-jobs) are submitted to BigJob by ASyncRE package independently of the BigJob resource allocation process. This means that there are no guarantees that replicas submitted to BigJob will be executed on the target resource in a certain time frame. After replica is submitted to the BigJob, it is packaged into a BigJob Compute Unit, which is a data structure representing sub-jobs internally. Compute Units are executed on the target system as soon as required resources are allocated. The main purpose of the ’run buffer’ is to ensure that there are no additional latencies associated with BigJob waiting for replicas to finish data exchanges.

Currently Amber module provides support for the two Amber engines: SANDER and PMEMD. It is possible to use MPI executables with ASyncRE package. MPI executables are automatically recognised when SUBJOB_CORES parameter is greater than one. The ASyncRE package provides a means for users to implement their own modules by writing derivative classes of the core package class async_re_job. In order to do this developers must implement a small set of functions.
Figure 2.6: Schema of the system of software layers used for MD simulations
Chapter 3

Implementation

3.1 ASyncRE NAMD module

This dissertation project involved the development of the extension module for the ASyncRE package, which would provide support for another MD engine. At the initial stage of the project the following candidate applications were considered - Gromacs [34], LAMMPS [35], NAMD [7] and Desmond [36]. Before deciding which application will be used as an MD engine for the extension module, a number of issues were taken into consideration. These issues are:
- Is the candidate application currently installed on available HPC systems?
- How much does it cost to obtain a license?
- Is the candidate application widely used and well known?
- How difficult it would be to learn to use this application?
- What is the availability of support materials (tutorials, user manuals, wiki’s, etc.) for RE simulations?
- How difficult it is to setup, run and modify RE simulations?
After evaluating application scores on the above questions, two applications appeared to be more competitive than others - Gromacs and NAMD. It was decided first to implement an extension module for NAMD and if time will allow it, a module for Gromacs. For NAMD ASyncRE module two schemes were selected: RE with temperature exchange and RE with Umbrella Sampling. First schema was selected due to its simplicity. Implementing a simple schema first typically is a good idea, since it allows to gradual increase of the code complexity during development phase. In addition to that, implementing a temperature exchange schema first, allows to check correctness of the algorithm of core ASyncRE module. Umbrella Sampling RE schema was selected in order to compare it with currently implemented RE-US module for Amber MD engine.
3.1.1 RE simulations with NAMD

NAMD tutorials provide information only on how to run RE temperature-exchange simulations. While there is no information on how to set up and run simulations for other schemes, lib/replica folder of NAMD 2.9 installation contains input and configuration files for one-dimensional and two-dimensional RE Umbrella Sampling schemes. These configuration scripts relatively easily can be modified and adapted for the chosen HPC system.

Typically NAMD RE simulation is set up using three configuration files:

- **base_file.namd** configuration file specifying most basic simulation parameters, such as number of steps per simulation cycle, time-step size and other. Also, in this file molecular structure (.psf), coordinates (.pdb) and parameters (.params) files are specified.

- **simulation.conf** configuration file defines main RE simulation control settings such as the number of replicas, upper and lower temperature limits, number of runs, how often simulation must be restarted and other. In this file are specified paths for replica output files and a pointer to **base_file.namd** file is set.

- **replica.namd** is the central part of the NAMD RE simulation. In this file are specified all details of the RE simulation, including implementation of the Metropolis scheme, MPI calls for data exchanges between replicas, details of the RE algorithm, simulation restart settings and other.

As we can see above, NAMD configuration files can be in .namd or .conf format. While it is possible to pass multiple configuration files to NAMD as command line arguments, typically only one configuration file is passed. In this file are supplied links to other configuration files. NAMD changes its working directory to the directory of the supplied configuration file, before the file is parsed. This means that things get complicated if several supplied configuration files are not placed in the same directory. At the same time, placing all configuration files in one directory is very convenient, since all file paths are relative to that directory, making file paths setup very straightforward.

NAMD configuration files are defined using Tcl scripting language.

Until recently, NAMD configuration files were order-independent, meaning that whole file (or set of files) were parsed before the simulation was run. Due to introducing support for more advanced simulations, the following order dependency rules were introduced for NAMD configuration files. NAMD files are order-independent until the point where one of the "action" commands (such as run) are specified. Then, input files are read and simulation is run. From this point configuration file is order-dependent and only a sub-set of commands may be specified. This change means that an extra care is required then creating or modifying existing simulation setup.

3.1.2 High-level design of the ASyncRE NAMD module

At a high-level, design of the NAMD module is very simple. The core class of NAMD module - **pj_namd_job (namd_async_re.py)** file is directly derived from the
The core ASyncRE class `async_re_job` (in `pj_async_re.py` file). In `pj_namd_job` class are defined:

- `_launchReplica()` function - in this function are specified input, error and log files for replicas; in this function is specified `compute_unit_description` describing BigJob Compute-Unit (sub-job) execution parameters, such as path to NAMD executable, working directory etc.; this function returns a Compute-Unit object

- `_hasCompleted()` function - as the name suggests, this function is checking if a simulation cycle has completed; this is done by checking for presence of `.coor`, `.vel`, `.xsc` and `.history` files; the main logic behind checks for existence of multiple output files is to verify that the simulation not only has completed, but has completed successfully; this is a generic function - it is used for both implemented RE schemes, despite the fact that output files for these schemes are slightly different; `_hasCompleted()` is a Boolean function, it is called multiple times during the simulation (as a part of `updateStatus()` function), not only after the wall clock time for simulation run has elapsed, but also before any jobs are submitted to BigJob - in order to build input files for individual replicas

- I/O functions - a number of functions for extracting simulation data from the NAMD output files for both RE schemes; in more detail these functions will be described later in this chapter

Python classes implementing both RE schemes are directly derived from the `pj_namd_job` class. NAMD module class inheritance diagram is provided in Figure 3.1.

### 3.1.3 Adapting Temperature Exchange RE NAMD simulation setup for ASyncRE NAMD module

In order to use NAMD as an MD engine for the ASyncRE module, NAMD RE simulation setup scripts must be modified. For the temperature exchange RE schema it was done as follows.

Although not a requirement, it was decided to merge four scripts of conventional NAMD simulation into one, in order to reduce amount of file manipulation operations and to simplify the simulation setup. As noted above, `replica.namd` script is the central part of the NAMD temperature exchange RE simulation setup. In this script is implemented algorithm for synchronous RE, where exchange acceptance probability is calculated based on the Metropolis criterion. This script is designed to be supplied to a single NAMD instance and to be executed in parallel by the specified number of replicas. Data exchanges between replicas are facilitated using MPI calls encoded into Tcl functions (e.g. `replicaSendrecv`). In the ASyncRE module a separate NAMD instance must be invoked for each replica and mechanism for data exchanges between replicas, is implemented in the ASyncRE package itself. In addition to that, in con-
Figure 3.1: ASyncRE NAMD module class inheritance diagram

```plaintext
proc replica_temp { i } {
    global num_replicas min temp max temp
    return [format "%.2f" [expr (smin temp * \n    exp( log(1.0*$max_temp/$min_temp)*(1.0*$i/($num_replicas-1)) ) )]]
}

set replica(temperature) [replica_temp $replica(index)]
set replica(temperature.a) [replica_temp $replica(index.a)]
set replica(temperature.b) [replica_temp $replica(index.b)]
```

Figure 3.2: Code snippet from the `replica.namd` script demonstrating how temperatures for replicas are set
Figure 3.3: Code snippet from the `alanin_base.namd` ASyncRE script demonstrating how input files for the simulation are set.

Conventional NAMD RE simulation setup NAMD is invoked just once per multiple runs, with number of runs specified in the script, but for ASyncRE package implementation, NAMD must be invoked for each run, since data exchanges are performed using output files. This means that significant changes must be done to `replica.namd` script in order to adapt (decompose) it for use with ASyncRE package. A step by step process of how `replica.namd` script was adapted for ASyncRE NAMD module will not be described here, since it is impractical and too detailed. In this section we will be concentrating on most important parts of the script, but for a curious reader both `replica.namd` script and `alanin_base.namd` ASyncRE script are provided in Appendix A.

In the first part of the `replica.namd` script general simulation settings are set, "left" and "right" neighbours of each replica determined and appropriate data structures for neighbouring replicas populated. All of this is irrelevant in the context of the ASyncRE NAMD setup, so won’t be discussed any further. Next in the `replica.namd` script are set temperatures for replica and it’s neighbours using procedure `replica_temp`, which can be found in Figure 3.2. This procedure was not included in ASyncRE NAMD script, but this mechanism for setting temperatures for replicas was implemented in `_buildNamdStates()` function of the `namd_async_re_job` class.

In `replica.namd` script, for the first simulation run is set only temperature value, but for subsequent runs are specified `.coor`, `.vel` and `.xsc` input files. This functionality was implemented in ASyncRE NAMD script. The corresponding code snippet from ASyncRE script can be found in Figure 3.3. Another important part of `replica.namd` script is provided in Figure 3.4. This script snippet implements an algorithm which determines if exchange will be accepted or not. Naturally this code snippet wasn’t made a part of ASyncRE script, but was used as a basis for implementation of the `_doExchange_pair()` function. As we can see from Figure 3.4, in conventional NAMD script values of temperatures and potential energies are obtained using inner data structures or by making MPI calls (e.g. `replicaRecv`). In the ASyncRE implementation each replica after each run redirects it’s temperature and potential energy values to corresponding output file (history file) and these files are used by the NAMD module to extract corresponding data values and to pass them to `_doExchange_pair()`.
If the decision regarding the exchange is positive (value of variable `swap` is set to 1), the actual exchange of temperatures between replicas is performed (Figure 3.5). In the rest of the `replica.namd` script are set restart files, deleted old output files and variables for monitoring simulation statistics are updated.

Figure 3.5: Code snippet from the `replica.namd` script demonstrating how exchanges of the temperatures are performed
3.1.4 Implementation of Temperature Exchange RE schema in ASyncRE NAMD module

Implementation of temperature exchange RE schema can be found in `namd_async_re_job` class of the `temp_ex_async_re.py` file. The schema is implemented as follows. The first function of `namd_async_re_job` class what gets invoked during program execution is `_checkInput()` function. In this function are parsed main simulation parameters, such as min and max temperature, which are specified in ASyncRE input file. For temperature exchange schema this input file is `namd-temperature.inp`. Parsed simulation parameters are assigned to corresponding instance variables. If any of the required parameters are missing or set incorrectly, `_checkInput()` function terminates program execution. Also here is called `_buildNamdStates()` function, which populates `stateparams` list with initial state parameters of each replica. State of each replica is defined only by its temperature. The reason for this is that only temperatures are exchanged during swaps. So in `_buildNamdStates()` function using formula from the NAMD simulation script provided in Figure 3.2, for each replica is calculated initial temperature. This temperature is then assigned to a single item dictionary, which in turn is appended to `stateparams` list, so that index of each replica corresponds to position of its dictionary in a `stateparams` list.

The exchanges of replicas are performed by switching `stateid`'s. `stateid` is simply an integer, which is stored in `status` data structure of the `async_re_job` core class. Initially `stateid` of each replica corresponds to its index - e.g. `stateid` of replica 3 equals 3. `stateid` is used as an index to retrieve state parameters of each replica from the list `stateparams`. This way when exchanges are performed, elements of `stateparams` data structure are not swapped, instead replica’s updated `stateid` is used to retrieve an appropriate element from the `stateparams` data structure.

After simulation parameters have been parsed by the `_checkInput()` function, NAMD input files for each replica are created using these parameters. This is done by the `_buildInpFile()` function. This function uses template file `alanin_base.namd` to substitute tokens in this file with simulation parameters and saves modified template file in work directory of currently processed replica. This function is invoked for each replica, since each replica must have it’s own NAMD simulation script. in this function also are set names for simulation output files, including history file, which is used to output simulation parameters for determining exchange probability. Function `_buildInpFile()` is a generic function - it is called to build NAMD simulation script for all simulation cycles. This places some additional requirements on the function and NAMD input script. If temperature exchange occurred, in NAMD input script must be issued command to rescale velocities by the given factor - `rescalevels [expr sqrt(1.0*$NEWTEMP/$OLDTEMP)]`. This requires a newly acquired temperature as well as old temperature to be provided in the script. New simulation temperature can be obtained by retrieving appropriate temperature value from `stateparams` structure according to the newly obtained `stateid`. In order to keep track of the old temperature values, which are ‘lost’ after the exchange, these temperature values along
with potential energies are redirected to history output files. Then old and new temperature values are obtained, they are compared to determine if exchange occurred or not. According to the result of this comparison, swap variable is set and passed to the NAMD script.

ASyncRE core module is designed to use Gibbs [37] sampling method for determining exchange probabilities for replicas. The implementation uses a sparse matrix of dimension-less energies called SwapMatrix. Each element of this matrix represents an energy of replica \( i \) in state \( j \). Replicas are represented by columns and states are represented by rows. The size of the matrix allows to include all participating replicas in all states, but during the simulation the matrix is populated with reduced energies of replicas only in a waiting state. Matrix elements for replicas in states other than waiting state are not populated. Each element of the swap matrix represents a reduced energy - a unitless value scaled by the inverse temperature. For RE with temperature exchange reduced energies are calculated by:

\[
    u_j(x_i) = \beta_j U(x_i)
\]  

(3.1)

where:
- \( \beta_j \) - is an inverse temperature for state \( j \) calculated by: \( \beta_j = 1/k_B T_j \)
- \( T_j \) - is a temperature of replica \( j \)
- \( k_B \) - is a Boltzmann’s constant
- \( U(x_i) \) - is a potential energy of replica \( i \), which is obtained from the simulation output

Swap matrix is created and populated after each simulation cycle by calling function \_computeSwapMatrix(). In this function required parameters and results are extracted in order to populate matrix elements. In order to obtain potential energies from the simulation output \_computeSwapMatrix() function calls \_getTempPot() function. Matrix elements are populated using values calculated by the \_reduced_energy() function, which uses the above equation for that purpose.

Swap matrix is calculated inside the \_doExchanges() function of the core ASyncRE class - async_re_job. Than it is passed along with other parameters to \_gibbs_re_j() function which actually determines if exchange will occur or not. \_gibbs_re_j() function for a given replica returns another replica, and if returned replica is different from the given replica an exchange of replicas stateid’s takes place.

In NAMD ASyncRE module are supported two methods for performing exchanges between replicas - a Gibbs sampler based method described above and a conventional method, which uses Metropolis scheme. In order to switch from one method to another slight changes in core file pj_async_re.py are required. In Figure 3.6 is provided code snippet demonstrating how to modify the code in \_doExchanges() function in order to switch from one implementation to another. Implementation of the Metropolis scheme based method for performing exchanges between replicas is defined in \_doExchange_pair() function of the namd_async_re_job class. This implementation does not make use of a swap matrix or \_computeSwapMatrix() function. The \_doExchange_pair() function for a given replica, returns a replica what has qualified for the exchange. A given replica is paired with each replica from
The list of replicas in a waiting state. For each pair of replicas is calculated:

$$\Delta \beta = \frac{(1/T_i) - (1/T_j)}{k_B}$$  \hspace{1cm} (3.2)

and

$$\Delta = \Delta \beta (U(x_j) - U(x_i))$$  \hspace{1cm} (3.3)

Then, probability of exchange is determined and assigned to Boolean variable `swp`. If `swp` equals "True", then replica currently paired with a given replica is returned by the `_doExchange_pair()` function. If from the list of waiting replicas none of the replicas qualified for the exchange, function returns a given replica. In this case no exchange of state id’s will take place.

At a very end of `temp_ex_async_re.py` file an instance of the `namd_async_re_job` class is created and `ASyncRE` input file specified as a command line argument is passed to it. Finally on that instance functions `setupJob()` and `scheduleJobs()` are called and simulation begins.
proc replica_bias { i } {
    return [list lenpot "centers [expr 17 + $i]"]
}

set replica(colvarbias) [replica_bias $replica(index)]
for { set i 0 } { $i < $replica(num_neighbors) } { incr i } {
    set replica(colvarbias.$i) [replica_bias $replica(index.$i)]
    set replica(colvarbias.$i) [replica_bias $replica(index.$i)]
}

Figure 3.7: Code snippet from the `umbrella.namd` script demonstrating how collective variable biases are set.

set replica(colvarbias) [replica_bias $bias_id]
for { set i 0 } { $i < $num_replicas } { incr i } {
    set replica(colvarbias.$i) [replica_bias $i]
}

Figure 3.8: Code snippet demonstrating how collective variable biases are set in the `alanin_base_umbrella.namd` ASyncRE script.

if {$i_run} { #restart
    bincoordinates $restart_root.$replica_id.coor
    binvelocities $restart_root.$replica_id.vel
    extendedSystem $restart_root.$replica_id.xsc
    colvarsInput $restart_root.$replica_id.colvars.state
} elseif { ![info exists input root] }{
    set ir [format $input_root $replica_id $replica_id]
    bincoordinates $ir.coor
    binvelocities $ir.vel
    extendedSystem $ir.xsc
} else {
    temperature $temperature
}

Figure 3.9: Code snippet from the `umbrella.namd` NAMD script demonstrating how input files are set.
3.1.5 Adapting RE with Umbrella Sampling NAMD simulation setup for ASyncRE NAMD module

Similarly as for the Temperature Exchange RE schema it was decided to merge all conventional NAMD simulation configuration scripts (scripts with filename extensions .namd and .conf) into one. The most important script of the conventional RE-US NAMD simulation setup is umbrella.namd script. In this script are defined the most important data flow mechanisms and orchestration of the exchanges between replicas is performed. This script is designed to be used by all participating replicas in parallel and to be processed by a single NAMD instance.

In this section will be described the most important steps in adapting umbrella.namd script for the ASyncRE NAMD module. For a more detailed understanding of the script decomposition process, both umbrella.namd NAMD script and alanin_base_umbrella.namd ASyncRE script are provided in Appendix A.

The first important part of the umbrella.namd script is where collective variable biases are set. This is done by calling function replica_bias, which is actually located in a different configuration file. The code snippet showing this procedure can be found in Figure 3.7. Function replica_bias returns a "bias expression" which is different for each replica. Collective variable biases in the ASyncRE script are set as depicted in Figure 3.8. As we can see from Figure 3.8, "colvarbias" is calculated for all participating replicas in ASyncRE script of each replica.

First cycle of the conventional RE-US simulation is started using coordinates, velocities and "extended system" input files. For subsequent runs are used output files of the previous run. A script snippet in which are set simulation input files is provided in Figure 3.9. In the ASyncRE script is used similar approach for setting input files. This can be found if Figure 3.10.

```bash
if {$i_run} { #restart
  set oldrun [expr {($cycle-1)}]
  set oldoutput $old_output_root$oldrun
  bincoordinates $oldoutput.coor
  binvelocities $oldoutput.vel
  extendedSystem $oldoutput.xsc
  colvarsInput $oldoutput.colvars.state
} else {
  if [info exists input root] {
    bincoordinates $input_root.coor
    binvelocities $input_root.vel
    extendedSystem $input_root.xsc
  } else {
    temperature $r_temperature
  }
}
```

Figure 3.10: Code snippet demonstrating how input files are set in the alanin_base_umbrella.namd ASyncRE script
Implementation of the algorithm for exchanges between replicas in RE-US NAMD script is provided in Figure 3.11. The main difference from the temperature exchange RE script is calculation of free energy differences, the result of which is assigned to the $ediff variable. For this calculation are used collective variable biases calculated previously for the "neighbouring" replicas. For the asynchronous RE we don’t know with which replica a given replica will be paired, so we must calculate collective variable biases for all participating replicas. The same applies to the calculation of the free energy differences. For conventional NAMD simulation a pair of replicas is predetermined, so $ediff can be calculated on the fly using precalculated collective variable biases. In the ASyncRE script calculation of the free energy differences is performed for each potential pair and the resulting energy differences are redirected to the "energies_x.out" output file (Figure 3.12). The algorithm provided in Figure 3.11 was used as a base for implementation of the _doExchange_pair() function. If exchange was accepted, the only procedure that takes place is the re-evaluation of the collective variable biases (Figure 3.13). This means that in essence only bias id’s of replicas are exchanged. For this reason in addition to the temperature and potential energy values, to history file is redirected bias id of a given replica.
for { set i 0 } { i < $num_replicas } { incr i } {
    set ediff [eval colvarbias [concat energydiff $replica(colvarbias.$i)]]
    puts $energy_file "$ediff"
}

Figure 3.12: Code snippet demonstrating how resulting energy differences are calculated in the `alanin_base_umbrella.namd` ASyncRE script

if { $doswap } {
    array set replica [replicaSendrecv [array get replica] $newloc $newloc]
    eval colvarbias [concat changeconfig $replica(colvarbias)]
}

Figure 3.13: Code snippet from the `umbrella.namd` script demonstrating how collective variable biases are re-evaluated after the exchange between replicas was accepted

### 3.1.6 Implementation of RE with Umbrella Sampling schema in ASyncRE NAMD module

A RE-US schema is implemented in `namd_us_async_re_job` class which can be found in `us_ex_async_re.py` file of the NAMD ASyncRE module. The implementation of the `_checkInput()` function is very similar to implementation of the same function for the temperature exchange schema. In this function are made checks for the MD engine type ("ENGINE") and replica exchange simulation type ("RE_TYPE"). Also in this function are set simulation parameters such as temperature, time step and number of steps per cycle. It is worth noting that, in contrast to the temperature exchange schema, current implementation of the RE-US schema requires all replicas to be initialised at the same temperature. Lastly, in this function is called `_buildNamdUsStates()` function to which is passed the number of replicas. Similarly to the `_buildNamdStates()` function, `_buildNamdUsStates()` initialises and populates `stateparams` list with initial state parameters of each replica. For the RE-US schema the state of each replica is defined only by it’s bias id. Initially bias id’s of replicas are equal to their replica number.

Next function of the `namd_us_async_re_job` class is `_buildInpFile()` function. As the name suggests, this function is designed to build an input file for each replica. All input files are based on the template file `alanin_base_umbrella.namd`. Tokens in a template file are substituted with individual parameters for each replica. Apart from the general simulation parameters parsed in the `_checkInput()` function, in `_buildInpFile()` function are set names for output files including history file and "energies" file. In this function is also incremented the value of the first time step parameter.

Next, follows the implementation of the `_doExchange_pair()` function. As men-
tioned previously this function implements the exchange algorithm illustrated in Figure 3.11. For a given replica \_doExchange\_pair() returns a replica with which an exchange of state parameters should occur. This function tries to find a pair from the list of all replicas in a waiting state. In case if no pair is found a function simply returns a given replica. Here \( \Delta \) is calculated by:

\[
\Delta = \beta (\Delta E(x_j) + \Delta E(x_i))
\]

(3.4)

where:

\( \beta = \frac{1}{k_B T} \)

\( \Delta E(x_j) \) is obtained by retrieving an \( i \)th element from the list of free energy differences, which is returned for each replica by the \_getEdiffData() function. Similarly, \( \Delta E(x_i) \) is an \( j \)th element of the list of free energy differences for replica \( i \). For determining exchange probability in \_doExchange\_pair() function is used Metropolis criterion.

In order to implement a Gibbs sampler based RE method, \_computeSwapMatrix() function must be defined. This function creates and populates a swap matrix, which is passed to the \_gibbs\_re\_j() function. In \_computeSwapMatrix() function are populated only elements corresponding to the replicas currently in a waiting state. Each element of the swap matrix represents a dimensionless energy of replica \( i \) in state \( j \) which is calculated using the above equation. For example a swap matrix element representing replica 4 in state 2 is calculated by:

\[
U[2][4] = \beta * (ediff_2 + ediff_4)
\]

where:

\( ediff_2 = ediff\_data\_2[4] \) is element with index 4 from the list of the free energy differences calculated for replica 2 using function \_getEdiffData()

\( ediff_4 = ediff\_data\_4[2] \) is element with index 2 from the list of free energy differences calculated for replica 4 \( \beta \) is an inverse temperature

In the final part of the us\_ex\_async\_re.py file can be found code that starts the actual RE-US simulation. This is done by creating an instance of the namd\_us\_async\_re\_job class. ASyncRE input file must be passed as an argument while creating this instance. Then, setupJob() function is called to initialise the simulation, that is started by calling scheduleJobs() function.
Chapter 4

Evaluation

4.1 ASyncRE package on Kraken

Evaluation of the implemented ASyncRE NAMD module was performed only on one HPC system - Kraken. Initially it was planned to analyse the package on both available Cray machines - Hector and Kraken. Unfortunately it was not possible to enable full functionality of BigJob on Hector by the time of this writing. Namely, despite the fact that pilot allocates several nodes for execution of sub-jobs, all sub-jobs are executed on a single node. In addition to that, on Cray machines such as Hector, BigJob experiences another limitation - only a single executable can be run on a node at any given time. Due to these issues evaluation of BigJob on Hector was not possible.

4.1.1 RE with Temperature Exchange simulations using NAMD

For both NAMD and ASyncRE RE with Temperature Exchange simulations were choosen a system, folding a 66-atom model of a deca-alanine helix. In a typical RE simulation exchanges are attempted every 400 to 2000 fs (time-steps). For both temperature exchange simulations a 2000 fs cycle length was chosen. While deca-alanine helix folding requires at least 10 ns to complete, due to limited time, total simulation length was set at 2 ns, meaning that a total of $2 \cdot 10^6$ simulation time-steps will be performed. NAMD requires a total number of requested CPU cores to be equal or greater to the number of replicas. For all simulation runs a number of cores was set to be equal to the number of replicas of the simulated system. For varying numbers of replicas time required to complete a simulation was measured and average simulation speeds for each number of replicas were calculated. In the context of this chapter simulation speed is the number of simulation time-steps performed in a wall-clock time unit. The results of the performed runs are provided in Figure 4.1. As we can see from Figure 4.1, increase in the number of replicas results in gradual decrease in the simulation speed.
4.1.2 RE with Temperature Exchange simulations using ASyncRE NAMD module

In order to evaluate implementation of the Temperature Exchange scheme in ASyncRE NAMD module the following experiments were conducted. For all experiments was chosen the same system folding a 66-atom model of a deca-alanine helix as for the NAMD simulation runs. In order for replicas to perform exchanges a number of replicas after a cycle must be in a waiting state. This means that for exchanges to occur a number of requested cores must be less than the number of replicas. Developers of the ASyncRE package recommend using 1:2 ratio of cores to replicas. In addition to that we must remember that on Kraken it is possible to launch only one replica (sub-job) per node at any given time. This means that if we plan to run 24 replicas and request 12 cores (one node), only one replica will be launched at a time. To overcome this limitation for each run we must request a number of nodes equal 1/2 of the total number of replicas. For example, if we are running a simulation involving 12 replicas we must request 72 CPU cores which corresponds to 6 nodes on Kraken. While it is possible to set a number of time-steps to be performed during each cycle, due to ASyncRE package specifics it is not possible to set the total number of simulation time-steps to be performed by each replica. The simulation is controlled by two parameters - `WALL_TIME` - a total number of wall-clock time minutes the simulation will run and `CYCLE_TIME` - specifying how long it will take framework to complete a cycle. Default value of the
cycle length is 30 seconds, but experiments demonstrated that this value is too small for the majority of simulation runs to finish. If simulation run is not finished after the cycle time has elapsed, the run is restarted. This significantly limits the simulation speed. Cycle length values were varied for all performed runs in order to maximise the average simulation speed. For all ASyncRE RE with TE simulations a number of time-steps per cycle was set at 2000. Again, while varying numbers of replicas, time required to complete a simulation was measured and average simulation speed calculated. The results of these experiments can be found in Figure 4.1. As we can see NAMD demonstrates significantly higher simulation speeds. For example, for the first runs NAMD simulation speed is 887 times higher. It is important to mention while systems used for simulations by both packages are equal in size (equal number of replicas), the number of CPU cores used by NAMD is two times larger, since for ASyncRE number of CPU’s is 1/2 of the total number of replicas. In addition to that, NAMD is able to use all 12 cores on each node, but ASyncRE is limited to one. Despite the fact that executing replicas are not directly communicating with each other (all communication is through files), this limitation might have some effect on the average simulation speed. Unfortunately it was not possible to compare performance of ASyncRE system using all cores on a node to launch replicas against ASyncRE system using only one core per node. Comparison of NAMD and ASyncRE using same amount of the computational power (the same number of cores) and running the simulations on "equal" hardware (both codes using only one core per node) would be useful as well. Although in this case it is quite
clear that NAMD performance would suffer due to increased communication latencies since in NAMD running replicas are exchanging data with each other. It is important to mention that significant amount of tuning was required in order to avoid the situation when replicas are re-launched for the ASyncRE package in the middle of the run. Due to this issue average simulation speeds were reduced dramatically. Also, increase in the number of time-steps per cycle would compensate for the large communication latencies of the ASyncRE package, but this was not investigated further since examples of the systems where a much larger number of time-steps per cycle is required were not found.

4.1.3 RE with Umbrella Sampling simulations using NAMD

For RE with Umbrella Sampling simulations using NAMD and ASyncRE was used Alanine stretching system from the NAMD library. As for previous experiments, cycle length was set at 2000 fs and total simulation length at 2 ns (1000 cycles). Again for all runs the number of CPU cores was set equal to the total number of replicas and total simulation time was measured while varying the number of CPU’s. Average simulation speeds for all performed runs are provided in Figure 4.2. In comparison with NAMD Temperature Exchange simulations, average speed values were substantially lower for all runs. Similarly as for the Temperature Exchange simulations, increase in the number of replicas results in the gradual decrease in the simulation speed. This leads to conclusion that regardless of the RE scheme type increase in number of replicas results in increased cost of communication. By comparing result presented in Figure 4.1 and Figure 4.2 we can conclude that simulation speed also depends on the scheme type and system complexity - for a more complex Alanine stretching system simulation speeds were lower.

4.1.4 RE with Umbrella Sampling simulations using ASyncRE

For ASyncRE were used 2000 fs cycle length and the same Alanine stretching system as for NAMD runs. Also, number of CPU cores used for each run was set to 1/2 of the number of replicas. Result of the performed runs can be found in Figure 4.2. As of the temperature exchange scheme we can observe a significant difference between NAMD simulation speeds and ASyncRE simulation speeds. This difference in simulation speeds is caused by the following factors:

- 50% less computational power
- higher communication latencies
- cycle length set using wall clock time - significant overhead
- some of the replicas are restarted
- only one replica executed per node (arguably)
Comparison of simulation speeds of both implemented RE schemes in ASyncRE are provided in Figure 4.3. As we can see for ASyncRE, RE-US scheme demonstrates much better performance results than RE-TE scheme. At the same time, graph of the RE-TE scheme is very similar in shape to NAMD TE graph. The same can’t be said about RE-US graph.

### 4.1.5 Analysis of NAMD RE simulations

In addition to the above experiments was examined performance of the NAMD RE simulations while fixing number of cores (replicas) and varying simulation lengths. Simulation runs were performed for both RE systems. Again, cycle length was set at 2000 fs. Initial number of cycles was set at 1000, but for the subsequent runs number of simulation cycles was doubled. Time taken to complete simulation runs was measured and results of the experiments are provided in Figure 4.4. For both RE schemes doubling of simulation length resulted in exponential increase in simulation time. Both graphs approximately can be defined by the following function: $T = 2^N$ where $T$ is simulation time and $N$ is simulation length.

For RE-US scheme was performed a comparison of simulation runs while having different simulation input files. For first system each run was started with initial input files. In Figure 4.5 this system is called "from start". For second system first run was started using initial simulation input, but subsequent runs were started using output of the pre-
Figure 4.4: Comparison of simulation run-times (in minutes) for Temperature Exchange and Umbrella Sampling NAMD RE schemes; varying simulation length; number of replicas is fixed at 12; number of CPU cores is fixed at 12; all runs are performed on one node.

Previous run. In Figure 4.5 this system is called "continuing". For both systems simulation length was fixed, but number of cores and replicas varied. Average simulation speeds for these runs are provided in Figure 4.5. In this figure we can observe a significant decrease in average simulation speeds for the system using output files of the previous runs as a simulation input. This means that for Alanine stretching system, simulation speed and total run-time for long runs increases not only due to increase in the total number of time-steps, but also due to increase in time required to perform a simulation step at later stages of the simulation.

From performed experiments with NAMD simulations we can conclude that simulation speed and simulation run-time depends on the following factors:

- Simulation length
- Type of RE simulation
- Complexity of the system used for the RE simulation
- Input parameters (at which point we are starting simulation)
- Underlying HPC architecture (communication latencies and CPU speed)
- Number of replicas (more replicas - more communication)
Figure 4.5: Comparison of speeds for Umbrella Sampling NAMD simulations; varying numbers of replicas and CPU cores; fixed simulation length; for a "from start" graph each of the simulation runs was performed using initial input files; for a "continuing" graph first run was performed using initial input files, but subsequent runs were performed using output of the previous run.
4.1.6 Comparison with ASyncRE Amber module for RE with Umbrella Sampling simulations

Unfortunately it was not possible to obtain simulation set up and input files to simulate the same system with both Amber and NAMD ASyncRE MD engines. In order to evaluate performance of ASyncRE Amber module were used results of the experiments with ASyncRE Amber module performed by E. Gallicchio et. al. [33], which are provided in Figure 4.6. According to results provided in Figure 4.6 ASyncRE Amber module demonstrates linear scaling for the number of concurrent replivas from 20 to 140. It is important to point out that in this case simulation speed was calculated by adding together the time-steps simulated by each replica and dividing it by the total simulation time. For all above experiments simulation speed was calculated on per replica basis - number of time-steps performed by each replica was divided by the total simulation time. In Figure 4.7 are provided average simulation speeds for ASyncRE NAMD module using the same methodology that was used for ASyncRE Amber module in Figure 4.6. As depicted in Figure 4.7 ASyncRE NAMD module for the majority of runs demonstrates scaling close to linear. In addition to that, for all simulation runs average speeds for NAMD module are higher than for Amber module. It is important to note, that these results are not directly comparable since for simulations are used different models.

4.1.7 Performance analysis of BigJob on Kraken

In order to evaluate performance of BigJob on Kraken, the following experiments were performed. For all experiments was used fixed number of Compute Units (sub-jobs) equal 192, while size of Pilot was incremented by 12 cores (node size on Kraken). for all experiments a total of three runs were performed. Average Compute Unit throughput values are provided in Figure 4.8. While increasing Pilot size up to 72 cores an increase in sub-job throughput can be observed. Increase of Pilot size from 72 to 84 cores results in decrease in sub-job throughput. There are two possible reasons for this decrease - first is specifics of Kraken architecture, second is bottleneck in BigJob sub-job launching mechanism.

In Figure 4.9 are provided average sub-job queuing times for the same runs. Up to 60 core queuing times are decreasing, but for Pilot size equal 72, a slight increase in queuing times can be observed. At the same time increase of pilot to 84 cores resulted in substantial decrease in queuing times. This means that there is no bottleneck in BigJob sub-job launching mechanism for this pilot size.

Taking into account limitations currently imposed by aprun, BigJob demonstrated a good performance and scalability on this machine. Application scaled to 96 CPU cores while demonstrating respectable performance results.
Figure 4.6: Scaling results for ASyncRE/BigJob with the AMBER MD engine employing a serial quantum mechanical/molecular mechanical potential. The total number of cores allocated for the full REMD run was varied, while each simulation was run on a single core. As should be expected, performance (as measured in ns/day) increases linearly (red fit line) with the number of simulations running at any given point in time. It is perhaps relevant to note that the performance of individual simulations is also quite consistent at different core counts (green fit line), verifying that the linear scaling is in fact representative of an increase in the number of simulations running and not just an increased core count [33]
Figure 4.7: Scaling results for ASyncRE with the NAMD MD engine using RE-US scheme; varying numbers of replicas and nodes; simulation speed calculated by dividing total number of simulation time-steps performed by all replicas, by total simulation run-time
Figure 4.8: Average Compute Unit (sub-job) throughput using BigJob with Saga-Python on Kraken; throughput in Compute Units per second; number of CUs fixed at 192; Varying Pilot size (number of BigJob allocated cores)
Figure 4.9: Average Compute Unit (sub-job) queuing times using BigJob with Saga-Python on Kraken; number of CUs fixed at 192; Varying Pilot size (number of BigJob allocated cores)
4.2 BigJob on Cray machines

One of the initial project aims was to develop a solution which would enable BigJob to run simultaneously multiple executables on one compute node of Cray XE/XT machines. This limitation is imposed by `aprune` - a job launching command used on Cray systems.

In order to investigate limitations of BigJob on Cray systems, two HPC machines were used - Hector and Kraken. On both of these machines Portable Batch System (PBS) is used as a job scheduling mechanism.

Initial experiments performed on Kraken demonstrated that, on any given node, sub-jobs requiring a small number of cores are launched one after another. This is despite the fact that technically it would be possible to execute several such sub-jobs simultaneously.

Experiments on Hector revealed that on this machine BigJob experiences even more limitations. In the case where sub-jobs require a number of cores which is less than the number of cores available on one node (32), despite the fact that BigJob requests several nodes to execute these sub-jobs, sub-jobs would be executed on one node one after another.

4.2.1 MPI with Python

In order to enable BigJob to run simultaneously multiple executables on one compute node, it was decided to parallelize the part of the program responsible for the submission of the sub-jobs. In BigJob, the class responsible for launching of the sub-jobs is the `bigjob_agent` class. The idea was for each core of the node to instantiate one MPI process and each of these processes would launch sub-jobs, which are pulled from the coordination service. Since BigJob is fully written in Python, it was decided that for this purpose the best candidate is `mpi4py` [38] package.

The simplest way to run a parallel MPI program is to specify the required number of cores when launching the program. Unfortunately in our case this approach would not work. Firstly, BigJob code is heavily object oriented and for the most part serial. Secondly, the parallel part of the code needs to be isolated in order to avoid unpleasant side-effects, such as execution of serial part of the code by multiple processes.

A more suitable approach for parallelization of the BigJob code is to use the Dynamic Process Management features of the mpi4py. In this case a master worker model would be used, where instance of the `bigjob_agent` class would be master, spawning the required number of processes and worker would be a separate script responsible for launching of the sub-jobs. Here basically worker script would perform the functions of the `execute_job()` function of the `bigjob_agent` class. In order to implement this approach a significant programming effort would be required due to a high level of the code inter-dependencies.

The idea of parallelizing BigJob code using `mpi4py` was abandoned, since the proposed solution would not provide any advantages over the currently implemented mech-
anism which uses the python threading interface.

### 4.2.2 Multiple aprun technique

An investigation on both available Cray machines was carried out to identify if it is possible to run multiple programs in parallel using a single job submission script as depicted in Figure 4.10. This is achieved by having multiple `aprun` commands in one job submission script. This technique is designed to be used for simultaneous simulations needing to run a large number of executables, each requiring a small number of cores. Simulations submitted using this technique are treated by the batch system as a single job.

Unfortunately this scripting technique would not enable BigJob to run simultaneously multiple executables on one compute node of Cray systems, because it requires each `aprun` command to be launched on a separate compute node. This is not possible due to the fact that on Cray XE systems only exclusive node access is allowed. Using this scripting technique only one instance of `aprun` can be launched per compute node during each script submission to the scheduler, meaning that BigJob already provides more flexibility than this approach.

```bash
#!/bin/bash --login
#PBS -N my_job_name
#PBS -l mppwidth=96
#PBS -l mppnppn=12
#PBS -l walltime=00:30:00
#PBS -A MY-BUDGET

# This prevents any system libraries from automatically
# using threading
export OMP_NUM_THREADS=1

cd $PBS_O_WORKDIR/simulation1/
aprun -n 24 -N 12 my_app < input1.dat > output1.txt &
cd $PBS_O_WORKDIR/simulation2/
aprun -n 24 -N 12 my_app < input2.dat > output2.txt &
cd $PBS_O_WORKDIR/simulation3/
aprun -n 24 -N 12 my_app < input3.dat > output3.txt &
cd $PBS_O_WORKDIR/simulation4/
aprun -n 24 -N 12 my_app < input4.dat > output4.txt &

# Wait for all simulations to complete
wait
exit 0
```

Figure 4.10: Multiple `aprun` script example
4.2.3 MPMD technique

In addition to the above technique `aprun` provides an option to submit jobs using Multiple Program Multiple Data mode. In this case, the `aprun` command is called only once and is followed by the executables together with the configuration flags, separated by colons (Figure 4.11). There are several limitations associated with this technique. First, compute nodes can’t be shared by executables - each executable must run on a separate compute node. Second, MPMD mode does not support system commands. Third, this mode requires the executables to be parallel MPI programs and at least to post an `MPI_Init()` and `MPI_Finalize()` calls.

4.2.4 TaskFarmer utility

For running multiple tasks in parallel Cray provides a TaskFarmer utility. Each task must be a serial job requiring only one processor to run. Before this utility can be used TaskFarmer module must be loaded on the target machine. The basic job submission script using TaskFarmer utility is provided in Figure 4.12. Here 64 cores are allocated to run 100 tasks. To TaskFarmer utility is submitted bash script, which allows to specify the name of the executable task, the location of the input files for each task and the location of the output files. A sample bash script can be find in Figure 4.13. As we can see in Figure 4.13 this scripts `$TF_TASKID` environment variable is unique to each task and can be used to specify separate input and output directories for each task. The number of tasks can be greater than the number of allocated processors. Unfortunately the TaskFarmer utility is not installed on Kraken and Hector. It is unlikely that the request to install it on any of these machines would be satisfied, since system administrators rarely comporomise system stability for the needs of a small group of users.

4.2.5 Multiple single-processor program technique

To the best knowledge of the author, there is only one PBS scripting technique that allows simultaneous execution of multiple programs on one compute node of Cray XE/XT machines. This technique only allows submission of single core programs. In the PBS submission script the `aprun` command is called just once, but with several additional flags (Figure 4.14):

```
aprun -d N ./prog_a : -n 64 -N 4 ./prog_b
```

```
Figure 4.11: MPMD `aprun` call example
```

- `-d N` - this flag allows to access all the cores of the node; `N` - must be substituted with the core count
Figure 4.12: PBS script using TaskFarmer utility

```bash
#!/bin/bash
#PBS -l mppwidth=64
#PBS -l walltime=2:00:00
#PBS -N my_job_name
#PBS -e my_job_name.$PBS_JOBID.err
#PBS -o my_job_name.$PBS_JOBID.out
#PBS -V

cd $PBS_O_WORKDIR

module load taskfarmer
cd $PBS_O_WORKDIR

tf -t 100 -n 2 -e serial.err -o serial.out ./my_task.sh
```

Figure 4.13: `my_task.sh` example bash script for TaskFarmer utility

```bash
#!/bin/sh

echo "my taskid is: " $TF_TASKID

if [ $TF_TASKID -le 9 ]
    then
        pre="dir0"
        dirname=$pre$TF_TASKID
        echo $dirname
    else
        pre="dir"
        dirname=$pre$TF_TASKID
        echo $dirname
fi

./task_exe < $dirname/infile > $dirname/outfile
```
Figure 4.14: PBS example script for running multiple single-processor jobs

```bash
#!/bin/csh
 PBS -A MY-BUDGET
 PBS -N run_serial
 PBS -l walltime=00:30:00,size=12
 PBS -j oe

 set echo
cd /path/to/work/directory/serial

# Use aprun to start a shell script which runs 12 copies of the
# same executable on a compute node
# Note: all aprun options specified below are required
# -n 1 # run on a single node
# -d 12 # allows the script to access all the cores on a node
# -cc none # allows each serial process to run on its own core
# -a xt # required by aprun to run a script instead of a program

aprun -n 1 -d 12 -cc none -a xt ./run_serial
```

- `cc none` - this enables each program to run on a separate core
- `a xt` - enables `aprun` to run script instead of a program

In addition to that, `aprun` command must be followed by the `-n 1` flag, ensuring that each program would run on a single core. Script that is submitted instead of executable is a simple shell script (Figure 4.15), in which required number of single-core programs is specified. It is important to note that it is possible to specify any number of executables that would not exceed total core count of a given node. This technique was proven to work on both Hector and Kraken using conventional PBS scripts.

It is possible to use this technique with BigJob, with minor changes to the source code, which take into account additional `aprun` flags. In order to use this technique, after installing modified version of BigJob, users only have to write a simple shell script where required number of single-core executables is specified.
#!/bin/sh  # This must be /bin/sh (other shells do not work)

# Run 6 copies of serial_code in the background
./serial_prog &
./serial_prog &
./serial_prog &
./serial_prog &
./serial_prog &
./serial_prog &

# Wait until all copies of serial_prog have finished
wait

Figure 4.15: Example shell script for running multiple single-processor jobs
Chapter 5

Conclusions

5.1 NAMD ASyncRE module

This MSc dissertation involved implementation of the NAMD based module for ASyncRE Python package. Implemented NAMD module supports Temperature Exchange and Umbrella Sampling RE schemes. Current implementation of RE-US scheme supports only one dimensional simulations. In order to support multi-dimensional RE-US schemes only minor code alternations would be required since current one-dimensional implementation is based on core NAMD configuration file which is used for both schemes. NAMD implementation of RE-US scheme demonstrated performance comparable with ASyncRE Amber module. At the same time NAMD implementations of both RE schemes were significantly slower than conventional NAMD. Despite this fact for sufficiently complex systems requiring significant amount of computational power, ASyncRE NAMD module potentially can be competitive with conventional NAMD simulations. Even if difference in average simulation speeds still would be significant, due to better sampling of the asynchronous RE model, ASyncRE module would require to perform fewer time-steps to arrive to solution.

5.2 BigJob on Cray machines

During this dissertation project, was made an attempt to develop a solution that would overcome limitation imposed by aprun on Cray machines, which only allows to launch one executable per node at any given time. A number of possible solutions were examined, including parallelization of BigJob with mpi4py. The only working solution identified was multiple single-processor program scripting technique. This solution was proven to work using conventional PBS scripts on both Hector and Kraken. In addition to that this solution was implemented in BigJob. Due to time limitations it was not possible to fully test this implementation. Unfortunately, even if implementation of this
technique would be fully functional, it would not be possible to use it for RE simulations with ASyncRE due to specifics of this package.

5.3 Future work

5.3.1 NAMD ASyncRE module

A very important improvement that would greatly complement developed NAMD module for ASyncRE package is a dedicated input/output (i/o) module. In current implementation of NAMD module, much of the complexity is handled by NAMD tcl scripts. As the result, input scripts are relatively complex and cumbersome for inexperienced user to understand. All required data is extracted from NAMD during the simulation and exchange decisions are made by manipulating this data. Arguably a better approach would be to develop a dedicated i/o module, which would allow to perform Python function calls in order to extract needed simulation data. For temperature-exchange scheme this may not be of a great importance, since all required data values can be easily redirected to `.history` file, but for RE Umbrella Sampling and other more complicated schemes i/o module would provide a convenient way to extract required data after the simulation has finished. Another important improvement would be enabling functionality to specify the required simulation length (number of time-steps) for all replicas. Currently NAMD module suffers from overheads associated with specifying cycle length as a wall clock time.

5.3.2 Gromacs ASyncRE module

As described above another candidate application for writing an ASyncRE extension module was Gromacs. Implementing a Gromacs ASyncRE module would be a useful exercise for several reasons. First, it would allow to compare the learning curves for NAMD and Gromacs, since author of this paper didn’t had any experience with any of those programs. Next, it would allow to compare amount of effort required to decompose existing simulation setup and to write the ASyncRE module itself. Even for existing Impact and Amber modules the amount of coding effort spent for writing a module for a single schema is vastly different, with Amber requiring several times more developer effort than Impact. Of course the choice of schema has some effects on the required coding effort, but even comparing the number of lines of code required, this effect is negligible. Another advantage of implementing a Gromacs ASyncRE module is possibility to identify some flaws in existing core module or some incorrect assumptions in how extension modules can be implemented.
Appendix A

Stuff which is too detailed

A.0.3 BigJob example script (Kraken)

```python
from pilot import PilotComputeService, ComputeDataService, State
import os, time, sys

NUMBER_JOBS = 4
COORDINATION_URL = "redis://kraken-gsi3:6379"

pilot_compute_service = PilotComputeService(COORDINATION_URL)

pilot_compute_description = {
    "service_url": "pbs://localhost",
    "number_of_processes": 24,
    "queue": "small",
    "allocation": "TG-MCB090174",
    "processes_per_node": 2,
    "working_directory": "/lustre/scratch/quentin/bj/agent",
    "walltime": 10
}

pilotjob = \\
pilot_compute_service.create_pilot(pilot_compute_description = \\
pilot_compute_description)

compute_data_service = ComputeDataService()
compute_data_service.add_pilot_compute_service(pilot_compute_service)
```

57
# start work unit
compute_unit_description = {
    "executable": "/lustre/scratch/antotre/bj/example-mpi",
    "arguments": [""],
    "number_of_processes": 1,
    "output": "stdout.txt",
    "error": "stderr.txt",
}

for i in range(NUMBER_JOBS):
    compute_unit = \
    compute_data_service.submit_compute_unit(compute_unit_description)
compute_data_service.wait()
compute_data_service.cancel()

A.0.4 NAMD RE with Temperature Exchange example script
- replica.namd

replicaBarrier

set nr [numReplicas]
if { $num_replicas != $nr } {
    error "restart with wrong number of replicas"
}
set r [myReplica]
set replica_id $r

if {{[info exists restart_root]}} { # restart
    set restart_root [format $restart_root $replica_id]
    source $restart_root.$replica_id.tcl
} else {
    set i_job 0
    set i_run 0
    set i_step 0
    if {{[info exists first_timestep]}} {
        set i_step $first_timestep
    }

    set replica(index) $r
    set replica(loc.a) $r
    set replica(index.a) $r
    set replica(loc.b) $r
set replica(index.b) $r
set replica(exchanges_attempted) 0
set replica(exchanges_accepted) 0

if { $r % 2 == 0 && $r+1 < $nr } {
    set replica(loc.a) [expr $r+1]
    set replica(index.a) [expr $r+1]
}
if { $r % 2 == 1 && $r > 0 } {
    set replica(loc.a) [expr $r-1]
    set replica(index.a) [expr $r-1]
}

if { $r % 2 == 1 && $r+1 < $nr } {
    set replica(loc.b) [expr $r+1]
    set replica(index.b) [expr $r+1]
}
if { $r % 2 == 0 && $r > 0 } {
    set replica(loc.b) [expr $r-1]
    set replica(index.b) [expr $r-1]
}

set job_output_root "$output_root.job$i_job"
first_timestep $i_step

proc replica_temp { i } {
    global num_replicas min_temp max_temp
    return [format "%.2f" [expr ($min_temp * \ 
        exp( log(1.0*$max_temp/$min_temp)* \ 
        (1.0*$i/($num_replicas -1)) ) )]]
}

set replica(temperature) [replica_temp $replica(index)]
set replica(temperature.a) [replica_temp $replica(index.a)]
set replica(temperature.b) [replica_temp $replica(index.b)]

proc save_callback {labels values} {
    global saved_labels saved_values
    set saved_labels $labels
    set saved_values $values
}
callback save_callback
proc save_array {} {
  global saved_labels saved_values saved_array
  foreach label $saved_labels value $saved_values {
    set saved_array($label) $value
  }
}

set NEWTEMP $replica(temperature)
seed [expr int(0*$rand(int(100000*$rand()) + \100*$replica_id) + 100000*$rand())]

langevinTemp $NEWTEMP
outputname [format $job_output_root.$replica_id $replica_id]

if { $i_run } { # restart
  bincoordinates $restart_root.$replica_id.coor
  binvelocities $restart_root.$replica_id.vel
  extendedSystem $restart_root.$replica_id.xsc
} else {
  temperature $NEWTEMP
}

outputEnergies [expr $steps_per_run / 10]
dcdFreq [expr $steps_per_run * $runs_per_frame]

source $namd_config_file

set history_file [open [format \"$job_output_root.$replica_id.history\" $replica_id] "w"]
fconfigure $history_file -buffering line

while { $i_run < $num_runs } {
  run $steps_per_run
  save_array
  incr i_step $steps_per_run
  set TEMP $saved_array(TEMP)
  set POTENTIAL [expr $saved_array(TOTAL) - $saved_array(KINETIC)]
  puts $history_file "$i_step $replica(index) \$NEWTEMP $TEMP $POTENTIAL"
  if { $i_run % 2 == 0 } {
    set swap a; set other b
if { $replica(index) < $replica(index.$swap) } {
    set temp $replica(temperature)
    set temp2 $replica(temperature.$swap)
    set BOLTZMAN 0.001987191
    set dbeta [expr ((1.0/$temp) - (1.0/$temp2)) / $BOLTZMAN]
    set pot $POTENTIAL
    set pot2 [replicaRecv $replica(loc.$swap)]
    set delta [expr $dbeta * ($pot2 - $pot)]
    set doswap [expr $delta < 0. || exp(-1. * $delta) > rand()]
    replicaSend $doswap $replica(loc.$swap)
    if { $doswap } {
        set rid $replica(index)
        set rid2 $replica(index.$swap)
        puts stderr "EXCHANGE_ACCEPT $rid ($temp) \ $rid2 ($temp2) RUN $i_run"
        incr replica(exchanges_accepted)
    }
    incr replica(exchanges_attempted)
}
if { $replica(index) > $replica(index.$swap) } {
    replicaSend $POTENTIAL $replica(loc.$swap)
    set doswap [replicaRecv $replica(loc.$swap)]
}

set newloc $r
if { $doswap } {
    set newloc $replica(loc.$swap)
    set replica(loc.$swap) $r
}

set replica(loc.$other) [replicaSendrecv \ $newloc $replica(loc.$other) $replica(loc.$other)]
set oldidx $replica(index)
if { $doswap } {
    set OLDTEMP $replica(temperature)
    array set replica [replicaSendrecv [array get replica] \ $newloc $newloc]
    set NEWTEMP $replica(temperature)
    rescalelevels [expr sqrt(1.0*$NEWTEMP/$OLDTEMP)]
    langevinTemp $NEWTEMP
puts stderr "iteration $i_run replica \n$replica(index) now on rank $r"
#
replicaBarrier

incr i_run

if { $i_run % ($runs_per_frame * $frames_per_restart) == 0 ||
    $i_run == $num_runs } { # restart
    set restart_root "$job_output_root.restart$i_run"
    output [format $restart_root.$replica_id $replica_id]
    set rfile [open [format "$restart_root.$replica_id.tcl" \n        "$replica_id"] "w"]
    puts $rfile [list array set replica [array get replica]]
    close $rfile
    replicaBarrier
    if { $replica_id == 0 } {
        set rfile [open [format "$restart_root.tcl" ""] "w"]
        puts $rfile [list set i_job [expr $i_job + 1]]
        puts $rfile [list set i_run $i_run]
        puts $rfile [list set i_step $i_step]
        puts $rfile [list set restart_root $restart_root]
        close $rfile
        if [info exists old_restart_root] {
            set oldroot [format $old_restart_root ""]
            file delete $oldroot.tcl
        }
    }
    replicateBarrier
    if [info exists old_restart_root] {
        set oldroot [format $old_restart_root $replica_id]
        file delete $oldroot.$replica_id.tcl
        file delete $oldroot.$replica_id.coor
        file delete $oldroot.$replica_id.vel
        file delete $oldroot.$replica_id.xsc
    }
    set old_restart_root $restart_root
}

set attempts $replica(exchanges_attempted)
if $attempts {
    set i $replica(index)
if { $replica(index.a) > $i } {
    set swap a
} else {
    set swap b
}
set temp $replica(temperature)
set temp2 $replica(temperature.$swap)
set accepts $replica(exchanges_accepted)
set ratio [expr 1.0*$accepts/$attempts]
puts stderr "EXCHANGE_RATIO $temp $temp2 \ $accepts $attempts $ratio"
}

replicaBarrier

A.0.5 ASyncRE RE with Temperature Exchange template script
  - alanin_base.namd

#######################################################
# basic simulation options for replica exchange
#######################################################

structure ../namd_re/alanin.psf
coordinates ../namd_re/unfolded.pdb

margin 10.0
stepspercycle @pr@
parameters ../namd_re/alanin.params
exclude scaled1-4
1-4scaling 0.4
switching on
switchdist 7.0
cutoff 8.0
pairlistdist 10.0

langevin on
langevinDamping 10.0

set steps_per_run @steps@
set replica_id @rid@
timestep @stp@

set job_output_root @somename@
set old_output_root @oldname@

set i_run @cycle@ firsttimestep @firststep@

set cycle [expr $i_run+1] set doswap @swap@

set outputname $job_output_root$cycle outputName $outputname

set OLDTTEMP @ot@ set NEWTEMP @nt@

if { $i_run } { # restart
set oldrun [expr ($cycle−1)]
set oldoutput $old_output_root$oldrun
bincoordinates $oldoutput.coor
binvelocities $oldoutput.vel
extendedSystem $oldoutput.xsc
} else {
    temperature $NEWTEMP
}

outputEnergies [expr ($steps_per_run / 10)]
dcdFreq [expr ($steps_per_run * 10)]

if { $doswap } {
    rescalelevels [expr sqrt(1.0*$NEWTEMP/$OLDTTEMP)]
}
langevinTemp $NEWTEMP

#################################################################
# this block is for putting output into history file
#################################################################

proc save_callback {labels values} {
    global saved_labels saved_values
    set saved_labels $labels
    set saved_values $values
}
callback save_callback

proc save_array {} {
global saved_labels saved_values saved_array
foreach label $saved_labels value $saved_values {
    set saved_array($label) $value
}

set history_file [open [format "@history@"] "w"]
fconfigure $history_file -buffering line

#################################################################

run $steps_per_run
save_array

set TEMP $saved_array(TEMP)
set POTENTIAL [expr $saved_array(TOTAL) - $saved_array(KINETIC)]

puts $history_file "$NEWTEMP $POTENTIAL"

A.0.6 NAMD RE with Umbrella Sampling example script
    - umbrella.namd

# validate replica_neighbors proc – works in tclsh
for { set i 0 } { $i < $num_replicas } { incr i } {
    set j 0
    foreach nbr [replica_neighbors $i] {
        if { $nbr < 0 } {
            error "replica_neighbors inconsistency detected: \ neighbor $j of replica $i is $nbr but should not \ be negative"
        }
        if { $nbr >= $num_replicas } {
            error "replica_neighbors inconsistency detected: \ neighbor $j of replica $i is $nbr but there are \ only $num_replicas replicas"
        }
    }
    set rnbrl [replica_neighbors $nbr]
    set rnbrc [llength $rnbrl]
    if { $j >= $rnbrc } {
        error "replica_neighbors inconsistency detected: \ neighbor $j of replica $i is $nbr but replica \ $nbr has only $rnbrc neighbors"
    }

65
set rnbr [lindex $rnbrl $j]
if { $rnbr != $i } {
    error "replica_neighbors inconsistency detected: \neighbor $j of replica $i is $rnbr but neighbor $j \of replica $nbr is $rnbr"
}incr j
}
puts "replica_neighbors proc passes internal consist check"

# bail if this is not NAMD
if { [catch numPes] } {
puts "Tcl interpreter does not appear to be NAMD \script exiting"
return
}

replicaBarrier

set nr [numReplicas]
if { $num_replicas != $nr } {
    error "restart with wrong number of replicas"
}
set r [myReplica]
set replica_id $r

if {{info exists restart_root]} { #restart
    set restart_root [format $restart_root $replica_id]
    source $restart_root.$replica_id.tcl
} else {
    set i_job 0
    set i_run 0
    set i_step 0
    if {{info exists first_timestep]} {
        set i_step $first_timestep
    }

    set replica(index) $r
set nnbr 0
foreach nbr [replica_neighbors $r] {
    set replica(loc.$nnbr) $nbr
    set replica(index.$nnbr) $nbr
    set replica(exchanges_attempted.$nnbr) 0
}
set replica(exchanges_accepted.$nnbr) 0
incr nnbr
}
set replica(num_neighbors) $nnbr

set job_output_root "${output_root}.job$i_job"
firsttimestep $i_step

set replica(colvarbias) [replica_bias $replica(index)]
for { set i 0 } { $i < $replica(num_neighbors) } { incr i } {
  set replica(colvarbias.$i) [replica_bias $replica(index.$i)]
  set replica(colvarbias.$i) [replica_bias $replica(index.$i)]
}

proc save_callback {labels values} {
  global saved_labels saved_values
  set saved_labels $labels
  set saved_values $values
}
callback save_callback

proc save_array {} {
  global saved_labels saved_values saved_array
  foreach label $saved_labels value $saved_values {
    set saved_array($label) $value
  }
}

seed [expr int(0*$rand(int(100000*$rand()) + \100*$replica_id) + 100000*$rand())]
outputname [format ${job_output_root}.$replica_id $replica_id]

if {$i_run} { # restart
  bincoordinates $restart_root.$replica_id.coor
  binvelocities $restart_root.$replica_id.vel
  extendedSystem $restart_root.$replica_id.xsc
  colvarsInput $restart_root.$replica_id.colvars.state
} elseif { [info exists input_root] } {
  set ir [format $input_root $replica_id $replica_id]
  bincoordinates $ir.coor
  binvelocities $ir.vel
  extendedSystem $ir.xsc
} else {

temperature $temperature
}

outputEnergies [ expr $steps_per_run / 10 ]
dcdFreq [ expr $steps_per_run * $runs_per_frame ]

source $namd_config_file

eval colvarbias [ concat changeconfig $replica(colvarbias) ]

set history_file [ open [ format \ "$job_output_root.$replica_id.history" $replica_id ] "w" ]

fconfigure $history_file -buffering line

while { $i_run < $num_runs } {

run $steps_per_run
save_array
incr i_step $steps_per_run
set TEMP $saved_array(TEMP)
set POTENTIAL [ expr $saved_array(TOTAL) - $saved_array(KINETIC) ]
puts $history_file "$i_step $replica(index) $TEMP $POTENTIAL"

set swap [ expr $i_run % $replica(num_neighbors) ]

set doswap 0
if { [ $replica(index) < $replica(index.$swap) ] } {
  set BOLTZMAN 0.001987191
  set ediff [ eval colvarbias [ concat energydiff \ $replica(colvarbias.$swap) ] ]
  set ediff2 [ replicaRecv $replica(loc.$swap) ]
  set delta [ expr ($ediff+$ediff2)/($BOLTZMAN * $temperature) ]
  set doswap [ expr $delta < 0. || exp(-1. * $delta) > rand() ]
  replicaSend $doswap $replica(loc.$swap)
  if { [ $doswap ] } {
    set rid $replica(index)
    set rid2 $replica(index.$swap)
    puts stderr "EXCHANGE_ACCEPT $rid $rid2 RUN $i_run"
    incr replica(exchanges_accepted.$swap)
  }
  incr replica(exchanges_attempted.$swap)
}
if { [ $replica(index) > $replica(index.$swap) ] } {
  set ediff [ eval colvarbias [ concat energydiff \
$r e p l i c a ( c o l v a r b i a s . S w a p ) ]$

`replicaSend $ediff $replica(loc.$swap)`

`set doswap [replicaRecv $replica(loc.$swap)]`

```tcl
}

set newloc $r
if { $doswap } {
    set newloc $replica(loc.$swap)
    set replica(loc.$swap) $r
}
```

for { set i 0 } { $i < $replica(num_neighbors) } { incr i } {
    if { $i != $swap } {
        set replica(loc.$i) [replicaSendrecv $newloc \ 
        $replica(loc.$i) $replica(loc.$i)]
    }
}
```

set oldidx $replica(index)
if { $doswap } {
    array set replica [replicaSendrecv \ 
    [array get replica] $newloc $newloc]
    eval colvarbias [concat changeconfig $replica(colvarbias)]
}
```

incr i_run
```
if { $i_run % ($runcycle * $frames_per_restart) == 0 || $i_run == $num_runs } { # restart
    set restart_root "$job_output_root.restart$i_run"
    set rroot [format "$restart_root.$replica_id $replica_id"]
    output $rroot
    set oroot [format "$job_output_root.$replica_id $replica_id"]
    file rename -force $root.colvars.state $rroot.colvars.state
    set rfile [open [format "$restart_root.$replica_id.tcl "$] \ $replica_id] "w"]
    puts $rfile [list array set replica [array get replica]]
    close $rfile
    replicaBarrier
    if { $replica_id == 0 } {
        set rfile [open [format "$restart_root.tcl ""] "w"]
        puts $rfile [list set i_job [expr $i_job + 1]]
        puts $rfile [list set i_run $i_run]
        puts $rfile [list set i_step $i_step]
        puts $rfile [list set restart_root $restart_root]
        close $rfile
```
if [ info exists old_restart_root ] {
    set oldroot [ format $old_restart_root "" ]
    file delete $oldroot.tcl
}
}

replicaBarrier
if [ info exists old_restart_root ] {
    set oldroot [ format $old_restart_root $replica_id ]
    file delete $oldroot.$replica_id.tcl
    file delete $oldroot.$replica_id.coor
    file delete $oldroot.$replica_id.vel
    file delete $oldroot.$replica_id.xsc
    file delete $oldroot.$replica_id.colvars.state
}

set old_restart_root $restart_root
}
}

for { set i 0 } { $i < $replica(num_neighbors) } { incr i } {
    set attempts $replica(exchanges_attempted.$i)
    if $attempts {
        set accepts $replica(exchanges_accepted.$i)
        set ratio [ expr 1.0*$accepts/$attempts ]
        puts stderr "EXCHANGE_RATIO $replica(index) \ $replica(index.$i) $accepts $attempts $ratio"
    }
}

replicaBarrier

A.0.7 ASyncRE RE with Umbrella Sampling template script - alanin_base_umbrella.namd

# simulation options for replica exchange
structure ../.namd_us/alanin.psf
coordinates ../.namd_us/alanin.pdb

margin 10.0
stepspercyle @pr@
parameters ../.namd_us/alanin.params
exclude scaled1 -4
1-4scaling 0.4
switching on
switchdist 7.0
cutoff 8.0
pairlistdist 10.0

colvars on
colvarsConfig ../.namd_us/colvars.conf

langevin on
langevinDamping 10.0

set steps_per_run @steps@
set replica_id @rid@
set bias_id @bid@
timestep @stp@
set num_replicas @replicas@

set job_output_root @somename@
set old_output_root @oldname@

set i_run @cycle@
firsttimestep @firststep@
set r_temperature @temper@
set cycle [expr $i_run + 1]

langevinTemp $r_temperature

proc replica_bias { i } {
    return [list lenpot "centers [expr 17 + $i"]"]
}

# this block is for putting output into history file

callback save_callback {labels values} {
    global saved_labels saved_values
    set saved_labels $labels
    set saved_values $values
}
callback save_callback

proc save_array {} {
    global saved_labels saved_values saved_array

    set saved_array [list]
    foreach l $saved_labels {
        set saved_array [list $saved_array [list $l]]
    }
    set saved_values [list]
    foreach v $saved_values {
        set saved_values [list $saved_values [list $v]]
    }
    callback save_callback labels $labels values $values
}

next_callback {labels values} {
    global saved_labels saved_values
    set saved_labels $labels
    set saved_values $values
}
callback next_callback

next_row_callback {
    global saved_labels saved_values
    set saved_labels [list]
    set saved_values [list]
}
callback next_row_callback

callback all

foreach label $saved_labels value $saved_values {
    set saved_array($label) $value
}

set history_file [open [format "@history@"] "w"]
fconfigure $history_file -buffering line

# energy output
set energy_file [open [format "@energy@"] "w"]
fconfigure $energy_file -buffering line

#######################################################

set replica(colvarbias) [replica_bias $bias_id]

for { set i 0 } { $i < $num_replicas } { incr i } {
    set replica(colvarbias.$i) [replica_bias $i]
    # puts $energy_file "$i $replica(colvarbias.$i)"
}

seed [expr int(0*$rand(int(100000*$rand()) + \100*$replica_id) + 100000*$rand())]
set outputname $job_output_root$cycle
outputName $outputname

set input_root "../namd_us/input/alanim.initial.$replica_id"

if {$i_run} { # restart
    set oldrun [expr ($cycle - 1)]
    set oldoutput $old_output_root$oldrun
    bincoordinates $oldoutput.coor
    binvelocities $oldoutput.vel
    extendedSystem $oldoutput.xsc
    colvarsInput $oldoutput.colvars.state
}
elseif { [info exists input_root] } {
    bincoordinates $input_root.coor
    binvelocities $input_root.vel
    extendedSystem $input_root.xsc
}
else {
    temperature $r_temperature
}
outputEnergies [expr $steps_per_run / 10]
dcdFreq [expr $steps_per_run * 10]

eval colvarbias [concat changeconfig $replica(colvarbias)]

run $steps_per_run
save_array

for { set i 0 } { $i < $num_replicas } { incr i } {
    set ediff [eval colvarbias [concat energydiff \ $replica(colvarbias.$i)]]
    puts $energy_file "$ediff"
}

set TEMP $saved_array(TEMP)
set POTENTIAL [expr $saved_array(TOTAL)−$saved_array(KINETIC)]
puts $history_file "$TEMP $POTENTIAL $bias_id"
Bibliography


[38] MPI for Python - mpi4py. Online at: http://mpi4py.scipy.org/ (referenced 26/07/2013)