TOWARDS FRAMEWORKS FOR LARGE SCALE ENSEMBLE-BASED EXECUTION PATTERNS

by

VIVEKANANDAN BALASUBRAMANIAN

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, 2015

ABSTRACT OF THE THESIS

Towards Frameworks for Large Scale Ensemble-based Execution Patterns

by Vivekanandan Balasubramanian

Thesis Director: Dr. Shantenu Jha

A major challenge in the field of chemical sciences is to bridge the gap between the ability to study matter at an atomic scale and to predict how these details behave and impact at a macroscopic scale. Molecular Dynamics (MD) simulations are a powerful tool for the study of macromolecular systems as they provide the ability to compute thermodynamic and kinetic parameters accurately.

In order for MD simulations to be effective, they must adequately sample, e.g., efficiently and accurately sample all conformational space for a molecule. A long standing debate in the MD community has been around approaches to effective sampling: for a given amount of compute time, which is likely to guarantee better sampling: a single long-running simulation or multiple smaller simulations? In this project, we provide support for the scenario when multiple small simulations are to be used.

Another important requirement faced by MD community is the need for iterative simulation and analysis stages, from which data can be extracted and, based on probability densities and weights, new set of trajectories can be generated. These scenarios result in the following computational challenges: (i) Executing large number of tasks concurrently, (ii) Support for heterogeneous resource, (iii) Effective data movement that is correlated with stage transitions.

To address these requirements, we developed the Ensemble MD Toolkit (EnMDTK). The EnMDTK has three primary design features: (1) Fundamental support for multiple concurrent simulations, (2) Support for different ensemble-based execution patterns, and (3) Execution Plugins which abstract, from the user the challenges /difficulties of managing the execution of these patterns on heterogeneous systems. The toolkit enables scientists to easily and scalably develop their own applications using pre-determined patterns supported by the EnMDTK.

Our contribution to this project is the development, testing and documentation of the specific iterative Simulation-Analysis pattern of EnMDTK. The development required extensive study and testing of the underlying RADICAL-Pilot framework in order to use it in the most effective form. Knowledge of the individual kernels, working and their specific dependencies were required for their deployment on the various resources. Testing revealed certain optimizations that could be done with the data movement. Performance characterization of the final toolkit was also done.

Acknowledgements

First and foremost I would like to thank my advisor Dr. Shantenu Jha for giving me the opportunity to work on this project. I am also grateful to Dr. Jha for the encouragement, motivation and guidance that helped me immensely in my thesis. Next, I would like to thank Ole Weidner and all members of the RADICAL team for their support and involvement in the development of this project. I would also like to thank Dr. Jordane Preto and Dr. Cecilia Clementi from Rice University, Dr. Ardita Shkurti and Dr. Charles Laughton from University of Nottingham, Iain Bethune and Dr. Elena Breitmoser from University of Edinburgh for their active involvement throughout the development and testing for this project. I thank XSEDE and EPCC for resources I used. I would like to thank my friends and family for all the love, support and encouragement.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The achievement of the sampling necessary to predict macroscopic properties from Molecular Dynamics (MD) simulations of large systems remains an enormous challenge. Uniform sampling has been used for long, as a method with which one can generate MD trajectories. Unfortunately, these methods leave many relevant regions of space unexplored. This stems from the separation of high-probability metastable regions by low-probability transition regions [1]. Enhanced Sampling methods have shown success to efficiently and accurately sample all conformational space for a molecule. In this method, interleaving large ensembles of short molecular dynamics simulations with novel analysis tools to direct the sampling on-the-fly, the sampling of macromolecular systems can be made to speed up[14].

Traditionally, scientists use bash scripts to submit their workloads as "jobs" onto a supercomputer or grid queue. The workload may be divided into one or more jobs based on the resource, time constraints, allocation constraints, etc. This generally leads to multiple drawbacks during implementation. One major drawback is that of queue waiting time. Scientists generally require many jobs to be executed in a sequence to produce useful data. Most of the times, these jobs have the same executable but work on different input data. Each of these jobs face a queue waiting time on the resource and the effective time to completion increases by a large factor. There are multiple disadvantages to using this method. A bash script creates a single point of failure for the entire workload. Many times, the conventional method is to launch the jobs as parallel MPI threads; if one of the threads fails, the entire workload fails. The same

holds true for serial If one decides to use another machine for the same purposes, it requires heavy changes to the script since every bash script is specific to the particular resource. Another drawback is inefficient resource utilization, due to various I/O and execution/scheduling policies, there might not be 100% utilization of the resources. Along with these execution level issues with this approach, there is also the requirement that these scientists have knowledge about scripting interfaces and the specifics of each resource they choose as the target machine. These problems discussed above serve as the motivation for this dissertation.

The above mentioned drawbacks give us a clear picture of our requirements, i.e., efficient and concurrent job execution methods, abstractions to execute independent jobs, resource level abstractions that provide interoperability, effective I/O transfer methods. To tackle this High Performance Distributed Computing (HPDC) problem, we develop EnsembleMD Toolkit (EnMDTK) that uses a Pilot Job framework as the underlying execution mechanism. A Pilot Job framework provides functionality to execute large number of computational tasks, concurrently, on a machine. Pilot jobs provide a much needed decoupling of the resource management from the workload execution. Additionally, we require that this Pilot framework be interoperable over multiple heterogeneous resources and hence we opt to use RADICAL Pilot[11], which uses an interoperability layer SAGA[8], to solve this problem. RADICAL Pilot provides I/O transfer methods to transfer data at the level of tasks. In EnMDTK, we present an easy to interface to the user where we scrape of the need to know resource specific or RADICAL Pilot specific details. The user can, therefore, concentrate solely on the science problem and not its deployment. More details about EnMDTK, its components and its advantages are presented in the following Chapters.

1.2 Objectives

The main objective of this thesis is to develop a framework to enable large scale execution of simulation and analysis patterns and , in the process, enable the user to develop applications using this framework by providing the bare building blocks. In this project,

we would be supporting two simulation tools - GROMACS[16], AMBER[15] and two analysis tools - LSDMap[14] and CoCo[9]. We aim to develop production grade software targeted to support ensemble-based execution patterns with the following capabilities :-

- Enable Large Scale execution of ensemble based tasks
- Provide interoperability to enable execution over multiple heterogeneous distributed computing infrastructure
- Enable the users to develop their own application by providing them with the building blocks
- Provide resource and tool level abstractions to minimize user application development time
- Provide effective data transfer methods
- Analyze the performance and scalability of the framework

1.3 Overview

The motivation and objectives for this thesis are discussed in Chapter 1. We compare conventional execution methods with Pilot Job frameworks, more specifically RADICAL Pilot, in Chapter 2. We also discuss the architecture and operation of RADICAL Pilot with a simple example. In Chapter 3, we develop an initial prototype by directly integrating the Simulation and Analysis tools with RADICAL Pilot. We focus on the specific "Pattern" at hand and how the prototype was architected around it. Once the similarities and requirements for different patterns are recognized, we develop and discuss the advanced prototype in chapter 4. We discuss the motivation, architecture and the API of EnsembleMD Toolkit. In chapter 5, we conduct strong and weak scaling tests using the toolkit. We conclude in chapter 6 with the learnings during this thesis and discuss on the future scope of the toolkit.

Chapter 2

Task Level Parallelism: Pilot-Abstraction

2.1 Conventional Methods

Distributed Computing Infrastructure (DCI) comprises of set of resources that changes in load, capability and availability, as opposed to the static utilization model of regular HPC clusters. Chemists, physicists and other scientists require the use of these DCI at a large scale to execute their experiments and analysis. It is imperative, therefore, to utilize DCIs efficiently for such applications.

There are many tools that utilize distributed computing infrastructures (DCI) successfully for a number of applications. These tools, although successful, do not use the DCI effectively and have a very complex design. These tools are not flexible with the different dimensions of scalability (scale-out, scale-across, etc.) or level of robustness required by the application. The tool might be usable at small scales and in controlled experiment, but they often fail to support the application at large scales. Applications using these tools rely heavily on the tool's ability for fault tolerance and security, but most of the tools operate under the assumption of no faults during deployment or execution of the application, thereby following a fail-stop behaviour. Most methods represent performance as measured only by peak performance. While peak performance is a significant metric in high-performance computing, there are additional metrics that could be considered as well. For example, in many applications that represent a continuous workload, one key metric is the throughput of the jobs during execution, i.e. the number of jobs launched per second. Also, most of the tools assume that there is only one application in the system. This assumption is generally not valid as any DCI is shared by multiple users at the same time and thus may require replication of data and

re-iteration of jobs in order to avoid the effects of loss of data during possible conflicts.

General solutions to the above concerns, consist of highly customized designs that are specific to particular resources. These solutions often follow a strict form. This makes addition of feature components, which might be required in the future, difficult to implement. One of the other major constraints is that many of these solutions include system level details, thereby, lacking any interoperability and reuse of code. Experiments on a different system would require heavy reprogramming and testing.

The toolkit being developed in this thesis should, therefore, aim to address the requirements of (i) extensibility, (ii) interoperability, (iii) scalability, (iv) support for different application designs (v) reuse of code. The key requirement here is that of abstractions at different levels of the hierarchy. Abstractions can be present in different levels - resource abstraction, task level abstractions, application level abstractions for extensibility, etc.

In this regard, Pilot Job (PJ) frameworks have been commonly used. A Pilot Job provides the ability to use a container job to execute a dynamically determined set of computational tasks. They provide a simple approach to decouple workload management from resource assignment, thus, effectively abstracting the dynamic execution and resource utilization[11]. Higher level abstractions also allow the representation of data dependencies in a logical form, rather than in terms of files. There are several PJ frameworks that are used for various applications, however, many of the Pilot-Job frameworks are all heavily customized and often tightly coupled to a specific infrastructure, and not extensible or usable across different systems, e.g. there is no such "unifying" and "extensible" tools that supports a range of application types and characteristics[11]. In this project, I have used RADICAL Pilot (RP) which uses an interoperability layer, SAGA, which comprises of set of adaptors and thus offers resource abstraction to the above RP layer.

2.2 RADICAL-Pilot

A Pilot-Job can be thought of as a container job for many sub-jobs (tasks), with sophisticated management to coordinate the launch and interaction of actual computational tasks within the container[10, 11]. A Pilot-Job acquires the resources necessary to execute the sub-jobs (thus, acquires the resources required to run the sub-jobs, rather than just one sub-job). If a system has a batch queue, the Pilot-Job is submitted to this queue. Once it becomes active, it can run the sub-jobs directly, instead of having to wait for each sub-job to queue. This eliminates the need to submit a different job for every executable and significantly reduces the time-to-completion. Pilots provide a fundamental abstraction for task-level parallelism, by supporting concurrent execution of large number of tasks. The PJ approach supports the decoupling of system-level task submission from resource assignment. PJ provides application level control to schedule each of the sub-jobs/tasks and management of the set of acquired resources. This feature, application level task scheduling control, can support many, if not all, patterns that the user application might have.

RADICAL Pilot is a PJ framework that sits above an interoperability layer called SAGA (Simple API for Grid Applications), that provides a high-level interface to the most commonly used distributed computing infrastructures[8]. SAGA provides mechanisms for distributed infrastructure components like job schedulers, file transfer and resource provisioning services. Given the heterogeneity of distributed infrastructure, SAGA provides a much needed interoperability layer that lowers the complexity and improves the simplicity of using distributed infrastructure whilst enhancing the sustainability of distributed applications, services and tools. Thus, it provides the building blocks to develop the abstractions and functionalities to support the characteristics required by distributed applications whether directly, or as tools in interoperable and extensible fashion.

2.2.1 Architecture

The RADICAL Pilot Architecture can be divided broadly into 3 layers : **Application Layer, Abstraction Layer** and **Infrastructure Layer**[4]. The Application Layer exposes RADICAL Pilot constructs to the user that can be used to develop the application execution logic. The Abstraction Layer consists of various components that form the core of RADICAL Pilot. It handles the deployment of the Pilot Job, scheduling of the tasks, coordination of the tasks, hides the complexity of execution on heterogeneous systems, etc. The Infrastructure Layer consists of components necessary for coordination of tasks, scheduling of tasks on the specific cores, etc.

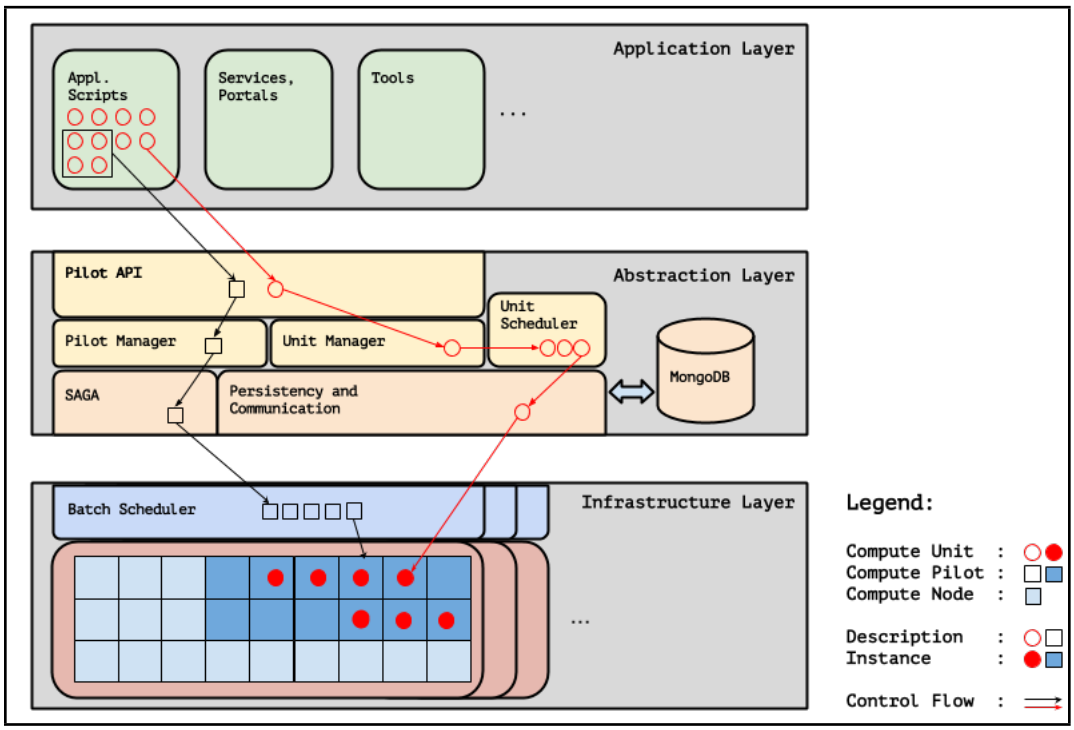

Figure 2.1: RADICAL Pilot Architecture[4])
Application layer consists of the actual application defined in terms of Compute Units. The Abstraction layer consists of RADICAL Pilot components that provide task level abstractions using Compute Units, decoupling of workload management from resource management using a Pilot Job and interoperability using SAGA (Simple API for Grid Applications). Infrastructure layer consists of a Pilot Agent that runs on the remote machine and launches each the Compute Units on to specific cores.

Main components of RADICAL Pilot are as follows :-

- Pilot (Pilot-Compute): The Pilot is the entity that gets submitted and scheduled on a resource. It is the empty container job that acquires the set of resources. Once acquired, it enables user level control and management of the resource.

- Compute Unit: A CU encapsulates a self-contained piece of work (a compute task) specified by the application, that is submitted to the Pilot-Job framework. The task can be either serial or parallel MPI job. There is no intrinsic notion of resource associated with a CU.

- Pilot Manager: The Pilot Manager is responsible for managing the interaction between the Pilots and for decisions regarding resource assignment. CUs can be combined and aggregated; the PM determines how to group them before executing them on the resource via the Pilot.

- Unit Manager: The Unit Manager manages the compute unit instances which represent the executable workload. The UnitManager connects the ComputeUnits with one or more Pilot instances (which represent the workload executors in RADICAL-Pilot) and a scheduler which determines which ComputeUnit gets executed on which Pilot.

- MongoDB: A central database is used for communication and coordination. The client registers compute units to be executed; the pilots eventually pull those requests from the database for execution.

- SAGA: SAGA supports a wide range of distributed, heterogeneous computing middleware via a flexible adapter architecture. SAGA provides a much needed interoperability layer that lowers the complexity and improves the ease of using distributed infrastructure whilst enhancing the sustainability of distributed applications, services and tools.

- Pilot Agent: The Agent is the actual container job that is submitted to the remote machine. Once Active, it bootstraps itself as a python process. It manages the scheduling and execution of Compute units on the individual cores.

2.2.2 Operation

The application submits a Pilot, of a certain size and a specific active duration, to the target resource. Once the Pilot is submitted, the application workload is defined in terms of Compute Units and is submitted to the Unit Manager. The logical sequence of the various executable components of the application can be realized by configuring the execution of the individual compute units. The Unit Manager maps each of the Compute Units to the Pilots. The Unit Manager consists of a scheduler which dictates the algorithm used in the mapping of the Compute Units to the Pilots. This scheduler is flexible and can be changed depending on the application behaviour. The Unit Manager is also responsible for staging the input data to the Compute Unit from either the local machine, any other storage systems or data movements within the remote machine. The Compute Unit, once ready for execution, is updated on the database.

Once the Pilot Agent is active on the remote machine, the Agent pulls the Compute Units from the database and based on the resource requirements of the Compute Units and current availability schedules or queues them. Each Compute Unit, apart from the required input data, consists of a shell script that 1) sets up the execution environment - this includes modules to be loaded, environment variables to be assigned 2) contains system specific executable to start execution of the actual workload.

As each of the Compute Units finish, the RP Agent updates the database and maintains synchronization with the Unit Manager which may contain more CUs to submit. The RP Agent is responsible for managing any output data from the Compute Units back to the local machine. Upon termination of the application, the Pilot on the remote machine is canceled.

2.2.3 RADICAL Pilot API : Bag of Tasks Example

In this section, we describe with an example, the API of RADICAL Pilot for the simple case of a 'Bag of Tasks'. We will create an example which submits N identical jobs (Bag

of Tasks) using RADICAL-Pilot[5]. This type of run is very useful if you are running many jobs using the same executable (but perhaps with different input files). Rather than submit each job individually to the queuing system and then wait for every job to become active and complete, you submit just one Pilot job that reserves the number of cores needed to run all of your jobs. When this pilot becomes active, your tasks are pulled by RADICAL-Pilot from the MongoDB server and executed.

2.2.3.1 Creating a session

```
import radical.pilot as rp


session = rp.Session(database_url="mongodb://my-mongodb-server.edu:27017")
c = rp.Context('ssh')
c.user_id = "tutorial_X"
session.add_context(c)
```

Create a new session. A radical.pilot.Session is the root object for all other objects in RADICAL Pilot. It has a tree structure with the Session object as the root and zero or more objects of radical.pilot.Context, radical.pilot.PilotManager and radical.pilot.UnitManager attached to it. A Session also encapsulates the connection(s) to a back end MongoDB server which provides persistence and coordination for RADICAL Pilot. Each Session has a unique identifier (uid) and methods to traverse its members. The Session uid can be used to disconnect and reconnect to a Session as required.

2.2.3.2 Creating a ComputePilot

A radical.pilot.ComputePilot is the Pilot Job entity for RADICAL Pilot. It is responsible for executing the ComputeUnits (task) assigned to it. ComputePilots can be launched either locally or remotely, on a single machine or on one or more HPC clusters, i.e., you could have multiple pilots running on same or several machines at the same time. ComputePilots are grouped in radical.pilot.PilotManager containers, so before you can launch a ComputePilot, you need to add a PilotManager to your Session. Just like a Session, a PilotManager has a unique id (uid) as well as a traversal method.

```
pmgr = radical.pilot.PilotManager(session=session)
```

In order to create a new ComputePilot, you first need to describe its requirements and properties. This is done with the help of a radical.pilot.ComputePilotDescription object.

```
pdesc = radical.pilot.ComputePilotDescription()
pdesc.resource  = "localhost"
pdesc.runtime   = 5
pdesc.cores     = 2
```

Here, we have listed the three mandatory parameters required to launch a Pilot Job:-

- **resource**: The name of the target machine or localhost
- **runtime**: The runtime, in minutes, that the Pilot should remain Active
- **cores**: The number of cores the Pilot would acquire.

A ComputePilot is launched by passing the ComputePilotDescription to the submit_pilots() method of the PilotManager. Like any other object in RADICAL-Pilot, a ComputePilot also has a unique identifier (uid)

```
pilot = pmgr.submit_pilots(pdesc)
```

2.2.3.3 Creating ComputeUnits

Once a ComputePilot is launched, you can now generate a few radical.pilot.ComputeUnit objects for the ComputePilot to execute. A ComputeUnit is very similar to an operating system process that consists of an executable, a list of arguments, and an environment along with some runtime requirements.

```
compute_units = []

for unit_count in range(0, 8):
    cu = radical.pilot.ComputeUnitDescription()
```

```
cu.environment = {"SLEEP_TIME" : "10"}
cu.executable  = "/bin/sleep"
cu.arguments   = ["$SLEEP_TIME"]
cu.cores       = 1


compute_units.append(cu)
```

Analogous to ComputePilots, a ComputeUnit is described via a ComputeUnitDescription object. The mandatory properties that you need to define are:

- **executable**: the executable to launch
- **argument**: arguments to the executable
- **cores**: cores required by the executable

This snippet creates a list of 8 Compute Units that perform a '/bin/sleep 10'.

2.2.3.4 Scheduling ComputeUnits

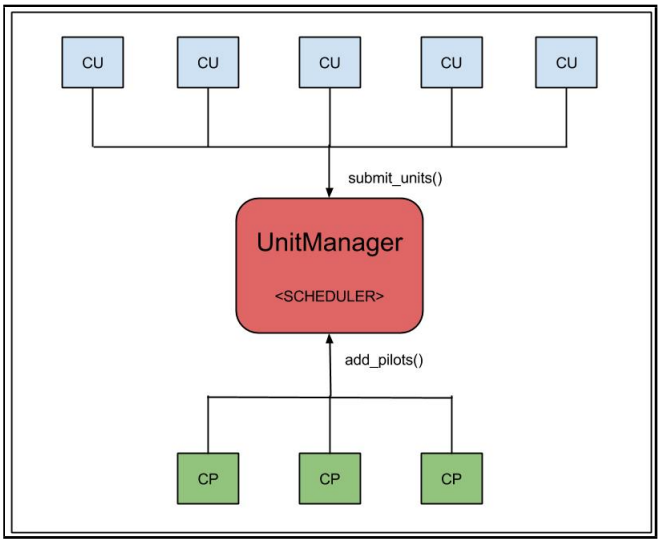

Figure 2.2: RADICAL Pilot UnitManager
Both the set of Pilots and the set of Compute Units are submitted to the UnitManager. The UnitManager, then, based on the selected algorithm binds each Compute Unit to a particular Pilot for execution.

In the previous steps we have created and launched a ComputePilot (via a PilotManager) and created a list of ComputeUnitDescriptions. In order to put it all

together and execute the ComputeUnits on the ComputePilot, we need to create a radical.pilot.UnitManager instance.

As shown in the diagram above, a UnitManager combines three things: UnitManager.submit_units() used to add the ComputeUnits, one or more ComputePilots, added via UnitManager.add_pilots() and a Unit Scheduler. The UnitManager then binds each of the submitted ComputeUnits to any of the ComputePilots based on a scheduling algorithm

Since we have only one ComputePilot, we dont need any specific scheduling algorithm for our example. We choose SCHED_DIRECT_SUBMISSION which simply passes the ComputeUnits on to the ComputePilot.

```
umgr = radical.pilot.UnitManager(session=session, scheduler=radical.pilot.
    SCHED_DIRECT_SUBMISSION)

umgr.add_pilots(pilot)
umgr.submit_units(compute_units)

umgr.wait_units()
```

The radical.pilot.UnitManager.wait_units() call blocks until all ComputeUnits have been executed by the UnitManager. Simple control flows / dependencies can be realized with wait_units()

2.2.3.5 ComputeUnit I/O

There are two points to be satisfied in order for RADICAL Pilot to transfer files. First you need to specify the respective input and output files for the Compute Unit in so called staging directives. Then, these staging directives need to be linked to each of the Compute Units using the input_staging and output_staging members. For this reason, a radical.pilot.ComputeUnitDescription allows the definition of input_staging and output_staging:

- **input_staging**: a list of local files that need to be transferred to the execution resource before a ComputeUnit can start running.

- **output_staging**: a list of remote files that need to be transferred back to the local machine after a ComputeUnit has finished execution.

Let's discuss using an example:-

```
INPUT_FILE_NAME  = "INPUT_FILE.TXT"
OUTPUT_FILE_NAME = "OUTPUT_FILE.TXT"


cud = radical.pilot.ComputeUnitDescription()
cud.executable = "/usr/bin/sort"
cud.arguments = ["-o", OUTPUT_FILE_NAME, INPUT_FILE_NAME]
cud.input_staging  = INPUT_FILE_NAME
cud.output_staging = OUTPUT_FILE_NAME
```

Here the staging directives INPUT_FILE_NAME and OUTPUT_FILE_NAME are simple strings that both specify a single filename. This transfers the file from the source local directory to the directory where the ComputeUnit is being executed. After the task has run, the file OUTPUT_FILE.TXT that has been created by the task, will be transferred back to the local directory.

The format of the staging directives can either be a string as above or a dict of the following structure:

```
staging_directive = {
    'source':   source,
    'target':   target,
    'action':   action,
    'flags':    flags,
    'priority': priority
}
```

- **source,target**: In case of the staging directive being used for input, then the source refers to the location to get the input files from, e.g. the local working directory on your laptop or a remote data repository, target refers to the working directory of the ComputeUnit.

- **action**: Depending on the relative location of the working directory of the source to the target location, the action can be COPY (local resource), LINK (same file system), MOVE (local resource), or TRANSFER (to a remote resource).
- **flags**: By passing certain flags we can influence the behavior of the action.
  - CREATE_PARENTS : Create parent directories while writing file.
  - SKIP_FAILED : Dont stage out files if tasks failed
- **priority**: This optional field can be used to instruct the backend to prioritize the actions on the staging directives

2.2.3.6 ComputePilot I/O

In addition to the constructs on Compute Unit-level RADICAL-Pilot also has constructs on Compute Pilot-level for data transfer. The reason behind this is that quite often there is (input) data to be shared between multiple Compute Units. Instead of transferring the same files for every Compute Unit, we can transfer the data once to the Pilot, and then make it available to every Compute Unit that needs it.

This works in a similar way as the Compute Unit-IO, where we use also use the Staging Directive to specify the I/O transaction The difference is that in this case, the Staging Directive is not associated to the Description, but used in a direct method call pilot.stage_in(sd_pilot).

```
sd_pilot = {'source': shared_input_file_url,
            'target': os.path.join(MY_STAGING_AREA, SHARED_INPUT_FILE),
            'action': radical.pilot.TRANSFER
}
pilot.stage_in(sd_pilot)
```

Chapter 3

Initial Prototype: Direct Integration with Radical Pilot

The first prototype of the framework was implemented directly using Radical Pilot. Resource acquisition was done using a Pilot. The different Simulation and Analysis engines were defined in terms of Compute Units. The data flow is maintained using Radical Pilot directives. A physicist or chemist using this tool for the purposes of science, does/should not require the knowledge of Radical Pilot or the underlying mechanisms to run the science application. To keep it so, we use a command line interface and use configuration files as the only method of user interaction. We discuss the detail of the architecture in greater detail in the forthcoming sections.

3.1 Simulation and Analysis Engines

Before we discuss the architecture of this native implementation, it is important to get an idea of the type of workload and the science behind it. The tool currently supports two MD Simulation Engines - **Gromacs**, **Amber** and two Analysis Engines - **LSDMap**, **CoCo**. By engines we mean the actual executable kernels on the target machine, we will be using them inter-changeably.

MD Simulation is a computational method that calculates the time dependent behavior of a molecular system. Scientists are interested in the changes and observations at a macroscopic level described usually in terms of temperature, pressure, etc. MD simulations provide detailed information on the fluctuations and conformational changes of proteins and nucleic acids. This information is at the microscopic level including atomic positions, velocities and momentum. The state of the molecule can be described in terms of these parameters/dimensions by a single point. MD simulations

generate a series of such points that are equivalent at a macroscopic level but differ at the microscopic level[13]. These methods are now routinely used to investigate the structure, dynamics and thermodynamics of biological molecules and their complexes. Gromacs and Amber both are MD packages that consist of various algorithms that do the same.

The connection between microscopic simulations and macroscopic properties is made via statistical mechanics which provides the rigorous mathematical expressions that relate macroscopic properties to the distribution and motion of the atoms and molecules[13]. The Analysis stage, in this tool, uses two such tools. CoCo analyses the distribution of an ensemble of structures in conformational space, and generates a new ensemble that fills gaps in the distribution[9]. These new structures provide possible, approximate, new solutions which is a refinement of the original data. LSDMap, on the other hand, constructs a ensemble based on the existing ensembles[14]. Based on probability densities, the algorithm assigns weights to each ensemble, the number of ensemble members to be considered can increase or decrease based on these analysis.

3.2 The Pattern : Simulation-Analysis Loop

A careful look at the workflow reveals an important information. The workflow follows an iterative pattern with two stages. The first stage is similar to a Bag of Tasks case, where each task performs a molecular dynamics simulation. Once all of the MD Simulation Tasks have completed, an Analysis is performed collectively on the outputs of all of the Simulation Tasks. The Analysis in our example is a single task but can also be converted into a Bag of Tasks in which case the output of the simulation stages are partitioned among the different analysis tasks. The above two stages form one iteration of the workflow.

There also exists a pre_loop and post_loop stage during which there is some pre-processing and post-processing of data. These occur outside of the loop and once per

entire workflow.

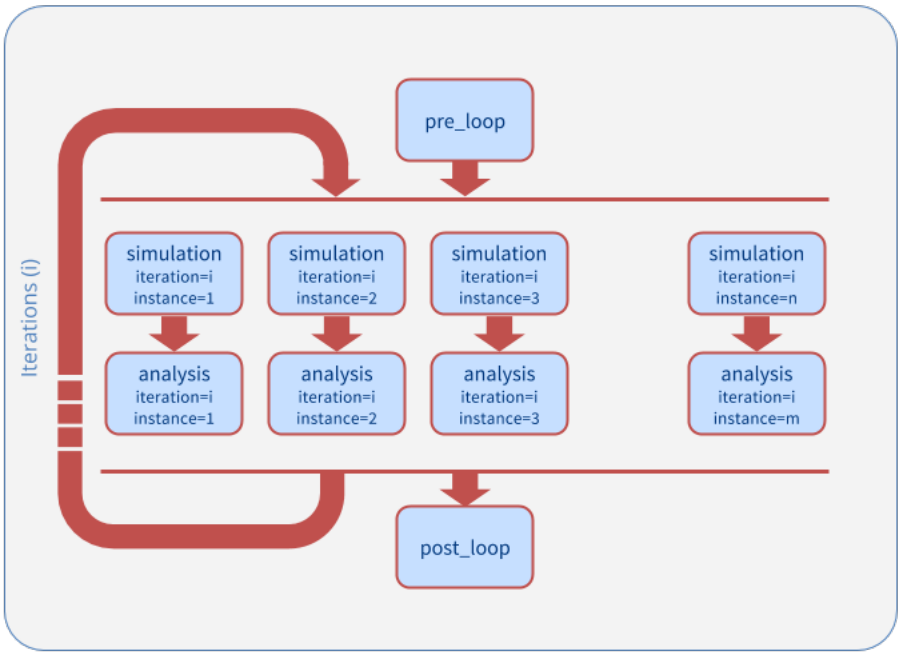

Figure 3.1: Simulation Analysis Loop[3]
The crux of the pattern consists of N bag of ensembles of MD simulations followed by M bag of analysis phases per one iteration. The pre_loop and post_loop perform preprocessing and post-processing functions.

3.3 Architecture

As mentioned before, ExTASY provides an configuration file based interface to users, so that they can take advantage of RADICAL Pilot without knowing the details of programming with RADICAL Pilot. ExTASY is designed to support the addition of new Simulators and Analyzers and they can be easily switched, by the user, from one to the other via the configuration files.

- Resource Configuration file: The file is used to specify the details required to launch a pilot. This mandatory file defines the target remote machine, runtime duration, number of cores, etc.

- Kernel Configuration file: This file is used to specify the details of the Simulation stage, Analysis stage and other workload specific parameters. These include

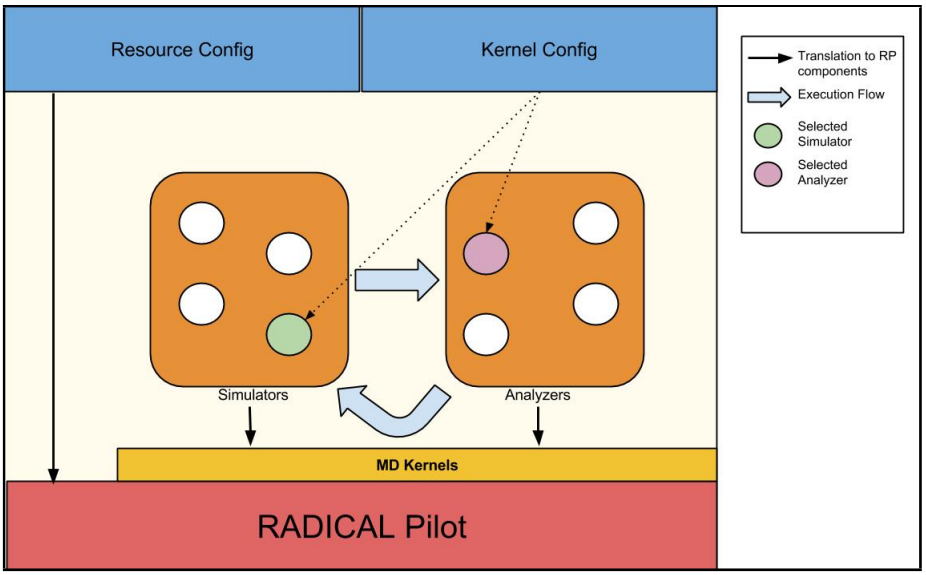

Figure 3.2: ExTASY Architecture
The ExTASY framework enables users to perform large scale ensemble based simulations followed by analysis algorithms. Two inputs: Resource configuration file that consists of parameters for launching the Pilot, Kernel Configuration file that consists of parameters required during simulation and analysis. MD Kernels is a set of dictionaries using which each Compute Unit can create the environment required to perform the simulation/analysis.

number of iterations, number of Compute Units, input filenames and simulation/-analysis specific parameters.

- Simulators: This is a bag of all supported Simulation kernels. It is extendable and currently supports Gromacs and Amber.

- Analyzers: This is a bag of all supported Analysis kernels. It is extendable and currently supports LSDMap and CoCo.

- MDKernels: This is a light layer which generates the exact the executable and environment setup for each remote target. This because, for the same application, the executables and the execution environment might be different for different resources. Hence, based on the resource, this layer sets up all the modules and environment variables and mentions the exact version of the executable to be used. The output of this layer is used to construct the Compute units to be submitted.

- RADICAL Pilot: This is the base layer of the framework and is used to execute the large number of simulations and analysis as well as manage the associated data efficiently.

3.4 Operation

Once ExTASY is installed and the necessary simulation and analysis files are available on the local machine, the user would first require to set up the resource configuration file with the details of target resource name, runtime, reservation amount, etc. Next, the user would have to set up the kernel configuration file. As mentioned before, this includes the arguments and filenames required in the simulation and analysis stages as well as details as to the number of compute units, number of iterations, etc.

The parameters mentioned in the kernel configuration file are used to create the compute units. The MDKernels package is used to add the system specific environment set up which includes the modules and PATH variables required for successful execution of the compute units. In this tool, first a bag of simulation tasks is submitted to the Pilot, and after the completion of all the tasks, an analysis compute unit is submitted. Therefore, a bag of tasks followed by an analysis constitutes one iteration and the tool can be configured to perform multiple iterations where one iteration uses the output of the last.

3.5 Other "Patterns"

3.5.1 Replica Exchange

Replica Exchange is another pattern which consists of multiple MD stages. In the first stage, a preprocessing is performed on the input file followed by "N" MD simulation steps. Once the simulations are completed, an exchange step is carried out, there are 3 exchange algorithms that can be used - Temperature Exchange, Salt Concentration, Umbrella Sampling. These Simulation and Exchange form one iteration/cycle of the pattern. A pictorial representation is given in Fig 3.3.

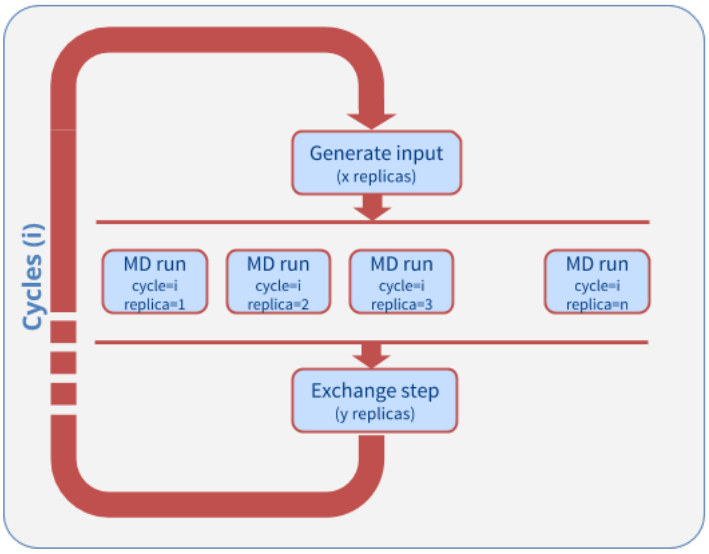

Figure 3.3: Replica Exchange Pattern[3]

3.5.2 All-pairs

The All-pairs problem is stated as:

All-Pairs(set A, set B, function F) returns a matrix M which is composed by comparing all elements of set A to all elements of set B using the function F. Otherwise stated as, M[i,j] = F(A[i],B[j])[12]. A pictorial representation is given in Fig 3.4.

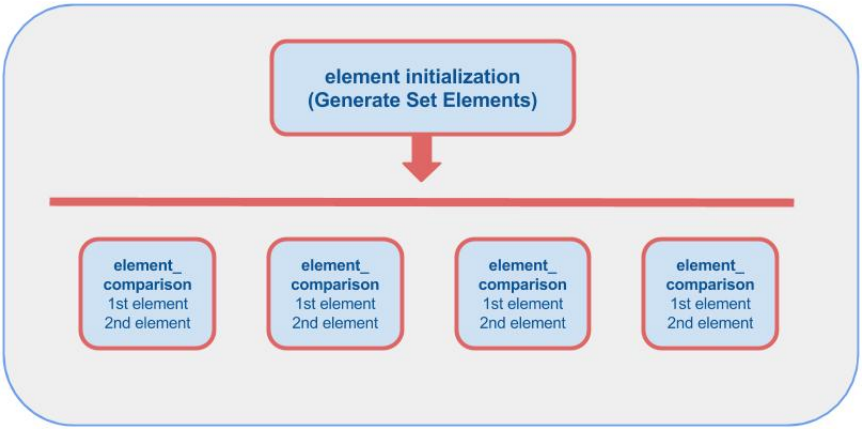

Figure 3.4: All Pairs Pattern[3]

Chapter 4

Advanced Prototype: Ensemble MD Toolkit

4.1 Motivation

The initial prototype, ExTASY, gave us a proof of the model. The model implemented a MD Simulation and Analysis loop pattern tool with RADICAL Pilot as the underlying execution manager with a command line interface to use this tool. This restricts the user to only 'use' such tools and not 'develop' them. To develop such tools the user would require knowledge of the underlying RADICAL Pilot framework. The questions, thus posed, are simple: (1) Can we enable the user to 'develop' such tools for science applications without requiring any knowledge of the layers underneath ? (2) What are the building blocks that need to be provided to the user ? (3) Can we provide a common tool that supports the 3 common MD patterns ?

4.2 Overview of Ensemble MD

The Ensemble MD Toolkit is a Python framework for developing and executing applications that are comprised of many simulations aka ensembles. It was designed with the goal to provide an intuitive programming and job execution framework for modeling and executing large-scale molecular dynamics (MD) applications. The architecture follows a top-down approach, starting from the top with the building blocks exposed to the user, support for the common patterns in the field of molecular dynamics down to the actual MD tools. The elements of RADICAL Pilot and the underlying layers are hidden from the user behind these high-level concepts. The toolkit follows a style of "pick the pattern that matches my problem and fill in the details".

The questions posed earlier are converted to framework requirements and defined. The toolkit should :-

1. **Concentrate on the science application**: This keeps the application development time short and meaningful for the scientists. The framework follows a "pick the template that matches my problem and fill in the details" development style.

2. **Decouple "what to execute" from "how to execute"**: The straightforward approach of defining the execution of tasks via the application logic has potential drawbacks. These include lack of concurrent execution, lack of overlapping data movement and computation, etc. It does not also give room to resource specific optimizations.

   Ensemble MD Toolkit addresses this problem by separating the Application Control from the Execution Strategy. The user facing script describes the Application Control and the Execution Strategy is described by "Execution Plugins". Decoupling the two allows the Execution Plugins to, based on the information about the application control and the associated data flow, execute strategically.

3. **Hide the heterogeneity**: There are two levels of heterogeneity. Firstly, the heterogeneity in the execution methods of the various target resources. The complication of execution in different systems should abstracted out in the lower layers. In Ensemble MD, this is handled by the SAGA layer which is used by RADICAL Pilot as a means of interoperability.

   Second, heterogeneity in the specific versions (and their dependencies) of the scientific tools. In Ensemble MD, this requirement is addressed via "Kernel Plugin" that use a pre-defined set of dictionaries that specify the modules and dependencies to be loaded on the different machines. At the user level, there should be no specific descriptions required when shifting from one resource to another.

4.3 Architecture

Now that we have the three main components describes, we shall now describe how these components fit along with each other to form the Ensemble MD Toolkit.

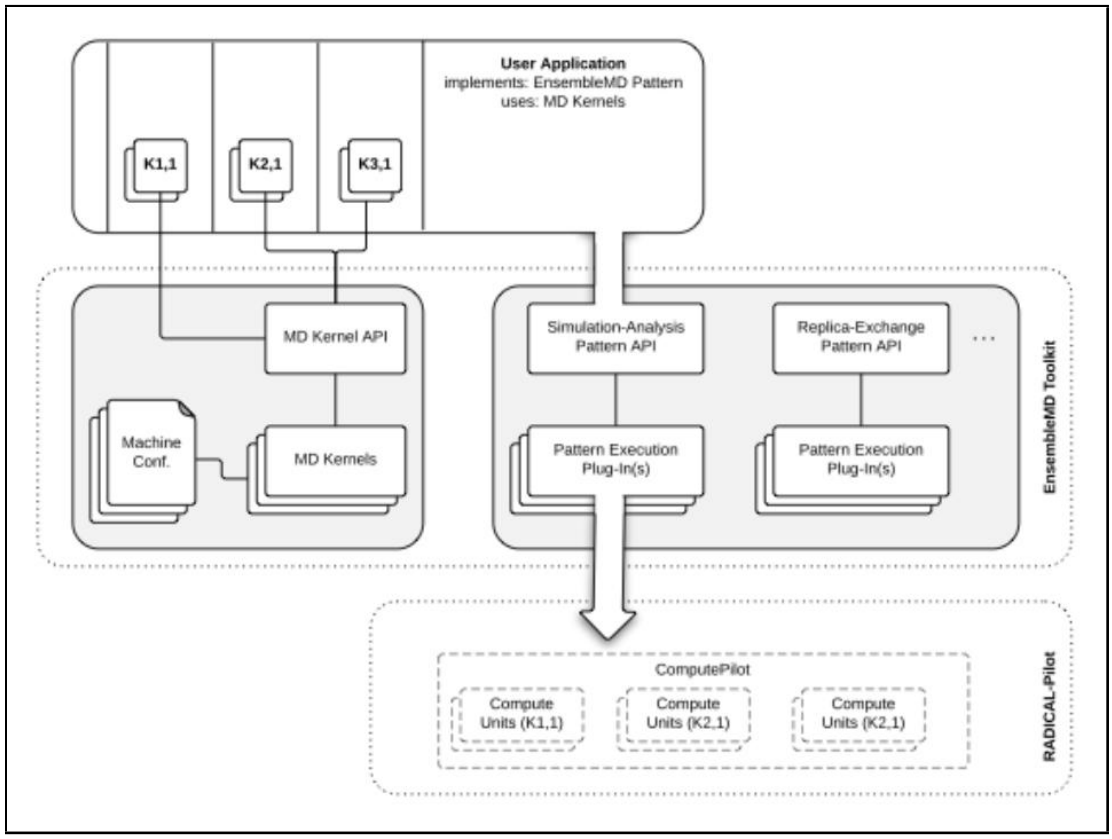

Figure 4.1: Ensemble MD Architecture[2]

Keeping the requirements and objectives in mind. We define 3 major components of the Ensemble MD Toolkit.

- **Application Patterns**: The Application Pattern is the user facing script and describes "what to do", i.e. the Application Control and Dataflow. An MD Application Pattern represents an executable set of tasks that exhibit a specific relationship among each other. Supported patterns will be:
  - Simulation-Analysis
  - Replica-Exchange
  - All-Pairs

An MD Application Pattern dictates the logic by which these tasks are scheduled to execute and transfer their respective data.

- **Execution Context**: The Execution Context decouples EnMD and radical.pilot. Within an Execution Context, a Bag of Tasks or an MD Application Pattern is executed. The Execution Context translates a Bag of Tasks into a corresponding set of radical.pilot.ComputeUnits. An MD Application Pattern (which internally consists of multiple, coupled Bag of Tasks) is also translated into a set of radical.pilot.ComputeUnits with the help of the respective Pattern Execution Plugin, which can also implement Pattern-specific optimization methods. Timing probes are added in specific parts of the execution plugin for such purposes.

  The Execution Context also takes care of resource allocation via radical.pilot. Here, too the Pattern Execution Plugins, can implement advanced resource management techniques, like for example allocating pilots for a future simulation iteration in advance. Since the plugin uses RADICAL Pilot, the requirement of addressing resource heterogeneity is accomplished which provides/ utilizes the interoperability layer, SAGA.

- **Kernel Plugins**: A Kernel Plugin represents an instantiation of a specific science tool along with its resource specific modules and environment variable definitions. The science tools supported are:-

  - AMBER
  - GROMACS
  - NAMD
  - LSDMAP
  - CoCo

  Kernel plugins hide tool-specific peculiarities across different clusters as well as differences between the interfaces of the various MD tools to the extent possible. Data dependencies (data-flow) is also defined between kernels, however, it is confined to the overarching control flow provided by the pattern.

4.4 Operation

The User Application sits on top of Ensemble MD. The application first "allocates" the resources. The pattern to be executed is selected by the user application. Then the required kernels are defined along with their arguments, in this case, (K1,1), (K2,1) and (K3,1). The MD Kernels API uses the resource information to select the specific kernel definitions from the dictionary of kernels. The resource specific kernel definitions are passed on to the execution plugin of the pattern. In the execution plugin, the kernel definitions as well as input/output specifications are converted to RADICAL Pilot Compute Units and submitted on to the target machine following their respective patterns.

4.5 Ensemble MD API : SimulationAnalysisLoop Pattern

In this section, we discuss the API of the toolkit with some examples and details that are paramount in developing applications using the Ensemble MD toolkit. In the Simulation Analysis Loop pattern, there four stages which need to defined :-

- **pre_loop** : Function is executed once before the Simulation Analysis loop
- **simulation_step**: Function which defines what needs to be executed during the simulation stage
- **analysis_step**: Function which defines what needs to be executed during the analysis stage
- **post_loop** : Function is executed once at the end of the Simulation Analysis loop

There are multiple data reference methods defined for the Simulation Analysis loop pattern.

4.5.1 Creating the Execution Context

Table 4.1: Simulation Analysis Loop : Step Referencing Variables

PRE_LOOP	Working directory of the pre_loop
PREV_SIMULATION	References the previous simulation step with the same instance number
PREV_SIMULATION_INSTANCE_Y	References instance Y of the previous simulation step
SIMULATION_ITERATION_X_INSTANCE_Y	References instance Y of the simulation step of iteration number
PREV_ANALYSIS	References the previous analysis step with the same instance number
PREV_ANALYSIS_INSTANCE_Y	References instance Y of the previous analysis step
ANALYSIS_ITERATION_X_INSTANCE_Y	References instance Y of the analysis step of iteration number X

```
cluster = SingleClusterEnvironment(
          resource="localhost",
          cores=1,
          walltime=30,
          username=None,
          allocation=None)

cluster.allocate()
```

The function **SingleClusterEnvironment()** is an Ensemble MD function that creates an Execution Context with one resource, 'localhost' in this case. It defines a fixed number of resources to be acquired in terms of 'cores' for a fixed 'walltime'. The parameters 'username' and 'allocation' are used when the resource (generally remote) requires a username to be accessed and an allocation number to be charged. The allocate() launches the RADICAL Pilot with the above mentioned parameters on to the target machine.

4.5.2 Defining the Pattern Details

```
randomsa = RandomSA(maxiterations=4,
                    simulation_instances=16,
                    analysis_instances=16)

cluster.run(randomsa)
```

We instantiate a class **RandomSA** with three parameters, maxiterations, simulation_instances and analysis_instances. This function (discussed in the next subsection) describes the specifications of Simulation Analysis pattern. 'maxiterations' stands for the number of iterations of the Simulation-Analysis loop. 'simulation_instances' stands for the number of simulation instances in 1 iteration. Similarly, 'analysis_instances' stands for the number of analysis instances in 1 iteration.

4.5.3 Creating the Simulation-Analysis Class

```
from radical.ensemblemd import SimulationAnalysisLoop

class RandomSA(SimulationAnalysisLoop):

    def __init__(self,
                 maxiterations,
                 simulation_instances=1,
                 analysis_instances=1):
        SimulationAnalysisLoop.__init__(self,
                                        maxiterations,
                                        simulation_instances,
                                        analysis_instances)
```

The Simulation Analysis class is created by importing the SimulationAnalysisLoop base class. The 'RandomSA' class is created using the imported base class by passing the maxiterations, number of simulation instances and the number of analysis instances.

4.5.4 Defining the 'pre-loop'

```
def pre_loop(self):
```

```
        k = Kernel(name="misc.mkfile")
        k.arguments = ["--size=1000","--filename=reference.dat"]
        return k
```

Important snippets of the misc.mkfile kernel plugin :-

```
_KERNEL_INFO = {
"name": "misc.mkfile",
"description": "Creates a new file of given size and fills it with random
    ASCII characters.",
"arguments": {
                "--size=":{"mandatory": True,
                        "description": "File size in bytes."},
                "--filename=":{"mandatory": True,
                                "description": "Output filename."}
            }
}
```

'pre_loop' is executed before the main simulation-analysis loop is started. In this example we create an initial 1 kB random ASCII file that we use as the reference for all analysis steps.

4.5.5 Defining the 'simulation_step'

```
def simulation_step(self, iteration, instance):
        k = Kernel(name="misc.mkfile")
        k.arguments = ["--size=1000",
        "--filename=simulation-{0}-{1}.dat".format(iteration, instance)]
        return k
```

The Kernel plugin is the same as in the pre_loop case.

The simulation step generates a 1 kB file containing random ASCII characters that is used to compare against the 'reference' file (created in the pre_loop) in the subsequent analysis step.

4.5.6 Defining the 'analysis_step'

```
def analysis_step(self, iteration, instance):

        input_filename  = "simulation-{0}-{1}.dat".format(iteration,
        instance)
        output_filename = "analysis-{0}-{1}.dat".format(iteration,
        instance)

        k = Kernel(name="misc.levenshtein")
        k.link_input_data = ["$PRE_LOOP/reference.dat",
        "$PREV_SIMULATION/{0}".format(input_filename)]

        k.arguments = ["--inputfile1=reference.dat","--inputfile2={0}".
        format(input_filename),"--outputfile={0}".format(output_filename)]

        k.download_output_data = output_filename
        return k
```

Important snippets of the misc.levenshtein kernel :-

```
_KERNEL_INFO = {
"name": "misc.levenshtein",
"description": "Calculates the Levenshtein distance between to strings.",
"arguments": {
                "--inputfile1=":{"mandatory": True,
                "description": "The fist input file."},
                "--inputfile2=":{"mandatory": True,
                "description": "The second input file."},
                "--outputfile=":{"mandatory": True,
                "description": "The output file containing the distance
                                value."}
                },
}
```

In the analysis step, we take the previously generated simulation output and perform a Levenshtein distance calculation between it and the 'reference' file. The placeholder '$PRE_LOOP' used in 'link_input_data' is a reference to the working directory

of pre_loop. The placeholder '$PREV_SIMULATION' used in 'link_input_data' is a reference to the working directory of the previous simulation step. It is also possible to reference a specific simulation step using '$SIMULATION_N' or all simulations via '$SIMULATIONS'. Analogous placeholders exist for 'ANALYSIS'.

'post_loop' function exists after the analysis step but is not used in this example. As described before, the 'post_loop' is executed once all the iterations of the simulation and analysis instances have been completed. The complete code for the above simple simulation analysis loop pattern is available in the appendix. Variations of this pattern have also been added.

4.6 Usability

In the previous subsection we discussed an example script for the Simulation Analysis Loop Pattern which determined the Levenshtein distance between two files. Changing what each of the steps do, requires only change in the kernel plugin. The user can use one of the kernel plugins already supported by EnMDTK or write one's own kernel plugin. Similarly, scientists running their application with a particular MD tool may switch to another with just a change in the kernel name. This enables the scientists to compare the MD tools and analysis the results of the same workflow with two different MD tools. Similarly, changing to the target machine can be done easily as well; only the details in the 'SingleClusterEnvironment()'' need to be changed.

Chapter 5

Experiments and Results

5.1 Hardware Resources

RADICAL Pilot and RADICAL Pilot based frameworks are very flexible and can utilize a number of heterogeneous resources. For this thesis, both the initial and advanced Prototype are supported on TACC Stampede[6] located in Austin, Texas and EPCC Archer[7] located in Edinburgh, UK. Extension to any other machines would, as discussed before, be only a matter of specifying the target in the user script along with minor changes to module load operations. For the purposes of experimentation, we chose to run on Stampede due to smaller queue wait times as compared to Archer. The TACC Stampede system is a 10 PFLOPS (PF) Dell Linux Cluster based on 6400+ Dell PowerEdge server nodes, each outfitted with 2 Intel Xeon E5 (Sandy Bridge) processors and an Intel Xeon Phi Coprocessor. More details about Stampede are presented in table 5.1.

Table 5.1: Stampede Hardware details[6]

Component	**Technology**	**Performance/Size**
Memory Distributed	32GB/node	205TB (Aggregate)
Shared Disk	Lustre 2.1.3, parallel File System	14 PB
Local Disk	SATA (250GB)	1.6PB (Aggregate)
Interconnect	InfiniBand Mellanox Switches/HCAs	FDR 56 Gb/s

5.2 Usecase 1: Gromacs-LSDMap

5.2.1 Experiments

In Gromacs-LSDMap, a large set of configurations/ensemble members (*.gro) is provided as input. A topology file (*.top), a gromacs parameter file (*.mdp) and a LSDMap

configuration file (*.ini) form the other set of inputs. The large set of ensemble members are uniformly divided and provided as input to each of the Compute Units. This implies that the amount of work done by each Compute Unit is a function of the number of total Compute Units.

Three experiments were carried out with Gromacs-LSDMap on Stampede.

- **Experiment 1**: In this first experiment, we keep the input file size constant, i.e., we keep the number of ensemble members constant. We keep the Pilot Size constant and vary the number of Compute Units used to complete the simulation. Hence, there will be multiple generations of Compute Units in the simulation stage. Each analysis stage uses 1 Compute Unit with cores equal to the Pilot size. This is a mark of the strong scaling behaviour of the toolkit.

- **Experiment 2**: In this experiment, we again keep the input file size constant. We again test the strong scaling behaviour of the toolkit while using Gromacs-LSDMap as the application kernels. But we vary the Pilot Size in each trial, a different dimension in strong scaling. We keep the ratio of Compute Units to the Pilot Size as 1. Hence, there will always be only 1 generation of Compute Units in each trial. Similar to experiment 1, we keep the number of analysis Compute Units as 1 with cores equal to the Pilot Size.

- **Experiment 3**: In the third experiment, we increase the input file size to observe the toolkitś behaviour. We also increase the Pilot Size by the same amount as the increase in the input size, in an attempt to observe if we can get similar execution times. Intuitively, this makes sense, since we increase the acquired resources by the same amount as the input size.

5.2.2 Results

Experiment 1: Execution time as a function of number of Compute Units

Description of graph:-

Detail : Gromacs-LSDMap on Stampede performed with 10000 initial ensemble members, 1 iteration

X-axis : Pilot Size/ Number of Compute Units
Y-axis(L) : Execution time for individual stages
Y-axis(R) : Total Execution time

In figure 5.1, we observe the behaviour of varying execution time with varying Compute Units. Each Compute Unit, has a gromacs preprocessing step followed by a MD simulation step. The execution time is a sum of both these steps. With increasing number of Compute Units, the size of the input file per Compute Unit decreases. We see a trade off between these two steps and observe an optimum point when number of Compute Units are 256 or 512. The analysis stage is executed over 16 cores in all the trials with the same size of input and hence remains similar at each data point. The conclusion of this experiment is that, there lies a "sweetspot" or an optimal distribution of work for every configuration that would reduce the execution time of the experiment.

Experiment 2: Execution time as a function of Pilot Size *Description of graph*:-
Detail : Gromacs-LSDMap on Stampede performed with 10000 initial ensemble members, 1 iteration
X-axis : Pilot Size/ Number of Compute Units
Y-axis(L) : Execution time for individual stages
Y-axis(R) : Total Execution time

In figure 5.2, we observe the behaviour of varying execution time with varying Pilot Size. Each Compute Unit, has a gromacs preprocessing step followed by a MD simulation step. The execution time is a sum of both these steps. In this experiment, we keep the number of Compute Units to be the same as the Pilot Size, i.e., there is only one generation of Compute Units. With increasing number of Pilot Size, and keeping the input size constant, we can observe noticeable decrease in the execution time. This is somewhat expected, as we increase the resources used in doing the same amount of work. The conclusion drawn from this experiment is that, it is possible to reduce the execution time of the experiment if more resources can be availed.

Experiment 3: Execution time as a function of Pilot Size and Input Size

Description of graph:-

Detail : Gromacs-LSDMap on Stampede performed with 300000 timesteps, 1 iteration

X-axis : Pilot Size/ Number of Compute Units/ Number of ensemble members

Y-axis(L) : Execution time for individual stages

Y-axis(R) : Total Execution time

Experiment 3 is almost a converse of experiment 2. In this experiment, we vary the pilot size and the input size by the same factor, thus keeping their proportion constant. We can observe that while going from 10,000 ensemble members to 20,000 and then 40,000, the simulation execution time remains similar in each case. This is so since we increase the resources by the same proportion, 16,32 and 64. We keep the number of Compute Units constant to remove the effect of distribution of work. The conclusion of this experiment is that, given large number of resources, it is possible to keep the execution times within an expected/desirable limit.

5.3 Usecase 2: Amber-CoCo

5.3.1 Experiments

In the case of Amber-CoCo, we use the configuration of the molecule "Vilin". In this scenario, each compute unit gets the same number of ensemble members regardless of the number of compute units as opposed to usecase 1. The simulation stage is divided into the minimization step and the MD simulation step. The configuration file, minimization parameters, simulation parameters and the topology file are provided to each of the compute units in the simulation stage. In the analysis stage, similar to gromacs-lsdmap, a configuration file is provided for CoCo which acts upon the output of the simulation stage.

Experiment: Similar to the experiments in the gromacs-lsdmap case, we vary the

number of Compute Units and the Pilot Size and observe the Execution Time. In this experiment, the amount of work done by an individual Compute Unit remains constant as the number of ensemble members per Compute Unit remains constant (1 in this case). This is a mark of the weak scaling behaviour of the toolkit for this application.

5.3.2 Results

Experiment: Execution time as a function of Pilot Size and Compute Units

Description of graph:-

Detail : Amber-CoCo on Stampede performed with 5000 timesteps, 1 iteration

X-axis : Pilot Size/ Number of Simulation Compute Units

Y-axis : Execution time for individual stages

In figure 5.4, we vary the number of Compute Units over different values of the Pilot Size. We observe that, for a combination of Pilot Size and Compute Units, if the number of generations of Compute Units is constant, so is the Execution Time. For example, for a Pilot Size = 16 and number of Compute Units = 16, the Execution Time is same as that of Pilot Size = 32, Compute Units = 32. The work done is twice though. This again points us to the fact that, given enough amount of resources, the Execution Time for increasing workload can be kept similar.

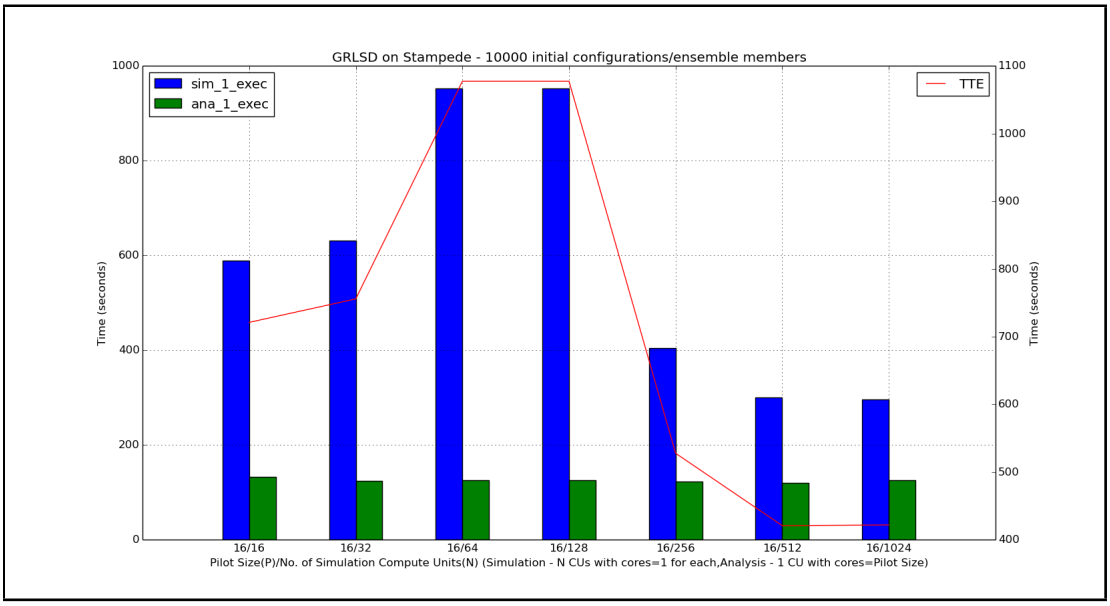

Figure 5.1: Gromacs-LSDMap : Exec time vs No. of Compute Units

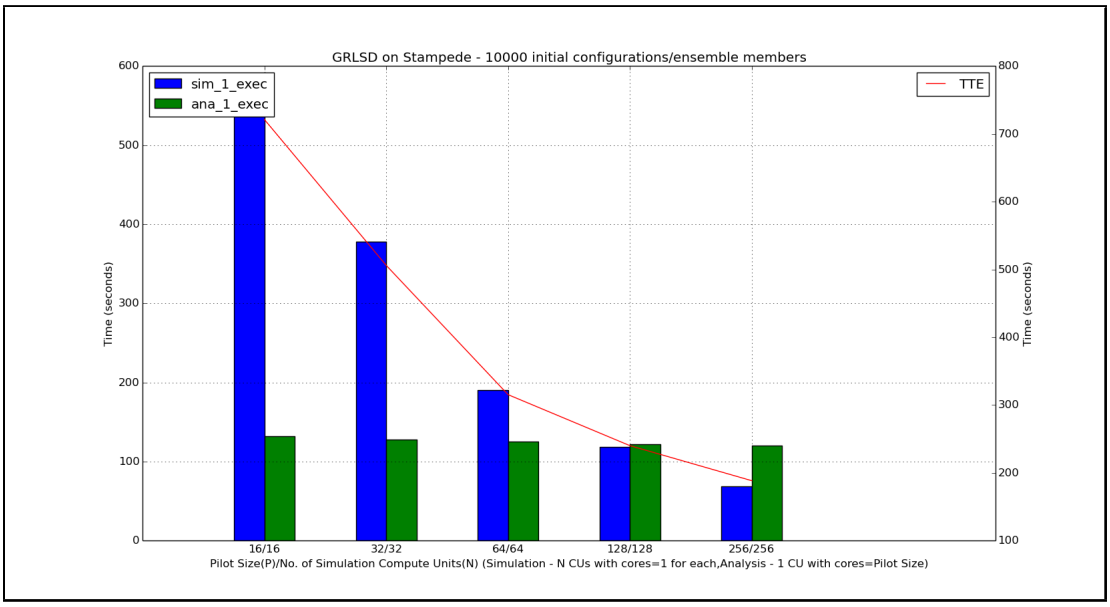

Figure 5.2: Gromacs-LSDMap : Exec time vs Pilot Size

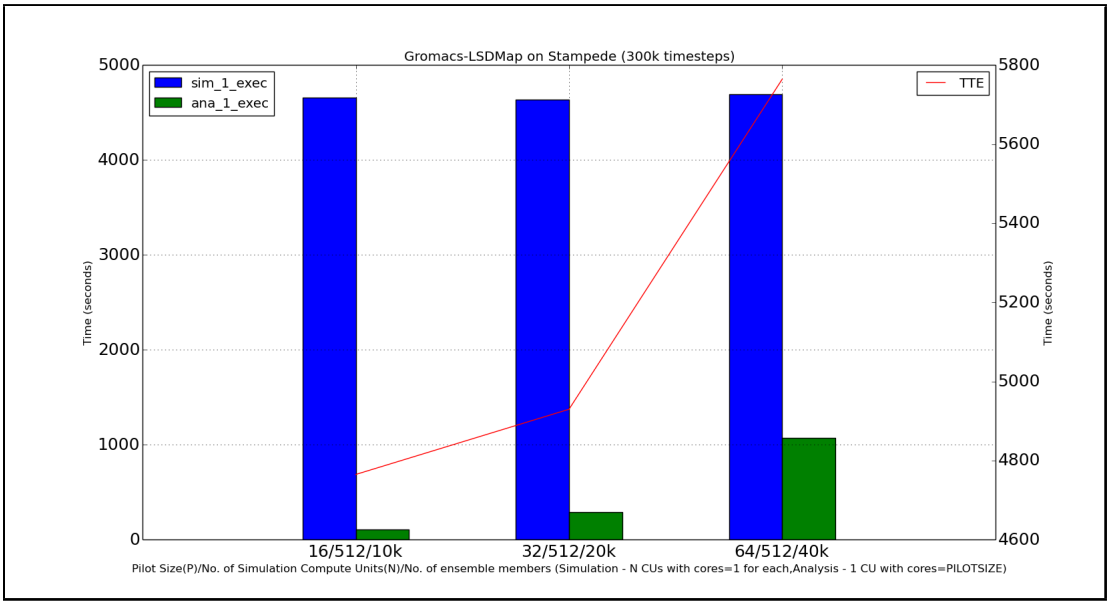

Figure 5.3: Gromacs-LSDMap : Exec time vs Pilot Size and Input Size

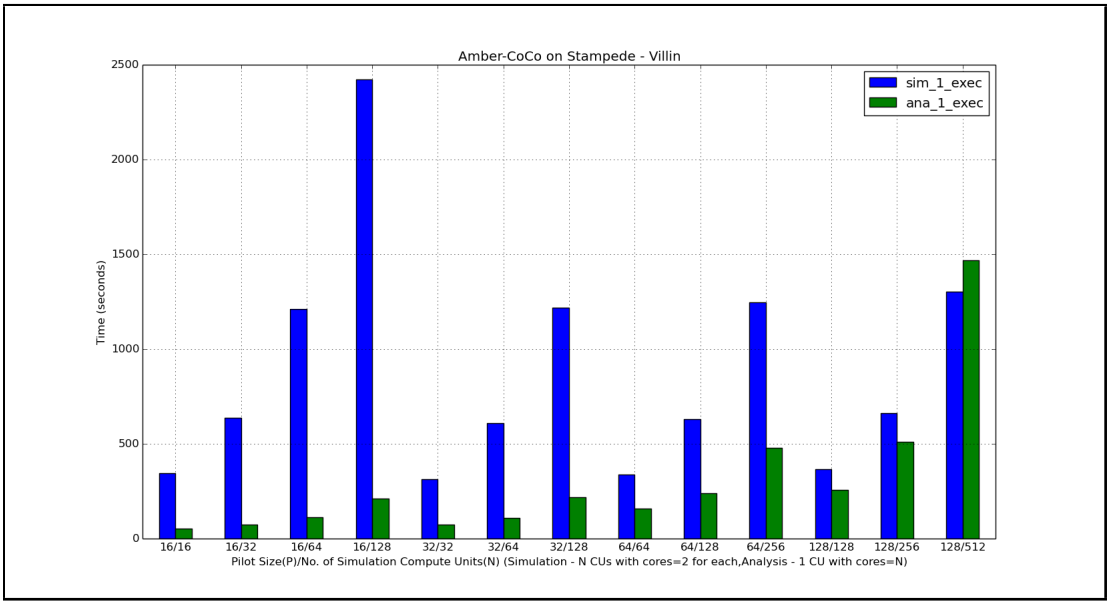

Figure 5.4: Amber-CoCo : Exec time vs Pilot Size/Compute Units

Chapter 6

Conclusion and Future Work

In this thesis, we presented a Framework to enable large scale ensemble based simulations, EnsembleMD Toolkit. We began by understanding the requirements of the science community, 1) to run large number of short simulations on a Distributed Computing Infrastructure, 2) to support, amongst others, a pattern which follows a Simulation Analysis Loop. We addressed the topic of task based parallelism as a promising alternative to conventional methods of job submission and execution. We use RADICAL Pilot to implement the task based parallelism principle. RADICAL Pilot enables us to Pilot Job frameworks which allow us to decouple resource management from workload management. RADICAL Pilot lays over the SAGA Layer which provides an interface to enable interoperability thus enabling scientists to use several Distributed Computing Infrastructures. Using Ensemble MD, we provide the bare essential building blocks required to develop a Molecular Dynamics application, not by the developer, but the scientists themselves. In order to minimize the overhead of development, resource and kernel level abstractions have been provided in Ensemble MD as well. Resource level abstractions have been discussed as part of RADICAL Pilot, whereas kernel level abstractions are provided in terms of Kernel plugins which implements the required environment setups to use the application on the specific resource. Easy to use and follow data transfer methods are provided in Ensemble MD to assist the scientists. Lastly, scalability tests, strong and weak scaling tests, have been performed on the Ensemble MD Toolkit Simulation Analysis Loop pattern and analyzed.

There is a lot of scope for future work in Ensemble MD toolkit. One of the aims is to implement a tool level abstraction, much similar to Kernel level abstractions, where

the tool specific argument methods can be removed from the user interface thereby making switches from one MD tool to another quite easy. Another direction of future work is to enable support for multiple Pilots so that execution can be performed over multiple heterogeneous machines simultaneously. Currently, the execution contexts are static and work over pre-known variables, the next versions of EnMD will concentrate on developing Dynamic Execution Contexts which can vary Pilot and Compute Unit specifications during runtime as to minimize the Time to Completion.

References

[1] Cameron Abrams and Giovanni Bussi. Enhanced sampling in molecular dynamics using metadynamics, replica-exchange, and temperature-acceleration. *Entropy*, 16(1):163–199, 2013.

[2] The RADICAL Group at Rutgers University. Ensemble MD Toolkit 0.2. `http://radicalensemblemd.readthedocs.org/en/0.2/index.html`, 2014.

[3] The RADICAL Group at Rutgers University. Ensemble MD Toolkit 0.2: The Patterns. `http://radicalensemblemd.readthedocs.org/en/0.2/the_patterns.html`, 2014.

[4] The RADICAL Group at Rutgers University. RADICAL Pilot 0.23. `http://radicalpilot.readthedocs.org/en/latest/intro.html`, 2014.

[5] The RADICAL Group at Rutgers University. RADICAL Pilot 0.23 : Getting Started. `http://radicalpilot.readthedocs.org/en/latest/examples/gettingstarted.html`, 2014.

[6] Texas Advanced Computing Center. Stampede User Guide. `https://portal.tacc.utexas.edu/user-guides/stampede`, 2011.

[7] Edinburgh Parallel Computing Centre. Archer User Guide. `http://www.archer.ac.uk/documentation/`.

[8] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Gregor Von Laszewski, Craig Lee, Andre Merzky, Hrabri Rajic, and John Shalf. Saga: A simple api for grid applications. high-level application programming on the grid. *Computational Methods in Science and Technology*, 12(1):7–20, 2006.

[9] Charles A Laughton, Modesto Orozco, and Wim Vranken. Coco: a simple tool to enrich the representation of conformational variability in nmr structures. *Proteins: Structure, Function, and Bioinformatics*, 75(1):206–216, 2009.

[10] André Luckow, Lukasz Lacinski, and Shantenu Jha. Saga bigjob: An extensible and interoperable pilot-job abstraction for distributed applications and systems. pages 135–144, 2010.

[11] Andre Luckow, Mark Santcroos, Andre Merzky, Ole Weidner, Pradeep Mantha, and Shantenu Jha. P: a model of pilot-abstractions. pages 1–10, 2012.

[12] Christopher Moretti, Jared Bulosan, Douglas Thain, and Patrick J Flynn. All-pairs: An abstraction for data-intensive cloud computing. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–11. IEEE, 2008.

[13] Swiss Institute of BioInformatics. Theory of Molecular Dynamics Simulations. `http://www.ch.embnet.org/MD_tutorial/pages/MD.Part1.html`, 1999.

[14] Jordane Preto and Cecilia Clementi. Fast recovery of free energy landscapes via diffusion-map-directed molecular dynamics. *Physical Chemistry Chemical Physics*, 16(36):19181–19191, 2014.

[15] www.ambermd.org. [last accessed on 04-10-2015]. .

[16] www.gromacs.org. [last accessed on 04-10-2015]. .