A Framework For Replica Exchange Simulations.

By

Antons Treikalis

A thesis submitted to the

Graduate School-New Brunswick

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Dr. Shantenu Jha

And approved by

____

____

____

____

New Brunswick, New Jersey

May, 2017

ABSTRACT OF THE THESIS

A Framework For Replica Exchange Simulations

By ANTONS TREIKALIS

Thesis Director:
Shantenu Jha

Replica Exchange (RE) simulations have emerged as an important algorithmic tool for the molecular sciences. Typically RE functionality is integrated into the molecular simulation software package. A primary motivation for the tight integration of RE functionality with simulation codes has been performance. This is limiting at multiple levels. First, advances in the RE methodology are tied to the molecular simulation code for which they were developed. Second, it is difficult to extend or experiment with various RE methods, since expertise in the molecular simulation code is required. We propose the RepEx framework which addresses the aforementioned limitations, while striking the balance between flexibility (RE pattern) and scalability (several thousand replicas) over a diverse range of HPC platforms. In this thesis is introduced the RepEx framework, the primary contributions of which are: (i) its ability to support different RE patterns independent of molecular simulation codes, (ii) the ability to execute different exchange types and replica counts independent of the specific availability of resources, (iii) a runtime system that has first-class support for task- level parallelism, and (iv) provide the required scalability along multiple dimensions.

Acknowledgements

I would like to thank my brother Adams for supporting me throughout my studies at Rutgers University. I also would like to thank my mother Ludmila and my father Viktors for their effort in raising me. I would like to thank Dr. Shantenu Jha for his advice and support. Without him this thesis would not be possible. Next, I would like to thank Dr. Matteo Turilli and Andre Merzky for their patience, help and feedback. Finally, I would like to thank all members for the RADICAL Group for their help in shaping of this thesis.

Table of Contents

Chapter 1
Introduction

Significant progress has been made, in recent years, to develop an understanding of the inner workings of biological systems, consequently, the significance of the biomolecular simulations increased dramatically. For the majority of proteins only amino-acid sequences are available [1], meaning that to obtain a detailed 3D structure of a protein, a protein folding molecular dynamics (MD) simulation must be carried out.

Hydrogen atoms in a biomolecule vibrate with a period of approximately 10 femtosecond (fs) [1], as a result, a time-step value of approximately 1 fs is required for an accurate and scalable integration for MD simulations. Additionally, simulation time necessary to observe the desired changes, must be in the range of microseconds. Simply to allow a protein to relax into the nearest low-energy state, an MD simulation must run for at hundreds nanoseconds [1]. As a result, simulation time required for protein folding simulations is approximately in the range from 1×10^6 to 1×10^9 time-steps.

To reduce the simulation time of the MD simulation, requiring millions of time-steps, one must take advantage of the capabilities of HPC clusters. In MD simulation, typically the problem size is fixed, creating a barrier for parallelization of the simulation.

For most proteins, obtaining samples at low temperatures using conventional MD simulations is hard, since simulations at low temperatures often get trapped at local minimum energy states. This sampling limitation can be mitigated by performing a random walk in temperature space, which in turn results in a random walk in energy space. Consequently, trapping in local minima energy states is avoided.

Replica Exchange (RE) [2] algorithm allows to perform a random walk in energy space (due to the non-Boltzmann weight factors) and as a result to overcome energy barriers. Although RE methods were introduced for Monte Carlo methods, their use with MD has grown rapidly. RE [2] is a popular technique to enhance sampling in molecular simulations. This is evidenced every year, by several hundred publications, using some variant of Replica Exchange in a range of scientific disciplines including chemistry, physics, biology and materials science.

REMD simulation starts with concurrent MD simulation of N non-interacting replicas of the original molecule (or system of the molecules). To enhance sampling, replicas are initialized at different thermodynamic configurations. We call this part of the REMD simulation an MD phase. After MD phase is done, replicas are attempting an exchange of their thermodynamic configurations. Acceptance of exchanges is determined using Metropolis criterion [3]. We refer to this part of the REMD simulation as exchange phase. After an exchange phase is done, the process in then repeated.

Initially, REMD [4] was used to perform exchanges of temperatures, but since has been extended to perform Hamiltonian Exchange [5], pH Exchange [6] and other exchange types.

Reinforcing the importance of REMD, many community MD engines [7, 1, 8], introduced support for REMD. Often these solutions are integrated with MD simulation engine and are using one of the popular parallel programming models (message passing, shared memory, etc.). As a result, these solutions demonstrate respectable performance, but share some important limitations. REMD functionality implemented in the same code space with MD engine, results in a tight integration of the RE algorithm with MD algorithm.

One of the major drawbacks of the integration of REMD capability with MD simulation engine is unnecessary duplication of development effort between ”competing” MD engines. Tight integration of implementation of REMD functionality with implementation of MD functionality, confines implementations of various REMD methods within a single code base, making it difficult to propagate this capability across community codes. MD engines [7, 1, 8] are highly optimized and specialized codes, often requiring tens of person-years of development. Domain scientists are typically unprepared for the full complexity of these MD engines, yet they are the ones most capable of algorithmic and methodological innovation.

To avoid the aforementioned limitations, REMD framework must decouple the RE methodology from the MD engine. To maximize execution flexibility, replicas should not be bounded by the computational resources (CPUs, GPUs, etc.) available. Unfortunately, this is not the case for many REMD frameworks. Furthermore, a REMD framework, at a minimum, should ensure that in the presence of a failure of a single replica (or multiple replicas), the entire simulation does not have to be stopped and then restarted.

While REMD is very well suited for parallelization and does not impose a requirement for a global synchronization barrier between MD and exchange phase. Existence of this barrier in most REMD software packages is an implementation decision, motivated by desire to reduce the complexity of the code. Removal of a global synchronization barrier introduces asynchronicity between MD and exchange phases. This asynchronicity can significantly improve resource utilization, when wallclock time of the replicas for MD phase is highly variable. Performance variability for replicas in REMD simulations can be caused by the heterogeneity of the computational resources or by the nature of the simulation. One such example, in quantum mechanics / molecular mechanics (QM/MM) [9] simulations, where variability in performance of individual replicas is caused by the nature of the QM/MM simulation.

To overcome the above limitations, we develop RepEx [10] software framework. RepEx is designed to satisfy the range of functional, performance and usability requirements for REMD frameworks. Conceptually, RepEx aims to decouple the implementation of the RE algorithm from the MD simulation engine. RepEx is not limited in the number and ordering of dimensions for multi-dimensional REMD simulations and currently supports three exchange types: salt concentration, temperature and umbrella exchange. Despite the fact, that we mainly were interested in Amber MD engine, to demonstrate extensibility and modularity of RepEx, we introduced support for NAMD MD engine.

Our framework relies on RADICAL-Pilot (RP) [11] as a runtime system to perform resource allocation, task scheduling and data movement. RADICAL-Pilot not only provides means for resource management via its API, but also enables separation of computational requirements of the simulation from the REMD implementation. Another distinctive feature of RepEx, partly arising from the use of RP, is fault tolerance: RepEx can either continue a simulation in the case of replica(s) failure or can relaunch a failed replica(s).

The functionality and flexibility come at a performance price, especially when compared to highly-customized approaches. We characterize the performance of our framework and argue that given the range of requirements, it is satisfactory. The design of RepEx facilitates implementation of new RE methods with a wide range of MD engines.

1.1 Aims

One of the major aims of this thesis is to develop a design of the framework for replica exchange. Proposed design should be motivated by the set of requirements for the REMD simulation frameworks and evaluated using a set of quantitative and qualitative metrics. Proposed design should facilitate re-usability of the existing modules, be independent of the MD simulation engine, and support asynchronous RE simulations.

To come up with a successful design for the REMD framework, it is necessary to understand: theoretical concepts central to MD and RE, limitations of the current software packages with integrated REMD capability and user requirements. Additionally, understanding of how computational resources will evolve in the future is of a major significance.

To evaluate the proposed design, we aim to develop an implementation of the REMD framework. It is essential to motivate implementation decisions, verify that software meets identified requirements and provide scientific validation.

1.2 Structure of the dissertation

There are seven chapters in this dissertation. In first chapter we provide motivation for the dissertation. We briefly introduce MD simulations and RE algorithm. We then highlight computational and software barriers limiting scientific progress. To finalize this chapter are outlined objectives and structure of the dissertation.

Second chapter is designed to provide background information, necessary to appreciate this thesis. We first introduce theoretical background of the MD and REMD simulations. We briefly discuss Quantum Mechanics / Molecular Mechanics (QM/MM) simulations, since we use QM/MM for our experiments. Next, we motivate the need for asynchronous RE and introduce multi-dimensional RE. Then, we provide an overview of the molecular simulation software packages with integrated REMD capability (Amber and NAMD). We also discuss some of the REMD software frameworks, which implement REMD algorithm outside of the MD engine. We finalize the background chapter with discussion of pilot systems and RADICAL-Pilot in particular.

Chapter three is dedicated to RepEx framework. We first outline requirements of the REMD software packages and categorize them into three types: functional, performance (scalability) and usability requirements. Next, we describe concepts, central to the design of the RepEx framework and introduce an asynchronous RE algorithm. We also provide implementation details of the RepEx framework and scientifically validate our implementation by performing a series of experiments.

In chapter four we present and discuss experiments performed to optimize the performance of the RepEx. We briefly describe performance optimization strategy utilized to reduce the total wallclock time. Our strategy consists of five optimization steps, and mainly is focused at minimization of file movement and minimization of the number of tasks. Next, we characterize performance of the RepEx framework for various REMD simulations and provide experimental evidence highlighting differences between synchronous and asynchronous RE.

In chapter five we analyze obtained results. To demonstrate how RepEx performs, in comparison with other software packages, we present a table in which we characterize six other software packages using eight features. We select these features using out intuition and user feedback.

We outline the key conclusions in chapter six. In this chapter we discuss the lessons learned while working on this dissertation. We discuss how well RepEx satisfies the set of requirements we have identified in chapter three and how well we achieved the objectives of the dissertation. Finally we highlight the main barriers for the uptake of the RepEx framework.

In the last chapter we present possible directions for the future development of the RepEx.

Chapter 2

Background

In this chapter we present concepts central to this thesis. We briefly discuss theory behind REMD simulations and discuss some of the software packages for REMD simulations. We also introduce the concept of a Pilot system, which plays an important role in the design of our framework.

First we introduce theoretical concepts behind MD simulations. Next we discuss REMD simulations from both theoretical and practical point of view. At a very high level we cover QM/MM simulations. Next we introduce the concept of asynchronous RE and explain the logic behind multi-dimensional REMD simulations. In section 2.6, we discuss two, arguably most popular molecular simulation software packages with integrated REMD capability, namely Amber and NAMD. We also discuss some of the modern software frameworks, implementing REMD functionality at a higher software layers. We finalize this chapter by discussing Pilot systems and RADICAL Pilot, as an implementation of a Pilot system.

2.1 Molecular Dynamics

Computer simulations are aimed at understanding the properties of molecules defined by their interactions and structure. Simulation techniques can be broadly divided into two groups: MD simulations and Monte Carlo simulations (MC). Naturally, there are other simulation techniques, but these two are the most popular ones. Unlike other simulation techniques, MD provides an insight into dynamic properties of the system.

Unlike MC simulation, MD simulation of a molecular system acting under potential energy $U(q)$ involves numerical, step-by-step, solution of the Newton's equations of motion:

$$p_i = \dot{q}_i m_i \qquad \dot{p}_i = -\frac{\partial U(q_i)}{\partial q_i} \tag{2.1}$$

Forces p_i are acting on atoms, and are derived from a potential energy $U(q^N)$, where $q^N = (q_1, q_2, ..., q_N)$ represents the complete set of atomic coordinates. Consequently, equation 2.1 yields trajectories in the phase space and is a Hamiltonian system.

2.1.1 Hamiltonian Systems

Hamiltonian systems are linear or non-linear systems with particular symmetry, what allows the stability of equilibrium points to be found and the solution curves to be drawn even though actual solutions are not obtained. For Hamiltonian systems the derivatives $\frac{dq_i}{dt}$ and $\frac{dp_i}{dt}$ are partial derivatives of a Hamiltonian function H. Hamiltonian system is defined as:

$$\frac{dq_i}{dt} = \frac{\partial H}{\partial p_i} \tag{2.2}$$

$$\frac{dp_i}{dt} = -\frac{\partial H}{\partial q_i} \tag{2.3}$$

Classical equations of motion form a Hamiltonian system, with Hamiltonian function defined as:

$$H(q,p) = U(q) + K(p) = U(q) + \sum_i \frac{p_i^2}{2m_i} \tag{2.4}$$

where: $U(q)$ is potential energy $K(p)$ is kinetic energy

We now show that equations 2.1 form a Hamiltonian system. First we note that:

$$p_i = m_i v_i \tag{2.5}$$

$$\frac{dp_i}{dt} = m_i \frac{dv_i}{dt} \tag{2.6}$$

Next we note that:

$$-\frac{\partial U}{\partial q_i} = F_i(q) \tag{2.7}$$

where F is a force vector

We now take a derivative of Hamiltonian H with respect to q_i:

$$-\frac{\partial H}{\partial q_i} = -\frac{\partial U}{\partial q_i} = F_i(q) = ma_i = m_i \frac{dv_i}{dt} = \frac{dp_i}{dt} \tag{2.8}$$

Next we take a derivative of Hamiltonian H with respect to p_i:

$$-\frac{\partial H}{\partial p_i} = \frac{\partial K}{\partial p} = \frac{p_i}{m} = \frac{m_i v_i}{m_i} = v_i = \frac{dq_i}{dt} \tag{2.9}$$

Now we write Hamiltonian system for the classical equations of motion:

$$\dot{p_i} = -\frac{\partial H}{\partial q_i} = F_i(q) \qquad \dot{q_i} = \frac{\partial H}{\partial p_i} = \frac{p_i}{m_i} \tag{2.10}$$

2.1.2 Phase space

The term **phase space** is often mentioned when MD simulations are discussed. A phase space is a $6N$-dimensional space in which there is an axis for every coordinate q_N and for every momentum p_N. If a system of Cartesian coordinates is used, then for each atom there are x, y, z and p_x, p_y, p_z axes.

Consequently we can describe system change over time as a trajectory in phase space. If we are dealing with a classical system, the energy is conserved and the phase space trajectory adheres to a surface of constant energy [12].

Ergodic systems, which can reach equilibrium, are capable of exploring all parts of phase space, having the same energy. In other words, a trajectory of a system, which is ergodic, will reach all points in phase space on the constant energy hypersurface with equal probability. If given system is not ergodic, some of the phase space points will remain unvisited and as a consequence the system will not be capable to examine the complete constant energy surface [12].

A **differential volume element** in phase space is a very small region of $6N$-dimensional volume that spans differential elements:

$$dq^N dp^N = dx_1 dy_1 ... dz_N dp_{x,1} dp_{y,1} ... dp_{z,N} \tag{2.11}$$

Configuration space is the subset of phase space corresponding to the $3N$ variables q^N.

2.1.3 Numerical Integration

A numerical integrator calculates an approximate solution trajectory of the system of ordinary differential equations given a time-step and initial positions and velocities of the atoms. The most widely used numerical integrators are: the Verlet algorithm [13], the Leapfrog algorithm [14] and the Velocity Verlet algorithm [15].

The Verlet algorithm offers better stability, time-reversibility and area preserving properties with little additional computational cost in comparison to the Euler method. The solution of Newton's equations of motion using the Verlet algorithm is based on a Taylor series expansion. The two updating steps of the Verlet method are:

$$q_i(t + \Delta t) = 2q_i(t) - q_i(t - \Delta t) + \frac{1}{m_i}\frac{\partial U}{\partial q_i}(t)\Delta t^2 \tag{2.12}$$

$$p_i(t) = m_i \frac{(q_i(t + \Delta t) - q_i(t - \Delta t))}{2\Delta t} \tag{2.13}$$

The Verlet Algorithm relies on two previous time steps, t and $t - \Delta t$, to advance the solution forward. One of the problems with the Verlet algorithm, is the fact, that due to potential loss of accuracy when implementing equation 2.12, the final term in the equation tends to be small and this term may be lost in the round-off error.

The Leapfrog algorithm was developed to correct some of the problems associated with the Verlet algorithm. The name of the algorithm comes from the fact, that q and p are "leapfrogging" over each other during integration. The two updating steps of the Leapfrog algorithm are:

$$q_i(t + \Delta t) = q_i(t) + \frac{p_i(t + \frac{\Delta t}{2})}{m_i} \Delta t \tag{2.14}$$

$$p_i(t + \frac{\Delta t}{2}) = p_i(t - \frac{\Delta t}{2}) + \frac{\partial U}{\partial q_i}(t) \Delta t \tag{2.15}$$

The two updating steps of the Velocity Verlet algorithm are:

$$q_i(t + \Delta t) = q_i(t) + \frac{p_i(t)}{m_i} \Delta t + \frac{1}{2m} \frac{\partial U}{\partial q_i}(t) \Delta t^2 \tag{2.16}$$

$$p_i(t + \Delta t) = p_i(t) + \frac{\Delta t}{2} (\frac{\partial U}{\partial q_i}(t) + \frac{\partial U}{\partial q_i}(t + \Delta t)) \tag{2.17}$$

All three numerical integration algorithms produce deterministic dynamical systems, meaning that there is no stochastic element. In addition, because these algorithms are solving a Hamiltonian system, they conserve the total energy [16]. We define the total energy as:

$$E(q, p) = U(q) + K(p) \tag{2.18}$$

Each of the three algorithms discussed in this section, when used for MD simulation, produces trajectories in a **microcanonical ensemble**: a set of all molecular configurations with constant energy. Due to the rounding errors of computer hardware and the choice of the step-size, the configurations in the trajectories do not have exactly the same energy. The advantage of the algorithms, presented in this section, is their ability to restrain the energy from divergence.

2.2 Replica Exchange Molecular Dynamics

While undertaking simulations of complex systems such as proteins, using MD or MC methods, it is problematic to obtain accurate canonical distributions at low temperatures [17]. Replica Exchange Markov chain Monte Carlo (MCMC) sampling 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 [2], but MD version of the method, which is known as REMD was first formulated by Sugita and Okamoto [4] in 1999. Today RE is applied in many scientific fields including chemistry, physics, biology, materials science and other.

In RE 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 initialized 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 [18]. RE 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 subsets 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, 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, RE method can be very efficiently mapped to distributed-memory architectures of HPC clusters.

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. 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 [18].

RE simulations may involve exchange 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 REMD. Due to better sampling of phase space, enabled by the use of RE method, inconsistencies in some widely used force fields were discovered [18]. Consequently, RE method now plays an important role in testing of new force fields for atomic simulations.

2.2.1 Theory of Replica Exchange method

In 2.4 we have defined the Hamiltonian function. With this Hamiltonian function, each state $x \equiv (q, p)$ at temperature T is weighted by the Boltzmann facotor:

$$W_B(x; T) = e^{-\beta H(q,p)} \tag{2.19}$$

where: β is the inverse temperature

We define the inverse temperature by:

$$\beta = \frac{1}{k_B T} \tag{2.20}$$

where: k_B is the Boltzmann constant

We define the average kinetic energy at temperature T by:

$$\langle K(p) \rangle_T = \langle \sum_{k=1}^{N} \frac{p_k^2}{2m_k} \rangle_T = \frac{3}{2} N k_B T \tag{2.21}$$

where:

N is the number of atoms

m_k is mass of atom $(k = 1, 2, ..., N)$

p is a momentum vector

In the RE method, are initialized M non-interacting replicas of the original system in the canonical ensemble, at M different temperatures T_m $(m = 1, 2, ...M)$. There is a one-to-one mapping between replicas and temperatures. We can summarize this by:

$$\begin{cases} i = i(m) & \equiv f(i), \\ m = m(i) & \equiv f^{-1}(i) \end{cases} \tag{2.22}$$

where:

i is a label for replicas $i = (1, 2, ...M)$ and is a permutation of label m for temperatures

m is a label for temperatures $m = (1, 2, ...M)$ and is a permutation of label i for replicas

$f(m)$ is a permutation function of m

$f^{-1}(i)$ is inverse of a permutation function

We define a state of an ensemble by:

$$X = (x_1^{[i(1)]}, ..., x_M^{[i(M)]}) = (x_{m(1)}^{[1]}, ..., x_{m(M)}^{[M]}) \tag{2.23}$$

where: superscript of x labes the replica

subscript of x labels the temperature

The state X is defined by the M sets of coordinates $q^{[i]}$ and momenta $p^{[i]}$ of N atoms in replica i in temperature T_m:

$$x_m^{[i]} = (q^{[i]}, p^{[i]})_m \tag{2.24}$$

By definition of the method, replicas are non-interacting. Consequently, the weight factor for the state X is given by the product of Boltzmann factors for each replica (or at each temperature):

$$W_{REM}(X) = exp\{-\sum_{i=1}^{M} \beta_{m(i)} H(q^{[i]}, p^{[i]})\} = exp\{\sum_{m=1}^{M} \beta_m H(q^{[i(m)]}, p^{[i(m)]})\} \tag{2.25}$$

where: $i(m)$ and $m(i)$ are the permutation functions.

Lets examine an exchange between a pair of replicas i and j, which are at temperatures T_m and T_n:

$$X = (..., x_m^{[i]}, ..., x_n^{[j]}, ...) \rightarrow X' = (..., x_m^{[j]'}, ..., x_n^{[i]'}, ...) \tag{2.26}$$

where: i, j, n and m are replated to the permutation functions in 2.22

The exchange of replicas i and j introduces a new permutation function f':

$$\begin{cases} i = f(m) & \rightarrow j = f'(m), \\ j = f(n) & \rightarrow i = f'(n) \end{cases} \tag{2.27}$$

We can expand an exchange of replicas as:

$$\begin{cases} x_m^{[i]} \equiv (q^{[i]}, p^{[i]})_m & \rightarrow x_m^{[j]'} \equiv (q^{[j]}, p^{[j]'})_m, \\ x_n^{[j]} \equiv (q^{[j]}, p^{[j]})_n & \rightarrow x_n^{[i]'} \equiv (q^{[i]}, p^{[i]'})_n \end{cases} \tag{2.28}$$

This is equivalent to an exchange of temperatures T_m and T_n for the corresponding replicas i and j:

$$\begin{cases} x_m^{[i]} \equiv (q^{[i]}, p^{[i]})_m & \rightarrow x_n^{[i]'} \equiv (q^{[i]}, p^{[i]'})_n, \\ x_n^{[j]} \equiv (q^{[j]}, p^{[j]})_n & \rightarrow x_m^{[j]'} \equiv (q^{[j]}, p^{[j]'})_m \end{cases} \tag{2.29}$$

In 2.28 and 2.29 momenta $p^{[i]'}$ and $p^{[j]'}$ are defined as:

$$\begin{cases} p^{[i]'} & \equiv \sqrt{\frac{T_n}{T_m}} \quad p^{[i]}, \\ p^{[j]'} & \equiv \sqrt{\frac{T_m}{T_n}} \quad p^{[j]} \end{cases} \tag{2.30}$$

Assignment in 2.30 means that velocities of all the atoms are rescaled uniformly by the square root of the ratio of the two temperatures, so that condition in 2.20 may be satisfied.

To ensure that exchange process converges towards an equilibrium distribution, it is sufficient to impose the detailed balance condition on the transition probability:

$$W_{REM}(X)w(X \to X') = W_{REM}(X')w(X' \to X) \tag{2.31}$$

From 2.4, 2.20, 2.24, 2.30 and 2.31 we get:

$$\begin{aligned} \frac{w(X \to X')}{w(X' \to X)} &= exp\{-\beta_m[K(p^{[j]'} + U(q^{[j]})] - \beta_n[K(p^{[i]'} + U(q^{[i]})] \\ &+ \beta_m[K(p^{[i]} + U(q^{[i]})] + \beta_n[K(p^{[j]} + U(q^{[j]})]\} \\ &= exp\{-\beta_m \frac{T_m}{T_n} K(p^{[j]}) - \beta_n \frac{T_n}{T_m} K(p^{[i]}) + \beta_m K(p^{[i]}) + \beta_n K(p^{[j]}) \\ &- \beta_m[U(q^{[j]}) - U(q^{[i]})] - \beta_n[U(q^{[i]}) - U(q^{[j]})]\} \\ &= exp(-\Delta) \end{aligned} \tag{2.32}$$

where:

$$\Delta \equiv [\beta_n - \beta_m](U(q^{[i]}) - U(q^{[j]})) \tag{2.33}$$

i, j, m and n are related by the permutation functions 2.22:

$$i = f(m), \quad j = f(n) \tag{2.34}$$

This is satisfied by the Metropolis criterion:

$$w(X \to X') \equiv w(x_m^{[i]}|x_n^{[j]}) = \begin{cases} 1, & for \Delta \leq 0, \\ exp(-\Delta) & for \Delta > 0 \end{cases} \tag{2.35}$$

Without loss of generality, we can assume $\beta_1 < \beta_2 < ... < \beta_M$. RE simulation is then performed by the following steps:

1. N replicas (with a fixed temperatures) are simulated simultaneously and independently for a certain number of MD steps

2. A pair of replicas at neighboring temperatures, for example $x_m^{[i]}$ and $x_{m+1}^{[j]}$ are exchanged with the probability $w(x_m^{[i]}|x_{m+1}^{[j]})$ as specified in 2.35.

For the second step, are exchanged only pairs of replicas with neighboring temperatures, since the acceptance ratio of the exchange decreases exponentially with the difference of the two βs. In addition, when an exchange is accepted, the permutation functions in 2.22 are updated.

The main advantage of the RE method stems from the fact, that the weight factor is known in advance. In other methods, such as simulated tempering, determination of weight factors can be very time consuming. To maximize sampling quality and to minimize required computational effort, distribution of temperatures and the number of replicas should be choosen with care. It was demonstrated that using a geometric progression for temperatures, so that $\frac{T_i}{T_j}$ is 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 [18]. Various studies [19] [20] showed that exchange acceptance probability around 20% results in best possible performance of the RE simulation.

We calculate the canonical expectation value of a physical quantity A at a temperature T_m $(m = 1, ..., M)$ as:

$$\langle A \rangle_{T_m} = \frac{1}{N_{sim}} \sum_{t=1}^{N_{sim}} \sum_{i=1}^{M} A\left[x_{f^{-1}(i;t)}^{[i]}(t)\right] \delta_{f^{-1}(i;t),m} \tag{2.36}$$

where:

N_{sim} is the total number of measurements made for each replica

$f^{-1}(i;t)$ is the permutation function from 2.22 at tth measurement

$\delta_{k,l}$ is Kronecker's delta function

We can also write equation 2.36 as:

$$\langle A \rangle_{T_m} = \frac{1}{N_{sim}} \sum_{t=1}^{N_{sim}} A\Big(x_m^{\left[f(m;t)\right]}(t)\Big) \tag{2.37}$$

For the expectation value at any intermediate temperature, we use the multiple-histogram reweighting technique as follows. Lets assume that we have made R-independent simulation runs at R different temperatures. If we denote the energy histogram as $N_m(E)$ and the total number of samples obtaied in the mth run as n_m, then $n_m = N_{sim}$ and the expectation vaue of a physical quantity A at any intermediate temperature $T = \frac{1}{k_B \beta}$ is given by:

$$\langle A \rangle_T = \frac{\sum_E A(E) P(E;\beta)}{\sum_E P(E;\beta)} \tag{2.38}$$

where:

$$P(E;\beta) = \frac{\sum_{m=1}^{R} g_m^{-1} N_m(E) e^{-\beta E}}{\sum_{m=1}^{R} n_m g_m^{-1} e^{f_m - \beta_m E}} \tag{2.39}$$

and

$$e^{-f_m} = \sum_E P(E;\beta_m) \tag{2.40}$$

where:

$g_m = 1 + 2\tau_m$ is the integrated autocorrelation time at temperature T_m

τ_m is the integrated autocorrelation time at temperature T_m

2.3 Quantum Mechanics/Molecular Mechanics

In hybrid quantum mechanics / molecular mechanics (QM/MM) simulations, the system is divided into two regions: the quantum mechanics region, in which the chemical process takes place, is simulated using quantum chemistry theory, the molecular mechanics region, constitutes the remainder of the system and is described by a molecular mechanics force field. QM/MM methods was originally introduced by Warshel and Levitt [21]. QM/MM enables the study of the chemical reactivity in large systems, such as enzymes [9]. QM/MM method is depicted in Figure 2.1.

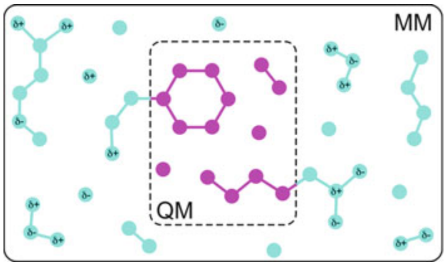


Figure 2.1: Illustration of the QM/MM method. A region, in which a chemical reaction occurs (cannot be described with a force field), is described by quantum mechanical theory. The remainder of the system is modelled using molecular mechanics [9].

In molecular mechanics we describe the potential energy of the system as:

$$U_{MM} = \sum_{i}^{N_{bonds}} U_i^{bond} + \sum_{j}^{N_{angles}} U_j + \sum_{l}^{N_{torsions}} U_l^{torsion} + \sum_{i}^{N_{MM}} \sum_{j>i}^{N_{MM}} U_{ij}^{coul} + \sum_{i}^{N_{MM}} \sum_{j>i}^{N_{MM}} U_{ij}^{LJ} \quad (2.41)$$

where:

N_{MM} - the number of atoms in the system

U^{bond} - bonds (modelled using harmonic functions)

U^{angle} - angles (modelled using harmonic functions)

$U^{torsion}$ - torsions (modelled using periodic functions)

U_{ij}^{coul} - the pairwise electrostatic interaction between atoms with a partial charge (Q_i)

U_{ij}^{LJ} - Lennard-Jones potential

We calculate U_{ij}^{coul} using Coulomb's law:

$$U_{ij}^{coul} = \frac{e^2 Q_i Q_j}{4\pi\epsilon_0 R_{ij}} \quad (2.42)$$

where:

R_{ij} - the inter-atomic distance

e - the unit charge

ϵ_0 - the dielectric constant

The Lennard-Jones potential is given by:

$$U_{ij}^{LJ} = \left(\frac{C_{12}^{ij}}{R_{ij}}\right)^{12} - \left(\frac{C_6^{ij}}{R_{ij}}\right)^{6} \tag{2.43}$$

where:

C_{12}^{ij} - repulsion parameter (depends on the atom types of the atoms i and j)

C_6^{ij} - attraction parameter (depends on the atom types of the atoms i and j)

In molecular mechanics electrons are ignored. Influence of electrons is modelled by empirical parameters. As a consequence, processes that involve electronic rearrangements, such as chemical reactions, cannot be described in molecular mechanics. To describe these processes are used quantum mechanics, where the electronic degrees of freedom can be defined. Evaluation of the electronic structure, is very computationally intensive and as a result the size of the system of interest often is restricted in size.

Many biochemical systems can't be easily described at a single level of theory due to their large size. In addition, molecular mechanics are lacking in flexibility when used to model processes in which chemical bonds are broken or formed [9].

The QM/MM method stems for the nature of the most chemical reactions. Often it is possible to make a clear distinction between a reaction region, where atoms that are directly involved in the reaction and a remainder of the system, where direct participation of the atoms in the chemical reaction is not observed. For most enzymes, the catalytic process is restricted to an active part, often located inside the protein. The rest of the system provides an electrostatic background that is not necessarily facilitating a reaction.

Potential energy in QM/MM is divided into three interaction types: interactions between atoms in the QM region, interactions between atoms in the MM region and interactions between QM and MM atoms. Atom interactions in QM and MM regions are described by the corresponding theories. Interactions between QM and MM atoms can't be easily described. To address this issue a number of approaches were proposed, which can be divided into two categories: subtractive and additive coupling schemes.

2.4 Asynchronous Replica Exchange

We now discuss synchronization patterns for RE simulations from software design perspective. A **synchronization pattern** specifies constraints for the order of execution of the tasks in RE. The simplest form of constraint is a barrier, which must be reached by all currently executing tasks, before next set of tasks can be submitted for execution. First, it is essential to justify the importance of this discussion and to motivate the need for asynchronous RE (Figure 2.2 (b)) from scientific perspective.

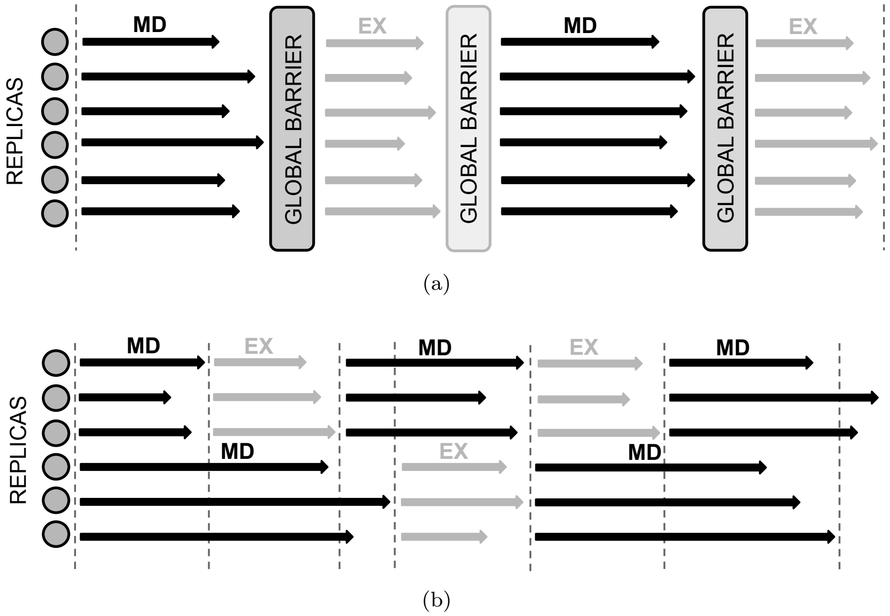
RE algorithms have historically been synchronous, viz., there is a global barrier between the simulation and exchange phases (Figure 2.2 (a)). Asynchronous RE (Figure 2.2 (b)) refers to the scenario when replicas can be in different phases. For example, a subset of replicas might be exchanging while some replicas might still be in simulation phase. In other words, the global synchronization of regular RE is relaxed. Asynchronous RE has the following scientific advantages:

Facilitates adaptive sampling. There are cases, where some replicas have already produced sufficient info and are no longer needed. For example, replicas simulating configuration space with very low probability may not need high accuracy hence only relatively small amount of sampling is required. Consequently these replicas should be terminated and their computational resource should be released. On the other hand, in the midst of simulations, new replicas may need to be created to cover the regions where more sampling is necessary. Obviously asynchronous algorithms are needed in such cases.

Enables integration of heterogeneous simulations. Nowadays multi-scale molecular simulations may consist of very different levels of theories hence different replicas may have significant differences in performance. For example, quantum mechanics calculations usually are slower than classical MD simulations. As a result, it is desired to have asynchronous RE algorithms to handle simulations with large mismatch in performance.

Handles fault-tolerance. Large-scale RE simulations, are more receptive to both hardware and software failures, which result in failures of individual replicas. Hence it is necessary to recover from such failures and continue simulation. Due to the nature of asynchronous algorithms, recovery time is significantly reduced compared to a synchronous RE, where in case of a failure all other replicas must wait at the barrier for a restarted replica.

Manages load-balance with fluctuation of available resources. Multi-dimensional RE simulations may require very large numbers of replicas, which could be larger than the available number of CPUs. In addition, both the number of running replicas and availability of a resource could change during simulation. Traditional synchronous algorithms are not capable to handle such cases. Asynchronous algorithms are needed to execute replicas at different time so that simulations of all replicas can be performed.



(a)

(b)

Figure 2.2: Schematic representation of RE Patterns: (a) Synchronous (b) Asynchronous. Over x-axis is shown wallclock time. Gray circles represent replicas, dark grey arrows MD phase propagation and light grey arrows exchange phase propagation. For both phases (MD and exchange), in synchronous pattern exists a global synchronization barrier. In this figure, for synchronous pattern, both MD and exchange are propagated concurrently but this is not a requirement for this pattern. In asynchronous pattern there is no barrier - MD and exchange can be propagated concurrently, meaning while some replicas run MD other replicas might be running exchange.

2.5 Multi-dimensional REMD

In Figure 2.3 is depicted a parameter exchange schema for a three-dimensional REMD. For illustration purposes we use artificial parameters A, B and C, which represent parameters in three respective dimensions. Although not shown in Figure 2.3 before an exchange of each parameter is performed an MD phase. In multi-dimensional REMD simulation parameters are exchanged sequentially - first is performed an exchange in first dimension, then in second and so on. In Figure 2.3 the total number of replicas is eight. While this simplifies the schema, it also limits the group sizes in all dimensions to two replicas. In each dimension, replicas are grouped based on the values of their parameters in other two dimensions. For example, replicas 0 and 4 are in the same group in dimension one (parameter A), since their parameters B and C are equal. It is important to note that group contents are changing dynamically - exchange

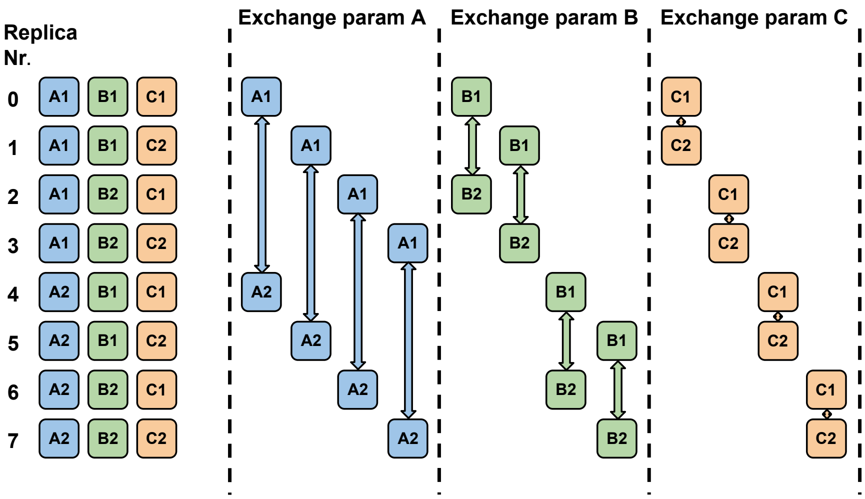


Figure 2.3: Schematic representation of the parameter exchange schema for the three-dimensional REMD. A, B and C are artificial parameters, which are exchanged between replicas belonging to the same group. For the exchange in any given dimension replicas are grouped based on their parameters in other two dimensions. To be grouped together in dimension 1, parameters of replicas in dimensions 2 and 3 must be equal. In this diagram, group sizes in all dimensions are equal to two. Generally, this is not a requirement - group sizes can exceed this number.

of replica parameter in any given dimension determines its group partners in other dimensions.

2.6 Molecular Simulation Software Packages with Integrated REMD Capability

2.6.1 Nanoscale Molecular Dynamics - NAMD

NAMD [1] is a parallel MD 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 analyzed 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. [22]

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 MD, 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

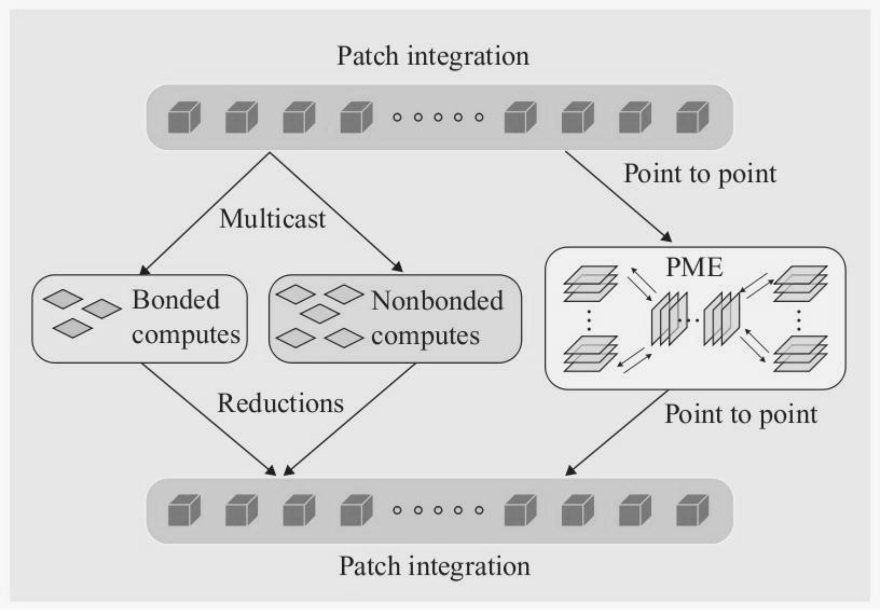


Figure 2.4: Schematic representation of NAMD parallelization mechanism [22]

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 [22]. 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, 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:

$$W = \frac{1}{t_c}\left(\frac{E}{1-E}\right)T_o \tag{2.44}$$

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

$$W = KT_o \tag{2.45}$$

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 utilize 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 [22]. 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 initialized 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 [22]. A schematic representation of this hybrid parallelization strategy utilizing *patches* and *compute objects* can be found in Figure 2.4.

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 utilize 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.

2.6.2 The Amber Molecular Dynamics Package

Computer MD simulations enable investigation of the dynamics and structure of proteins. Some examples are enzyme reaction mechanisms, ligand binding and protein refolding. Amber is one of the most popular MD simulations software packages. Amber is a collection of the various programs working together to setup, run and analyze MD simulations for proteins, carbohydrates and nucleic acids. In addition Amber also is a name of a set of classical molecular mechanics force fields, developed for the simulation of biomolecules. Interestingly, some other MD software packages have implemented the Amber force fields and Amber itself can be used with other force fields.

Amber is a result of a joint effort of the more than 40 researchers, who are working on advancing MD methodology. MD simulation part of Amber package consists of the four tightly coupled programs: `sander`, `sander.MPI`, `pmemd` and `pmemd.cuda`. Popularity of the Amber force fields, and their adoption by other MD software packages motivated the decision to separate Amber's setup and analysis tools into a separate package named AmberTools.

In Figure 2.5 is provided a schematic representation of the information flow in Amber. A typical simulation with Amber can be divided into three steps: system setup, MD simulation and trajectory analysis. Tools at the top layer of the Figure 2.5 are responsible for the system preparation. Middle layer of the figure represents MD simulation tools. At the bottom layer of the Figure 2.5 are depicted trajectory analysis tools. Naturally, collaborative nature of the development, resulted in separation of Amber into various programs. This approach enabled the usage of different programming languages and software engineering practices by developers of these programs. For example, LEaP is written in C using X-window libraries, MD simulation programs are written in Fortran 90, but mm-pbsa is written in Perl. Historically, system

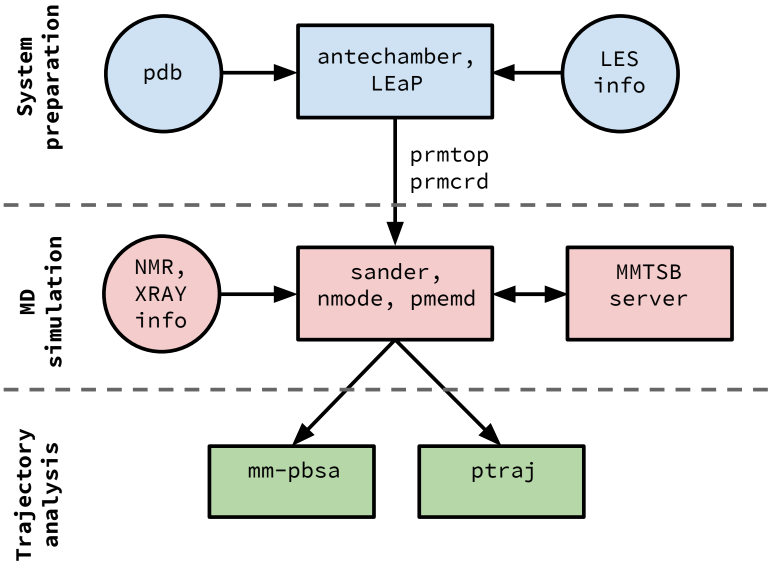


Figure 2.5: Information flow in the Amber program suite

preparation and analysis programs were considered less computationally intensive in comparison with simulation programs. As a result, most of the performance optimization effort went into MD simulation programs. In contrast with MD programs, which run on HPC clusters, system preparation and analysis programs often are executed on a users workstation. Amber's development approach enables integration of other MD software into Amber's software stack. For example, Amber tools and force fields can be used by NAMD. To achieve this, NAMD must parse and interpret information provided by .prmtop files. In addition, there is a number of software packages, which are aimed at improving Amber's preparation interface. Some of these extensions use portion of LeAP's code, others are entirely written from scratch.

As shown in Figure 2.5, in Amber there are four system preparation programs. Out of these four programs, major programs are LEaP and antechamber. The former is responsible for construction of biopolymers from the component residues, preparation of lists of force field terms and solvation of the system. The latter is responsible for creation of the force fields for residues. System preparation phase generates two files: parameter-topology files (.prmtop format) and coordinate file (.prmcrd format). As the name suggests, the latter contains coordinates of all atoms in the system. The former contains information required to compute forces and energies, such as: mass of the atom, atom name, lists of bonds, force field parameters, angles, and dihedrals. Despite being fairly basic, Graphical User Interface (GUI) of LEaP not only provides a visual representation of the PDB files, but also allows to perform interactive altering of the structure. In addition to GUI, LEaP also provides a text mode, which is the main usage mode of this program.

Central MD program in Amber is `sander`. This program is written in Fortran 90. `sander` reads input parameters as a label-value pairs. There are more than 150 different input parameters in `sander`, but in most simulations around 1/5 of these parameters is actually used.

A parallel version of sander is called `sander.MPI`. In `sander.MPI` to each process is assigned a certain number of atoms, while all atoms are organized in a single data structure. Coordinates of all atoms are available to all active processes. During the simulation, for each simulation step processes calculate potential energy and corresponding gradients. Force vector is calculated by gathering individually computed data by all processes, which results in each process obtaining the full force vector for its assigned atoms. Next, processes perform a MD simulation step and communicate updated positions of their assigned atoms to all the processes. This enables multiple approaches for collective calculation of force fields by active processes.

Frequent all-to-all communication hinders scalability of `sander.MPI`. In addition, system size has a significant effect on the size of the data structures used for the communication and on the amount of communicated data itself. Consequently, explicit solvent MD simulations with `sander.MPI` do not scale linearly when number of nodes is greater than 12. Due to a smaller number of force fields and coordinates, implicit solvent MD simulations demonstrate a much better scalability.

To address performance limitations of `sander` was developed `pmemd`. In this code were made optimizations for both parallel and serial performance. While `pmemd` supports only the most popular simulation types, it provides significant performance improvements in comparison with `sander`.

Amber's analysis program is called `ptraj`. There is a number of challenges associated with MD trajectory analysis. First, is management of trajectory files. Files are generated at different times and the number of files can be quite large. In addition, the size of the trajectory files can be very large. Next, multiple types of analysis may be required as the simulation is propagated. The `ptraj` is capable to process Amber and CHARMM trajectory file formats (.prmtop and .psf respectively). The `ptraj` also is capable of composing trajectories from partial trajectories and removing unnecessary parts of trajectories (solvent) if needed. The `ptraj` provides capability for users to add new analysis commands. The `ptraj` is a cluster of programs, which perform analysis such as energy estimation, entropies estimation, etc.

2.7 REMD Software Frameworks

In this section, we focus on frameworks designed to implement RE algorithm outside of the MD engine; we defer a discussion of REMD simulations using molecular simulation software packages with integrated RE capability till later. We start with CHARMM based implementation for 2D REMD. Then we introduce Multiple Copy Algorithm (MCA) implementation with NAMD engine and finally we discuss implementations of asynchronous RE.

2.7.1 CHARMM

Ref. [23] presents an implementation of a 2D US/H-REMD method, implemented in REPDSTR module of the CHARMM [24]. REPDSTR uses an MPI level parallel/parallel mode where to each replica are assigned multiple MPI processes and dedicated I/O routines. Exchanges of parameters between neighboring replicas in first dimension are implemented using odd-even rule and in second dimension using even-odd rule. To improve sampling efficiency exchange attempts are performed alternatively along the two dimensions.

Implementation was tested on IBM Blue Gene/P supercomputer using the binding of calcium ions to the small protein Calbindin D_9k. Obtained results show that 2D US/H-REMD significantly improves the configurational sampling for biological potential of mean force (PMF) calculations and as a result facilitates convergence of the simulation.

Authors presented strong scaling performance of 2D US/H-REMD, involving 4096 replicas and utilizing up to 131072 CPUs with nearly linear scaling.

2.7.2 MCA implementation with NAMD

A Charm++ based implementation designed to run MCA was presented in [25]. It is tightly bound to the NAMD simulation engine. Charm++ is used to run concurrently multiple NAMD instances, which are exchanging messages via a point-to-point communication functions of Tcl scripting interface. Tcl scripting enables users to implement REMD algorithms without modifying the source code.

Authors demonstrated strong scaling behavior of the swarms-of-trajectories string method implementation using the full-length c-Src kinase system utilizing up to 524288 cores on Blue Gene/Q supercomputer. Results of temperature exchange REMD simulations with peptide acetyl-$(AAQAA)_3$-amide [26] in TIP3 solvent on Blue Gene/Q (utilizing up to 32768 cores) were presented. Implementation of two-dimensional Hamiltonian RE with Umbrella Sampling (US/H-REMD) on irregular-shaped distribution of umbrella windows was discussed.

2.7.3 ASyncRE package

Ref. [27, 28] presented ASyncRE package, developed to perform large-scale asynchronous REMD simulations on HPC systems.

ASyncRE has an emphasis on asynchronous RE. Package supports Amber [7] and IMPACT [29] MD engines. It implements two REMD algorithms, namely multi-dimensional RE umbrella sampling with Amber and BEDAM λ RE alchemical binding free energy calculations with the IMPACT. AsyncRE uses a similar runtime system as RepEx, is capable of launching more replicas than there are CPU cores allocated and is fault tolerant: failure of a single (or multiple) replicas does not result in failure of a whole simulation. Upon the user request failed replicas can be ignored or relaunched.

In ASyncRE duration of the simulation phase is defined as a real time interval. This allows to perform synchronous RE simulations with ASyncRE, by specifying a sufficiently large duration for the MD simulation phase and allocating enough CPU cores to run all replicas concurrently. The major drawback of running synchronous RE simulations with ASyncRE are substantially increased simulation's Total Time to Completion (TTC) and significant under utilization of allocated CPU cores on a remote system.

2.7.4 Asynchronous Replica Exchange Package for Volunteer Computing Grids and HPC clusters

Ref. [30] introduced another REMD package targeted at asynchronous RE, optimized for volunteer computing resources. The package can be used on HPC clusters as well. It is customized for IMPACT and supports both 1D and 2D REMD simulations.

Distinctive features are fault tolerance, the ability to use a dynamic pool of resources and to use fewer CPU cores than replicas. Input files are generated on a user's workstation and exchange phase is performed on coordination server, meaning that output data must be moved from target resource to coordination server. Implementation specifics enforce the length of the MD simulation phase to be greater than 1ps. Implementation was evaluated using statistical inefficiency analysis [31] and divergence of the binding energy distribution from a target distribution, using the Kullback-Leibler divergence [32] for both 1D REMD and 2D REMD BEDAM [33] simulations using b-cyclodextrin-heptanoate model and using up to 240 replicas.

2.7.5 Summary

As seen, there are multiple existing packages designed to perform REMD simulations. A significant number of them however, support a single MD engine, and their design makes it difficult to substitute simulation engines. The tight binding of RE methods to a particular engine raises a barrier for the development of new RE algorithms.

Historically, tight integration has prevailed due to the perception that performance trumps all other features. There are emerging examples of important biomolecular problems however, that involve multi-state equilibria, and for which the interpretation of experiments requires scanning control variables such as temperature, ionic conditions, and pH in addition to geometrical or Hamiltonian order parameters[34]. These applications have the added challenge that sampling along the space of the order parameters needs to be statistically converged at all points. Here, the REMD method offers the added advantage that equilibrium between simulations is enhanced through the exchanges. An illustrative example is the "problem space" associated with biocatalysis whereby conformational equilibria, metal ion binding and protonation events lead to an active state that can catalyze the chemical steps of the reaction [35]. Thus, these applications require not only the elucidation of the free energy landscape of the chemical reaction itself [36, 37], but also the characterization of the probability of finding the system in the catalytically active state as a function of system variables [38].

To address these novel applications and scenarios, a flexible and efficient multi-dimensional REMD framework is required, that can be used for both system control variables and generalized coordinates. Currently, there is no REMD framework capable of providing the necessary flexibility in composing the range of RE methods with MD engines as needed, while providing adequate performance.

2.8 Pilot Systems

Traditionally many HPC systems are designed to support workloads consisting of a single parallel executable. At the same time, there are numerous use-cases in various scientific domains, which can take advantage of the capability to run workloads of multiple heterogeneous tasks. Often these tasks have complex dependencies, vary in computational requirements and are created on demand. To execute these workloads, is required an API, which enables fine-grained management of tasks and control over allocated computational resources. Currently HPC systems are lacking such interface. Pilot systems enable execution of complex, multi-task workloads by decoupling of workload specification, resource selection, and task execution via job placeholders. Pilot systems provide an interface to HPC systems, which enables a flexible usage of the available computational resources, while complying to usage policies of these systems. To access control over computational resources, pilot system submits job placeholders (i.e., pilots) to the schedulers of the HPC resources. First, pilots are waiting in schedulers queue until there are enough CPUs available. Then, pilots perform setup procedure (i.e. bootstrap) and are ready to execute tasks. Tasks are executed within the time frame requested by the user.

There are many scientific workloads consisting of large ensembles of tasks (both single and multi-core) which have computational requirements exceeding the size of the allocatable resources. Execution of such workloads on HPC clusters, using conventional techniques is cumbersome or even impossible.

From a user perspective, pilot systems provide two clear advantages. First, pilot systems allow to reduce the total time to completion (TTC) for ensemble of jobs. Two major components of the TTC are queue waiting time and total execution time of individual jobs. Second, pilot systems provide means to define complex workloads at application level and manage their execution on HPC systems.

Historically most of the pilot systems were developed for a specific user communities, resources or workloads. The first pilot system was developed as early as 1996, when AppLeS [39] system introduced features such as application-level scheduling and a concept of a resource placeholder. Later, more pilot systems, targeting various communities and resources emerged. Some of the well known pilot systems are: DIANE [40], Falkon [41], DIRAC [42] and BigJob [43]. Two notable examples of pilot systems are HTCondor [44] and Glidein [45]. These systems introduced concurrent execution of workloads on multiple resources and currently are amongst the most widely used pilot systems. Another notable example is ATLAS PanDA [46], which is used to run millions of jobs a week on a Worldwide LHC Computing Grid (WLCG). An example of a workload specific pilot system is CRAM [47]. CRAM is designed to execute ensembles of MPI tasks on HPC clusters and was developed specifically for the Sequoia supercomputer at Lawrence Livermore National Laboratory (LLNL). Another examples is Pegasus-MPI-Cluster (PMC) [48], which is an MPI-based Master/Worker framework that can be used in combination with Pegasus workflow management system [49]. It runs large-scale workflows of small tasks (limited to a single node) on HPC resources. An example of a pilot system for HPC resources, which provides an API is Falcon [41]. Application developers can use Falcon's API to develop distributed applications for execution of their workloads. One of the Falcon's limitations is focus on a single-core tasks.

Features of pilot systems make them a good workload execution mechanism for applications designed to perform REMD simulations. In particular, decoupling of the workload specification and task execution. This enables application developers to significantly increase fault tolerance of the respective applications.

Despite the fact that the notion of the pilot system is fairly established, the industry standard model for pilot systems doesn't exist. Designs of the pilot systems often have significant differences. This can be explained by the fact that these systems often targeting specific resources, workload and user communities. As the result, portability and extensibility of these systems is limited. Additionally, many pilot systems introduce new definitions and concepts, making it almost impossible to compare them in a meaningful way. 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 [50]. The main goal of P* Model is to provide a convenient means to qualitatively compare and analyze various pilot systems.

2.9 RADICAL-Pilot

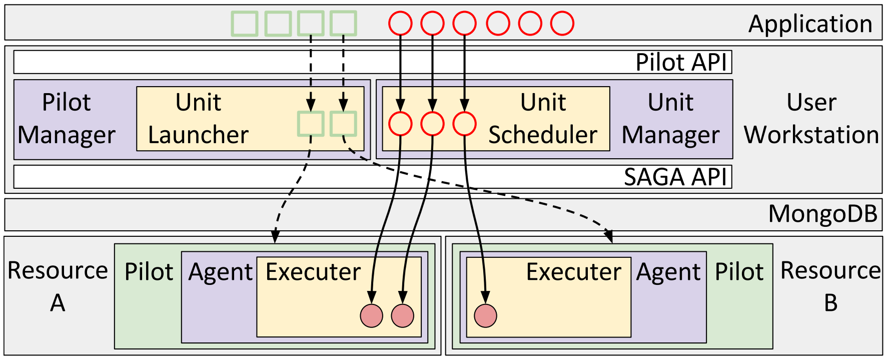


Figure 2.6: RP overview. An application uses the Pilot API to describe pilots (green squares) and units (red circles). The PilotManager instantiates (dash arrow) pilots, the Unit-Manager instantiates (solid arrow) units. Both managers are executed on the user workstation. Pilots are launched (dashed arrows) on resources A and B via SAGA API, an Agent is bootstrapped for each pilot, units are scheduled (solid arrow) to the Agents via MongoDB and executed by the Agents Executer. Boxes color coding: gray for entities external to RP, white for APIs, purple for RPs modules, green for pilots, yellow for modules components.

RADICAL-Pilot (RP) is a portable, modular and extensible pilot system written in Python programming language. RP is capable of spawning more than 100 tasks/second and the steady-state execution of up to 8,000 concurrent tasks. RP can be used stand-alone, as well as integrated with other application-level tools as a runtime system. A distinct feature of RP is its API - ”Pilot API” [51], which can be used by application developers to orchestrate execution of their workloads on HPC systems. Pilot API is designed to enable users to describe pilots and workloads (comprised of multiple tasks), which are then passed to the RPs runtime system. RP utilizes a Simple API for Grid and Distributed Applications (RADICAL-SAGA [43], [52]) as an interface to the HPC systems. Unlike some other pilot systems, RP does not provide workload management capabilities. RP supports execution of ensembles consisting of both MPI (including multi-node tasks) and non-MPI tasks on various HPC resources.

In RP, pilots are represented as collections of computational resources, which are independent from the architecture of the target resource. Workloads are represented as a collections of units, which are executed by a pilot. Both pilots and units have a state model associated with them. States of the pilots are managed by the PilotManager, but states of the units by Unit Manager and Agent (Figure 2.6). To launch pilots on HPC resources, PilotManager is using SAGA API. SAGA is providing adapters for various scheduling systems and types of the resources, which enable management of jobs and files. As a communication channel between UnitManager and Agents RP uses a MongoDB database. UnitManager is responsible for scheduling units for execution on available pilot instances. First, pilot is submitted to the queue of a resource scheduler. When pilot becomes active, Agent gets bootstrapped on allocated CPU cores. Then Agent starts pulling units from the MongoDB database and executes them on CPU cores allocated by the pilot.

RP is a distributed system, with PilotManager and UnitManager running on a user workstation and Agent running on a remote resource. Launcher component of the PilotManager specifies the number, properties and placement of the Agent components (Stager, Scheduler, and Executer) on a Pilot. To achieve this, Launcher is using resource configuration files, which are defined for various resource types. Placement of the Agent components depends on the resource type. On HPC clusters, Agent components can be placed on the cluster's MOM nodes, login nodes or compute nodes. Agent components use ZeroMQ for communication, which facilitates transitions of the units. After Agent becomes active, its Scheduler queries the resource manager for the number of cores allocated by the pilot and their topology on a cluster. Scheduler is capable to get resource info from the following resource managers: Sun Grid Engine (SGE), PBS Professional, Simple Linux Utility for Resource Management (SLURM), TORQUE, Cray CCM, LSF and LoadLeveler. Currently RP supports two scheduling algorithms: "Torus" for CPUs comprising an n-dimensional torus (such as on IBM BG/Q) and "Continuous" for CPUs connected continuously. Once Scheduler obtains resource information, it passes the units to Executer components, which launch units using launching method, specified in the resource configuration file. RP supports the following launching methods: IBRUN, ORTE, APRUN, MPIRUN, MPIEXEC, CCMRUN, RUNJOB, DPLACE, RSH, SSH, POE, and FORK. Executer uses one of the two spawning mechanisms for unit execution: Popen (Python based) and Shell (based on /bin/sh). Executer monitors unit execution, collects exit codes and reports to Scheduler once units finish executing.

Chapter 3
RepEx framework

In this chapter we present our framework for REMD simulations. We start this chapter by specifying a set of requirements we have identified in collaboration with potential users. We broadly categorize requirements into three categories: functional, performance (scalability) and usability. Next, we present design of RepEx framework, which is built around the following concepts: RE pattern, Pilot system and flexible execution mode. In this section we discuss these three concepts in detail. Next, we introduce an asynchronous RE algorithm for multi-dimensional REMD simulations. While this algorithm is designed for REMD simulations with multiple dimensions, it can be used for one-dimensional simulations as well. In this chapter we also describe implementation of RepEx framework. We introduce and describe three types of modules used to our implementation: Execution Management Modules (EMM), Application Management Modules (AMM) and Remote Application Modules (RAM). To aid the description of our implementation, we provide a task execution diagram and two UML diagrams, namely class diagram and control flow diagram. We close this chapter by scientific validation of RepEx implementation, where we perform a number of experiments and use free energy profiles for comparison with published results.

3.1 Requirements

In this section, we motivate and define requirements of a REMD simulations software. We describe three types of requirements: functional, performance (scalability) and usability. We have identified the following functional requirements:

Generality is a requirement to maximize a range of replica-exchange methodologies as well as MD engines. A general purpose framework should support: (i) different types of exchange parameters, (ii) multiple exchange parameters in a single REMD simulation, and (iii) multiple MD engines. A corollary of this requirement is the decoupling of advances in replica-exchange methodology to MD engines, and thus the potential for broader (greater number of REMD applications) and deeper impact (enable new research opportunities).

Execution flexibility arises from the need to decouple the number of CPU cores from the number of replicas. Alternatively, the ability to set-up a REMD simulation with a desired number of replicas, independent of the number of CPU cores available at a given instance of time. For reasons ranging from a queue waiting time to limitations in the number of allocatable CPU cores on a given cluster, it should be possible to use as many or as few CPU cores as needed, irrespective of the number of replicas. The same principle also applies to individual replicas: support for both single-core and multi-core replicas should be provided. Currently, **all** REMD frameworks require at least as many CPU cores as replicas; furthermore, the number of replicas that are actively simulated is fixed and statically determined.

Synchronization. To enable a wider range of REMD simulations support for asynchronous RE is required without loss of generality or execution flexibility.

Interoperability. Most REMD simulations are executed on supercomputers which vary in scheduling systems, middleware and software environment. To support community production grade science, a REMD framework should work on multiple high-end machines as well as small HPC clusters, while retaining functionality and performance.

The above functional requirements have to be balanced with the following performance requirements:

Scalability with the number of replicas. To obtain high sampling quality, REMD simulations should support the ability to run a large number of replicas. Furthermore, given that the number of replicas needed for a REMD simulation scales as $\approx N^d$, where d is the dimensionality (of exchange), the need to support a large number of replicas is greater when applied to multi-dimensional simulations. Achieving good scalability for REMD frameworks is a challenging task, especially when preserving the four functional requirements outlined above.

Scalability with the number of CPU cores. Whereas the primary performance metric is the scalable execution of a large number of replicas, it is often the case that each replica is multi-node (in Ref. [53], each replica was 768 cores); multi-node replicas are important to simulate large physical systems. Any framework should provide scalability in the number of replicas simulated and the number of CPU cores utilized.

Usability. Relative to the simulation phase, the exchange phase is significantly more complex. Thus, not only should a framework for REMD separate the logic of the exchange mechanisms from the simulation mechanisms, it should not expose the complexity of exchange mechanism, should be automated as much as possible and must be fully specified by configuration files. Definition of configuration files should be intuitive and should include a minimal set of parameters.

3.2 Design

To satisfy the requirements outlined in the previous sub-section, we discuss the three concepts underpinning the unique design of RepEx:

- **RE Pattern:** explicit support for synchronization patterns between simulation and exchange phases.
- **Pilot system:** a multi-stage mechanism for workload execution via the use of an initial placeholder job (the "pilot") and thus dynamically allocating computational resources for replicas.
- **Flexible Execution Mode:** The ability to execute different patterns and number of replicas independent of the underlying resources available, i.e., flexible spatial and temporal mapping of workload (tasks) to the allocated CPUs.

3.2.1 Replica Exchange patterns

The RepEx framework captures the distinction between different synchronization scenarios using two RE patterns and exposes them to end-users, independent of MD simulation engine and the resources available.

Synchronous RE Pattern

Synchronous RE pattern depicted in Figure 2.2 (a), corresponds to the scenario where all replicas must finish simulation phase, before any of the replicas can transition to the exchange phase. There is a global synchronization barrier, which forces the replicas arriving at the barrier early to wait for the lagging replicas. Once all replicas are done in the simulation phase all of them transition to the exchange phase. This cycle is then repeated. The synchronous pattern is the conventional way of running REMD simulations, partly because of the implementation simplicity.

Asynchronous RE Pattern

Asynchronous RE Pattern, shown in Figure 2.2 (b), does not have a global synchronization barrier between simulation and exchange phase. While some replicas are in the simulation phase, others might be in the exchange phase. Based on some criterion, a subset of replicas transition into the exchange phase, while other replicas continue in the simulation phase. Selection of replicas that will transition may be based on a FIFO principle, e.g. first N replicas transition into an exchange phase. Alternatively, only replicas which have finished a predefined number of simulation time-steps (2 ps) during some real time interval (1 minute) transition into exchange phase.

3.2.2 Pilot system

We have discussed the pilot system concept in section 2.8 and RADICAL-Pilot, as an example of a pilot system, in section 2.9. Pilot systems have been generalized [54] to provide a variety of capabilities, but the two most important are: management of dynamically varying resources and execution of dynamic workloads. A Pilot system supports the execution of workloads with multiple, heterogeneous and dependent tasks[11].

To execute its workloads, RepEx relies on RADICAL-Pilot. Usage of RPs API, enables separation of computational requirements from the replica count in REMD simulation. Integration of a pilot system in our design facilitates modularity of the framework and reusability of RE patterns for various MD engines.

3.2.3 Flexible Execution Modes

Decoupling the workload size from the available resources requires: (i) the details of workload be kept separate from the details of resources — type, quantity and availability, and (ii) the ability to execute a workload of a given size (say N replicas) independent of the specific resources available. As alluded to, Pilot-job systems enable the former; we now discuss how the pilot abstraction allows the latter.

Depending upon the relative size of the resources available (R) to the size of simulations (S = number of replicas x resource requirement of each replica), REMD simulations are executed differently. Thus, there are two Execution Modes: when $R > S$ (Execution Mode I), and when $R < S$ (Execution Mode II), each of which can be used with any of the two RE patterns.

Execution Mode I

In Execution Mode I the number of allocated CPU cores satisfies execution requirements of all replicas at a given instant of time. For example, if each replica requires a single CPU core to run, in this mode enough cores are allocated to run all replicas concurrently. Figure 3.1 illustrates capturing Execution Mode I in the context of a Synchronous RE pattern.

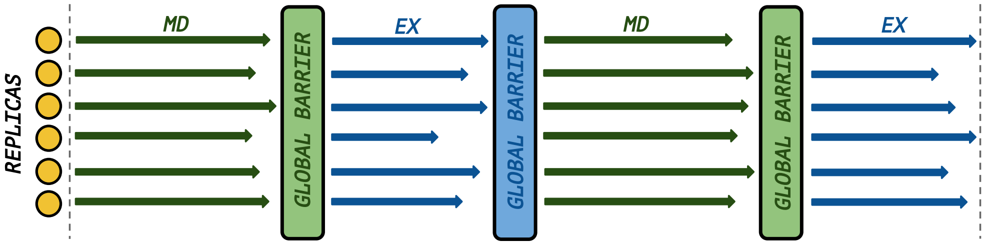


Figure 3.1: Schematic representation of Execution Mode I. On the x-axis is time. Gray squares represent replicas, blue arrows MD phase propagation and green arrows exchange phase propagation. Both MD and exchange phase for all replicas are performed concurrently. After MD and exchange phase is placed a global barrier, ensuring that all replicas enter next phase simultaneously.

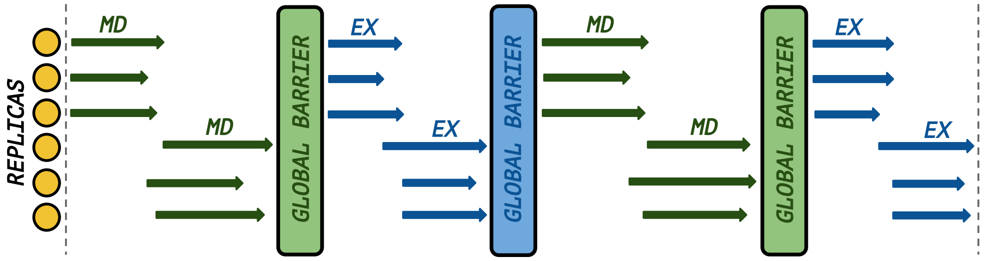


Figure 3.2: Schematic representation of Execution Mode II. On the x-axis is time. Gray squares represent replicas, blue arrows MD phase propagation and green arrows exchange phase propagation. Replicas don't propagate MD and exchange phase concurrently. Batch size for each phase is determined by the number of CPU cores allocated. A global synchronization barrier is present after both MD and exchange phase, ensuring that all replicas enter next phase simultaneously.

Execution Mode II

Execution Mode II supports the scenario when there are not enough CPU cores to run all replicas concurrently. The ratio of cores to replicas is a user defined variable, but typically is a term of a geometric series, e.g. $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$. As a result, only a fraction of replicas can propagate simulation or exchange phase concurrently. A schematic representation of Execution Mode II is illustrated in Figure 3.2; for simplicity, we depict the synchronous RE Pattern, but Execution Mode II can be used with any of the two available RE Patterns.

While users are given the option to select an execution mode, exact execution details are determined by execution management module of RepEx. The Execution Mode abstraction hides the gory details of the execution, which differ based upon the relative values of R and S. The implementation of Execution Modes vary in the spatial and temporal mapping of workload (tasks) to the allocated CPUs. Specifically, they differ in:

- Order and level of concurrency for task execution
- Number of Pilots used for a given simulation
- Number of concurrently used target HPC resources

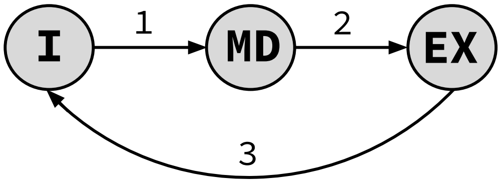


Figure 3.3: Finite state machine for asynchronous RE algorithm. Each replica can be in one of three states: 'I' - idle; 'MD' - MD simulation; 'EX' - exchange. From idle state replica can only transition to an MD simulation state (state transition 1). From MD simulation state, the only transition is to exchange state (state transition 2), but from exchange state replica can transition only to idle state (state transition 3).

Execution Mode is a subset of execution options decoupling simulation requirements from the resource availability and enabling flexible usage of allocated HPC resources. A user should be able to switch between available Execution Modes without any refactoring. In addition to providing conceptual simplicity by hiding details of the execution, Execution Modes provides a significant practical functionality: it permits the study of systems not otherwise possible thanks to the ability to launch more replicas then there are allocatable CPU cores on a target HPC cluster. This might be particularly useful when a user has access to small HPC clusters, but is interested in running REMD simulations involving a large number of replicas. For example, a user can assign as many cores to each replica as needed, or if only a small HPC cluster comprising 128 cores is available user still can perform a simulation involving 10000 replicas.

3.2.4 Asynchronous RE algorithm

In this section we present an algorithm for asynchronous REMD simulations. Algorithm can be used for both one-dimensional and multi-dimensional simulations.

We first develop a finite state machine for replica, which is shown in Figure 3.3. Each replica can be in one of three states: 'I' - idle; 'MD' - MD simulation; 'EX' - exchange. From idle state replica can only transition to an MD simulation state (state transition 1). From MD simulation state, the only transition is to exchange state (state transition 2), but from exchange state replica can transition only to idle state (state transition 3). Individual replicas can be in any of the three states at a given moment: replicas 1..4 can be in state 'MD', replicas 5 and 6 can be in state 'EX' and replicas 7 and 8 can be in state 'I'. In addition, individual replicas, can be in different dimensions at a given moment (this applies to both MD and exchange phases).

Asynchronous RE algorithm is presented in algorithm 3.2.4. Prior to entering main simulation loop we set the state of all replicas to 'I' (lines 1-3). At line 4, is entered the main simulation loop, which terminates only when loop runtime (elapsed_time) exceeds, simulation_time specified by the user.

In the main simulation loop, first is performed a check, for the size of the ex_repicas array. If there are replicas in this array, these replicas enter exchange phase (lines 5-19). ex_replicas array is populated, when the number of replicas which have finished MD phase exceeds or is equal to the wait_size. Variable wait_size is specified by the user as the ratio of replicas, currently propagating an MD phase (typically, this ratio is $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{32}$, etc.). This ratio is then converted to the number of replicas in a given simulation, which is then assigned to the wait_size variable. If ex_repicas array is not empty, replicas are grouped together into arrays, by their current dimension, since replicas in ex_repicas array, can be in different dimensions. Grouped replicas are then appended to ex_dims array (2D array).

For each array in ex_dims we perform the following (lines 7-15). We prepare an exchange task(s) for replicas in array and execute this task(s). We wait for this task(s) to finish, since it does not make sense to enter MD phase for current set of replicas before that. When exchange task(s) is done, Fro each replicas in array, we update state to 'I' and update current dimension. Finally for each replica in ex_replicas we update their group index. This is required, since after an exchange of the parameters, group index of replicas in other dimensions has changed.

Next, we iterate through all the replicas and if replicas is in state 'I' it enter an MD phase (lines 20-26). For each replicas in state 'I' we prepare an MD task, execute that task and update the state of the replica to 'MD'. At this point we don's wait for replicas to finish an MD task, execution tall is non-blocking.

At line 27 we initialize an array md_done. next we enter a while loop, which terminates only when the size of md_done is equal to or greater than the wait_size (lines 28-35). In this while loop, first each replica, which finished an MD task is appended to the md_done array. Then replicas in md_done array are sorted by their group index and groups which have only a replica are removed from the array. This is done to ensure that each replica has an opportunity for an exchange. Next in md_done array, groups having an odd number of replicas are reduced in size by one. This step is necessary to ensure, that remaining members of the group will have exchange partners at the next exchange attempt.

At line 36 we assign md_done to ex_replicas array. At this point number of replicas, which has finished MD phase is equal to or greater than the wait_size and these replicas can enter an exchange phase. For each replica in ex_replicas we change state to 'EX' and as a final step in main simulation loop we update the simulation_time variable.

Algorithm 1 Asynchronous algorithm for multi-dimensional RE.

```
for all replicas do
    replica.state = 'I'
end for
while (elapsed_time < simulation_time ) do
    if ex_replicas is not empty then
        group replicas in ex_replicas by dimension and append resulting arrays to ex_dims
        for each array in ex_dims do
            prepare EX task
            execute EX task
            wait for EX task to finish
            for replica in array do
                replica.state = 'I'
                update current dimension for replica
            end for
        end for
        for each replica in ex_replicas do
            update group index
        end for
    end if
    for all replicas do
        if replica.state == 'I' then
            prepare MD task
            execute MD task
            replica.state = 'MD'
        end if
    end for
    init. array md_done
    while (size of md_done < wait_size ) do
        for each finished MD task do
            add replica to md_done
        end for
        sort replicas in md_done by group index a given dimension
        remove single replica groups from md_done
        reduce size of groups with odd number of replicas in md_done by one
    end while
    ex_replicas = md_done
    for each replica in ex_replicas do
        replica.state = 'EX'
    end for
    update elapsed_time
end while
```

3.3 Implementation

In this section we describe the implementation of our REMD framework. Due to a temporary creative stupor we named out framework RepEx, which is an acronym for RE. We first present different types of modules available in RepEx. Then we discuss how RepEx is interacting with RADICAL-Pilot to execute its tasks. Next we present a hierarchy of classes and finally we walk through a flow of control in a typical RepEx simulation.

3.3.1 RepEx Modules

At the core of RepEx are three module types: Execution Management Modules (EMM), Application Management Modules (AMM) and Remote Application Module (RAM).

Execution Management Modules (EMM)

EMMs enable a separation of execution details, namely resource management and workload configuration, from the simulation. Majority of the resource management calls (RP API) are performed in EMM. EMM is responsible for launching a pilot(s) on a target resource, staging-in files at the beginning of the simulation, submission of MD and exchange tasks to a target resource and for the management of the workload dependencies. Due to EMM's ability to encapsulate synchronization routines, each of the two RE Patterns is fully specified by a single EMM. Additionally, most of the profiling calls are inserted in EMMs. At the time of this writing in RepEx are available four EMMs:

- **ExecutionManagementModule**. This module is a parent class for all other EMMs. It define a set of attributes common to all EMMs and implements a single function `launch_pilot()`.
- **ExecutionManagementModulePatternS**. This module specifies a synchronous RE pattern. This module is used to run simulations with up to three dimensions with any ordering of these dimensions. Module implements a single function `run_simulation()`.
- **ExecutionManagementModulePatternA**. As the name suggests this module specifies an asynchronous RE pattern. Any number of dimensions is supported by the pattern as well as an arbitrary ordering of exchange types. Module implements a single function `run_simulation()`.
- **ExecutionManagementModulePatternSgroup**. This module is designed to pack replicas belonging to the same group into a single task. Only a synchronous RE pattern

is supported by this module.

EMMs are running on the client-side of the RepEx.

Application Management Modules (AMM)

AMM is the most complex module of the RepEx. For each supported MD engine is defined a single AMM. AMMs explicitly support *generality* requirement by performing the following operations:

- translation of the user provided parameters to simulation setup
- instantiation of the replicas (replica objects) according to the provided parameters
- management of the simulation input/output files and file movement patterns
- preparation of the Compute Units (tasks) for both simulation phases (MD and exchange)
- exchange of the parameters between replicas
- support of the different exchange types and dimension counts
- support simulation restart from the previous checkpoint in case of failure

Similarly as EMMs, AMMs are running on the client-side of the RepEx.

Remote Application Modules (RAM)

As the name suggests modules of this type are running on the server-side (remote system). RAMS are responsible for the following operations:

- generation of individual input files for MD simulation phase
- generation of restraint (.RST) files for MD simulation phase (umbrella exchange)
- calculation of single-point energies (salt concentration exchange)
- reading energy and temperature values from the simulation output files
- performing exchange phase calculations

Due to the differences in formats of input/output files, most of the RAMs are MD engine specific. At the same time most of the exchange calculations are MD engine independent, resulting in reusability of the RAMs. For example, `GlobalExCaculator` module is exactly the same for all MD engines and simulation types, irrespective of the dimension count.

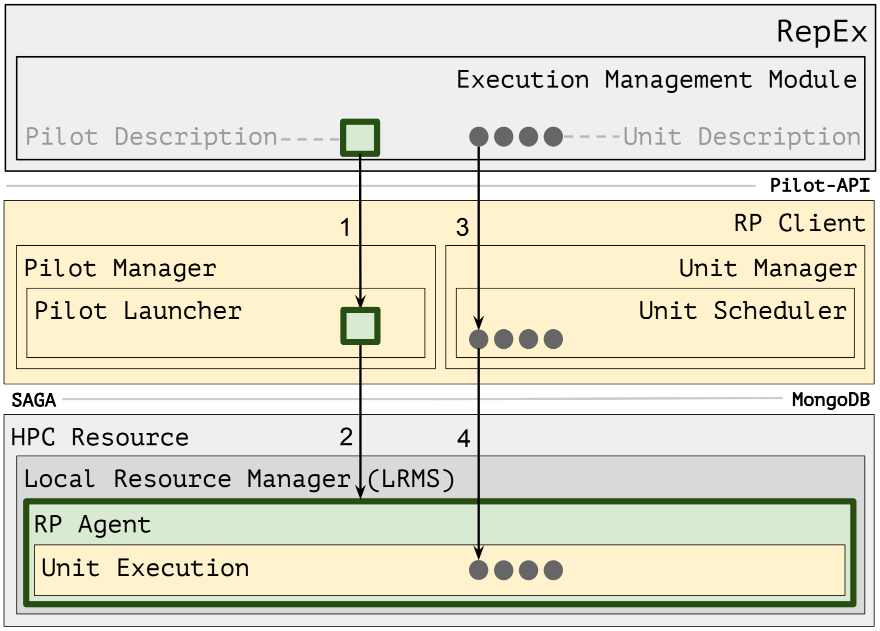


Figure 3.4: RepEx task execution diagram. Top gray square represents RepEx package, middle (yellow) square represents RP API and bottom square represents a target resource. The numbered arrows represent control flow. First in Execution Management Module is created a Pilot description, which is then submitted to RPs Pilot Manager (step 1). Pilot Manager launches a Pilot instance on a target resource (step 2). After a Pilot is active, RepEx workload can be submitted for execution. From Execution Management Module to RPs Unit Manager are submitted Compute Units (step 3), which are defined using a Compute Unit Description. Then Unit Manager submits Compute Units for execution to RPs Agent (step 4).

3.3.2 Task Execution Diagram

In Figure 3.4 is presented a diagram showing how RepEx interacts with RADICAL-Pilot to execute its workloads. The gray square on the top of the diagram is RepEx package. Middle yellow square represents RPs API and bottom square an HPC resource. In EMM of RepEx, according to the user provided resource configuration file is created a `Compute Pilot Description`. Then to launch a pilot is performed a call to RPs API (`rp.PilotManager.submit_pilots()`). This call submits a `Compute Pilot Description` to Pilot Manager (step 1), which in turn launches a pilot (RP Agent) on a target resource (step 2). Once RP Agent becomes active on a resource, it is ready to execute Compute Units (tasks) on allocated CPU cores. Next, from EMM of RepEx for execution are submitted Compute Unit Descriptions (generated according to the user defined simulation input file) to RPs Unit Manager (step 3). These Compute Unit Descriptions can be of any type. Finally, Unit Manager executes these Compute Units on a target resource.

3.3.3 UML Class Diagram

A simplified UML class diagram is presented in Figure 3.3.3. Execution Management Modules are packaged together in EMMs package (Green square). Application Management Modules can be found in AMMs package (blue square) and Remote Application Modules are organized into two packages - NAMD RAMs and AMBER RAMs (red squares). The only class which does not belong to any package is `Replica` class. These classes are instantiated by AMMs on a client-side of RepEx and are used to record parameters and simulation files associated with a given replica. Any type of REMD simulation requires at least two `Replicas` objects. As mentioned above there are four EMMs in RepEx. The parent class `ExecutionManagementModule`, which implements `launch_pilot()` method and child classes which implement `run_simulation()` method. Child classes must also implement one of the two RE patterns. Between EMMs and AMMs is a uni-directional association: EMMs know about AMMs, but AMMs are not aware of the EMMs. Multiplicity value for this association is 1, since there is a one-to-one mapping between EMM and AMM. Typically only a single instance of EMM and AMM is created during the simulation. A uni-directional association is also between AMMs and RAMs: RAMs are completely self-contained and are not aware of any other classes. All RAMs are instantiated by AMMs. Multiplicity values for association between AMMs and RAMs depend on the class type: RAMs responsible for generation of input files for replica or performing a part of exchange calculations on a per replica basis have multiplicity value of two or more (we must have at least two replicas). Other RAMs, which mainly are responsible for "global" exchange calculations have a multiplicity value of one or more (we have at least one simulation cycle).

3.3.4 UML Control Flow Diagram

We now discuss a control flow in a typical REMD simulation. In Figure 3.3.4 is depicted a simplified UML control flow diagram for one cycle of the T-REMD simulation with AMBER MD engine. Resource configuration file and simulation input file are passed to a command line tool `repex-amber`. Next in `repex-amber` is created an object of the `ApplicationManagementModuleAmber` class and an object of the `ExecutionManagementModuleS` class. The latter corresponds to a synchronous RE pattern. Next, using an instance of the AMM class, is called a method `initialize_replicas()`, which returns to `repex-amber` a list of replica objects. After that, to launch a pilot on a target resource, is called `launch_pilot()` method of the `EMM`. Once a pilot becomes active on a target resource, is called `rum_simulation()` method of the `EMM`. To this method is passed as an argument a list of replica objects and an instance of the AMM class. Within run_simulation() method, using an instance of the AMM class is first called a prepare_shared_data(). This method populates two lists: shared_urls and shared_files. The former contains URLs of the simulation input files and the latter the names of these files. These two lists are used to create RPs data directives by both EMM and AMM.

Next is called AMMs function get_all_groups(). This function returns a 2D list of replicas grouped according to their group index in a given dimension. Grouping of replicas is important, since part of the exchange phase is performed as a post processing step of the MD phase. We might not have enough CPU cores to launch all tasks concurrently, which can result in a "self-locking" of the simulation. For an exchange phase, replicas depend on the output files of the MD phase generated by the replicas belonging to the same group. If we don't have enough CPU cores to launch all tasks of the MD phase concurrently, we only launch concurrently a subset of groups. Once replicas are grouped, we can prepare tasks for an MD phase.

Now we are ready to prepare Compute Units for the MD phase. Using an instance of the AMM, from the EMM for each replica is called prepare_replica_for_md() method, which returns a Compute Unit Description. Once all Compute Unit Descriptions are generated, they are submitted to RPs Unit Manager via submit_units() method. Immediately after submit_units() in EMM is placed a blocking wait_units() call, which returns only after all Compute Units are done.

In this example, execution of each Compute Unit for MD phase has three stages: pre-execution, execution itself and post-execution. At the pre-execution stage is executed InputFileBuilder Remote Application Module, which as the name suggests creates AMBER simulation input file (.mdin) for a given replica. If exchange type is Umbrella exchange, then for the first simulation cycle this module also creates a restraint (.RST) file. At execution stage, is invoked AMBER, which is used to perform a certain number of MD simulation time-steps. Finally, at the post-execution stage is executed MatrixCalculatorTempEx RAM. To obtain energy and temperature values for the replicas in the current group, first in this module is called get_historical_data() method. This data is used to to calculate reduced energy using a reduces_energy() method call. Reduced energy values are used to populate a column of the swap matrix for the given replica.

After all Compute Units of the MD phase are done, can be performed global calculations of the exchange phase. First is called AMMs method prepare_global_ex_calc() to prepare a Compute Unit for these calculations. This Compute Unit is then submitted to

RPs Unit Manager via `submit_units()` call. After `submit_units()` call is placed blocking `wait_units()` call. Unit Manager executes on a target resource Remote Application Module `GlobalExCalc`. This module performs exchange calculations using Gibbs sampling method and determines pairs of replicas, which should exchange their respective parameters. In `GlobalExCalc` is firsts called `do_exchange()` method, which returns a list of pairs replicas. In `do_exchange()` is called `gibbs_exchange()` method, which for each replica returns a replica to exchange parameters with. `GlobalExCalc` returns `pairs_for_exchange.dat` file in which are written indexes of replicas, which should exchange parameters.

When `GlobalExCalc` is done, in EMM is called `do_exchange()` method of the AMM, which performs an actual exchange of temperatures. At this point a full simulation cycle involving both MD and exchange phases is done. After the whole simulation is done, is called `move_output_files()` method, which moves all files generated in a working directory on the client-side to a single directory called `simulation_output`.

3.4 Validation

In this section we validate implementation of RepEx. 3D-REMD was performed using order parameters of temperature, and umbrella sampling in the ϕ and ψ torsion angles (as shown in Figure 3.7). In the temperature (T) dimension, six windows were chosen from 273K to 373K by geometrical progression. In both umbrella (U) dimensions, eight windows were selected uniformly between $0°$ and $360°$, where each window corresponds to a harmonic restraint centered on it with a force constant of 0.02 kcal·mol^{-1}·degree^{-2}. The total number of replicas is therefore $6{\times}8{\times}8{=}384$. Each replica was previously equilibrated for > 1 ns. In the production run, we set the exchange attempt interval (cycle) to be 20000 steps (20 ps) and in a 15-hour run with 400 cores (25 nodes) on Stampede [55], the simulation finished 90 cycles (1.8 ns). The acceptance ratios of exchange attempts are approximately 3% for T dimension and 25% for U dimensions. Free energy profiles were then generated from the last 1 ns of production data using the maximum likelihood approach implemented in the vFEP package [56, 57]. Free energy profile of the alanine dipeptide system along the ϕ-ψ angles has been studied extensively by theoretical and experimental methods. Figure 3.7 demonstrates identical free energy profiles at 300K, compared with other recent computational studies [58], [59]. The enthalpy contribution of $\alpha R/c7_{eq}$ transition has not been reported in any computational studies. Nevertheless, the temperature dependency of free energy profiles in Figure 3.7 has been utilized to estimate this enthalpy contribution as 1.2 kcal/mole, comparable to the known experimental reported value

1.1 kcal/mole. [60]

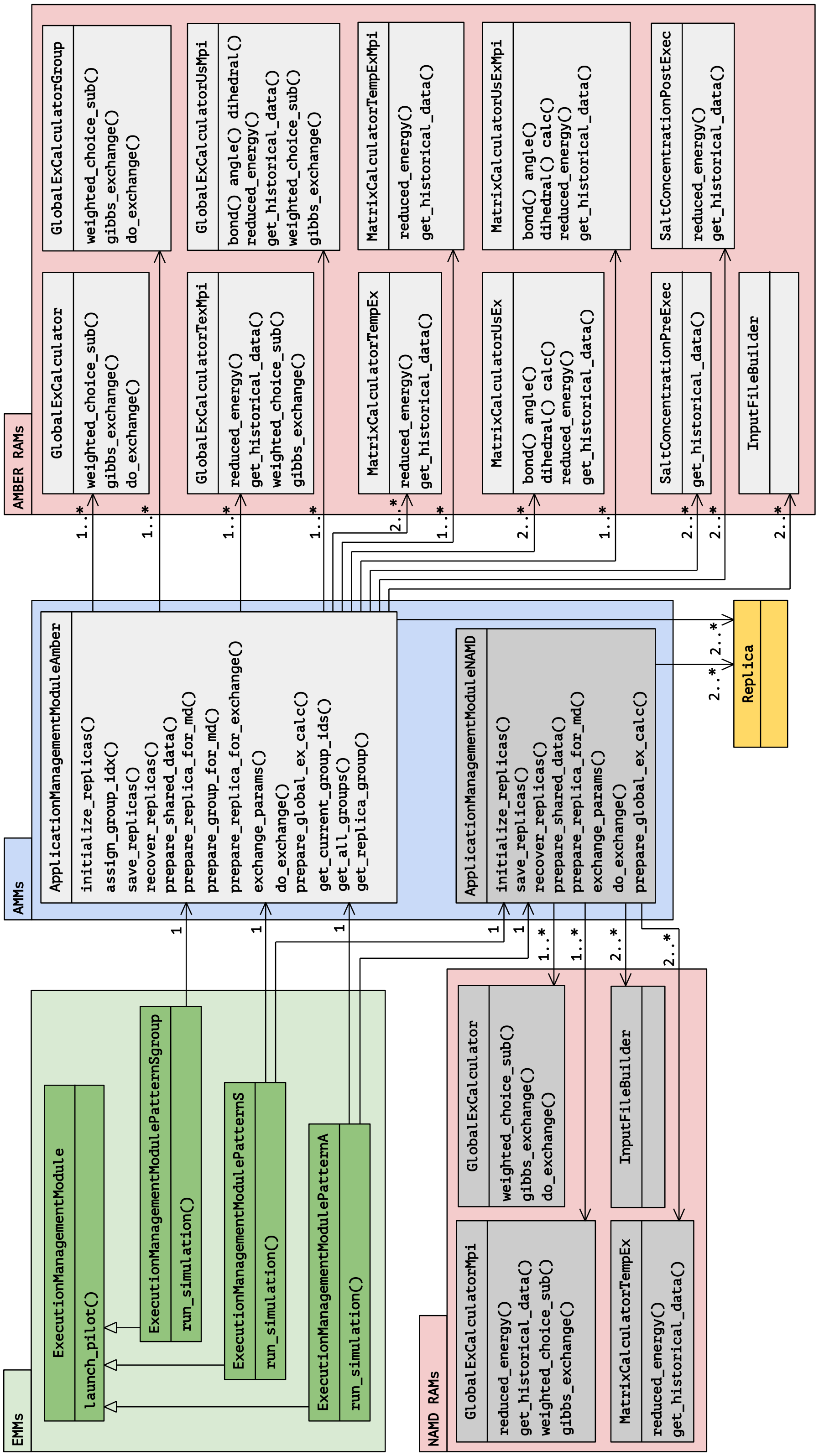


Figure 3.5: UML class diagram. Execution Management Modules are in green boxes. Remote Application Modules are red boxes and Application Management Module is in blue box.

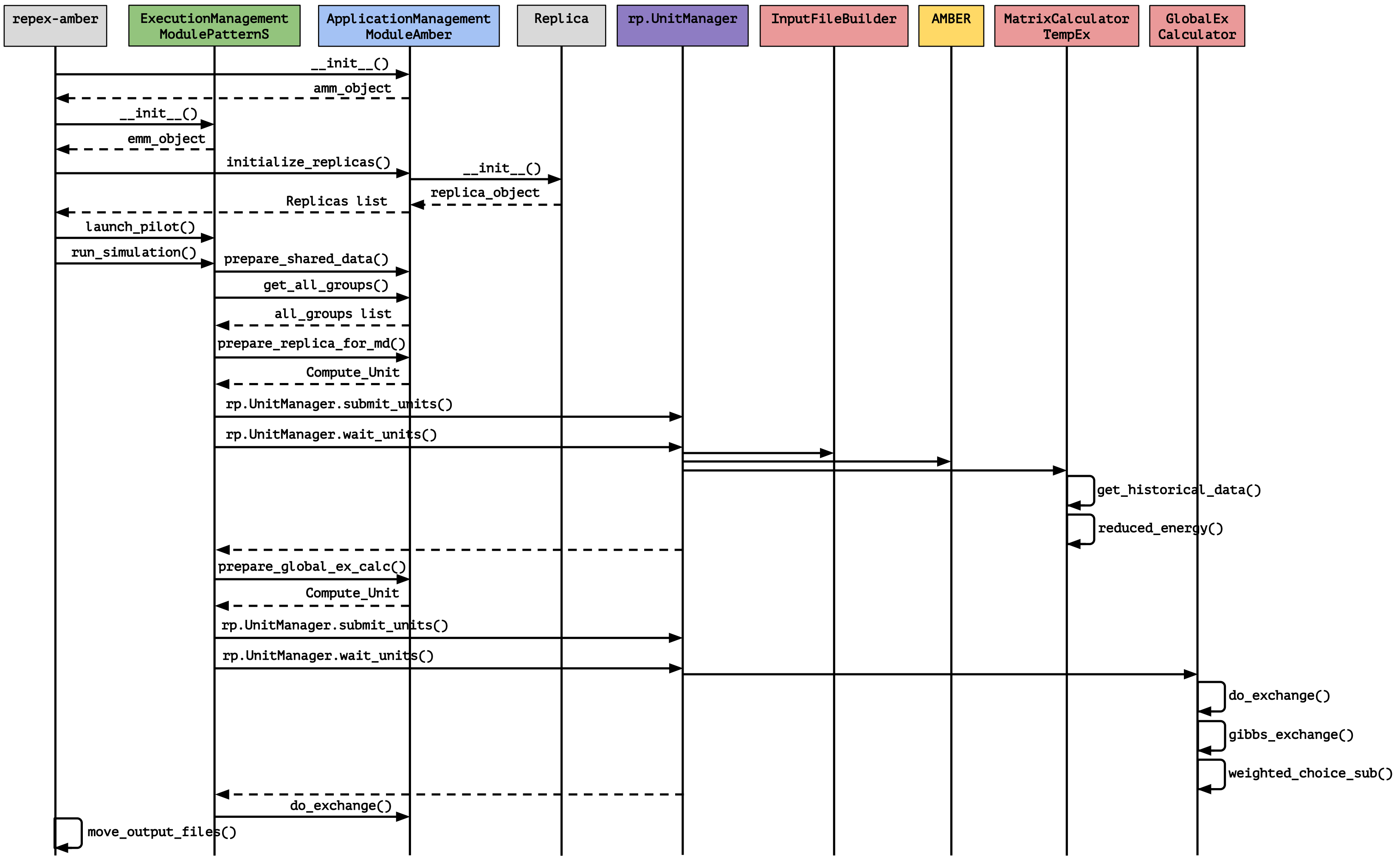
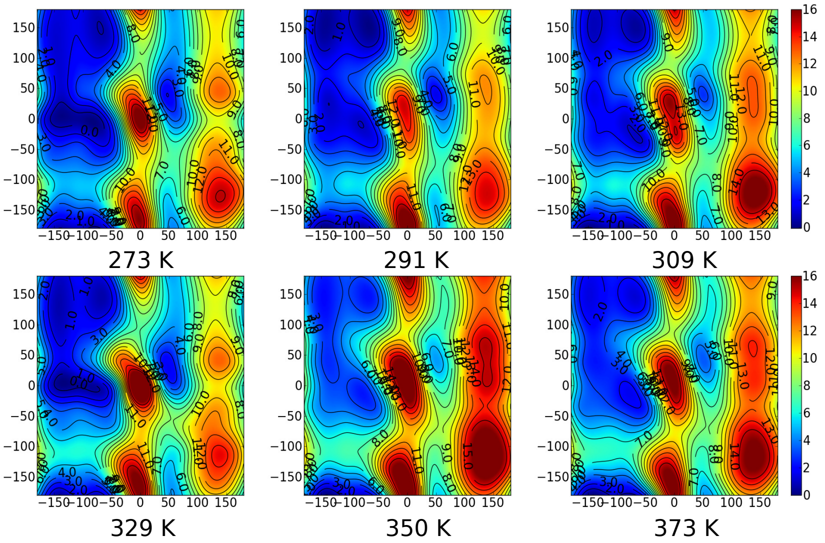


Figure 3.6: UML Control Flow Diagram.



273 K

291 K

309 K

329 K

350 K

373 K

Figure 3.7: Free energy profile of alanine dipeptide backbone torsion at six different temperatures. In all six subplots, the x and y-axes correspond to ϕ and ψ torsion angles, respectively. The range of energies is from 0 kcal/mol to 16 kcal/mol while each level in the contour corresponds to a 1 kcal/mol increment.

Chapter 4

Experiments

Now we present a set of experiments performed to evaluate, optimize and characterize performance of RepEx framework. We first present results of initial evaluation of our framework. We measure average simulation cycle times for multi-dimensional TUU-REMD simulations with various replica counts. Using these results we perform a number of optimizations, targeted at reduction of average simulation cycle times. We perform five optimizations and present result for four of them.

After obtaining acceptable performance (as a result of the performed optimizations), we characterize performance of our framework. First, we present overheads associated with our implementation. We measure file staging times for each of the available exchange types. In addition, we measure overheads associated with RepEx implementation (for 1D and 3D simulations) and overheads associated with RADICAL Pilot.

Next, we present performance results for 1D REMD simulations with both Amber and NAMD MD kernels. We also plot parallel efficiency for all three exchange types. We characterize performance of multi-dimensional REMD simulations by performing weak and strong scaling experiments. In addition, we present results for simulations with multi-core replicas.

Final section of this chapter is dedicated to experiments with asynchronous RE. We fist compare resource utilization of synchronous and asynchronous RE. Next, we quantify how exchange phase is affected by the use of asynchronous RE.

4.1 Performance Optimization

4.1.1 Initial Evaluation

To evaluate our initial implementation we perform 3D REMD simulation runs using Amber as MD engine on Stampede. We use TUU-REMD simulation, comprising one temperature dimension and two umbrella sampling dimensions. For all experiments, we measure and plot average REMD simulation cycle time, which is an average of 4 simulation cycles. Experiments were performed using alanine dipeptide (Ace-Ala-Nme) solvated by water molecules and comprising 2881 atoms. In all three dimensions we perform 6000 time-steps between exchanges, which takes on average 136.64 seconds (in each dimension). Order of dimensions is fixed: umbrella sampling exchange is followed by temperature exchange, which is followed by second umbrella sampling exchange. For all simulation runs, we use the Synchronous RE pattern and Execution Mode I: number of CPU cores is greater or equal to the number of replicas, meaning that all replicas propagate both simulation end exchange phases concurrently. For initial experiments, number of replicas in each dimension is equal, which results in the total number of replicas being a number raised to the power of three. We use RADICAL-Pilot 0.33 and sander as Amber executable.

Input files: For our experiments parameters (file extension .prmtop) and coordinates (file extension .inpcrd) files are shared between all replicas, meaning that only a single instance of each file needs to be transferred from local system to Stampede. It is important to note, that for certain simulation types, each replica may require an individual coordinates file. Amber input files (file extension .mdin) are generated locally for each replica and then transferred to Stampede. This is done for each MD cycle in each dimension. For alanine dipeptide these files are 334 bytes large. For umbrella exchange, locally are generated Amber restraint files (file extension .RST) and transferred from local system to Stampede once, at the beginning of the simulation. Each of these files is 213 bytes in size.

Output files: During the exchange phase each of the replicas generates a matrix_column_d_c.dat file. Size of this file depends on the number of replicas used for a given simulation: for 64 replicas run file size is 331 bytes, for 216 replicas run file size is 962 bytes, for 512 replicas run size is 2.2 Kilobytes. After exchange is finished, these files are transferred from Stampede to laptop for final processing. This is done for each dimension and for each simulation cycle. Pairs of replicas for exchanges of parameters are determined locally.

For M-REMD simulations, T_c is comprised of the 1D cycle time for each dimension, since simulations are performed only in one dimension at any given instant of time.

Initially we define a simulation cycle time as:

$$T_c = T_{MD} + T_{EX} + T_{data-init} + T_{data} + T_{RepEx-over} + T_{RP-over} \tag{4.1}$$

where:

- T_{MD} - MD simulation time, time to perform X simulation time-steps
- T_{EX} - Exchange time. Time for calculations required to determine exchange partners

- T_{data} - Data time. Time to perform data movement procedures, which are mostly remote-to-remote. For example, Amber's .mdinfo files to "staging area" which is accessible by subsequent tasks
- $T_{data-init}$ - Initial data transfer time (from users work station to remote system). Time required to transfer files required to as inputs for MD phase and Exchange phase from local system to remote system (HPC cluster). Note: initial data staging is performed once per simulation and not for every cycle.
- $T_{RepEx-over}$ - RepEx overhead. Time to prepare tasks for execution and time to perform local RepEx method calls
- $T_{RP-over}$ - RP overhead. Time required for task launching on a target resource and time for internal RP communication

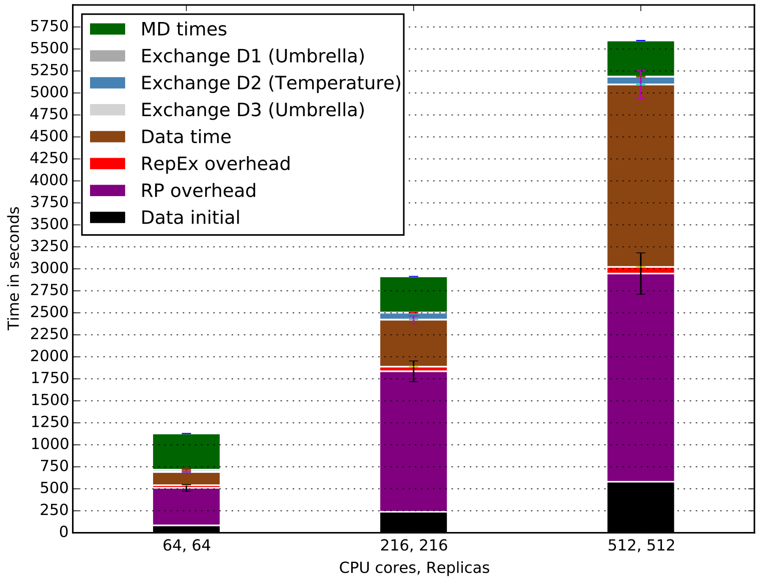


Figure 4.1: Initial performance results: Decomposition of average simulation cycle time into contributing factors for alanine dipeptide TUU-REMD. All simulation runs are performed on Stampede.

Initial performance results are presented in Figure 4.1. For all three runs, MD times are less than 50 % of the cycle time. For simulation run with 64 replicas, MD time comprises ∼35 % of the cycle time, for the run with 216 replicas ∼16 % and for the run with 512 replicas ∼7 %. All timings, with exception of MD simulation time demonstrate exponential growth. As seen in Figure 4.1, for all three runs, initial data staging takes a significant amount of time. For experiment with 512 replicas, initial data staging takes around 550 seconds, which is sub-optimal.

Clearly, there is a lot of potential for performance improvement. In the next section we outline and present our performance optimization strategy.

4.1.2 Optimization I

As a first performance optimization step, we reduce time required for data movement. We eliminate the need to transfer from local system to remote HPC cluster, simulation input files (.mdin) for each replica. Instead, we transferring a single, shared input file template and a script which generates an input file for each replica according to specified simulation parameters. This means that we need to add a pre-execution step for MD simulation tasks, which would run the script to generate simulation input file for each replica. In input file template, parameters specific to individual replica are substituted with placeholders. A script substitutes placeholders in input file template with parameters corresponding to a given replica. This procedure occurs at a pre_exec (pre-execution) stage of a Compute Unit. For each cycle and in each dimension we generate simulation input file on a remote HPC cluster (Stampede).

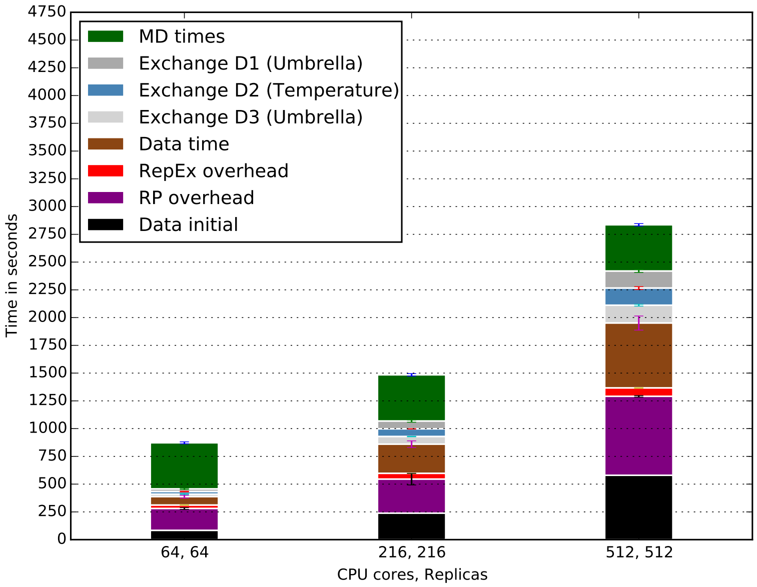


Figure 4.2: Optimization I: remote generation of simulation input files. Decomposition of average simulation cycle time into contributing factors for alanine dipeptide TUU-REMD. All simulation runs are performed on Stampede.

Performance results of this optimization are presented in Figure 4.2. As depicted in Figure 4.2, this optimization resulted in substantial decrease in data times and RP overhead times. All other timings remained the same. RP overhead timings were reduced, since task launching delay is affected by the number and size of files which are staged-in prior to task execution. Data time for simulation run with 64 replicas is now reduced by ∼70 seconds, but RP overhead time by ∼190 seconds. For run with 216 replicas, performance gains are more apparent. Data time decreased by ∼240 seconds, but RP overhead by ∼1280 seconds. Lastly, for the run with 512 replicas, data time was reduced by ∼1400 seconds and RP overhead by ∼1380 seconds.

4.1.3 Optimization II

To further reduce time for file staging, we eradicate transferring of restraint (.RST) files for umbrella exchange dimensions. Similarly as with simulation input files, from the local system we transfer to remote HPC cluster only a restraint file template and a script, which generates restraint file based on provided parameters. Both, restraint file template and a script are shared among all replicas. In restraint file template, parameters specific to individual replica are substituted with placeholders. Before first simulation cycle, each task (RP's Compute Unit) runs a script, which substitutes placeholders in restraint file template with parameters corresponding to a given replica. This procedure is performed only once and is specified as a pre_exec (pre-execution) stage of a Compute Unit. As a result, we create all restraint files on a remote cluster and don't need to transfer all of the from local system.

In addition, we also reduce the number of file transfers for the exchange phase. Previously, for exchange, we transferred N (number of replicas) matrix_column_d_c.dat (d is dimension, c is cycle) files, from remote cluster to local system to finalize exchange phase. Now we determine exchange partners remotely, thus eliminated the need to transfer N matrix_column_d_c.dat files. As a result we transfer a single file with replica ids. During the exchange phase, when all individual exchange tasks are complete, we execute an additional task (Remote Application Module), which reads data from individual matrix_column_d_c.dat files and determines pairs of replicas for exchange of parameters.

Resulting timings are provided in Figure 4.3. Intuitively, the aforementioned optimizations allowed to reduce initial data time and data time. All other timings were not affected by this optimization. The initial data times decreased by 71, 235 and 508 seconds for simulations involving 64, 216 and 512 replicas respectively. Due to elimination of restraint file transfers from local system to remote cluster, initial data times are now independent of replica counts and take ∼12 seconds for all three runs.

Elimination of matrix_column_d_c.dat file transfers, resulted in substantial decrease in data times. For a simulation run with 512 replicas, data times decreased by ∼510 seconds.

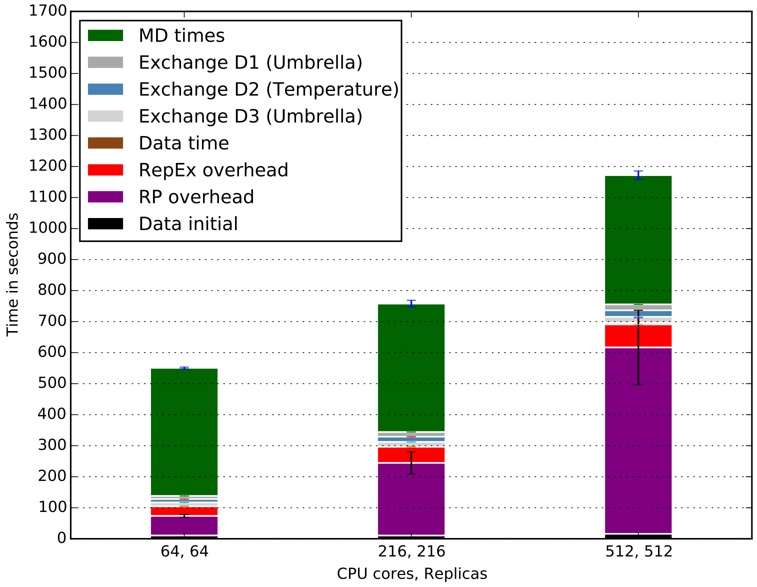


Figure 4.3: Optimization II: remote generation of restraint files and remote determination of exchange partners. Decomposition of average simulation cycle time into contributing factors for alanine dipeptide TUU-REMD. All simulation runs are performed on Stampede.

4.1.4 Optimization III

Analysis of the Compute Unit execution profiles, revealed that there is a considerable delay between execution start-up times of the Compute Units. Despite being submitted for the execution simultaneously, Compute Units will not start executing at the same time. In addition, as the number of simultaneously submitted Compute Units increases, difference between execution start of the first unit and execution start of the last unit, is increasing proportionally to the number of units.

In RepEx, for both MD and exchange phases we create a separate set of Compute Units, which is equal in size to the number of replicas. To mitigate the effect of the execution start-up delay of RP's Compute Units, we merge MD phase units with exchange phase units. As a consequence, exchange phase tasks now are performed as a post-execution (post_exec) step of the MD compute units.

Average simulation cycle times after optimization III are shown in Figure 4.4. As shown in Figure 4.4, RP overhead times decreased for all three runs. For the run with 64 replicas, RP overhead decreased by 58 seconds. For simulation run with 216 replicas, by ∼90 seconds and for the run with 512 replicas, by ∼270 seconds.

Execution start-up delay for Compute Units, currently is the largest contributing factor to

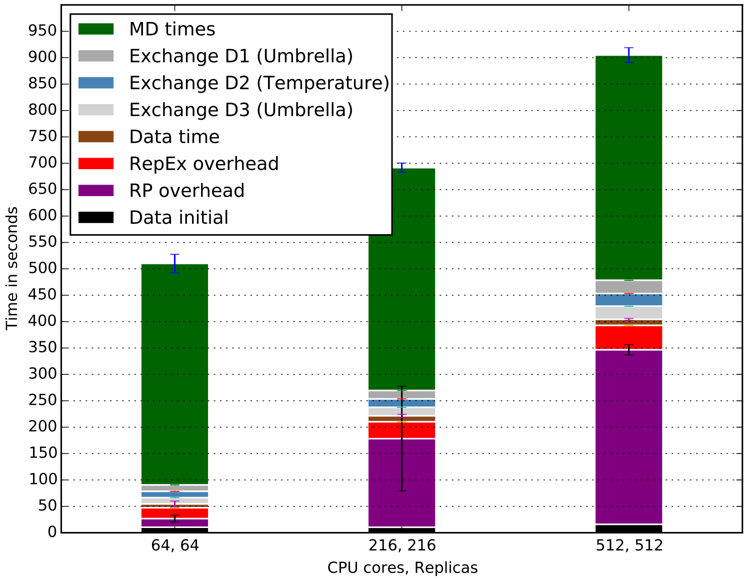


Figure 4.4: Optimization III: merging of MD simulation tasks with exchange tasks. Decomposition of average simulation cycle time into contributing factors for alanine dipeptide TUU-REMD. All simulation runs are performed on Stampede.

the simulation overhead. In ideal case, MD time should be responsible for the ∼90% of the average REMD simulation cycle time. Currently, for the run with 512 replicas, MD time is responsible for ∼47% of the total simulation cycle time, despite the fact, that we have reduced execution start-up delay for nearly a half for all runs.

4.1.5 Optimization IV

All simulation runs for optimizations I,II and III were performed using default configuration of RADICAL-Pilot agent. It is possible to modify this configuration by increasing a number of execution and data staging workers, thus improving the performance of the application. Experimentally it was determined that for the Stampede cluster the best configuration is with 8 components for each of the workers. In addition, we now are using a development branch of the latest RADICAL-Pilot version (v0.42-143-gcefbb3b@devel).

In Figure 4.5 are presented results after making the above changes. As we can see, for all three runs we achieved a substantial decrease in RP overhead: for simulation run involving 216 replicas, RP overhead is now reduced by ∼140 seconds, but for the run with 512 replicas by ∼290 seconds.

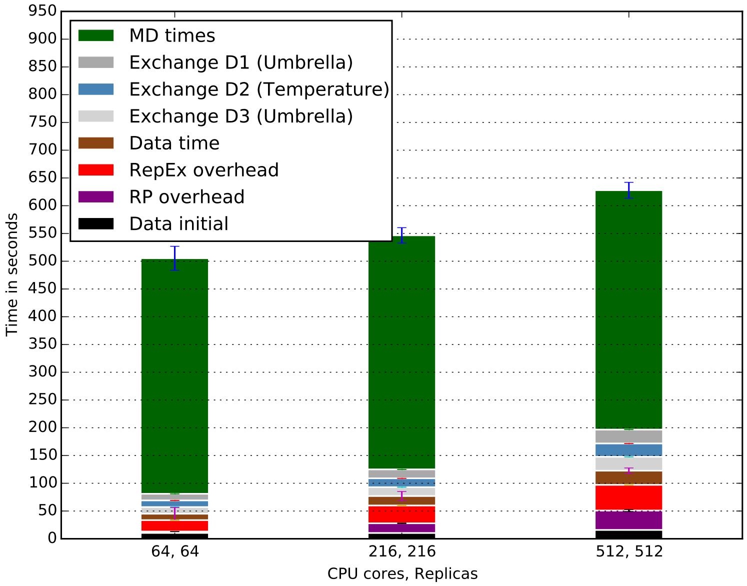


Figure 4.5: Optimization IV: Increasing the number of execution and data staging workers for RP agent. Using a development branch of the latest RP version. Decomposition of average simulation cycle time into contributing factors for alanine dipeptide TUU-REMD. All simulation runs are performed on Stampede.

4.1.6 Optimization V

To further decrease RP overhead, RepEx overhead, data times and exchange times, we decided to merge multiple MD and exchange tasks into a single Compute Unit. We merge together only tasks which are belonging to the same group in the current exchange dimension. More information on grouping of replicas for multi-dimensional REMD simulations can be found in section 2.5. For a simulation involving 1000 replicas, this approach allows to reduce the number of tasks by a factor of 10.

To merge MD and exchange tasks for multiple replicas into a single Compute Unit for each exchange type we write a special Remote Application Module. This module is a Python MPI script, which is responsible for creation of simulation input files, restraint files, concurrent execution of Amber executables and exchange calculations.

This approach also substantially reduces the number of I/O operations, since for a group of size N, N files are read now by a single script, instead of by N scripts.

One major drawback of this approach is that it only can be used for MD tasks requesting a single CPU core. This means that if we are using Amber, we can't use `pmemd.MPI` or `pmemd.cuda`.

Unfortunately, we were not able to achieve consistent performance improvements for this optimization. There are two reasons for that. First, performance of MPI tasks launched by RP, when the number of requested CPU cores is less than one node is inconsistent. Second, launching of MPI Compute Units by RP is slower than launching of single core Compute Units. Nevertheless, we believe that when the aforementioned problems will be resolved, this optimization might be useful.

4.1.7 Summary

In this section we outlined steps, taken to reduce the time to perform an RE simulation cycle. To identify possibilities for performance improvement we decomposed a simulation cycle into contributing factors: MD simulation time, Exchange time, Data time, Initial data transfer time, RepEx overhead and RP overhead. We next performed an initial evaluation of our framework using three-dimension REMD (TUU). In first two optimizations we reduced the amount of file movement. This alone, allowed to significantly reduce average simulation cycle time. For example, for 512 replicas optimizations I and II together, reduced simulation time by ∼ 4400 seconds. As next optimization step, we reduced RP overhead by merging together MD tasks with exchange tasks, reducing a simulation cycle by ∼270 seconds (512 replicas). In optimization IV, we increase the number of execution and data staging workers for RP, and switch to a latest development version of RP. We achieve a decrease in simulation cycle time by ∼290 seconds. Next we attempted to further reduce simulation time, by merging MD tasks for replicas belonging to the same group together. This approach allows to reduce the total number of tasks and as a consequence the RP overhead. Unfortunately we were not able to achieve consistent performance improvements using this approach. As a result of performance optimization activity we reduced a simulation cycle time by ∼600, ∼2300 and ∼4900 seconds for 64, 216 and 512 replicas respectively.

4.2 Performance characterization

Having validated both the design and implementation of RepEx, in this section we discuss a series of experiments used to demonstrate the unique functional capabilities and characterize it's performance.

For all experiments, we measure and plot the average REMD simulation cycle time, which is an average of 4 simulation cycles. Experiments were performed using alanine dipeptide (Ace-Ala-Nme) solvated by water molecules on Extreme Science and Engineering Discovery

Environment [55] (XSEDE) allocated systems: Stampede and SuperMIC. For all experiments was used Synchronous RE pattern.

Simulation cycle time is calculated using formula defined in section 4.1.1. For M-REMD simulations, T_c is comprised of the 1-D cycle time for each dimension, since simulations are performed only in one dimension at any given instant of time.

We calculate weak scaling efficiency as:

$$E_w = \frac{T_1}{T_N} \times 100\% \tag{4.2}$$

where:

- T_1 - time to complete simulation cycle involving minimal number of replicas R_{min} with number of CPU cores equal to the number of replicas, e.g. 8 replicas on 8 CPUs
- T_N - time to complete simulation cycle involving N replicas with N CPU cores

We calculate strong scaling efficiency as:

$$E_s = \frac{T_1}{(M/N_{min}) \times T_N} \times 100\% \tag{4.3}$$

where:

- T_1 - time to complete simulation cycle involving N replicas with minimal number of CPU cores N_{min}, e.g. 1024 replicas on 8 CPUs
- T_N - time to complete simulation cycle involving N replicas with M CPU cores, where $N_{min} < M$
- M - number of used CPU cores

Results obtained in Chapter 4 can be reproduced by following instructions at [61]. All experiments (except asynchronous RE experiments) were performed with RP version 0.35. Last version of RP is 0.40.1 and thanks to various optimizations it is capable of substantially better performance. These optimizations, however, only alter the RP overhead timings (as well as data timings) presented in this section and will have minimal impact on the overall performance characterization of RepEx.

4.2.1 Characterization of Overheads

There are three factors which contribute to the T_c as a result of design decisions we have made. These factors are data time, RepEx overhead and RP overhead. In this subsection, we summarize how these factors influence the T_c.

Figure 4.6 presents the values of data times, RepEx overheads and RP overheads for simulation runs involving 64, 216, 512, 1000 and 1728 replicas on SuperMIC. For all runs, we use a single CPU core per replica and use Execution Mode I with synchronous RE pattern.

Values of data times depend on the exchange type, since data movement patterns differ for each exchange type. As depicted in Figure 4.6, data times for temperature exchange are shorter than for umbrella exchange and salt concentration. For all replica counts, data times are relatively small: longest data transfer time is 6.3 seconds. This is due to the fact, that majority of the transfers are happening within the cluster/resource. Consequently, data times change as a function of a target system, since largest contributing factor, is the performance of a parallel file system.

RepEx overhead depends on the total number of replicas and on simulation type. For all 1D simulations, values of RepEx overhead are nearly identical, since the number of operations to perform task preparation is very similar. RepEx overhead times for 3D simulations are longer, since there are more data associated with each replica, complexity of data structures is increased and more computations are performed during task preparation.

RP overhead depends only on the number of replicas (tasks) launched concurrently. As we can see in Figure 4.6, RP overhead is proportional to the number of replicas.

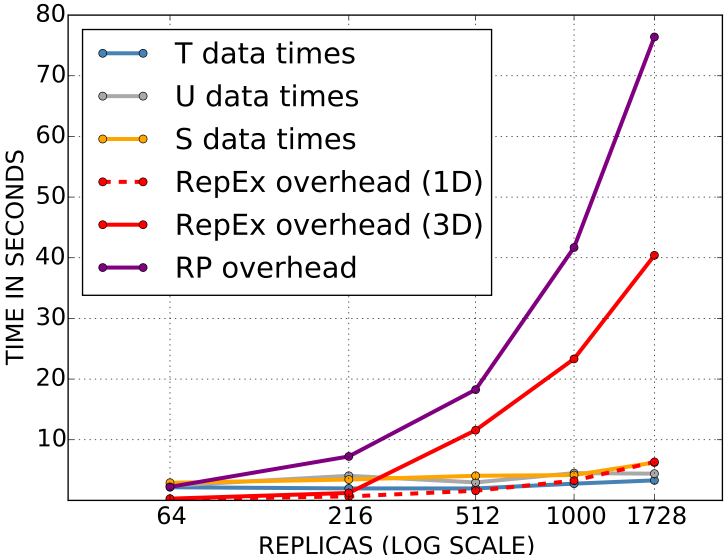


Figure 4.6: Characterization of overheads: Data times, RepEx overhead and RP overhead.

4.2.2 Performance Characterization of 1D-REMD

In this subsection, we characterize performance of 1D REMD simulations with RepEx. For each of the three available 1D-REMD simulations: temperature exchange (T-REMD), umbrella exchange (U-REMD) and salt concentration exchange (S-REMD) we measure average cycle times. We perform simulation runs involving 64, 216, 512, 1000 and 1728 replicas in Execution Mode I. All runs are conducted with a single CPU core per replica and **sander** as the Amber executable. We use alanine dipeptide solvated by water molecules comprising a total of 2881 atoms and perform 6000 simulation time-steps between exchanges. We perform all runs on SuperMIC supercomputer [55]. Results of these experiments are presented in Figure 4.7.

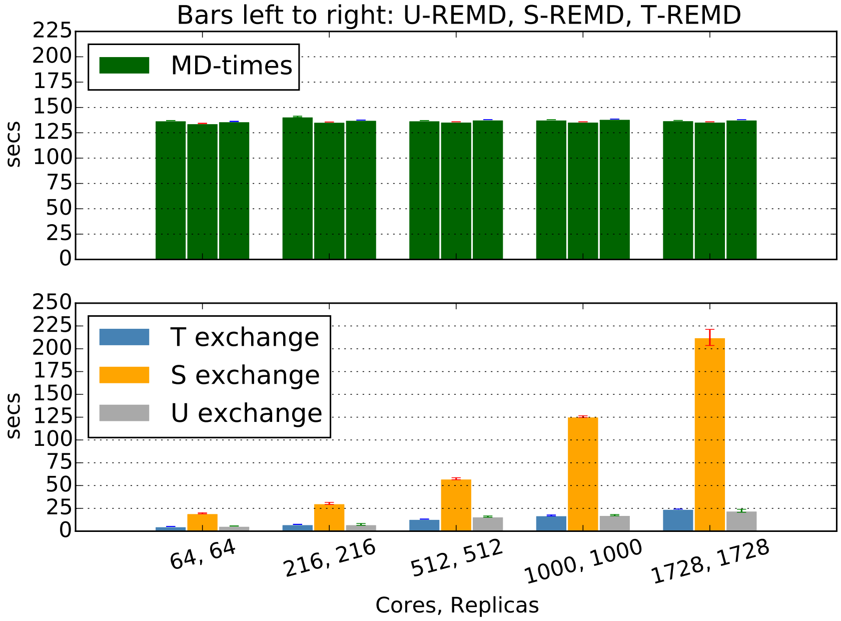


Figure 4.7: 1D REMD experiments with RepEx: weak scaling. Decomposition of average simulation cycle times T_c (in seconds) into MD simulation time and exchange time for umbrella sampling, salt concentration and temperature exchange. For all simulation runs the number of replicas is equal to the number of CPU cores (e.g. 1 core per replica) and both vary from 64 to 1728. All simulation runs are performed on SuperMIC supercomputer.

As we can see, for all three exchange types, the time to perform 6000 time-steps is nearly identical, as evidenced by the almost similar average heights of dark green bars in Figure 4.7 (139.6 seconds).

Next we discuss exchange timings for different exchange parameters, as seen in the lower panel of Figure 4.7. Timings for temperature and umbrella exchange are similar and have a nearly linear growth rate. For both exchange types, we use a single MPI task to perform an exchange. In case of U-REMD we have implemented a single point energy calculation internally. Despite the fact that U-REMD exchange is more involved, we don't see a significant difference in exchange timings between U-REMD and T-REMD.

Due to the mathematical complexity, the single point energy calculation for S-REMD is calculated using Amber for each replica in each state. This implies that for each replica, an additional task is required. Since we are using Amber's group files, this task requires at least as many CPU cores as there are potential exchange partners for each replica. Consequently, the exchange times for S-REMD are substantially longer, but nonetheless have a nearly linear growth rate.

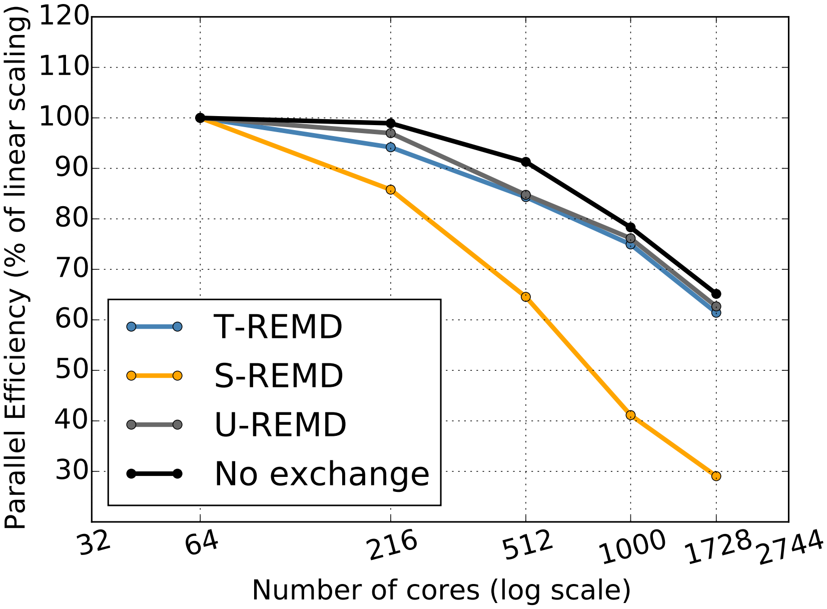


Figure 4.8: Parallel Efficiency (% of linear scaling) for Temperature Exchange REMD (1D), Salt Concentration REMD (1D) and Umbrella Sampling REMD (1D) using Amber MD engine on SuperMIC supercomputer.

The parallel efficiency results for the 1D-REMD simulations are presented in Figure 4.8. We calculate parallel efficiency for the weak scaling scenario and use average cycle time for simulation with 64 cores as starting point, e.g. 100% efficiency. We also present efficiency results for simulations without an exchange phase (black line). This quantifies the influence of exchanges on the efficiency of 1D simulations. Since all tasks have overheads associated with them, we observe a decrease in efficiency even if there is no exchange. For all three exchange types we observe decrease in efficiency while increasing the number of cores. Efficiency values for T-REMD and U-REMD are similar and demonstrate linear behavior. Efficiency for S-REMD is lower. This is caused by specifics of exchange phase, discussed earlier.

4.2.3 T-REMD with NAMD engine

To demonstrate RepEx's ability to use different MD engines for REMD simulations we perform weak scaling experiments using T-REMD with NAMD engine. We run our experiments on

SuperMIC, use NAMD-2.10 and perform a total of 4000 time-steps between exchanges. We perform runs with 64, 216, 512, 1000 and 1728 replicas. For each replica we use a single CPU core and we allocate enough cores to run all replicas concurrently (Execution Mode I). Results of these experiments are provided in Figure 4.9.

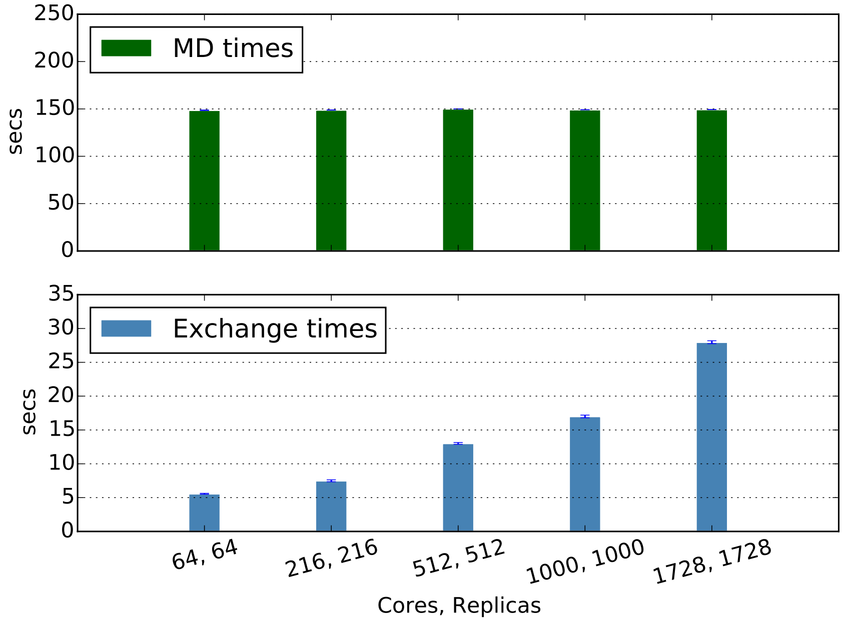


Figure 4.9: Experiments with NAMD engine. Decomposition of average simulation cycle times T_c (in seconds) into MD simulation time and Exchange time for weak scaling scenario. Experiments are performed on SuperMIC supercomputer, using T-REMD. For MD simulation are used single-core replicas.

As expected, MD times for all cores/replicas pairs are nearly equal. The growth rate for exchange times can't be characterized as monomial.

4.2.4 M-REMD performance characterization

Similar to 1D-REMD experiments, we use alanine dipeptide to characterize M-REMD performance and 6000 simulation time-steps between exchanges. We perform weak and strong scaling experiments for TSU-REMD on Stampede.

Weak Scaling: To characterize the weak scaling performance of M-REMD simulations, the number of replicas in each dimension is kept equal, thus, as the number of replicas in one dimension varies from 4, 6, 8, 10 and 12, it results in the total number of replicas equal to 64, 216, 512, 1000 and 1728 respectively. We use Amber 12.0, and **sander** as Amber executable, as for each replica we use a single CPU core. The experiments are performed in Execution mode I, i.e., with enough cores to run all replicas concurrently. Results of experiments are provided in Figure 4.10.

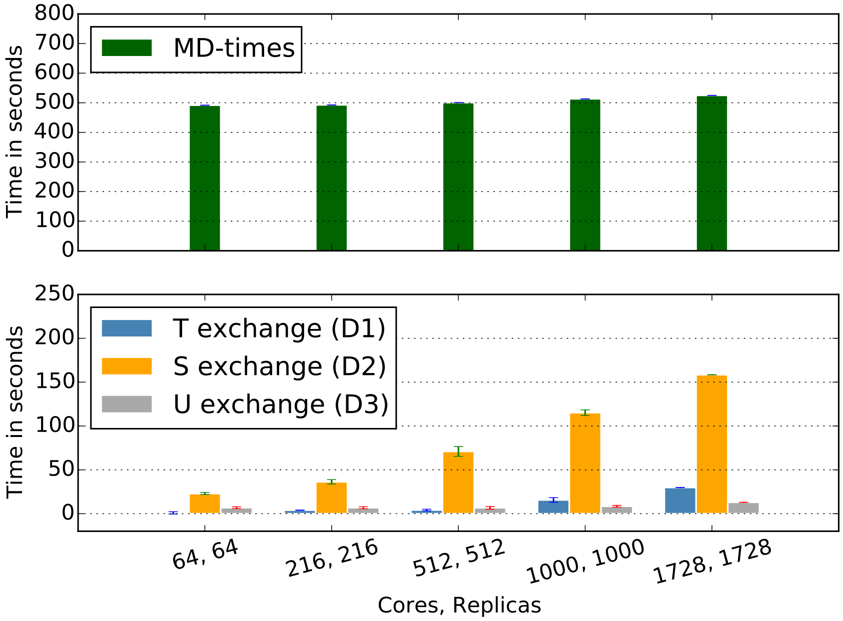


Figure 4.10: Multi-dimensional REMD experiments with RepEx - weak scaling. TSU-REMD on Stampede using Amber MD engine. For all simulation runs the number of replicas is equal to the number of CPU cores and both vary from 64 to 1728. For all simulation runs are used single-core replicas. In figure is shown decomposition of average simulation cycle times T_c (in seconds) into MD and exchange times.

For all simulation runs MD times are nearly identical: ∼ 495.0 seconds. It is expected, since variation in the number of replicas should not affect MD time.

We observe a nearly linear scaling for exchange timings in all three dimensions. While temperature and umbrella exchange timings are very similar, salt concentration exchange takes substantially more time. As mentioned in Subsection 4.2.2, for salt concentration exchange we use Amber to perform a single point energy calculations, which results in doubling of tasks and higher computational requirements for this exchange type.

Parallel Efficiency results are presented in Figure 4.12(a). We observe a rapid decrease in efficiency with increase in the number of cores. This can be explained by the influence of performance for salt concentration exchange. Despite that, for all core counts, efficiency is above 50 %.

Strong Scaling: To characterize the strong scaling performance of M-REMD RepEx, the number of replicas is fixed at 1728 with 12 replicas in each dimension, but the number of cores is varied: 112, 224, 432, 864 and 1728. Again, we use Amber 12.0, and **sander** as Amber executable, since for each replica a single CPU core is used. The experiments are performed using Execution Mode II, as we have fewer cores than replicas for all cores/replicas pairs, except the last one. Results of these experiments are provided in Figure 4.11.

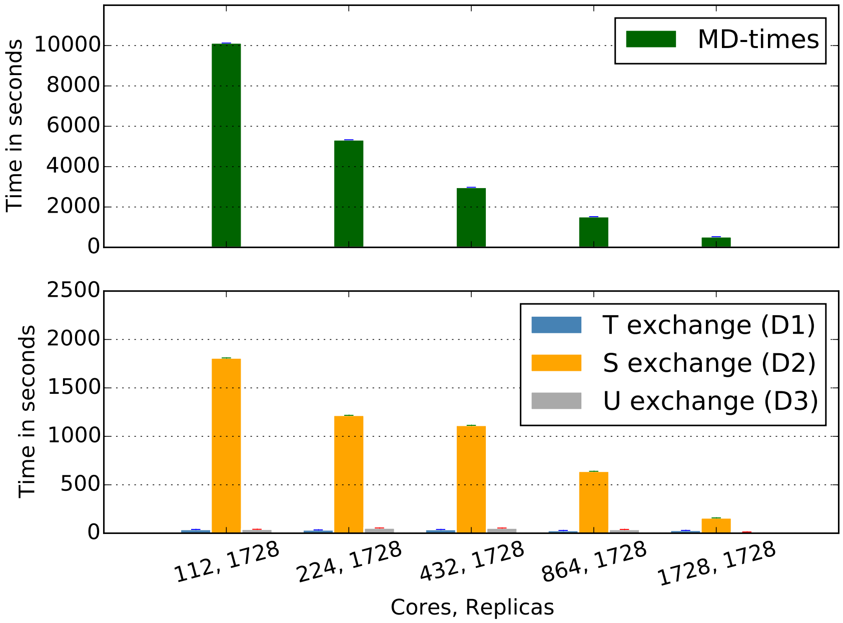
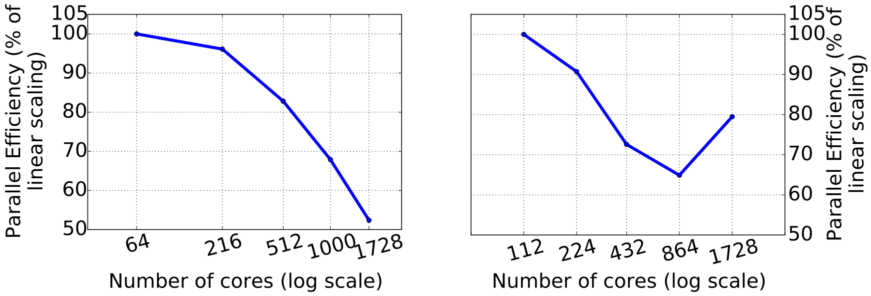


Figure 4.11: Multi-dimensional REMD experiments with RepEx: strong scaling. TSU-REMD on Stampede using Amber MD engine. Number of replicas is fixed at 1728, but the number of CPU cores is increased from 112 to 1728. For all runs are used single-core replicas. In figure are shown MD simulation and exchange times. RepEx enables users to vary the size of computational resources independently of the simulation size. Allocating more CPUs reduces the T_c.

As illustrated in Figure 4.11, decrease in MD time is proportional to the number of cores: the doubling of the number of CPU cores, results in decrease of MD time by nearly a half.

Exchange time in temperature exchange and umbrella exchange dimensions is almost equal for all numbers of CPU cores. This highlights the fact, that implementation of temperature exchange and umbrella exchange are very similar. Due to task launching delay and grouping of replicas by parameter values in each dimension, the exchanges largely overlap with MD. As a result, tasks which have finished simulation phase sooner can perform certain exchange procedures, before an exchange is finalized. Compared to temperature exchange and umbrella exchange, salt concentration exchange times are significantly higher: at 112 cores, salt exchange time takes nearly 1800 seconds.

Parallel Efficiency results are presented in Figure 4.12(b). As we can see, efficiency graph is non-linear. We observe a decrease in efficiency up to the last data point where number of CPUs is equal to the number of replicas. For the last data point, efficiency increases. This behavior is caused by the MPI task scheduling issue of RP. In the next release of RepEx this issue will



(a)

(b)

Figure 4.12: Parallel Efficiency (% of linear scaling) for TSU-REMD on Stampede using Amber MD engine - (a) weak scaling, (b) strong scaling.

be addressed.

4.2.5 REMD with Multi-core Replicas

To demonstrate RepEx's capability to execute replicas using multiple CPU cores we use solvated alanine dipeptide with 64366 atoms. We perform a total of 20000 time-steps between each exchange. Experiments are performed on Stampede using Amber 12.0 and pmemd.MPI as Amber executable for multi-core replicas and sander for single-core replicas. We use different executables, since pmemd.MPI can't be run on a single CPU core.

We perform weak scaling experiments using multi-core replicas and multi-dimensional TUU-REMD with one temperature dimension and two umbrella dimensions. We perform simulation runs with fixed number of replicas and change number of CPU cores per replica. For all runs, we use 216 replicas, but the number of cores per replica varies from 1 to 64. Results of these experiments are provided in Figure 4.13.

We observe a substantial drop in MD times when we use multiple cores per replica. We attribute this drop due to RepEx's ability to support replicas running over multi-core/multi-nodes, as well as using a highly efficient pmemd.MPI code. Further increase of CPU cores per replica doesn't demonstrate a linear behavior. This is not a limitation of the RepEx framework but attributable to the size of the alanine dipeptide, which although relatively larger than the earlier physical system, is small in absolute terms and thus makes it difficult to gain significant performance improvements by using more CPUs.

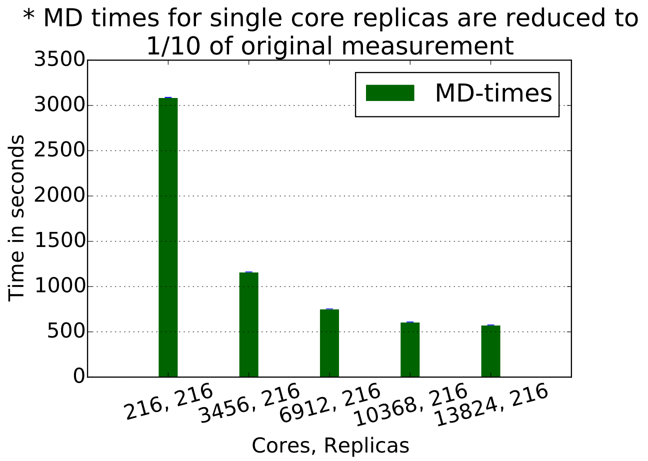


Figure 4.13: Multi-core replica experiments using TUU-REMD with Amber engine. MD times for weak scaling scenario. Experiments are performed on Stampede supercomputer. Number of replicas is fixed at 216, but the number of CPUs per replicas is increased from 1 to 64.

4.2.6 Asynchronous REMD

This section is aimed at identification and quantification of advantages and disadvantages associated with the use of asynchronous RE pattern.

First, we compare and contrast synchronous and asynchronous RE patterns in terms of resource utilization. We define resource utilization as a percentage of ideal resource utilization, which is obtained by measuring total simulation time, while running MD simulation on a target resource 100% of the time. Next, we quantify the differences in exchange metrics, such as crosswalks, accepted exchanges and attempted exchanges between the two patterns.

We define a crosswalk as a transition of a single replica into all possible thermodynamic states. For example, if we are performing a temperature exchange REMD, a crosswalk is obtained, if a given replica received all initial temperature values after a number of exchanges. At a very minimum a crosswalk requires N accepted exchanges, where N is the number of replicas (and the number of initial temperatures). Normally to obtain a crosswalk a number of accepted exchanges is greater than N.

We record an exchange as accepted, if after an exchange phase, current thermodynamic state of the replica was changed. There are cases, in which despite multiple exchanges replica ends up with its initial thermodynamic state. For example, the following pairs of replicas perform an exchange: (2,3), (3,4), (4,2) and (3,4). As a result of this sequence of exchanges all of the replicas remain with their initial temperatures and none of the exchanges are counted as accepted.

Finally we record an exchange attempt if during the exchange phase, every time a replica exchanges its temperature with another replica, regardless of the final temperature value replica receives at the end. For example the following pairs of replicas perform an exchange: (2,3) and (3,2). After two exchanges temperatures of replicas remain unchanged, but for each replica we record two exchange attempts.

Resource Utilization

First, we compare resource utilization for synchronous and asynchronous RE pattern. For both patterns we use TUU-REMD with Amber MD engine. We run a Quantum Mechanics / Molecular Mechanics (QM/MM) simulation with a truncated dinucleotide molecule. We simulate a non-enzymatic transesterification reaction, which is a model reaction for the RNA cleavage reaction. The QM method we are using is called AM1/d-PhoT which is a semi-empirical method that is specially parameterized for this type of reaction.

We calculate resource utilization as:

$$U = \frac{U_{pattern}}{U_{max}} \times 100\% \tag{4.4}$$

where:

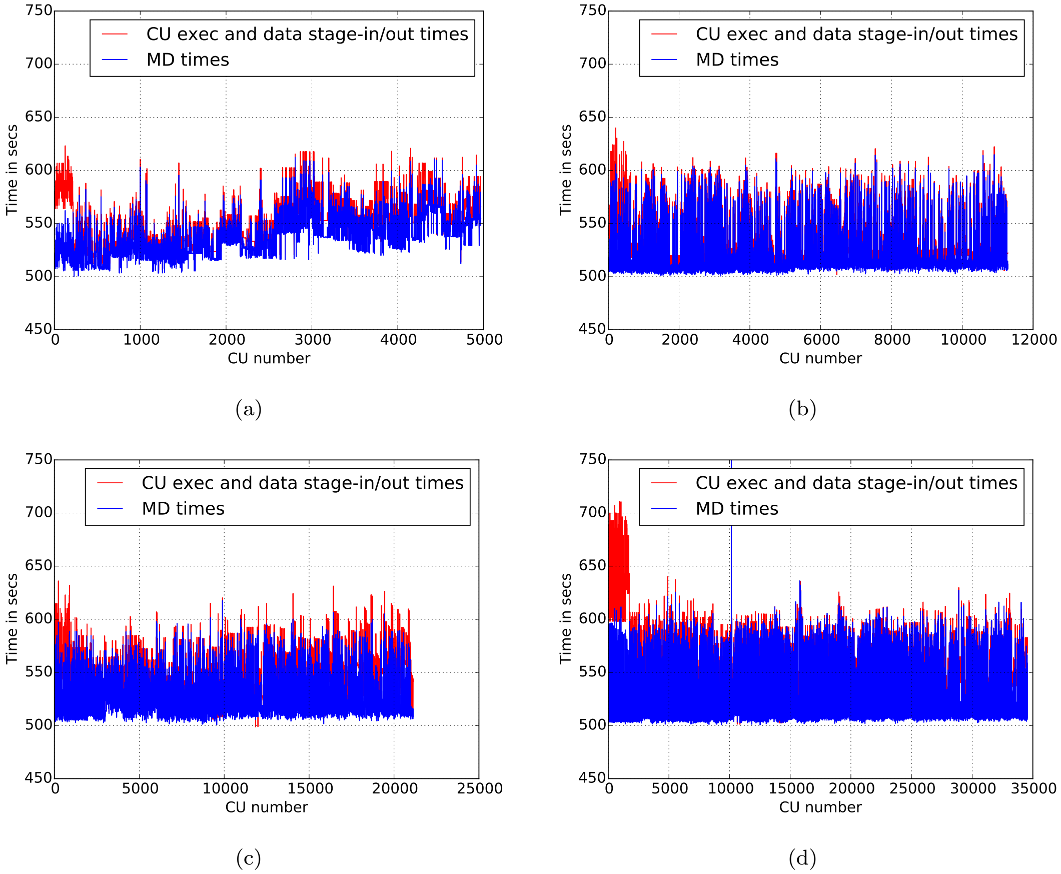
- $U_{pattern}$ - utilization using (async/sync) RE pattern. Total simulation time obtained on N CPU cores.
- U_{max} - maximal (ideal) utilization, when we run MD on all allocated CPU cores 100% of the time.

For asynchronous RE, we denote the ratio of replicas which have finished MD phase (before entering an exchange phase) to the total number of replicas as W_r.

In RepEx interaction with Amber MD engine (or any other MD engine) is done through Amber input/output files. Thus, it is imperative to understand by how much a shared file system of a given HPC cluster affects performance of RepEx and how the role of a file system changes at scale.

We perform all simulation runs on Stampede HPC cluster. For all runs we use a single CPU core per replica and each run is 240 minute long. To demonstrate fluctuation in execution times of MD tasks, we perform weak scaling runs with 216, 512, 1000 and 1728 replicas while simulating 2000 time-steps in between exchanges. Obtained results are presented in Figure 4.14.

In Figure 4.14 (a) are depicted execution times for MD Compute Units, as they are measured



(a)

(b)

(c)

(d)

Figure 4.14: Execution times for MD tasks when running QM/MM REMD with synchronous RE pattern and using **216 (a), 512 (b), 1000 (c) and 1728 (d)** replicas: times to run Amber (sander) executable (blue bars); times to run Amber executable plus times for file staging (red bars). For all simulation runs all replicas are running concurrently. All runs are performed on Stampede HPC cluster.

internally by Amber and execution times together with file staging times as they are measured by RP profiler. For the first 216 Compute Units timings with file staging (red line) are much higher, then for the rest of the units. This can be explained by the setup of the shared file system on Stampede. For the rest of the units file staging times on average are ∼6 seconds. As we can see, difference in execution times between MD Compute Units can be up to 100 seconds. Another observation is increase in execution times for units 3000-3216. This can only be explained by the system noise at the time of the measurement.

Similar results were obtained for simulation run with 512 replicas (Figure 4.14 (b)). For the first 512 Compute Units file staging times are longer, than for the rest of the units. On average file staging times are still relatively short, but difference in execution times does not exceed 100 seconds.

Results of the simulation run with 1000 replicas are depicted in Figure 4.14 (c). In contrast to two previous runs, file staging times are now much larger (red bars). For some units file staging times can be as long as 50 seconds. At the same time, fluctuation in execution time is very similar to what we have previously observed.

Finally we present execution times for simulation run with 1728 replicas. As shown in Figure 4.14 (d), for the first 1728 Compute Units file staging times are substantially longer. In some cases, file staging takes more than 100 seconds. The main reason for this behavior is the setup of the shared file system on Stampede. For the rest of the Compute Units, file staging times are larger than before, but not by a significant amount. In addition, in this simulation run, there is a single Compute Unit (nr. 10003), for which execution time, is much larger than for all other units. This is not uncommon for QM/MM simulations, where some replicas might not even finish the certain number of simulation time-steps within simulation time limits.

We have quantified and measured fluctuation in execution times for REMD QM/MM simulations with various replica counts. Our experiments demonstrate, that for these simulations there is scope for improvement in resource utilization. We have also investigated how file staging times are varying as the simulation progresses and the number of replicas is increased. For all runs, file staging times for the first set of units are substantially longer than for the rest of the units. This is caused by the creation and staging of restraint files during the first simulation cycle and setup of the shared file system on Stampede. For subsequent simulation cycles, the longest file staging times were observed for the run with 1000 replicas (up to 40 seconds).

Now we compare resource utilization of synchronous and asynchronous RE patterns. We perform simulation runs with 215, 512, 1000 and 1728 replicas, while varying the number of simulation time-steps from 500 to 2000. In case of asynchronous RE, for all runs we set $W_r = 0.125$. Results of these runs are presented in Figure 4.15.

As shown in Figure 4.15, for both RE patterns, increase in the number of replicas results in decrease in resource utilization. This can be explained by the increase of the overheads and data times while increasing the number of replicas. In addition, for both RE patterns, highest utilization values are demonstrated while performing 2000 simulation time-steps between exchanges. Larger number of simulation time-steps results in longer MD times and a higher ratio of MD time to overheads. Consequently, resource utilization is higher.

From Figure 4.15 we can see that utilization numbers for both RE patterns are very similar. In addition, our results show that in terms of resource utilization neither of the two patterns is superior to another one. It was exected, that asynchronous RE pattern would demonstrate substantially better utilization numbers. There are multiple factors which can hinder the expected results. First, is the implementation of the asynchronous RE pattern. Current implementation

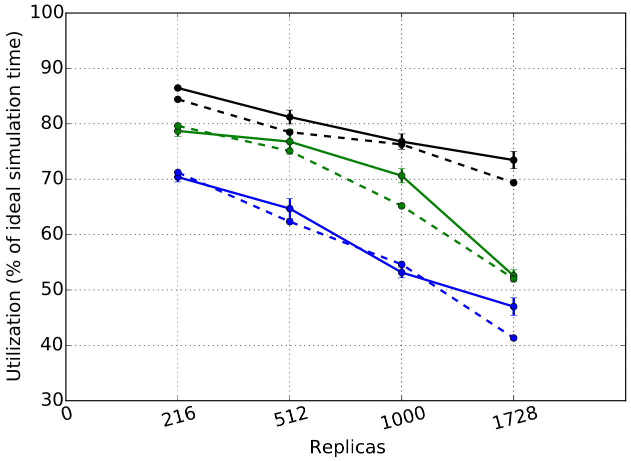


Figure 4.15: Resource utilization using synchronous and asynchronous RE patterns. Utilization is defined as a percentage of maximal (ideal) simulation time, which is obtained by running MD 100 % of the time. We use a single-core replicas for all runs and we always allocate enough CPU cores to run all the replicas concurrently. We vary the total number of replicas from 216 to 1728. Utilization for synchronous RE pattern is shown using solid lines, but for asynchronous RE pattern using dashed lines. We use black color for Runs with 2000 simulation time-steps between exchanges are shown in black color, runs with 1000 steps using green color and runs with 500 steps using blue.

must be revised and optimized for performance. Second, is insufficient mismatch in performance between individual replicas. Altering the simulation setup might result in substantially better utilization numbers for asynchronous RE pattern. Finally, for all asynchronous runs W_r was fixed at 0.125. It is possible that fine-tuning the value of W_r would allow to achieve a better resource utilization.

Sampling efficiency

To understand how sampling efficiency is affected by the choice of RE pattern (synchronous or asynchronous) we examine the number of accepted exchanges, the number of attempted exchanges and the ratio of accepted exchanges to the number of attempted exchanges for 1D temperature exchange REMD.

Here crosswalk is calculated for each replica and is defined as a transition from the current state to every other available state. In other words, a given replica has to exchange temperature with every other replica. A crosswalk takes at least N (number of replicas) exchanges.

As before, we use alanine dipeptide and for all runs we perform 6000 time-steps between exchanges. All runs are performed on SuperMIC HPC cluster. For all runs we set the range of temperatures from 300 to 460 degrees. We perform weak scaling runs, where each replica is run on a single CPU core and we use 40, 80 and 160 replicas. Each run takes 480 minutes of wall clock time. For each number of replicas we perform runs with W_r equal to 0.5, 0.25 and 0.125.

In this experiment, there is no fluctuation in execution times for MD tasks and asynchronous RE pattern doesn't have any advantage in terms of performed simulation time. Nethertheless, this simple simulation setup enables us to investigate how the choice of synchronization mechanism affects exchange metrics.

We first show crosswalk values both RE patterns (Figure 4.16). For synchronous pattern, for all simulation runs, increase in the number of replicas results in increase in the number of obtained crosswalks. This is expected, since the range of temperatures is the same (from 300 K to 460 K) for all runs, meaning that for runs with larger number of replicas difference in temperatures between neighboring replicas is smaller. As a result more exchanges are accepted and probability of obtaining a crosswalk is higher. Each time temperature difference between neighboring replicas is reduced by 50 % and we observe similar increase in the number of crosswalks.

As we can see from Figure 4.16, for asynchronous RE pattern change in the number of crosswalks largely depends on the value of W_r. While increasing the number of replicas we don't observe increase in the number of crosswalks. For all simulation runs with $W_r = 0.5$ number of crosswalks does not exceed seven.

For simulaion runs with $W_r = 0.25$ we observe increase in the number of crosswalks, similar to the synchronous RE pattern. Increase in the number of replicas, results in doubling in the number of crosswalks. Despite that, for all three runs, obtained number of crosswalks is less than for synchronous RE pattern. This can be explained by shorter total simulation time in comparison with synchronous RE pattern.

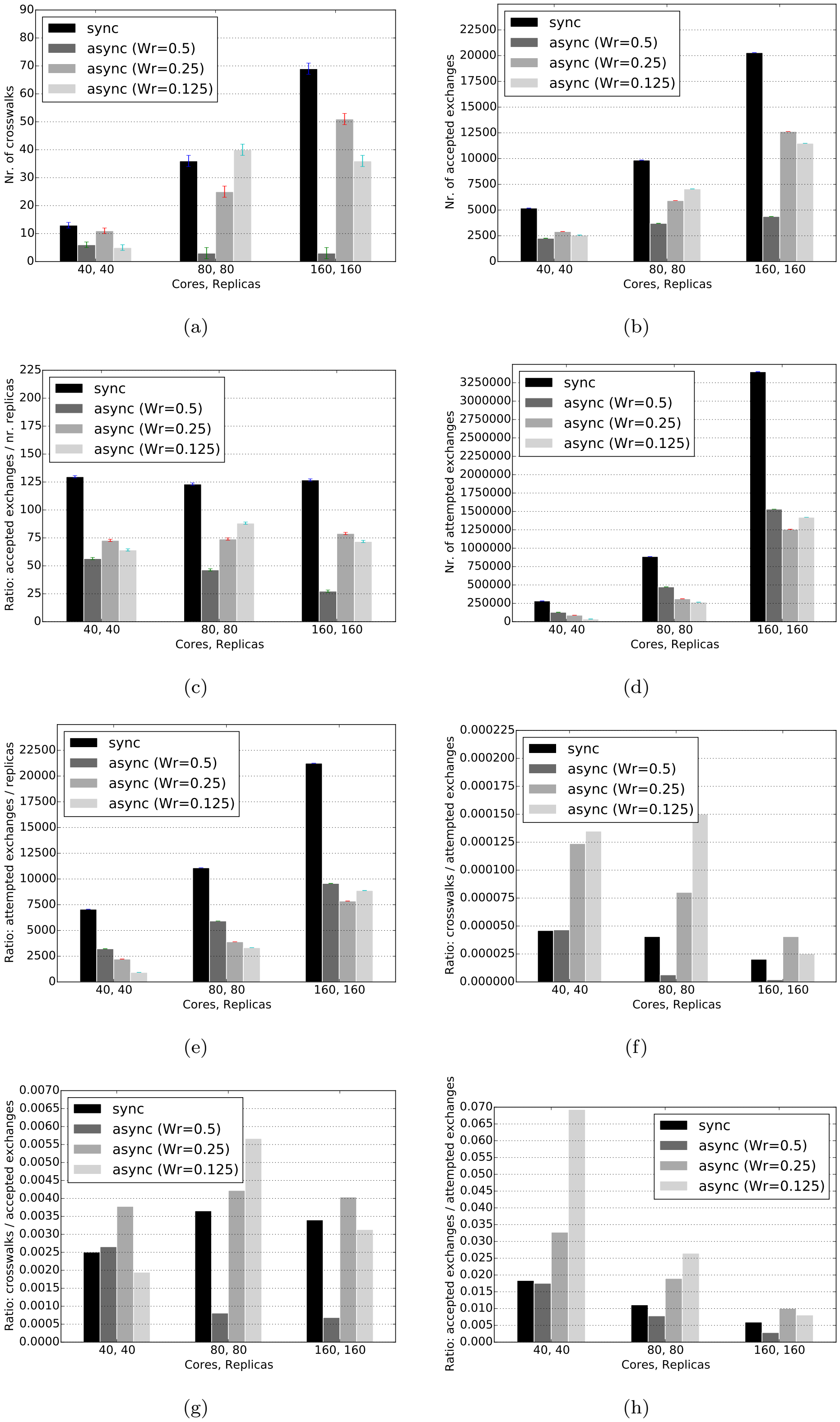
For simulation runs with $W_r = 0.125$ we first observe a substantial increase in the number of crosswalks, but further increase in the number of replicas results in decrease in the number of crosswalks. For the run with 80 replicas, number of crosswalks is larger than for a synchronous RE pattern. This is the only asynchronous run, for which was obtained larger the number of crosswalks, than for synchronous pattern.

Obtained results show, that asynchronous RE pattern is very sensitive to the value of W_r. For this experiment we recommend $W_r = 0.25$, since rate of change of the number of crosswalks with increase in replica count is very similar to synchronous RE pattern.

Next, we demonstrate the number of accepted exchanges for the same set of simulation runs (Figure 4.16 b). Fro synchronous RE pattern, doubling of the number of replicas results in the doubling in the number of accepted exchanges. For asynchronous RE pattern and $W_r = 0.5$

doubling the number of replicas results only in marginal increase in the number of accepted exchanges. This explains why the number of crosswalks is not increasing when we use $W_r = 0.5$. For simulation runs with $W_r = 0.25$, doubling the number of replicas results in the doubling in the number of accepted exchanges. Finaly, for simulation runs with $W_r = 0.125$, we observe an increase in the number of accepted exchanges, but this increase is not linear. It is worth pointing out that for the run with 80 replicas, the number of accepted exchanges for asynchronous pattern is smaller than for synchronous pattern by ~ 2600. At the same time, for this run number of crosswalks for asynchronous pattern is 40, which is greater than for synchronous pattern.

In Figure 4.16 (f) is shown the ratio of the number of crosswalks to the number of attempted exchanges. For synchronous RE pattern this ratio does not increase linearly and is in the range from 0.0025 to 0.0036. For asynchronous RE pattern this ration varies with W_r, but in most cases is higher than for synchronous RE pattern. From this we can conclude that using asynchronous RE pattern for M accepted exchanges number of crosswalks will be higher than for synchronous RE pattern.



(a)

(b)

(c)

(d)

(e)

(f)

(g)

(h)

Figure 4.16: 1D T-REMD with synchronous and asynchronous RE patterns. All runs performed on SuperMIC cluster. Runs with synchronous RE pattern (black bars), asynchronous RE pattern with $W_r = 0.5$ (dark grey bars), $W_r = 0.25$ (light grey bars) and $W_r = 0.125$ (extra light gray bars) **(a)** Number of crosswalks. **(b)** Number of accepted exchanges. **(c)** Ratio of accepted exchanges to the number of replicas **(d)** Number of attempted exchanges. **(e)** Ratio of attempted exchanges to the number of replicas. **(f)** Ratio of crosswalks to attempted exchanges. **(g)** Ratio of crosswalks to accepted exchanges. **(h)** Ratio of accepted exchanges to attempted exchanges.

Chapter 5
Results and Analysis

RepEx is designed to address functional, performance and usability requirements outlined in Chapter 3. We have demonstrated capabilities of RepEx and characterized its performance for 1D and 3D REMD simulations. We saw the range of exchange parameters that it supports and the flexibility in their ordering (e.g., TUU versus TSU). Furthermore we demonstrated support for Amber and NAMD MD engines with minimal conceptual or implementation changes.

We saw the ability to utilize different RE patterns and Execution Modes, which is made available by decoupling exchange parameters (T/U/S), dimensionality and algorithm (sync. vs. async.) from the resource management and execution details. As such, it is accurate to say that RepEx satisfies the functional and usability requirements.

Most of the recent developments of molecular simulation software packages for RE, are developed with aim to demonstrate ability to efficiently utilize computational resources at large scales and/or using large number of replicas. While these developments provide new scientific insights and are advancing the field of molecular simulation software, they often are tightly coupled with a particular setup.

First, software is developed for a particular HPC cluster and is targeting a specific hardware configuration. Often, this means that the code is not portable and it is unlikely that demonstrated results will be achieved on another cluster. While this is not a trivial endeavor, usefulness of this approach is limited. Our aim was to develop a software package which is capable of demonstrating high scalability and performance on variety of HPC clusters.

Second, development can be done with a particular RE algorithm in mind. Performance results in this case can be attributed to infrequent communication pattern or small amount of data, which is sent during the exchange. It is important to ensure, that REMD software not only supports different RE methods, but also demonstrates good scalability and performance results for these methods.

In Table 5.1, we have summarized the most important features of seven existing packages used for REMD, some of which are used by communities of hundreds, if not thousands of users.

	Amber	**Gromacs**	**LAMMPS**	**VCG async**	**CHARMM**	**Charm++/ NAMD MCA**	**RepEx**
Max replicas	$\sim$2750	$\sim$900	100	240	4096	2048	3584
Max CPU cores	$\sim$5490	$\sim$150000	76800	1920	131072	524288	13824
Fault tolerance	n/a	n/a	n/a	basic	n/a	n/a	basic
MD engines	Amber	Gromacs	LAMMPS	IMPACT	CHARMM	NAMD	Amber, NAMD
RE patterns	sync	sync	sync	sync, async	sync	sync	sync, async
Nr. Exec. Modes	1	1	1	2	1	1	2
Nr. dims	2	2	2	2	2	2	3
Exchange params	3	2	2	2	2	2	3

Table 5.1: Comparison of molecular simulation software packages with integrated REMD capability. We characterize each of the seven packages based on 8 features. For each feature we provide a numerical value or a name corresponding to that feature.

In this table, we have included three popular MD simulation engines, namely Amber, LAMMPS and Gromacs that have been extended to provide RE capabilities and four REMD packages that have been designed to be external to MD engines.

For each feature (except ”Fault tolerance”, ”MD engines” and ”RE patterns”) we have provided a numerical value of that feature. For ”MD engines” we provide actual engine name, for ”RE patterns” we specify supported patterns and for ”Fault tolerance” we indicate if there are existing mechanisms enabling graceful handling of failures.

As can be seen from Table 5.1, a majority of the packages are designed to address a subset of features we identified as necessary to be flexible and general purpose. Many packages have eschewed generality for performance. For example, Charm++/NAMD MCA package can utilize O(100,000) cores but does not provide flexible resource utilization nor asynchronous RE capabilities. On the other hand, VCG RE package is one of the few packages, which supports asynchronous RE but it has limited scalability (both in the number of replicas and CPU cores) and is tightly coupled to IMPACT which is not an open source MD engine. Similar to most other existing solutions, both VCG and Charm++/NAMD are limited in the number of exchange parameters as well as in flexibility in the ordering of exchange parameters.

Clearly a balance between performance and functional requirements needs to be maintained. On the evidence of Table 5.1 we believe that RepEx provides an optimal balance.

Chapter 6
Conclusion

We have designed and implemented a framework for REMD simulations. To overcome limitations of software packages with integrated REMD functionality, we base our design on the following concepts: **RE pattern**, which enables explicit selection of different synchronization options independently of the MD engine; **flexible execution mode**, which decouples the number of replicas from the number of computational resources (CPUs, GPUs, etc.); **pilot system**, simplifying management and execution of REMD simulation workflows on HPC clusters.

Separation of MD simulation engine from the implementation of RE algorithm simplifies integration of new MD simulation engines and facilitates reuse of RE patterns and Execution Modes.

RE patterns available in our framework, can be used interchangeably with any of the two MD engines. To the best of our knowledge none of the currently available REMD implementations have this capability.

Use of a pilot system as a runtime, not only enables separation of the algorithm and workload management from the resource management and runtime complexity, but also significantly reduces code complexity.

We believe that design decisions we have made, enable our framework to provide a high level of generality as evidenced by the Table 5.1. RepEx is designed to be a research vehicle for the domain scientists and to enable them to develop and test REMD methods unavailable in other software frameworks. Design and modularity of the code significantly lower the barrier for implementation of new features and reduce development time.

In Chapter 4 we have demonstrated scalability of the RepEx using both 1D and 3D REMD simulations on various HPC clusters, including petascale machines such as Stampede. While obtained performance results are not ideal, we believe that our framework is not limiting potential users in their ability to do research. In terms of performance, RepEx is bounded by the RADICAL-Pilot and without major design changes unlikely will demonstrate significantly better performance.

Main performance bottlenecks of RepEx are associated with file staging and task launching. To overcome the former, usage of a shared file system of the HPC cluster should be minimized. When possible, individual tasks should write to a local file system of compute nodes. This alone would greatly reduce file staging time. To further reduce file staging time, individual tasks should read data from files, stored in a local file system of a compute node, and communicate this data using a communication protocol.

Task launching delay is a bottleneck arising from the use of RADICAL-Pilot and it increases as a function of the number of tasks simultaneously submitted for execution. We have made an attempt to reduce the number of tasks by grouping multiple replicas into a single task, but this did not result in performance improvement. RADICAL-Pilot is a new pilot system and is under constant development. As evidenced by the results in section 4.1, during the development of the RepEx, RP's task launching delay was significantly reduced. We believe that in a near future this issue will be mitigated even further.

For the majority of the experiments, duration of the MD phase exceeded 2 minutes of wall clock time. While this duration is reasonable, especially for large proteins, there are many use cases, where frequency of exchanges is much higher and consequently duration of the MD phase is shorter. Undeniably, for these simulations overheads of the RepEx play a more significant role.

Chapter 7
Future Work

The most beneficial task in the context of this project would be developing a better understanding of the asynchronous RE. It is crucial to identify scenarios and use cases in which asynchronous RE is superior to synchronous RE. Also, it is important to provide a clear guidelines on how to maximize benefits arising from the usage of asynchronous RE pattern. In addition to utilization experiments presented in section 4.2.6, there is a number of scenarios which we haven't covered. For example, we can compare resource utilization using asynchronous and synchronous RE, while changing the ratio of replicas to CPU cores.

Another direction for future work is multi-cluster REMD simulations. Scenario when two or more HPC clusters are used for the same REMD simulation (with single MD engine) might not be of significant interest, since a single modern HPC cluster is capable to satisfy requirements of majority of users.

A more realistic usecase for multi-cluster simulations is multi-dimensional REMD simulations involving multiple MD engines. Due to licensing issues, many specialized MD engines are not available on shared HPC resources. These MD engines are available only on clusters, accessible to limited number of users. To simultaneously utilize these MD engines and computing power of large HPC clusters, we can use multi-cluster simulations.

A particularly suitable usecase for multi-cluster simulations is multi-level RE. In multi-level RE we simultaneously run two multi-dimensional REMD simulations. To steer simulations in the right direction, during the simulation we periodically exchange simulation data between two REMD simulations. These exchanges are less frequent than traditional RE exchanges and as a result make usage of multiple clusters beneficial for this simulation type.

In terms of features and functionality, there are multiple directions for the development of the RepEx framework:

- Internal implementation of the single point energy calculations for salt concentration exchange

- Addition of the new exchange parameters, such as pH, to enable support for new types of multi-dimensional REMD simulations.

- Support for other MD simulation engines
- Explicit support for the simulations concurrently utilizing multiple HPC resources
- Addition of interfaces exposing REMD capability of MD engines

Bibliography

[1] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten, “Scalable molecular dynamics with namd,” *J. Comput. Chem.*, vol. 26, no. 16, pp. 1781–1802, 2005.

[2] R. H. Swendsen and J.-S. Wang, “Replica monte carlo simulation of spin-glasses,” *Physical Review Letters*, vol. 57, no. 21, p. 2607, 1986.

[3] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, “Equation of state calculations by fast computing machines,” *The journal of chemical physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[4] Y. Sugita and Y. Okamoto, “Replica-exchange molecular dynamics method for protein folding,” *Chemical physics letters*, vol. 314, no. 1, pp. 141–151, 1999.

[5] H. Fukunishi, O. Watanabe, and S. Takada, “On the hamiltonian replica exchange method for efficient sampling of biomolecular systems: application to protein structure prediction,” *The Journal of chemical physics*, vol. 116, no. 20, pp. 9058–9067, 2002.

[6] Y. Meng and A. E. Roitberg, “Constant ph replica exchange molecular dynamics in biomolecules using a discrete protonation model,” *J. Chem. Theory Comput.*, vol. 6, no. 4, pp. 1401–1412, 2010.

[7] R. Salomon-Ferrer, D. A. Case, and R. C. Walker, “An overview of the amber biomolecular simulation package,” *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 3, no. 2, pp. 198–210, 2013. [Online]. Available: http://dx.doi.org/10.1002/wcms.1121

[8] S. Pronk, S. Páll, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. Kasson, D. van der Spoel *et al.*, “Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit,” *Bioinformatics*, vol. 29, no. 7, pp. 845–854, 2013.

[9] G. Groenhof, “Introduction to qm/mm simulations,” *Biomolecular simulations: methods and protocols*, pp. 43–66, 2013.

[10] "Repex on github," https://github.com/radical-cybertools/radical.repex, accessed: 2015-11-11.

[11] A. Merzky, M. Santcroos, M. Turilli, and S. Jha, "RADICAL-Pilot: Scalable Execution of Heterogeneous and Dynamic Workloads on Supercomputers," 2015, (under review) http://arxiv.org/abs/1512.08194.

[12] "Principles of modern molecular simulation methods," https://engineering.ucsb.edu/~shell/che210d/, accessed: 2016-10-10.

[13] H. Grubmüller, H. Heller, A. Windemuth, and K. Schulten, "Generalized verlet algorithm for efficient molecular dynamics simulations with long-range interactions," *Molecular Simulation*, vol. 6, no. 1-3, pp. 121–142, 1991.

[14] D. Fincham, "Leapfrog rotational algorithms," *Molecular Simulation*, vol. 8, no. 3-5, pp. 165–178, 1992.

[15] N. S. Martys and R. D. Mountain, "Velocity verlet algorithm for dissipative-particle-dynamics-based models of suspensions," *Physical Review E*, vol. 59, no. 3, p. 3733, 1999.

[16] B. P. C. Isard, *Theory and practice in replica-exchange molecular dynamics simulation.* ProQuest, 2008.

[17] Y. Sugita, A. Kitao, and Y. Okamoto, "Multidimensional replica-exchange method for free-energy calculations," *The Journal of Chemical Physics*, vol. 113, no. 15, pp. 6042–6051, 2000.

[18] D. J. Earl and M. W. Deem, "Parallel tempering: Theory, applications, and new perspectives," *Physical Chemistry Chemical Physics*, vol. 7, no. 23, pp. 3910–3916, 2005.

[19] N. Rathore, M. Chopra, and J. J. de Pablo, "Optimal allocation of replicas in parallel tempering simulations," *The Journal of chemical physics*, vol. 122, no. 2, p. 024111, 2005.

[20] A. Kone and D. A. Kofke, "Selection of temperature intervals for parallel-tempering simulations," *The Journal of chemical physics*, vol. 122, no. 20, p. 206101, 2005.

[21] A. Warshel and M. Levitt, "Theoretical studies of enzymic reactions: dielectric, electrostatic and steric stabilization of the carbonium ion in the reaction of lysozyme," *Journal of molecular biology*, vol. 103, no. 2, pp. 227–249, 1976.

[22] H. Yu and L. Kale, "Scalable molecular dynamics with namd on the ibm blue gene/l system," 2008.

[23] W. Jiang, Y. Luo, L. Maragliano, and B. Roux, “Calculation of free energy landscape in multi-dimensions with Hamiltonian-exchange umbrella sampling on petascale supercomputer,” *J. Chem. Theory Comput.*, vol. 8, pp. 4672–4680, 2012.

[24] B. R. Brooks, C. L. Brooks, A. D. MacKerell, L. Nilsson, R. J. Petrella, B. Roux, Y. Won, G. Archontis, C. Bartels, S. Boresch *et al.*, “Charmm: the biomolecular simulation program,” *J. Comput. Chem.*, vol. 30, no. 10, pp. 1545–1614, 2009.

[25] W. Jiang, J. C. Phillips, L. Huang, M. Fajer, Y. Meng, J. C. Gumbart, Y. Luo, K. Schulten, and B. Roux, “Generalized scalable multiple copy algorithms for molecular dynamics simulations in namd,” *Computer physics communications*, vol. 185, no. 3, pp. 908–916, 2014.

[26] W. Shalongo, L. Dugad, and E. Stellwagen, “Distribution of helicity within the model peptide acetyl (aaqaa) 3amide,” *Journal of the American Chemical Society*, vol. 116, no. 18, pp. 8288–8293, 1994.

[27] B. K. Radak, M. Romanus, E. Gallicchio, T.-S. Lee, O. Weidner, N.-J. Deng, P. He, W. Dai, D. M. York, R. M. Levy, and S. Jha, “A Framework for Flexible and Scalable Replica-Exchange on Production Distributed CI,” ser. XSEDE ’13, 2013, pp. 26:1–26:8.

[28] B. K. Radak, M. Romanus, T.-S. Lee, H. Chen, M. Huang, A. Treikalis, V. Balasubramanian, S. Jha, and D. M. York, “Characterization of the Three-Dimensional Free Energy Manifold for the Uracil Ribonucleoside from Asynchronous Replica Exchange Simulations,” *Journal of Chemical Theory and Computation*, vol. 11, no. 2, pp. 373–377, 2015, http://dx.doi.org/10.1021/ct500776j. [Online]. Available: http://dx.doi.org/10.1021/ct500776j

[29] J. L. Banks, H. S. Beard, Y. Cao, A. E. Cho, W. Damm, R. Farid, A. K. Felts, T. A. Halgren, D. T. Mainz, J. R. Maple *et al.*, “Integrated modeling program, applied chemical theory (impact),” *J. Comput. Chem.*, vol. 26, no. 16, pp. 1752–1780, 2005.

[30] J. Xia, W. F. Flynn, E. Gallicchio, B. W. Zhang, P. He, Z. Tan, and R. M. Levy, “Large-scale asynchronous and distributed multidimensional replica exchange molecular simulations and efficiency analysis,” *J. Comput. Chem.*, vol. 36, pp. 1772–1785, 2015.

[31] M. P. Allen and D. J. Tildesley, *Computer simulation of liquids.* Oxford university press, 1989.

[32] S. Kullback and R. A. Leibler, "On information and sufficiency," *The annals of mathematical statistics*, pp. 79–86, 1951.

[33] E. Gallicchio, M. Lapelosa, and R. M. Levy, "Binding energy distribution analysis method (bedam) for estimation of protein- ligand binding affinities," *J. Chem. Theory Comput.*, vol. 6, no. 9, pp. 2961–2977, 2010.

[34] C. Bergonzo, N. M. Henriksen, D. R. Roe, J. M. Swails, A. E. Roitberg, and T. E. Cheatham III, "Multidimensional replica exchange molecular dynamics yields a converged ensemble of an RNA tetranucleotide," *J. Chem. Theory Comput.*, vol. 10, pp. 492–499, 2014.

[35] M. T. Panteva, T. Dissanayake, H. Chen, B. K. Radak, E. R. Kuechler, G. M. Giambaşu, T.-S. Lee, and D. M. York, *Multiscale Methods for Computational RNA Enzymology.* Elsevier, 2015, ch. 14.

[36] B. Ensing, M. De Vivo, Z. Liu, P. Moore, and M. L. Klein, "Metadynamics as a tool for exploring free energy landscapes of chemical reactions," *Acc. Chem. Res.*, vol. 39, no. 2, pp. 73–81, 2006.

[37] E. Vanden-Eijnden, "Some recent techniques for free energy calculations," *J. Comput. Chem.*, vol. 30, no. 11, pp. 1737–1747, 2009. [Online]. Available: http://dx.doi.org/10.1002/jcc.21332

[38] T. Dissanayake, J. M. Swails, M. E. Harris, A. E. Roitberg, and D. M. York, "Interpretation of pH-Activity Profiles for Acid-Base Catalysis from Molecular Simulations," *Biochemistry*, vol. 54, pp. 1307–1313, 2015.

[39] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao, "Application-level scheduling on distributed heterogeneous networks," in *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on.* IEEE, 1996, pp. 39–39.

[40] J. T. Mościcki, "Diane-distributed analysis environment for grid-enabled simulation and analysis of physics data," in *Nuclear Science Symposium Conference Record, 2003 IEEE*, vol. 3. IEEE, 2003, pp. 1617–1620.

[41] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a fast and light-weight task execution framework," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing.* ACM, 2007, p. 43.

[42] A. Casajus, R. Graciani, S. Paterson, A. Tsaregorodtsev *et al.*, “Dirac pilot framework and the dirac workload management system,” in *Journal of Physics: Conference Series*, vol. 219, no. 6. IOP Publishing, 2010, p. 062049.

[43] A. Luckow, L. Lacinski, and S. Jha, “Saga bigjob: An extensible and interoperable pilot-job abstraction for distributed applications and systems,” in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on.* IEEE, 2010, pp. 135–144.

[44] D. Thain, T. Tannenbaum, and M. Livny, “Distributed computing in practice: the condor experience,” *Concurrency and computation: practice and experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[45] I. Sfiligoi, “glideinwmsa generic pilot-based workload management system,” in *Journal of Physics: Conference Series*, vol. 119, no. 6. IOP Publishing, 2008, p. 062044.

[46] P.-H. Chiu and M. Potekhin, “Pilot factory–a condor-based system for scalable pilot job generation in the panda wms framework,” in *Journal of Physics: Conference Series*, vol. 219, no. 6. IOP Publishing, 2010, p. 062041.

[47] J. Gyllenhaal, T. Gamblin, A. Bertsch, and R. Musselman, “Enabling high job throughput for uncertainty quantification on bg/q,” *ser. IBM HPC Systems Scientific Computing User Group (SCICOMP)*, 2014.

[48] M. Rynge, S. Callaghan, E. Deelman, G. Juve, G. Mehta, K. Vahi, and P. J. Maechling, “Enabling large-scale scientific workflows on petascale resources using mpi master/worker,” in *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond.* ACM, 2012, p. 49.

[49] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny *et al.*, “Pegasus, a workflow management system for science automation,” *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[50] A. Luckow, M. Santcroos, A. Merzky, O. Weidner, P. Mantha, and S. Jha, “P: a model of pilot-abstractions,” in *E-Science (e-Science), 2012 IEEE 8th International Conference on.* IEEE, 2012, pp. 1–10.

[51] “Pilot api,” http://radicalpilot.readthedocs.org/, accessed: 2015-11-11.

[52] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. Von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf, "Saga: A simple api for grid applications. high-level application programming on the grid," *Computational Methods in Science and Technology*, vol. 12, no. 1, pp. 7–20, 2006.

[53] J. B. Swadling, D. W. Wright, J. L. Suter, and P. V. Coveney, "Structure, dynamics, and function of the hammerhead ribozyme in bulk water and at a clay mineral surface from replica exchange molecular dynamics," *Langmuir*, vol. 31, no. 8, pp. 2493–2501, 2015.

[54] M. Turilli, M. Santcroos, and S. Jha, "A Comprehensive Perspective on Pilot-Jobs," 2015, http://arxiv.org/abs/1508.04180.

[55] "Extreme science and engineering discovery environment," https://www.xsede.org/resources/overview, accessed: 2015-11-11.

[56] T.-S. Lee, B. K. Radak, A. Pabis, and D. M. York, "A new maximum likelihood approach for free energy profile construction from molecular simulations," *J. Chem. Theory Comput.*, vol. 9, pp. 153–164, 2013.

[57] T.-S. Lee, B. K. Radak, M. Huang, K.-Y. Wong, and D. M. York, "Roadmaps through free energy landscapes calculated using the multidimensional vFEP approach," *J. Chem. Theory Comput.*, vol. 10, pp. 24–34, 2014.

[58] W. Sinko, Y. Miao, C. A. de Oliveira, and J. A. McCammon, "Population Based Reweighting of Scaled Molecular Dynamics," *J. Phys. Chem. B*, vol. 117, no. 42, pp. 12 759–12 768, 2013.

[59] X. Peng, Y. Zhang, H. Chu, and G. Li, "Free energy simulations with the AMoeba polarizable force field and metadynamics on GPU platform." *J. Comput. Chem.*, vol. 37, no. 6, pp. 614–622, Mar. 2016.

[60] T. Takekiyo, T. Imai, and Y. Taniguchi, "Temperature and PressureEffects on Conformational Equilibria of Alanine Dipeptide in Aqueous Solution," *Biopolymers*, vol. 73, pp. 283–290, 2004.

[61] "Repex experiments," https://github.com/radical-cybertools/radical.repex/blob/master/EXPERIMENTS.md, accessed: 2015-11-11.

Appendix A
Appendix

A.1 Abbreviations

- **Target/Remote system** - HPC cluster which user can access via communication network and which is used to execute simulation tasks
- **Local system** - laptop or desktop to which user has physical access
- **Pilot** - task-container launched on compute nodes of HPC cluster and executing tasks according to their description
- **Simulation phase** - first phase of REMD simulation, during which replicas propagate MD simulation. Simulation phase can be defined as a number of time-steps, e.g. 2 ps or as a fixed real time period, e.g. 1 minute
- **Exchange phase** - second phase of REMD simulation, during which replicas exchange thermodynamic parameters.
- **Simulation Cycle** - simulation phase followed by exchange phase for 1D-REMD and **M** simulation phase, exchange phase pairs for M-REMD
- **Task** - a unit of execution. There is a one-to-one mapping between tasks and RP's Compute Units.
- **Total Simulation Time** - total time required to perform **N** Simulation Cycles
- **RE Pattern** - REMD simulation type definition based on synchronization mode between simulation and exchange phases
- **Execution Mode** - a set of execution options decoupling simulation requirements from the resource availability and enabling flexible usage of allocated CPU's on HPC resources
- **Bulk submission** - all tasks of a current step are submitted at once (as a Python list) to RADICAL-Pilot for execution - `unit_manager.submit_units(tasks)`

- **Sequential submission** - tasks of a current step are submitted sequentially, one by one to RADICAL-Pilot for execution
- **RP overhead** - task launching delay on a target resource and communication delay
- **RepEx overhead** - task preparation time, e.g. association of data and parameters with a task and translation of task description to RP task description (creation of Compute Units); time to perform RepEx method calls inside simulation loop on a local system
- **Synchronization pattern** - specifies constraints for the execution order of the tasks in RE

A.2 I/O patterns for different types of REMD simulations with Amber MD engine

In this section we present and discuss file movement patterns for three exchange types supported for the use with Amber MD engine: Temperature exchange, Umbrella exchange and Salt Concentration exchange. Understanding the file movement requirements for a particular exchange type is a fundamental step in enabling efficient and scalable implementation. It is important to point out, that despite being similar, file movement patterns for different exchange types have some important differences. We only discuss file movement patterns for one-dimensional simulation types, since file movement in any multi-dimensional simulation is a sequential concatenation of one-dimensional patterns.

For all three file movement patterns simulation is decomposed into five stages - initialization, MD simulation, individual exchange, global exchange and post processing. Intuitively, initialization is performed on a local system. At this stage simulation input files are identified according to the user input. MD simulation stage is done on a remote system. This stage requires as an input a set of input files transferred from a local system and other files which are generated dynamically on a remote system, just before the MD simulation. Individual exchange is a part of the exchange phase, which can be performed concurrently by multiple processing units. A subset of MD simulation output files is used as an input for individual exchange stage. Contents of this subset depend on the exchange type. For all three exchange types the output of the individual exchange stage is a single file - `matrix_column.dat`. As the name suggests, each of these files contains a single column of the Gibbs ”swap” matrix, which is used during the exchange procedure. For global exchange stage are required all `matrix_column.dat` files, which are used to compose a swap matrix. Global exchange stage is responsible for the majority of exchange calculations. This stage as an output produces a single file - `exchange_pairs.dat`. In this file are provided id's of the pairs of replicas, which must exchange their respective parameters. Finally, `exchange_pairs.dat` file is transferred back to the local system, where actual exchange of parameters is performed enabling generation of the next set of Compute Units for MD simulation.

We first present a file movement pattern for Temperature exchange, then for Umbrella exchange and finally for Salt Concentration exchange.

A.2.1 Temperature exchange

File movement pattern for temperature exchange simulation is depicted in Figure A.2.1. Initialization stage requires a user to specify at least three shared input files: parameters file (.prmtop), input file template (.mdin) and coordinates file (.inpcrd). These files are required as in input of the MD simulation stage. In addition to these three files, just before the MD simulation stage is generated a simulation input file `init_input.mdin`. Starting with second simulation cycle, instead of initial input coordinates file, MD simulation stage requires as input a restart file of the previous cycle - `restart_id_c.rst`. After the MD simulation stage are generated four files: `output.mdout`, `trajectory.mdcrd`, `restart_id_c.rst` (is used as an input for the next cycle) and history.mdinfo. The latter file is required as an input of the individual exchange stage. It is important to note that each individual exchange task requires a history.mdinfo of each replica in it's group. That is if we perform an individual exchange calculations for replica with $id = 1$ and the group to which this replica belongs is $G = \{1, 2, 3, 4\}$, then history.mdinfo files of replicas 1, 2 and 4 are required as input for individual exchange of replica 1. As mentioned previously individual exchange for each replica produces a `matrix_column.dat` file. Which is required as input for the global exchange stage. The global exchange stage, in turn produces a single `exchange_pairs.dat` file, which is used for local post-processing.

A.2.2 Umbrella exchange

In Figure A.2.2 is presented a file movement pattern for umbrella exchange simulations. In addition to parameters file (.prmtop), input file template (.mdin) and coordinates file (.inpcrd), initialization stage requires user to specify a restraints template file (.RST). Before MD simulation stage, this file is used to generate an individual restraints (`restraint.RST`) file for each replica. In addition to the files specified for MD simulation stage of the temperature exchange file movement pattern, umbrella exchange requires two extra files: restraints template file `rstr_template.RST` and `restraint.RST`. Similarly as for temperature exchange, first simulation cycle of MD stage requires `init_coors.inpcrd` file, but subsequent cycles must use `restart_id_c.rst` of the previous cycle. Individual exchange stage, as an input takes two files: `history.mdinfo` file for current replica and `restraint.RST` for all replicas in current group. As an output of the individual exchange stage is generated a single `matrix_column.dat` file. File requirements of global exchange and post-processing stage are identical with temperature exchange pattern described above.

A.2.3 Salt Concentration exchange

I/O pattern for salt concentration exchange is shown in Figure A.2.3. As we can see, I/O for initialization stage and MD simulation stage is exactly the same as for temperature exchange. For salt concentration exchange, individual exchange is more involved than for other REMD types and consists of three steps. First is `pre_exec` step, where `inp_energy.mdin` files for Amber's group execution are generated. Next step is actual execution, where single point energies for replicas in current group are calculated. This step produces `inp_energy.mdinfo` files as an output. The final step of the individual exchange stage is `post_exec`. In this step, single point energies from the `inp_energy.mdinfo` files for all replicas in the current group are read-in and is produced a `matrix_column.dat` file. Global exchange and post processing stages are exactly the same as for other REMD types.

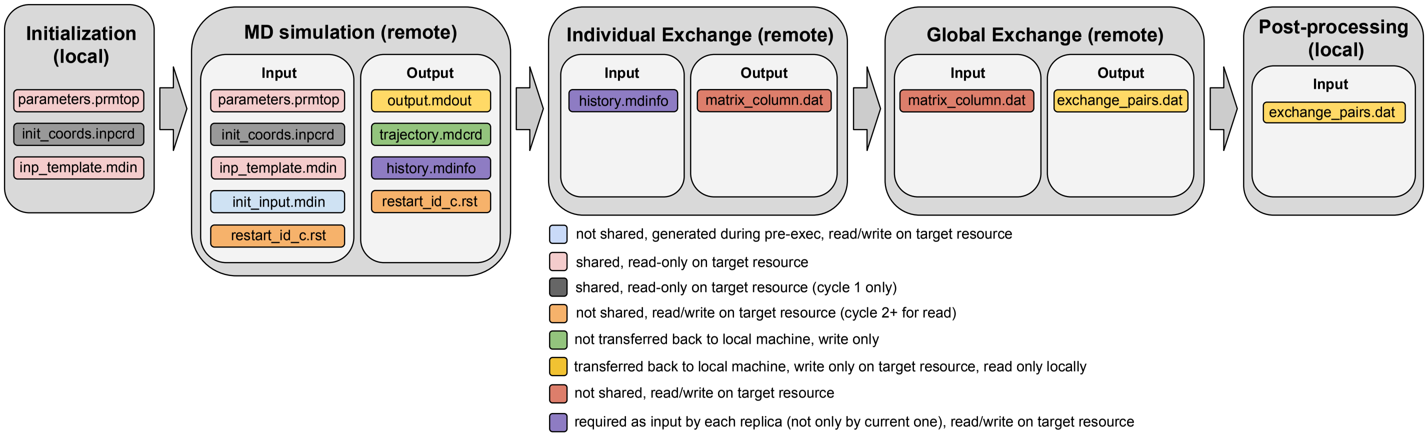

Initialization (local)
parameters.prmtop
init_coords.inpcrd
inp_template.mdin
MD simulation (remote)
Input
parameters.prmtop
init_coords.inpcrd
inp_template.mdin
init_input.mdin
restart_id_c.rst
Output
output.mdout
trajectory.mdcrd
history.mdinfo
restart_id_c.rst
Individual Exchange (remote)
Input
history.mdinfo
Output
matrix_column.dat
Global Exchange (remote)
Input
matrix_column.dat
Output
exchange_pairs.dat
Post-processing (local)
Input
exchange_pairs.dat
not shared, generated during pre-exec, read/write on target resource
shared, read-only on target resource
shared, read-only on target resource (cycle 1 only)
not shared, read/write on target resource (cycle 2+ for read)
not transferred back to local machine, write only
transferred back to local machine, write only on target resource, read only locally
not shared, read/write on target resource
required as input by each replica (not only by current one), read/write on target resource

Figure A.1: File movement pattern for Amber MD engine: Temperature exchange.

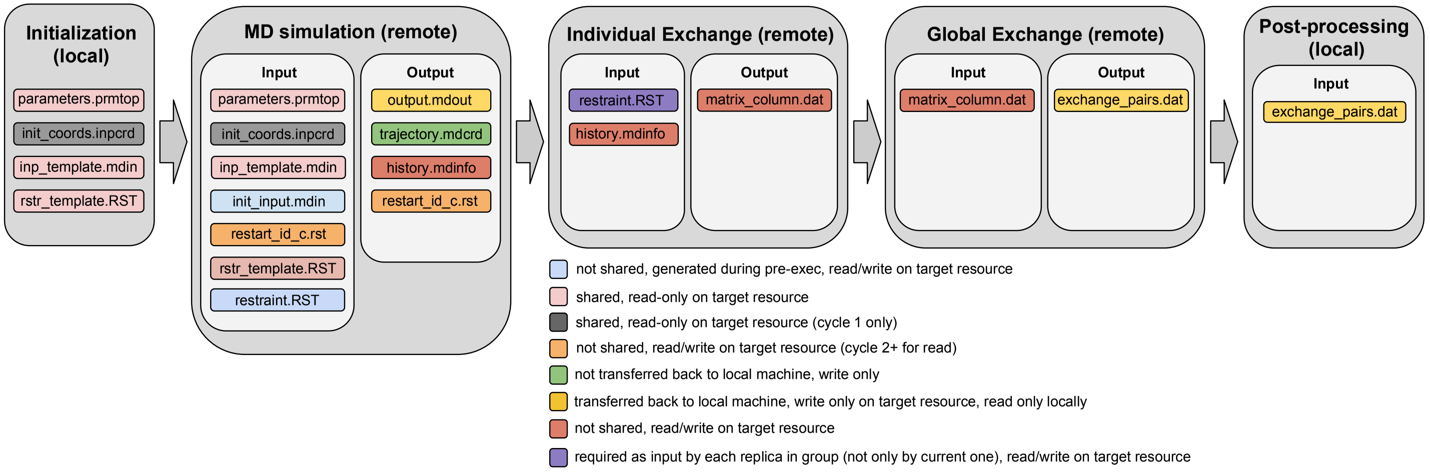

Initialization
(local)
parameters.prmtop
init_coords.inpcrd
inp_template.mdin
rstr_template.RST
MD simulation (remote)
Input
parameters.prmtop
init_coords.inpcrd
inp_template.mdin
init_input.mdin
restart_id_c.rst
rstr_template.RST
restraint.RST
Output
output.mdout
trajectory.mdcrd
history.mdinfo
restart_id_c.rst
Individual Exchange (remote)
Input
restraint.RST
history.mdinfo
Output
matrix_column.dat
Global Exchange (remote)
Input
matrix_column.dat
Output
exchange_pairs.dat
Post-processing
(local)
Input
exchange_pairs.dat
not shared, generated during pre-exec, read/write on target resource
shared, read-only on target resource
shared, read-only on target resource (cycle 1 only)
not shared, read/write on target resource (cycle 2+ for read)
not transferred back to local machine, write only
transferred back to local machine, write only on target resource, read only locally
not shared, read/write on target resource
required as input by each replica in group (not only by current one), read/write on target resource

Figure A.2: File movement pattern for Amber MD engine: Umbrella exchange.

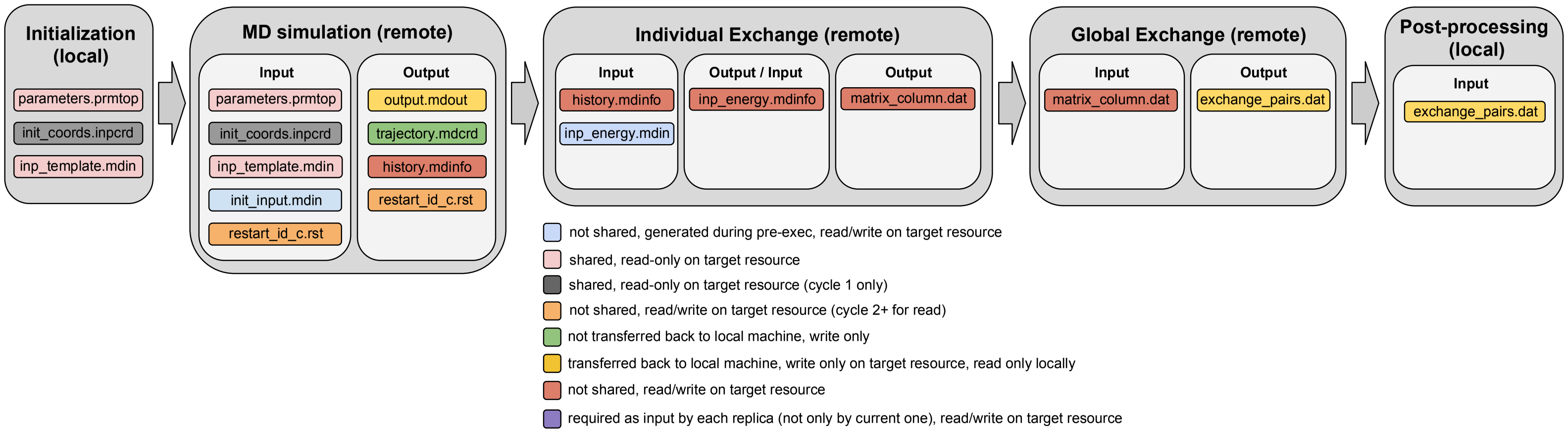

Initialization (local)
parameters.prmtop
init_coords.inpcrd
inp_template.mdin
MD simulation (remote)
Input
parameters.prmtop
init_coords.inpcrd
inp_template.mdin
init_input.mdin
restart_id_c.rst
Output
output.mdout
trajectory.mdcrd
history.mdinfo
restart_id_c.rst
Individual Exchange (remote)
Input
history.mdinfo
inp_energy.mdin
Output / Input
inp_energy.mdinfo
Output
matrix_column.dat
Global Exchange (remote)
Input
matrix_column.dat
Output
exchange_pairs.dat
Post-processing (local)
Input
exchange_pairs.dat
not shared, generated during pre-exec, read/write on target resource
shared, read-only on target resource
shared, read-only on target resource (cycle 1 only)
not shared, read/write on target resource (cycle 2+ for read)
not transferred back to local machine, write only
transferred back to local machine, write only on target resource, read only locally
not shared, read/write on target resource
required as input by each replica (not only by current one), read/write on target resource

Figure A.3: File movement pattern for Amber MD engine: Salt Concentration exchange.