Louisiana State University
LSU Digital Commons

LSU Master's Theses | Graduate School

2011

Efficient replica-exchange across distributed production infrastructure

Abhinav S. Thota
Louisiana State University and Agricultural and Mechanical College, athota1@tigers.lsu.edu

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_theses

Part of the Computer Sciences Commons

Recommended Citation
Thota, Abhinav S., "Efficient replica-exchange across distributed production infrastructure" (2011). *LSU Master's Theses*. 1456.
https://digitalcommons.lsu.edu/gradschool_theses/1456

This Thesis is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Master's Theses by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.

EFFICIENT REPLICA-EXCHANGE ACROSS DISTRIBUTED PRODUCTION INFRASTRUCTURE

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Systems Science

in

The Department of Computer Science

by
Abhinav Thota
B.E., Osmania University, Hyderabad, India, 2008
May 2011

Acknowledgements

First and foremost, I express my gratitude to my advisor, Dr. Shantenu Jha for giving me the opportunity to work under him. His continuous guidance, encouragement and patience helped me immensely in my research. I sincerely thank Dr. Andre Luckow upon whose previous work I continued to build on and for his support and involvement in this work. I thank the members of my committee, Dr. Gabrielle Allen and Dr. Supratik Mukhopadhyay for their valuable time. I would like to thank my collaborators and friends Sharath Maddineni, Dr. Joohyun Kim, Dr. Soon-Heum Ko and Dr. Nayong Kim for the excellent discussions over the time. I thank all the people in the SAGA-Devel team, who were always responsive and helpful. I thank LONI and Teragrid for the HPC resources I used. Lastly, I thank my parents for all the love, support and encouragement.

Table of Contents

Acknowledgments . . . . **ii**

List of Tables . . . . **v**

List of Figures . . . . **vi**

Abstract . . . . **viii**

1 Introduction **1**

2 Background and Related Work **4**
- 2.1 Background 4
- 2.2 Related Work 4

3 Replica-Exchange Algorithms **6**
- 3.1 Mathematical Model 6
- 3.2 Synchronous Replica-Exchange 7
- 3.3 Asynchronous Replica Exchange 8

4 Replica-Exchange Framework **9**
- 4.1 SAGA - A Simple API for Grid Applications 9
- 4.2 SAGA BigJob - A Pilot-job Framework 10
- 4.3 Replica-Exchange Manager 11
  - 4.3.1 Centralized RE-Manager 11
  - 4.3.2 Decentralized RE-Manager and the Replica-Agent 12

5 Characterizing the Implementation and Performance Details of the RE Framework **15**
- 5.1 Synchronous RE 16
- 5.2 Asynchronous (decentralized) RE 17
- 5.3 Asynchronous (centralized) RE 19

6 Experiments and Results **21**
- 6.1 Scale-Up: Increasing the number of replicas on a single machine 21
  - 6.1.1 Experiments 21
  - 6.1.2 Results 22
- 6.2 Scale-Out: Increasing the number of machine used 24
  - 6.2.1 Experiments 24
  - 6.2.2 Results 25
- 6.3 Heterogeneity in Resources 30

6.4 Crosswalks and Temperature Mixing . . . 32
6.4.1 Crosswalks . . . 32
6.4.2 Temperature Mixing . . . 32
6.5 Experience using LONI and Teragrid Resources . . . 34
6.5.1 SAGA-BigJob Setup . . . 34
6.5.2 Running Simulations . . . 35

7 Conclusion and Future work . . . 37

Bibliography . . . 39

Vita . . . 42

List of Tables

5.1 Average values of terms in Eqn 3.2 for the three RE algorithms. T_{MD}=time replica takes to complete 500 time-steps; T_W=synchronization time; T_{EX}=time to make a pairwise exchange; T_{find}=time to find/lock a partner; T_{file}=time to write/transfer files; T_{state}=time to update states after exchange; T=total time-to-completion. 16

6.1 The table shows the terms that change when scaling-up and scaling-out. S = synchronous; AC = asynchronous (centralized); AD = asynchronous (decentralized); T_{MD} = time replica takes to complete 500 time-steps; T_W = synchronization time; T_{EX} = time to make a pairwise exchange; T_{find} = time to find/lock a partner; T_{file} = time to write/transfer files; T_{state} = time to update states after exchange; T = total time-to-completion. 24

List of Figures

4.1 **The RE Framework:** The framework consists of the RE-Manager and the SAGA BigJob framework. BigJob is used to efficiently execute replica sub-jobs. The RE-Manager manages replica sub-jobs and performs replica exchanges. 10

4.2 **Centralized Coordination:** Synchronous and asynchronous RE are implemented using the centralized coordination style. The RE Manager (master) manages the replicas and coordinates exchanges. It is centralized coordination, since the master controls everything. 12

4.3 **Decentralized Coordination:** Asynchronous RE is implemented using the decentralized coordination style also. Here, the master is only required for initially launching the replica-agents. The later coordination is done peer-to-peer via the Advert Service. 13

5.1 **Synchronous RE State Diagram:** This is a 3 state model. A replica is in `new` state when it is submitted but not yet started. Once the replica is started `running` and `done` states follow. The exchanges are attempted when *all* the replicas are in `done` state. 17

5.2 **Asynchronous (decentralized) RE State Diagram:** This is a four state model. The replica goes into the `running` state when it is started by the replica-agent. It is marked as `done` after it completes running. The state `pending` is assigned when an exchange is in progress and `complete` is assigned when the exchange is completed. 18

5.3 **Asynchronous (centralized) RE State Diagram:** This is a 3 state model. A replica is in `new` state when it is submitted but not yet started. Once the replica is started `running` and `done` states follow. The exchanges are attempted while a replica is `done` state. 20

6.1 **Scale-Up Performance for 4 to 256 Replicas:** The graph shows the runtimes for the different RE implementations. The ratio between the number of exchanges and number of replicas is kept constant. Each replica is assigned 16 processors and run 500 time-steps. The asynchronous decentralized RE implementation shows the best scaling behavior. The synchronous RE scales less well mainly due to the synchronization step, while the limitations of the single master, which becomes a bottleneck, effect both synchronous and asynchronous RE to some extent. 22

6.2 **Scale-out performance for 8, 16 and 32 replicas, synchronous:** The experiments were done on LONI resources and repeated at least 5 times. The error bars denote standard error. As the number of machines increases, the time-to-completion increases in general mainly due to higher exchange costs (T_{EX}) caused by e. g. remote file copies and additional synchronization costs. 26

6.3 **Scale-out performance for 8, 16 and 32 replicas, asynchronous (decentralized):** The experiments were done on LONI resources and repeated at least 5 times. The error bars denote standard error. As the number of machines increases, the time-to-completion remains constant as there are no remote file copies or synchronization costs. 27

6.4 **Scale-out performance for 8, 16 and 32 replicas, asynchronous (centralized):** The experiments were done on LONI resources and repeated at least 5 times. The error bars denote standard error. As the number of machines increases, the time-to-completion increases in general mainly due to higher exchange costs (T_{EX}) caused by e. g. remote file copies. 28

6.5 **Different replicas were launched with different number of processors:** The experiments were done on *QueenBee* and repeated 5 times. The error bars denote standard error. The heterogeneous experiments were done by launching 16 replicas with 16 processors and 16 replicas with 8 processors. In the homogeneous experiments all the replicas were launched with 16 processors. The synchronous RE is most adversely effected by the slower replicas. The other two at least take some advantage of the the faster replicas. 31

6.6 **Average number of unique exchanges and total exchanges per replica:** These exchanges were modeled according to the synchronous and asynchronous RE algorithms. The top two lines show the average number of total exchanges per replica for asynchronous and synchronous RE, respectively. It shows that asynchronous RE makes more number of exchanges per replica. The bottom two lines show the average number unique exchanges per replica. Given that the maximum possible exchanges is ≈ 42, for smaller N_R, both algorithms perform similarly. But with larger number of replicas, the asynchronous RE shows better temperature mixing/sampling. 33

Abstract

Replica-Exchange (RE) methods represent a class of algorithms that involve a large number of loosely-coupled ensembles and are used to understand physical phenomena – ranging from protein folding dynamics to binding affinity calculations. We develop a framework for RE that supports different replica pairing and coordination mechanisms, that can use a wide range of production cyberinfrastructure concurrently. Additionally, our framework uses a flexible pilot-job implementation, which enables effective resource allocation for multiple replicas. We characterize the performance of two different RE algorithms - synchronous and asynchronous - at unprecedented scales on production distributed infrastructure (Teragrid and LONI). The synchronous RE algorithm is implemented with a centralized master, while the asynchronous RE algorithm is implemented with both centralized and decentralized replica management schemes.

We evaluate the performance of the different algorithms and implementations when we scale-up the number of replicas (up to 256) on a single machine and when we scale-out across 2 and 4 machines. Both the synchronous and asynchronous algorithms perform similarly when the number of replicas is small. But as the number of replicas increase, in the synchronous RE, the synchronization cost increases the total time to completion. In the centralized asynchronous RE, the cost of managing many replicas in a centralized manner increases the time to completion but not as much as in the synchronous RE. The decentralized asynchronous RE scales much better with increasing number of replicas. When scaled-out across many machines, the performance of synchronous RE depends on whether the machines are homogeneous or heterogeneous. A heterogeneous infrastructure means increased synchronization costs. We also run tests to see if one of the algorithms is better suited to achieve more crosswalks and temperature mixing – better sampling.

Chapter 1

Introduction

Today there are many applications that are computationally intensive, requiring hundreds, if not thousands of processors. Common and not so common applications include weather forecasting, molecular modeling, simulations, banking, etc. One such application is *Replica-Exchange Molecular Dynamics (RE)* [1, 2]. The RE methods represent a class of algorithms that involve a large number of loosely-coupled ensembles, called replicas. By loosely-coupled, we mean that the replicas need to communicate with each other and are bound to one-another in some way. Here each replica could be a molecular-dynamics application. The replicas are essentially the same simulation, but with minor differences, such as temperature, etc. RE simulations are used to understand a range of physical phenomena – ranging from protein folding dynamics to binding affinity calculations.

In RE, each replica could be a molecular-dynamics application. RE is a bit more complicated than a straight-forward computationally intensive application, in that after each run the replicas communicate with other replicas and exchange information such as energy or temperature and restart. There could be tens or hundreds of coupled-ensembles in a simulation. Also, each MD application typically needs a lot of processing power.

To solve this kind of computationally intensive problems, researchers use powerful computers called supercomputers or high-performance computers (HPC). These machines are very powerful and expensive. Government, military and corporations commission special-purpose machines, but individual researchers usually cannot afford them. Individual researchers use shared resources, provided by universities and national scientific research grids such as the Teragrid [3], LONI [4], etc. The shared resources operate under certain policies and guidelines which the user needs to follow.

Another approach to solving computationally intensive problems can be distributed computing, where resources distributed physically or logically are leveraged to achieve performance. Distributed computing presents many challenges to the scientific community. The machines usually are not of the same architecture, there are issues of authentication, connectivity, communication and coordination.

With policies and guidelines in place, it is not trivial to develop and run a RE application on shared HPC machines , where there is a need to coordinate different replicas. Thus the design and development of most RE implementations [5] is influenced and constrained by the programming systems and the infrastructure it is developed against. Breaking this coupling between the development and the underlying infrastructure, to enable applications

to be flexible (across infrastructure), extensible (to new methods of communication and coordination) and scalable is an important design objective of distributed applications – both logically distributed and physically distributed.

If the implementation is not efficient, the replicas could end up waiting large amounts of time for other replicas to start, complete and make the exchange. Application formulations that are scalable while being flexible and extensible are better suited to using the diverse range of traditional and hybrid infrastructure (e.g., grid-cloud and heterogeneous resources). Along with application formulations that facilitate the flexible utilization of a range of infrastructure, it is imperative to have the correct runtime abstractions that support flexible deployment of these applications.

In support of flexible and scalable formulations of the RE class of algorithms, we develop a RE Framework that supports multiple formulations, is extensible to a broad range of infrastructure and as we shall show scales-up and scales-out.

Our RE Framework uses a flexible pilot-job implementation to support the execution of the ensembles. It supports scalable implementation of RE that can use a range of distributed infrastructure concurrently that supports different exchange coordination mechanisms (synchronous versus asynchronous), and thereby different variants of the RE algorithm. SAGA stands for Simple API for Grid Applications. SAGA is an API that provides the basic functionality, tools and frameworks required to build distributed applications, so as to be independent of the details of the underlying infrastructure. BigJob is a SAGA-based Pilot-Job framework [6] – which works across multiple, heterogeneous, distributed grid and cloud infrastructure [7]. Using a pilot-job helps avoid prior co-scheduling of resources for each application and allows for concurrent execution of the sub-jobs. Here jobs that are run under the pilot-job are referred to as sub-jobs. It should also be noted that because the ensembles are restarted after each run, a pilot-job framework helps avoid repeated waiting at the resource manager.

Our RE framework supports different variants of the RE algorithm. They involve different exchange coordination mechanisms - synchronous and asynchronous. In the synchronous RE the exchanges take place only after all the replicas finish running and in the asynchronous RE [8, 9], the exchanges can happen between any two replicas that have finished running. We compare the two algorithms under different circumstances.

The RE framework also includes different kinds of replica management schemes - centralized and decentralized. In the centralized scheme, a master controls all the replicas and the simulation. In the decentralized replica management, each replica is managed individually. The two implementations are compared to understand how they perform with increasing number of replicas and with increasing degree of distribution.

We characterize the performance of the two different RE algorithms - synchronous and asynchronous - at unprecedented scales on production distributed infrastructure. The synchronous RE algorithm is only implemented with a centralized master, while the asynchronous RE algorithm is implemented in both centralized and decentralized fashions. We evaluate the performance of the different algorithms and implementations when we scale-up the number of replicas (up to 256) on a single machine and when we scale-out across 2 and 4 machines. We also introduce heterogeneity artificially in the resources by assigning different number of processors to different groups of replicas and evaluated the performance of the different algorithms.

We also evaluate if one of the algorithms helps produce better scientific results. We modeled the algorithms to test if one of them makes more exchanges - unique and total - than the other algorithm. More exchanges would mean better sampling and consequently better results.

Both the synchronous and asynchronous algorithms perform similarly when the number of replicas is small. But as the number of replicas increase, in the synchronous RE, the synchronization cost increases the total time to completion. The asynchronous RE algorithm does not a problem with large number of replicas. In the centralized asynchronous RE, the cost of managing many replicas in a centralized manner increases the time to completion but not as much as in the synchronous RE. The decentralized asynchronous RE scales much better with increasing number of replicas.

When scaled-out across many machines, the performance of synchronous RE depends on whether the machines are homogeneous or heterogeneous. A heterogeneous infrastructure would mean increased synchronization costs. When scaled-out across a homogeneous infrastructure, both the RE algorithms should perform similarly.

At this point, I would like to clarify my contribution to this work. The SAGA-BigJob Framework was previously developed in Ref. [6]. Synchronous RE was implemented in Ref [7]. I used this synchronous RE framework to run the simulations and collected the data. I developed asynchronous RE framework with centralized and decentralized replicas management schemes and used this framework to run the simulations and collect the data for various results. We have submitted this work for publication and the draft can be accessed here [10].

The rest of the thesis is organized as follows: in Chapter 2, we quickly go over the background and related work; in Chapter 3, we propose a mathematical model for RE and extend it to synchronous and asynchronous RE. In Chapter 4, we describe the SAGA BigJob framework and then we describe the RE framework which is based on it. In Chapter 5, we describe the implementation and performance of the RE framework and its variations. In Chapter 6, we describe the experiments and present the results and analysis. Finally, in Chapter 7, we conclude this work.

Chapter 2

Background and Related Work

2.1 Background

Typical MD simulations these days use 100 to 1000 processors to simulate $\approx$ 100 nanoseconds in less than a week. The largest simulations use special purpose machines like *Anton* [11] or intend to use supercomputers like *Blue Waters* [12]. But with shared production resources, access to the machines will be through resource managers and the number of processors a user can request is limited. For applications like RE, where all the replicas need to run concurrently, we need efficient methods to request and manage the resources. Any reasonably useful framework for RE should be able to leverage a wide variety of distributed and heterogeneous resources. It should be able to scale-up (retain performance when the number of replicas is increased on the same machine) and scale-out (retain performance when the number of distributed machines used is increased) reasonably well. The framework should not be limited to supporting a particular version of MD and should support multiple processor jobs. It should also be able to efficiently manage the periodic communication between replicas.

Distributed computing can be of two varieties: (i) distributed computing across desktop and CPU grids and (ii) distributed computing across high-performance computing (HPC) machines. Using large number of distributed computers across grids or large-scale HPC machines is called high-throughput computing (HTC). The traditional method of scaling out is to use large number of small grid resources. A lot of applications use this method to scale-out [13, 14] The other method is to scale-up and scale-out across HPC machines. We call this high-throughput of high performance computing. This method is not as commonly used as distributed computing across desktop or CPU grids, but would provide higher throughput.

2.2 Related Work

In this section, we list a variety of existing MD and RE applications of different sizes, some of which use production resources while others use desktop or CPU grids. We also present details of implementations which use private unshared resources at scale. But before that, we would like to highlight that our RE framework is infrastructure independent and can used across a variety of distributed environments such as the Teragrid [3] and LONI [4], can

be used to scale-up and scale-out across many machines.

Ref. [15] describes a simple implementation using the MPI [16] library to coordinate the exchanges. But it uses small local machines and does not take into consideration the queuing and other issues associated with a shared resource. Also, it cannot be used to run large scale simulations across many machines. Ref. [8] uses campus-wide desktop grids and Beowulf clusters [17] to run their simulations and the Salsa [18] framework for coordination and communication – this is an infrastructure specific framework.

Refs. [13, 14, 19] use desktop or CPU/GPU grids, which harness idle processing power from volunteer CPUs and gaming systems. The MD jobs in this case are more light weight in nature and can use only one processor. There are considerable communications costs associated as the data needs to be transferred to and from the processing element via the internet. Ref. [20] uses a modified synchronous RE algorithm, as the synchronous RE algorithm is not suitable for running on distributed and heterogeneous systems (distributed CPU grids). They launch multiple replicas with the same temperatures to avoid the synchronization step. This approach is scalable but the MD jobs can only use one processor at a time. This limits the throughput that is achievable, while the resource availability is uncertain.

Another method is using shared high-throughput grid infrastructure, such as EGEE [21], where the machines are highly distributed in nature and the jobs use only one processor. Ref. [22, 23] make use of the WISDOM [24] project to utilize EGEE infrastructure. But this is an infrastructure specific implementation tied to EGEE. Another approach is to buy the machines or use unshared resources, which may not be feasible for individual researchers. Ref. [25] describes the implementation of large scale replica exchange - with 256 replicas using 8192 cores on an unshared resource. The communication during exchanges is carried out using the MPI library.

Each of the above approaches have their advantages and disadvantages, but none if them addresses the issues involved in leveraging shared HPC resources. Our framework is an attempt to provide an infrastructure independent solution to utilize distributed HPC resources at scale.

Chapter 3

Replica-Exchange Algorithms

The RE class of algorithms involve the concurrent execution of *replicas* - which are defined as instances of essentially similar simulations but with minor differences, such as the defining temperature of the replica. These replicas are loosely-coupled, in that there are infrequent exchanges between pairs of replicas. In addition to the frequency of communication between the replicas being low (relative to that of within a single replica), the amount of information/-data exchanged between replicas is small (a couple of bytes) compared to the simulation's operating data-set size. In this chapter we try to develop a mathematical model for RE, first based on an ideal scenario, where we assume zero communication and coordination costs. We then extend it to consider a typical implementation of RE, where we take into consideration the different costs that effect and limit the performance of the RE application. We then apply this model to synchronous and asynchronous RE algorithms. We use this model through out this report to analyze the performance of the different implementations.

3.1 Mathematical Model

In this section we develop a mathematical model that captures the primary components that make up the total runtime of a RE experiment. In an ideal scenario, the total time-to-completion of an experiment would be equal to the concurrent runtime of the ensemble of replicas and there would be no overhead associated with the coordination of the replicas. We also assume that the resources, network and other components are homogeneous. If an ensemble contains N_R replicas and the total number of pairwise exchanges is defined to be N_X and the runtime of a replica to complete a defined number of time steps which is defined to be T_{MD}, then the total time-to-completion of an experiment T would be:

$$T = \frac{1}{p} \times (T_{MD} \times \frac{N_X}{\frac{N_R}{2}}) \tag{3.1}$$

where p is defined as the probability of a successful exchange – not all exchanges are accepted/successful. The decision to accept an exchange or not is made using the Metropolis scheme [26], which is a well known way of accepting a proposed change of state, even when energetically not favorable.

$\frac{N_R}{2}$ is the maximum number of independent exchange events that can occur concurrently for N_R replicas. $\frac{N_X}{\frac{N_R}{2}}$ is the number of ensemble runs needed to complete N_X exchanges. For example, if N_R is 4 and N_X is 16, 2 exchanges are possible after an ensemble run and 8 such runs are required to complete 16 exchanges.

However, any RE production run will entail some overhead of job-submission, termination, coordinating the replicas and exchanges, etc. Therefore, we propose that the time to complete a RE simulation of N_X exchanges successfully is:

$$T = \frac{1}{p} \times [(T_{MD} \times \frac{N_X}{\frac{N_R}{2}}) + (T_{EX} + T_W) \times \frac{N_X}{\eta}] \tag{3.2}$$

where T_{EX} is the time it takes to perform a *pairwise* exchange. It includes the following components: (i) the time to find a partner (T_{find}), (ii) the time to exchange/write/transfer files (T_{file}) and (iii) the time to manage state updates (e.g., in a central database) and conduct book-keeping operations associated with replica pairing/exchanging (T_{state}). T_{state} may arise due to different reasons – which may be related to implementation of the algorithms. In summary, $T_{EX} = T_{find} + T_{file} + T_{state}$. The component T_W is the time spent waiting by a replica to synchronize with other replicas that are not ready to make exchanges, e.g., maybe still running. η is the number of concurrent exchanges taking place. At maximum η is $\frac{N_R}{2}$; it can however be lower if exchanges are not conducted concurrently. It can be seen from Equation 3.2 that as T_{MD} increases, the cost of coordination becomes less relevant. In this work, we used a fixed value of p (=1) when running simulations to characterize the performance and used a value of p=$\frac{1}{3}$ when running simulations to model the degree of mixing of the temperatures in the ensemble.

We will explain how the terms in Equation 3.2 vary in the synchronous and asynchronous RE algorithms in the following two sections.

3.2 Synchronous Replica-Exchange

Traditionally, RE algorithms have been implemented such that the exchanges have been synchronous. If the number of replicas is N_R, a *fixed* number ($\frac{N_R}{2}$) of replica pairs are generated. When *all* the replicas in the ensemble reach a pre-defined state (e.g., the Molecular Dynamics (MD) simulation completes a defined number of steps), an exchange of temperatures between fixed pairs of replicas with neighboring temperatures is attempted using the Metropolis scheme. If the exchange attempt is successful, parameters such as the temperature are swapped. After all the possible exchanges are completed, the replicas are restarted.

For the synchronous RE formulation, all replicas must reach a pre-determined state before exchanges are performed. T_W is the time spent waiting for all the replicas in the ensemble to reach the same state. It is non-zero, as in the synchronous RE, all replicas need to finish running before any exchanges are attempted. This synchronization step has the potential to become a serious bottleneck when the ensemble contains large number of replicas. In any implementation, it is highly likely that the coordination of the synchronization step of a large number of replicas will be a costly operation.

In contrast to parameter sweep applications in which coordination commonly only occurs at the beginning and end – a pattern commonly referred to as scatter-gather – RE requires

periodic coordination. In the synchronous RE formulation, coordination occurs at defined intervals – such as when all the replicas have finished running.

A major limitation of this model is that the replicas are paired in fixed groups and thus exchanges take place between pre-determined pairs of replicas. As a consequence of replica pairs being fixed before an exchange, T_{find} is 0. But. this limits the number of possible exchange partners that are available for a given replica. This inhibits exchanges between replicas with non-nearest temperatures, and ultimately reduces the possibility of crosswalks – where a crosswalk is said to occur when a replica originally with a low temperature reaches the upper temperature range and then returns to the lower temperature range [8]. This also effects the number of possible exchange partners and in turn the total umber of exchanges that are possible.

In addition to limitations in modeling the physics, rigid replica-pairing is efficient only in homogeneous environments; for heterogeneous environments and systems, where resource availability and processor performance fluctuates, the need for synchronization leads to slow-down and inefficiencies. We show how these limitations are overcome in the asynchronous (exchange) formulations of RE.

3.3 Asynchronous Replica Exchange

In the asynchronous RE algorithm, a replica does not have to wait for *all* other replicas to reach a pre-determined state. An exchange can occur whenever a replica reaches a pre-determined state. The replica then attempts an exchange with any other available replica in the ensemble.

Thus, in the asynchronous algorithm, each replica after completing a run has to search and find a *new* partner, $T_{find} \neq 0$, however T_W is 0 because there is no synchronization involved. In other words, a reduction in *synchronization* (wait) times comes at the cost of increased *coordination* (replica pairing) costs. The specific value of the term T_{EX} (Equation 3.2) differs from the synchronous formulation.

The issue of *static* versus *dynamic* pairing in principle is independent of synchronous or asynchronous exchanges, in that one could have synchronized exchanges but with different replica every time; equivalently we could have fixed pairs, but no global synchronization, i.e., asynchronous exchange between fixed pairs. However, in this work, we will equate synchronous exchange with static pairing, and asynchronous exchanges with dynamic pairing.

We do not see an obvious performance bottleneck in the asynchronous RE. T_{EX} is inherently dependent on the implementation and thus with an efficient implementation we could see a good performance with the asynchronous RE even with large number of replicas (N_R).

Chapter 4

Replica-Exchange Framework

An important motivation for this work is the design and implementation of a framework that provides the capability to support different RE algorithms and exchange mechanisms. In turn, the framework is an important contribution of this work. The framework is independent of the underlying infrastructure and thus supports the use of multiple heterogeneous resources – which are typically made available to the end-users.

The RE Framework provides all the capabilities that we mention in Chapter 2. It provides the capability to efficiently manage communications between the replicas, with a devoted and accessible key/value store. It can be used with any MD code or with another application in its place. It natively supports MPI [16] jobs. We also implement centralized and decentralized management of replicas for asynchronous RE, so as to take full advantage of the asynchronous algorithm.

It is useful to highlight that we differ from other RE implementations (e.g. Ref. [8]) in that we use *production-grade* national and regional cyberinfrastructure, such as the TeraGrid [3] and LONI [4]. We use general purpose tooling and standard capabilities that are available on these production infrastructure.

In this chapter we outline the architecture, implementation and the basic performance characteristics of the RE framework when used to implement the different RE algorithms (synchronous and asynchronous) and replica management schemes (centralized and decentralized). The RE Framework consists of the RE-Manager and the SAGA BigJob framework [6] – which is a pilot-job framework. The BigJob framework is common for both synchronous and asynchronous RE. The RE-Manager's functionality varies based on the replica management schemes – centralized or decentralized.

4.1 SAGA - A Simple API for Grid Applications

SAGA [27] is an open source, high-level API that provides the basic functionality required to build distributed applications, tools and frameworks so as to be independent of the details of the underlying infrastructure. SAGA can be used to provide simple access layers for distributed systems and abstractions for applications. It can be used to address the fundamental application design objectives of interoperability across different infrastructure, distributed scale-out, extensibility and adaptivity whilst preserving simplicity. SAGA is

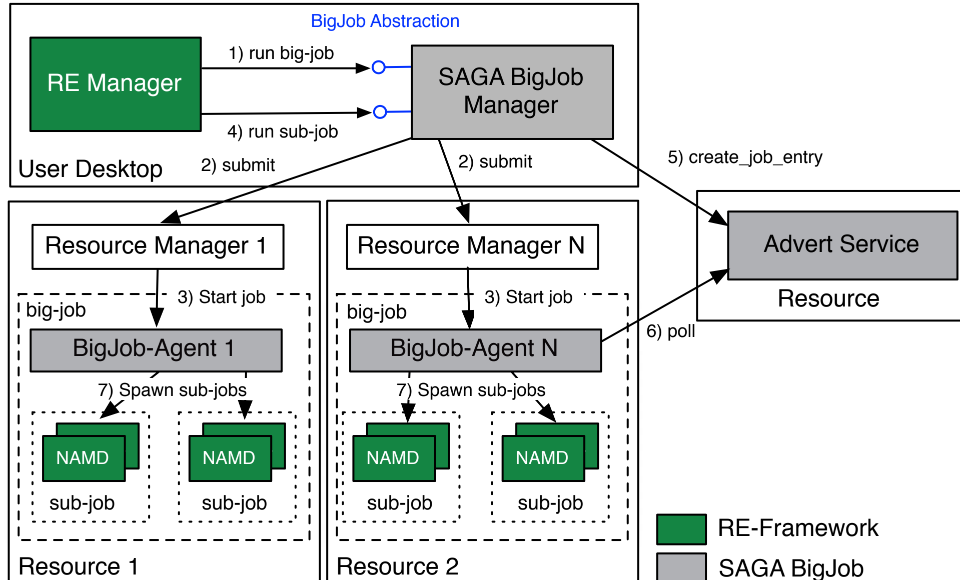

Figure 4.1: **The RE Framework:** The framework consists of the RE-Manager and the SAGA BigJob framework. BigJob is used to efficiently execute replica sub-jobs. The RE-Manager manages replica sub-jobs and performs replica exchanges.

an API standardization effort within the Open Grid Forum (OGF) [28], an international standards development body concerned primarily with standards for distributed computing.

SAGA provides a range of capabilities and we use only a small sub-set of those. Some of the functions we carry out using SAGA APIs are submitting jobs to the resource managers, monitoring the jobs, managing the communication between the replicas, staging of files to different locations, etc.

4.2 SAGA BigJob - A Pilot-job Framework

The usage of the SAGA-based pilot-job framework has been previously demonstrated in Ref. [6]. It is called the SAGA-BigJob Framework, and it has been used to run synchronous RE simulations across multiple, heterogeneous, distributed grid and cloud infrastructure [7]. Figure 4.1 shows the architecture of the RE Framework – comprised of the RE-Manager and SAGA BigJob framework.

The SAGA BigJob framework consists of three components: (i) the BigJob-Manager, (ii) the BigJob-Agent and (iii) the advert service which is a central key/value store used for communication between the BigJob-Manager and the BigJob-Agent.

The various tasks that are carried out using the SAGA APIs include file staging, job spawning and conducting the exchange attempts. Here we use the SAGA BigJob framework to efficiently request and manage computational resources for multiple replicas and it enables the use of a range of infrastructure. For example, we can submit multiple big-jobs on multiple

resources and manage them from one location.

The RE-Manager requests a defined number of big-jobs from the BigJob-Manager; for each big-job a regular batch job is submitted to the resource manager (step 1-3 in Figure 4.1). When the big-job becomes active, the BigJob-Manager can start to process sub-jobs. For each new sub-job an advert entry storing the description of the sub-job is created by the BigJob-Manager (step 4-5). The BigJob-Agent periodically polls for new jobs (step 6). If a new job is found and resources are available, the BigJob-Agent runs the job (step 7). If resources/processors are not available, the job is queued in the advert service. It is possible that there can be more sub-jobs than a big-job can accommodate. Further, the BigJob-Agent continuously monitors the running sub-jobs and updates the sub-job states in the advert service. Once a sub-job finishes running, the compute resources are freed and marked as available.

4.3 Replica-Exchange Manager

The RE-Manager is the master process which in addition to controlling the different components – SAGA BigJob framework, the individual replicas, etc., also defines and implements the coordination/exchange mechanism employed. The actual tasks that the RE-Manager performs depends not only on the RE algorithm employed, but also on the replica management scheme being supported. The RE-Manager supports two different replica management schemes: centralized and decentralized. In centralized, the RE-Manager manages all the replicas and performs the exchanges, while in the decentralized case, a *replica-agent* manages each replica individually as well making and performing exchange decisions.

As we will discuss in Chapter 5, the synchronous and asynchronous RE algorithms were implemented using the centralized RE-Manager. The decentralized replica management was used to support only the asynchronous RE algorithm. This is because in the synchronous algorithm, even with a decentralized replica management, all the replicas will have to synchronize with a master and this reduces the efficiency of a decentralized implementation.

4.3.1 Centralized RE-Manager

The control flow of a centralized replica management mechanism is shown in Figure 4.2. A replica can be in one of these three states: (i) `new` (submitted but not started), (ii) `running` and (iii) `done`. Once the big-job(s) is active and the replicas are `running`, the RE-Manager constantly queries the BigJob-Manager for the latest replica states. When the RE-Manager finds a replica that has finished running, it collects the energy and temperature of that replica by reading the output file.

We implemented both the synchronous and asynchronous RE using the centralized version of the framework. Even though both use the same framework, the implementation of the asynchronous RE-Manager is different from the synchronous RE-Manager.

In synchronous RE, once *all* the replicas have finished running, the RE-Manager performs the exchanges by swapping temperatures and writing new configuration files. The new configuration files are staged to the appropriate location. The RE-Manager then submits

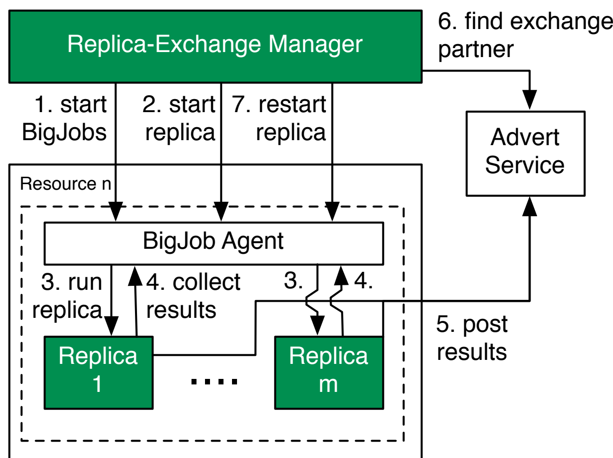

Figure 4.2: **Centralized Coordination:** Synchronous and asynchronous RE are implemented using the centralized coordination style. The RE Manager (master) manages the replicas and coordinates exchanges. It is centralized coordination, since the master controls everything.

the replicas for restarting, and the BigJob-Manager restarts them. The RE-Manager keeps count of the successful exchanges, until the required number of exchanges are done.

In asynchronous RE, the RE-Manager e. g. is not required to wait for *all* replicas to finish running before performing *any* exchanges. Whenever the asynchronous RE-Manager finds a replica that has finished running, it tries to find a partner to make an exchange. To find a partner, the RE-Manager goes over the list of all the replicas in the ensemble. If it finds a replica available, it attempts an exchange. If a replica is not found available, the RE-Manager queries the BigJob-Manager for the latest replica states and updates its local list. It then loops over the list to find a replica that has finished running and a partner to exchange with that replica. If the exchange is successful, the replicas are submitted to be restarted.

4.3.2 Decentralized RE-Manager and the Replica-Agent

In the decentralized implementation or in the decentralized management of replicas, in order to conduct the exchanges the RE-Manager launches multiple replica-agents (in lieu of replicas directly). Replica-agents then take control of replica start/re-start and exchange attempts. Figure 4.3 shows the control-flow of decentralized replica management mechanism. The replica-agents run the replicas upon launch; a list of nodes that is used to carry out the MD run is passed to the replica-agent as an argument at startup. The replica-agent constantly monitors the replica, and when the replica finishes, it updates the advert server with the current state of that replica. It also reads the temperature and energy from the output files, and posts the values to the advert server. The RE-Manager is primarily responsible

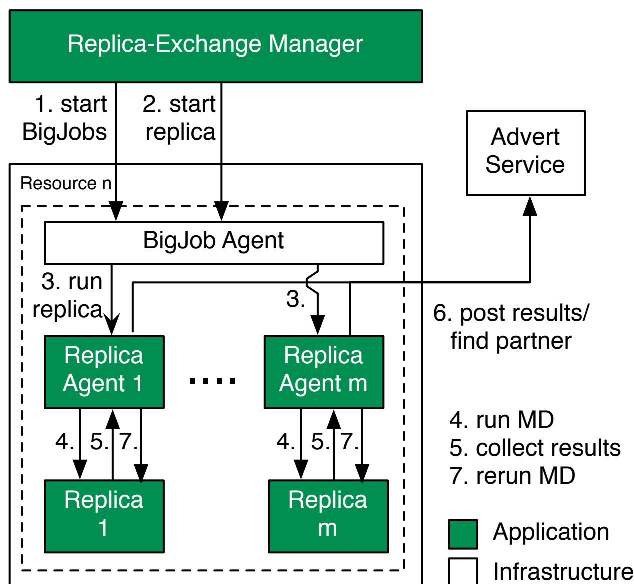

Figure 4.3: **Decentralized Coordination:** Asynchronous RE is implemented using the decentralized coordination style also. Here, the master is only required for initially launching the replica-agents. The later coordination is done peer-to-peer via the Advert Service.

for keeping a count of the number of exchanges performed; when the desired number of exchanges are done, the RE-Manager ends the experiment.

A replica can be in one of the these four states: in addition to states (i) `running`, (ii) `done`, which were found in the centralized case, there are also (iii) `pending` (for an exchange to complete) and (iv) `complete` (exchange has been completed) states. When a replica reaches a pre-determined state it (the replica-agent acts as the proxy for the replica) transitions from `running` to `done`. It initiates the search for a partner, and scans the list of replicas randomly, so as to avoid contention if multiple replicas have initiated a search for a partner.

If a replica finds a potential exchange partner, it must reverify that both states are still in the `done` state. This step is necessary to avoid concurrency issues that arise if multiple replicas attempt to exchange with the same partner. If both replicas are still in `done` state, the exchange can proceed and their states are set to `pending`. If the state of any replica has already been changed (to `pending`), the exchange attempt is aborted.

Additional replica states are needed to implement and maintain decentralized coordination/control. The states that a replica can be in the decentralized RE coordination mechanisms are different to those in centralized mechanism. In the centralized implementation, there is no `complete` state and in the decentralized implementation there is no `new` state. There is no `new` state in the decentralized implementation because the replica-agent starts running only after the processors become available and the replica is started and put directly in the `running` state.

After the exchange is performed (the temperature of both replicas in the advert server has been changed), their states are set as `complete`. Here the state `complete` is a marker which tells the replica-agent that the exchange has been made, the configuration files are in place and to restart the replica. The associated replica-agents write new configuration files

with updated temperatures, and restart their replicas. The replica-agent that initiated the exchange increments the exchange count. The RE-Manager constantly queries the advert server for the latest exchange count and when all exchanges have been made, it stops the experiment.

It should be noted that even in the decentralized implementation, the advert service is still centralized. Only the replica-management and exchange coordination are decentralized. We did not observe any increase in latency with respect to the increasing number of connections when accessing the advert service. But the communication times do vary depending on the physical distribution of the advert-service. Also, maintaining data consistency is not an issue because there is only one advert-server. Two write actions cannot be carried out simultaneously on a single value. It should also be noted that the exchanges are non-deterministic in nature. Firstly, because the replicas available for exchange are found in a random order and secondly, because in principle the Metropolis scheme [26] is used to match the replicas.

Chapter 5

Characterizing the Implementation and Performance Details of the RE Framework

In Chapter 4, we presented the basic components of the RE framework and discussed the control flow for the three implementations of the two algorithm using different exchange coordination mechanisms. In this chapter we provide more details of the working of RE Framework. We also give a basic characterization of the performance of the RE Framework for the three formulations. The work presented in this chapter is necessary to build up an understanding of the different factors that effect the performance of the RE Framework. This chapter provides a detailed analysis of every action in a typical experiment using the RE Framework implementation and the cost of that action.

For the basic characterization, the following experimental configuration is used: (i) Infrastructure: Our experiments were performed on LONI and the Teragrid shared resource *QueenBee (QB)*. A highly scalable, parallel MD code – NAMD [29], was used to perform the simulations for each replica (although, it is important to mention that any other MD or Monte Carlo code could be used just as simply and effectively with the RE Framework). (ii) Replica-Exchange Configuration: The total number of replicas (N_R) in the ensemble is 32 and the total number of pairwise exchanges (N_X) is 128. The ensemble of replicas are run concurrently and 16 pairwise exchanges are possible after each run. Thus, each replica on average is restarted 7 times to complete 128 exchanges. Each replica is configured to run 500 time-steps and is allocated 16 processors. One big-job of size 512 processors is requested. On average each 500 time-step run takes 71 s. For all implementations, in the event of a successful exchange, jobs are restarted with new temperature values. In the case of an unsuccessful exchange, jobs are restarted without exchanging the configuration. (iii) The physical system that we use as benchmark is the Hepatitis-C Virus that was examined in [7]. It is to be noted that in actual biological science simulations, the number of time-steps between exchanges is often higher than 500. Thus, the amount of computation performed in between exchanges is typically higher, lowering the cost and relevance of performing the exchange.

Each production run was repeated multiple times (≈ 10) and the start time of each run is measured only after the big-job becomes active. As we are interested in understanding the scale-up and scale-out properties of synchronous and asynchronous RE, we do not con-

	synchronous	**asynchronous (decentralized)**	**asynchronous (centralized)**
T_{MD}	71 s	71 s	71 s
T_W	2.8 s	0 s	0 s
T_{EX}	0.6 s	7.9 s	1.9 s
T_{find}	0 s	7.1 s	1.3 s
T_{file}	0.4 s	0.6 s	0.4 s
T_{state}	0.2 s	0.2 s	0.2 s
T	**1003 s**	**631 s**	**811 s**

Table 5.1: Average values of terms in Eqn 3.2 for the three RE algorithms. T_{MD}=time replica takes to complete 500 time-steps; T_W=synchronization time; T_{EX}=time to make a pairwise exchange; T_{find}=time to find/lock a partner; T_{file}=time to write/transfer files; T_{state}=time to update states after exchange; T=total time-to-completion.

sider queue wait-times. As explained in Section 3.1, the relative performance of the RE implementations is determined by the waiting time (T_W) and the time for conducting the exchange (T_{EX}). In the following sections, we analyze the average values for T_W and T_{EX} for each replica pair. Table 5.1 summarizes the results. We will discuss the different RE implementations in the following sections. The source-code for implementing the different RE algorithms using the RE Framework is available at: https://svn.cct.lsu.edu/repos/saga-projects/applications/async-re/.

5.1 Synchronous RE

Synchronous RE is implemented using the centralized replica management scheme. Figure 5.1 shows the state changes involved in the synchronous RE implementation. Since this is a centralized implementation, the RE-Manager manages everything. There are only three states a replica can be in: (i) `new` (submitted but not started), (ii) `running` and (iii) `done`. Initially, when the big-job becomes active all the replicas are marked as `new`, followed by `running` and `done` states. When all the replicas have finished running and after all exchanges are complete, the replicas are resubmitted and the process repeats.

In a homogeneous environment the waiting time T_W for the synchronous RE implementation is primarily determined by the fact that the RE-Manager is only able to process one replica at a time, i. e. there is generally a delay between the startup of one replica and the next. Since the post-processing of the replica ensemble takes longer than the delay between the start-up of first and last replicas, the post-processing time determines the overall time spent waiting (T_W) for other replicas. Post-processing involves various state updates as well as the stage-out of the output file and requires on average 1.4 s per replica. Thus T_W, which is defined for a pair of replicas, is 2.8 s. For an ensemble of 32 replicas (with 16 pairs) the delay between the first and last replica transitioning to `done` state adds up to 44.8 s.

T_{EX} comprises of three sub-components: T_{find}, T_{file} and T_{state}. In synchronous RE, T_{find} is 0 s due to the fact that there is fixed replica pairing. The updating and stage-out of the configuration files has been observed to take approximately 0.2 s per replica. Thus, T_{file} is

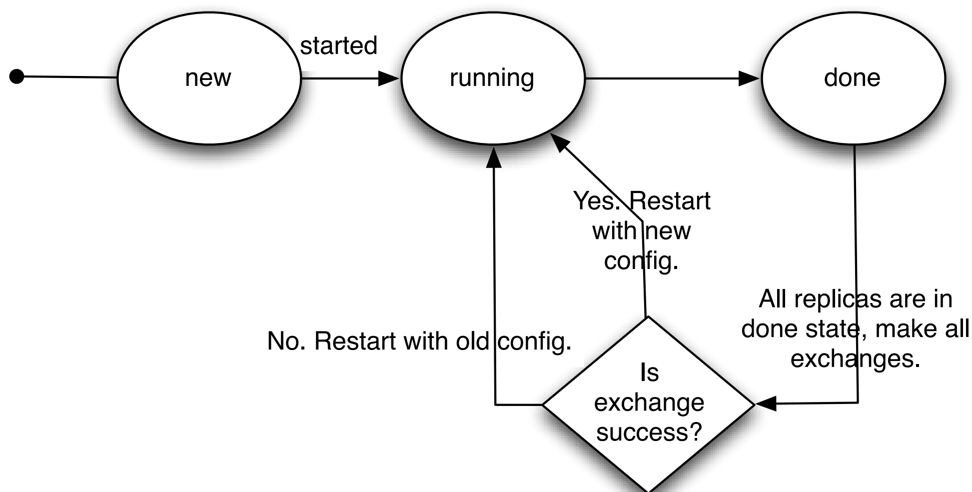

Figure 5.1: **Synchronous RE State Diagram:** This is a 3 state model. A replica is in `new` state when it is submitted but not yet started. Once the replica is started `running` and `done` states follow. The exchanges are attempted when *all* the replicas are in `done` state.

0.4 s per replica pair (the transfer of the input files is done sequentially). T_{state} is the time required by the RE-Manager to post a job description to the advert service. On average T_{state} amounts to 0.1 s per replica, i. e. 0.2 s per replica pair. Thus, T_{EX} is: $0 + 0.4 + 0.2 = 0.6$ s.

Although there are $\frac{N_R}{2}$ concurrent pairs, the exchanges at the RE-Manager are carried out sequentially; thus the effective number of concurrently exchanging pairs (η) is 1. Substituting the above values in equation 3.2, we get:

$$T = \frac{1}{p} \times [(71 \times \frac{128}{16}) + (0.6 + 2.8) \times \frac{128}{1}] = \frac{1}{p} \times 1003\, s. \tag{5.1}$$

5.2 Asynchronous (decentralized) RE

In this section, we describe the implementation of asynchronous RE with decentralized replica management. Figure 5.2 shows the state changes involved in the asynchronous (decentralized) RE implementation. Since this is a decentralized replica management scheme, the replica-agents negotiate exchanges between themselves. A replica can be in any one of these four states: (i) `running`, (ii) `done`, (iii) `pending` (for an exchange) and (iv) `complete` (exchange has been completed) states. The replica directly goes in to the `running` state when the replica-agent starts a replica. When it completes the run it goes into `done` state. Then the replica-agent looks for partners and if it finds a partner in `done` state, it once again reverifies the states of *both* replicas. If both the replicas are still in `done` state, both the replicas states are updated to `pending`. After the exchange is completed, both replica states are updated to `completed`, which is a signal for a replica-agent to restart its replica. If, at anytime in between, the replica-agent finds that the replica state is `completed`, that means

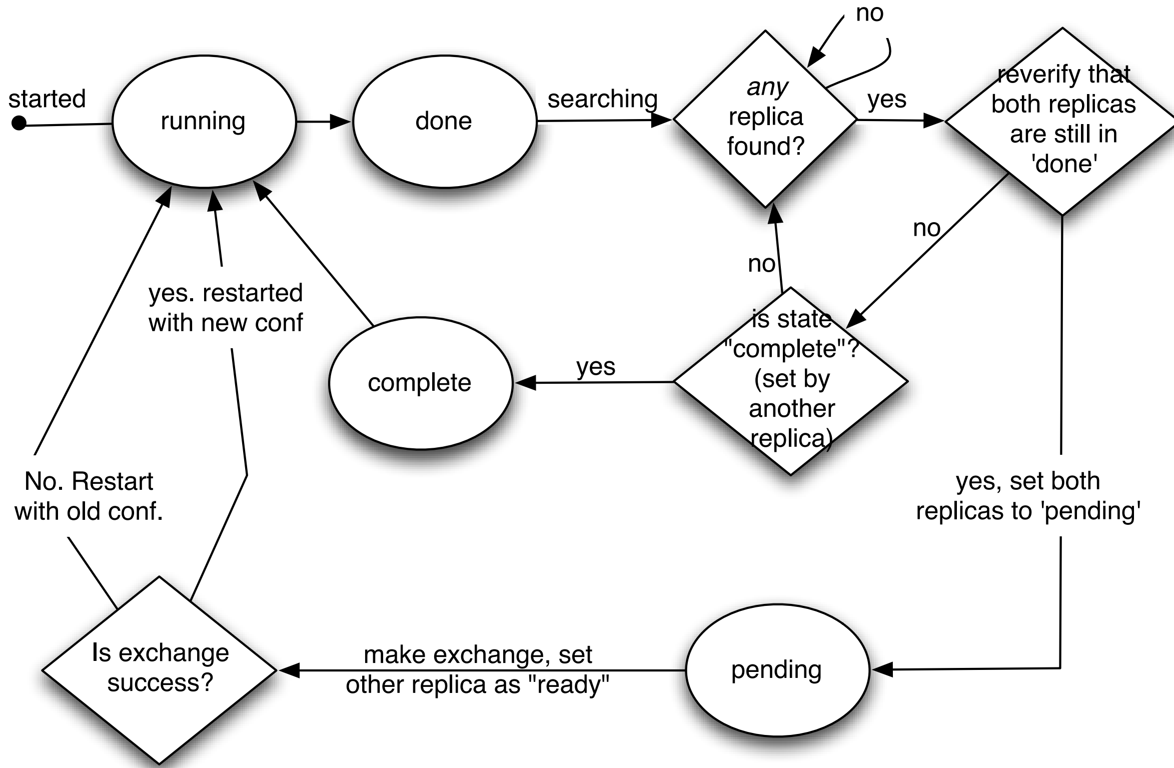

Figure 5.2: **Asynchronous (decentralized) RE State Diagram:** This is a four state model. The replica goes into the `running` state when it is started by the replica-agent. It is marked as `done` after it completes running. The state `pending` is assigned when an exchange is in progress and `complete` is assigned when the exchange is completed.

another replica-agent has made the exchange and updated the state.

A fundamental difference between the synchronous and asynchronous formulation of RE is in the synchronization barrier for the replicas before exchanges in the synchronous RE. As alluded to earlier, we equate fixed replica pairing with synchronous algorithm, and dynamic pairing with the asynchronous algorithm. Thus, $T_W = 0$ for asynchronous formulations, and this comes at the cost of a higher T_{find}, the time required to search and find a new exchange partner. The lack of a synchronization barrier also leads to a more involved implementation to dynamically pair replicas.

T_{find} comprises of two components: the pre-exchange management and the actual search. As described, in the decentralized implementation of asynchronous RE algorithm, the replicas are managed and exchanged by replica-agents. The following pre-exchange steps are necessary: after a replica completes its run, the agent updates the replica state at the advert service, which takes $\approx 0.1\,\text{s}$. The replica-agent then retrieves the energy and temperature from the output file requiring $\approx 2 \times 0.2\,\text{s}$. It then posts both values to the advert service, which takes $\approx 2 \times 0.1\,\text{s}$. The total time for these pre-exchange operations is: $0.1 + 0.2 \times 2 + 0.1 \times 2 = 0.7\,\text{s}$.

The other component of T_{find} is the actual search time. When searching for a replica randomly, on average it takes $\frac{N_R}{2}$ attempts to find a replica that is in the `done` state. However, finding a replica in the `done` state is not enough to complete an exchange attempt. Since there are several concurrent exchange attempts, contention and conflicts are common. Thus, a re-verification step must occur before the exchange is committed. During the re-verification, both replica-agents that are involved in an exchange ensure that there has been no change of state to either of the two replicas involved. Often, the re-verification step leads to an aborted exchange attempt. The exact number of necessary attempts is a random variable, determined by the number of replicas, the distribution of states and whether the attempt to find a replica is random or sequential. Empirical observations suggest that there are between 2 and 4 find and re-verification attempts before a successful exchange occurs. Also, replicas contend with each during this phase: the higher the numbers of replicas, the higher the contention. This means that T_{find} increases with increasing N_R. Specifically, for the scenario with 32 replicas and a random search, T_{find} is $\approx 2 \times N_R \times 0.1$ (where 0.1 is the typical time to set/get a value to/from the advert service), that is $6.4 + 0.7 = 7.1$ s.

T_{file} includes the time to the temperature from the advert service and the time to write a new configuration file: T_{file} is $0.1 \times 4 + 0.2 = 0.6$ s. T_{state} is the time required to update the state in the advert service, which is $0.2\,s$. In sum, T_{EX} is $7.1 + 0.6 + 0.2 = 7.9\,s$. It should be noted that T_{EX} is highly dependent on the actual implementation. While the current implementation is kept simple on purpose, this value can be improved in a more sophisticated implementation.

This being a decentralized implementation, there are $\frac{N_R}{2}$ concurrent exchanges and η is 16. Substituting the above values in equation 3.2, we get:

$$T = \frac{1}{p} \times (71 \times \frac{128}{16}) + (0 + 7.9) \times \frac{128}{16} = \frac{1}{p} \times 631\ s. \tag{5.2}$$

5.3 Asynchronous (centralized) RE

In this section we describe the implementation of the asynchronous RE with centralized replica mangement. Figure 5.3 shows the state changes involved in the asynchronous RE using a centralized master. This is similar to the synchronous RE implementation, which also features centralized replica management. This is again a three state model, where the replica can be in one of the following states: (i) `new`, (ii) `running` and (iii) `done`. The only difference to the synchronous RE implementation is that in this case, the RE-Manager attempts to find partners and make exchanges whenever it finds a replica in `done` state.

As in the decentralized implementation, the asynchronous centralized RE formulation does not require synchronization between *all* replicas in order to make pair-wise exchanges. Thus, $T_W = 0$ by definition. Using a centralized replica management mechanism, the time to find an exchange partner (T_{find}) can be reduced; however, this comes at a tradeoff that there is some contention at the master in the form of increased workload – as in managing the replicas. Also, we observed some delays at the BigJob-Agent during the startup of the sub-jobs mainly due to the fact that the BigJob-Agent is single-threaded and is busy processing other replicas after their termination. Specifically, the time-to-submit a replica-pair to the BigJob-Manager, i. e. two replica sub-jobs, is on average 1.1 s for the asynchronous

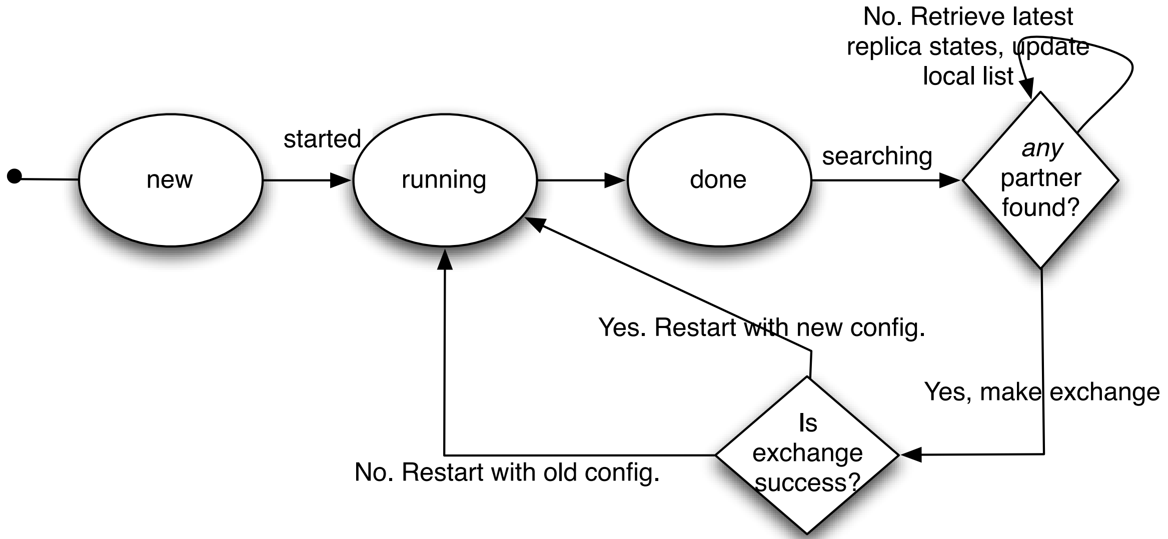

Figure 5.3: **Asynchronous (centralized) RE State Diagram:** This is a 3 state model. A replica is in `new` state when it is submitted but not yet started. Once the replica is started `running` and `done` states follow. The exchanges are attempted while a replica is `done` state.

(centralized) formulation. This is 0.5 s longer than in cases without contention at the BigJob-Agent – in such cases the submission of two sub-jobs requires only 0.6 s.

In contrast to the synchronous case, T_{EX} for the asynchronous centralized case has a T_{find} component since the replicas are dynamically paired and not statically fixed. T_{find} depends on the overall number of replicas (N_R), which determines the number of records the RE-Manager has to scan to find an available replica. If random search is used, the RE-Manager must search through $\frac{N_R}{2}$ replicas on average, before it finds a partner. Although the search for a replica is random like in the decentralized implementation, there is no need to reverify, as due to centralized control there are no conflicts in replica pairing, i.e., exchanges are made by the RE-Manager and only one exchange takes place at a time. An advert query for a replica state takes 0.01 s. Note this is a factor of 10 less than the decentralized implementation. This is because in the centralized case, all the values are located in the same advert directory; where as in the decentralized case, each replica's information is located in a distinct advert directory, so as to enable other replica-agents to find it. For 32 replicas, the RE-Manager requests on average 16 other replica states before it finds a partner; thus, T_{find} is in total $0.01 \times 16 + 1.1 = 1.3\,s$. Both T_{file} and T_{state} are same as in the synchronous case, i. e. T_{EX} is thus: $1.3 + 0.4 + 0.2 = 1.9\,s$.

As in the synchronous RE, the effective number of concurrently exchanging pairs is 1 due to the fact that exchanges are sequentially carried out by the RE-Manager. Substituting above values in equation 3.2, we get:

$$T = \frac{1}{p} \times [(71 \times \frac{128}{16}) + (0 + 1.9) \times \frac{128}{1}] = \frac{1}{p} \times 811\ s. \tag{5.3}$$

Chapter 6

Experiments and Results

In this chapter we discuss the experiments performed and analyze their results. To evaluate the scaling properties of the different RE implementations, we conducted several experiments on Teragrid and LONI resources. We initially increased the number of replicas on a single machine ("scale-up"); for the asynchronous-centralized implementations we varied the number of distributed machines used ("scale-out") for different replica counts. For both scale-up and scale-out experiments we find that even though implementation details are important, the asynchronous RE algorithms have better scaling properties than the synchronous algorithms. We also the address the effect of heterogeneity in resources and the crosswalk/temperature mixing performance of the different algorithms.

6.1 Scale-Up: Increasing the number of replicas on a single machine

6.1.1 Experiments

Given that RE applications are often used with large number of replicas, it is an important consideration that the given RE framework be able to cope with large number of replicas. It may be that one algorithm is better suited to scaling-up than the other, but the framework used must be able to provide a capability to run large number of replicas. In this section, we evaluate and compare the scale-up performance of the algorithms in conjunction with our framework.

To understand the scaling behavior of the different RE formulations, we analyze total time to completion (T) for different replica counts from 4, 8, 16, 32, 64, 128 and 256 making 16, 32, 64, 128, 256, 512 and 1024 exchanges, respectively. This fixes the ratio of the number of exchanges to the number of replicas ($\frac{N_X}{N_R}$) to 4. Each replica is configured to run 500 time-steps to complete a run and is allocated 16 processors. Experiments up to 64 replicas were performed on *QueenBee*, while experiments with 128 and 256 replicas were done on *Ranger*; this is because 128 replicas would require 2048 cores and *QueenBee* only allocates a maximum of 2048 processors per job request. As getting a 2048 processor allocation involves extremely long waiting times in the queue, *Ranger* was used. We have normalized the data to factor in the difference in performance of *Ranger* and *QueenBee*.

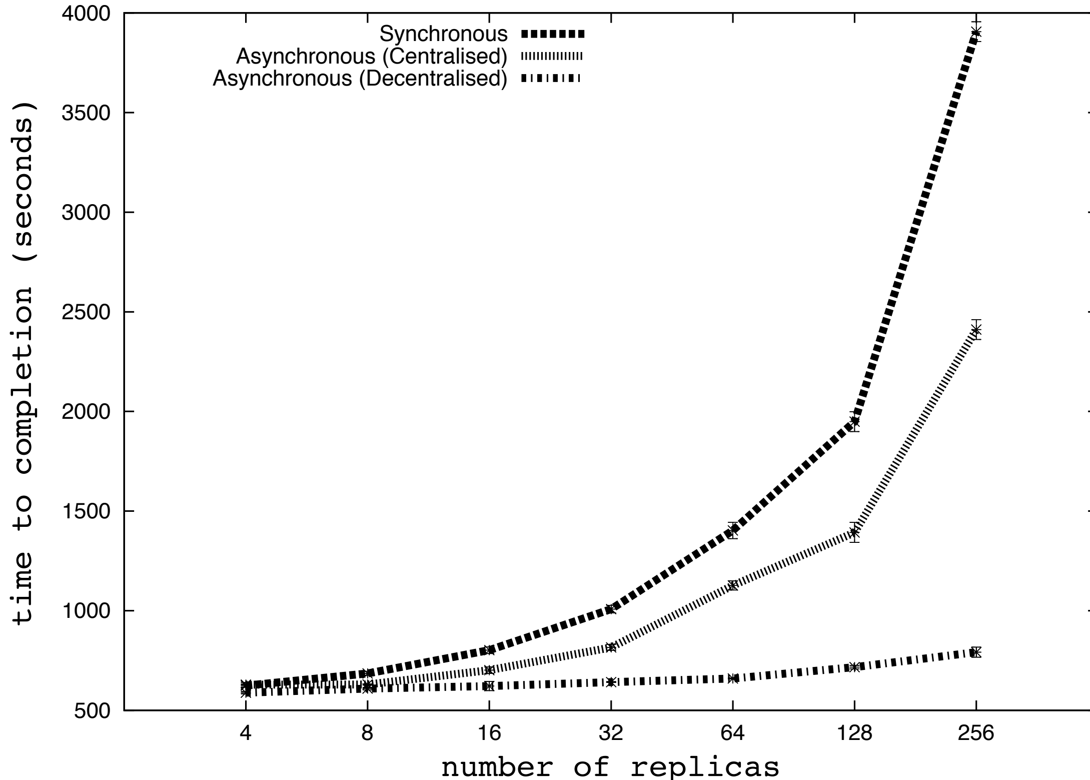

Figure 6.1: **Scale-Up Performance for 4 to 256 Replicas:** The graph shows the runtimes for the different RE implementations. The ratio between the number of exchanges and number of replicas is kept constant. Each replica is assigned 16 processors and run 500 time-steps. The asynchronous decentralized RE implementation shows the best scaling behavior. The synchronous RE scales less well mainly due to the synchronization step, while the limitations of the single master, which becomes a bottleneck, effect both synchronous and asynchronous RE to some extent.

6.1.2 Results

As a consequence of the ratio of the number of attempted exchanges to the number of replicas being constant, the number of times each replica is restarted to complete all exchanges remains constant; hence comparison between different cases will reveal differences in the coordination cost (T_W and T_{EX}). The increase, however, is not uniform across the three implementations: it is largest for synchronous RE, and the least for asynchronous (decentralized) RE. We analyzed T and the values of its components for 32 replicas in Section 5; we use this analysis as the basis to understand the scale-up behavior of the three formulations. The results of the experiments are shown in Figure 6.1. The increase in T as N_R increases is non-linear for synchronous and asynchronous (centralized) RE and linear for asynchronous (decentralized) RE. This is because there are issues that become bottlenecks for synchronous and and asynchronous (centralized) implementation – the synchronization step and the cen-

tral master respectively, whereas there are no such issues in the asynchronous (decentralized) implementation.

Synchronous RE

In the synchronous RE algorithm, there is an explicit synchronization of all replicas. As seen in Table 5.1, T_W is a major component of the T. As N_R increases, the number of replicas that need to be synchronized at every exchange step increases as well; consequently, the total coordination time at each exchange step increases. The cause for the increase in T_W is mainly algorithmic and arises due to the synchronization step and consequently contention at the centralized RE-Manager as well as the single-threaded BigJob-Agent process. Further, some implementation details, such as the fact that the current implementation of the RE-Manager is single-threaded, also effect T_W. If $N_R = 64$ and $N_X = 256$; using Equation 3.2, T_W can be approximated to $(0.6 + 2.8) \times 256 = 870.4\ s$. According to our model, T_W for $N_R = 32$ is 485 s, which is consistent with the change in T observed in the experiments depicted in Figure 6.1. A similar analysis was performed for different replica and exchange counts and the values obtained are in agreement with the empirical data in Figure 6.1.

Asynchronous RE - Decentralized

T_{EX} and in particular T_{find} effect the performance of the decentralized asynchronous RE. For asynchronous (decentralized) RE, T_W is 0. T_{find} has a strong N_R dependence. The finding operation involves several queries to the advert service; T_{find} can be approximated as follows: $2 \times N_R \times 0.1$ (where 0.1 is the typical time to set/get a value to/from the advert service). From Section 5.2 and Equation 5.2, it can be derived that for 64 replicas, T_{find} is approximately $0.7 + 2 \times 64 \times 0.1 = 13.5\,\text{s}$. The difference in the coordination cost $(T_{EX} \times \frac{N_X}{\frac{N_R}{2}})$ between a N_R value of 64 and 32 is $\approx 45\ s$, which accounts for the bulk of the increase in overall T. The observed performance can be attributed to both the algorithm, and to the specific implementation details, such as the usage of a central advert service.

Asynchronous RE - Centralized

In contrast to the asynchronous (decentralized) RE, T_{find} in asynchronous (centralized) RE is only weakly dependent on N_R; however, since the RE-Manager makes the exchanges serially, we still see an increase in the time-to-completion with increasing N_R. For $N_R = 64$ and $N_X = 256$, Equation 5.3 leads to a new coordination $(T_W + T_{EX})$ time of $(0 + 1.9) \times 256 = 486.4\,s$, up from 243.2 for $N_R = 32$. This change accounts for a large component of the difference in T as N_R goes from 32 replicas to 64 replicas, as shown in Figure 6.1. The actual difference observed in T from 32 to 64 replicas is 311 s. We have verified this scale-up model for different number of replicas and found it to be consistent with the data in the Figure 6.1.

Now that we have seen the factors that influence the performance, we show in Table 6.1 the terms that are effected when scaling-up and terms that are effected when scaling-out. In the next chapter we present and analyze the scale-out results.

	Scale-Up			Scale-Out		
	S	AC	AD	S	AC	AD
T_{MD}	No	No	No	May	May	May
T_W	Yes	No	No	Yes	No	No
T_{EX}	No	Yes	Yes	Yes	Yes	Yes
T_{find}	No	Yes	Yes	No	May	May
T_{file}	No	No	No	Yes	Yes	May
T_{state}	No	No	No	May	May	May

Table 6.1: The table shows the terms that change when scaling-up and scaling-out. S = synchronous; AC = asynchronous (centralized); AD = asynchronous (decentralized); T_{MD} = time replica takes to complete 500 time-steps; T_W = synchronization time; T_{EX} = time to make a pairwise exchange; T_{find} = time to find/lock a partner; T_{file} = time to write/transfer files; T_{state} = time to update states after exchange; T = total time-to-completion.

6.2 Scale-Out: Increasing the number of machine used

6.2.1 Experiments

We have stated earlier that a RE framework be able to utilize large number of distributed resources. Given that MD applications are computationally intensive, it is important that the RE framework provide the capability to the user to leverage distributed resources. The RE framework also needs to ensure that this capability is provided without seriously effecting the performance of the application.

To evaluate the performance of the different RE implementations when scaling out across many machines, we used the following TeraGrid and/or LONI resources: *QueenBee, Eric, Louie* and *Oliver*. The experiments were conducted with 8, 16 and 32 replicas making 32, 64 and 128 exchanges. These exchanges were repeated on 1, 2 and 4 machines while distributing the number of replicas equally on each machine. It is important to note that all experiments are conducted using four BigJobs, irrespective of the number of machines used; varying the number of BigJobs as well as the ratio of replicas per BigJob effects overall performance. Another important point to note is that only the experimental runs where all the four big-jobs become active within $30\,s$ of each other on submission to the resource manager are considered and included in the results. This is to remove the queue wait time from the equation and focus on the runtime. Each experiment is repeated 5 times.

Although the resources are homogenous and all belong to LONI, they are geographically distributed. This has a minor impact on the latency observed in the communication with the advert service (which runs on a desktop in LSU): From *QueenBee*, *Eric* and *Oliver* the creation of an advert entry takes 0.01 s, from *Louie* it requires 0.02 s. We are mentioning these communication times here, as they are the same in the synchronous and asynchronous experiments.

We analyze the performance of the different RE implementations using a varying number of distributed resources (1, 2 and 4 respectively) while keeping the number of replicas constant. After factoring in the performance advantage due to the additional big-jobs/BigJob-Agents, the analysis provided in 5.3 can be used to understand the behavior using the model

developed in Chapter 3.

6.2.2 Results

Synchronous

The results are shown in the Figure 6.2. We see that as we go from 1 to 2 and 4 machines, the time to completion increases. The difference in performance that we see as we go from 1 to 2 and 4 machines could be caused by: (a) increased synchronization costs (T_W), (b) additional time taken for remote file staging (T_{file}), and (c) higher communication costs to contact the advert server. But the machines we used are similar in nature and the NAMD runtimes are almost close to each other. But still, the NAMD runtimes are not constant and vary slightly from one to another. And with distributed machines, this effect is more pronounced than what we would observe if the experiments were conducted on a single machine. We do observe slight differences in the communication times with the advert server, but as mentioned earlier, these are negligible. Therefore, the factors effecting the T are the additional time to stage the configuration files and the additional cost added to the synchronization cost.

To copy the configuration file locally, it takes 0.01 s and to a remote machine it takes 0.38 s, approximately. The difference in time is $\approx$0.37 s. In the Figure 6.2, going in the 8 replica case, going from 1 machine to 2 machines, the T changes from 608 to 623 s. In the 2 machine case, the RE Manager would have to stage 4 configuration files remotely every exchange step. That would be $4 \times 8 = 32$ times in the whole simulation. That is an additional $32 \times 0.37 \approx 12$ s. And, 608+12= 620 s. A similar analysis holds for 16 and 32 replica configurations. We only see the effect additional synchronization and file transfer costs, but in heterogeneous environments, there could be other factors effecting the performance. Also, the times are too close to each other – closer than the 30 s fluctuation in the BigJob start-up times.

Asynchronous - Decentralized

Since this is the asynchronous algorithm, there are no synchronization costs. As the machines involved are approximately homogeneous in performance, we assume the same T_{MD} on all machines. Also, in the decentralized implementation, we take advantage of the replica-agents and transfer the temperatures through the advert service and avoid remote file transfers. Therefore, in principle, T should be generally be constant as we increase the number of machines to 2 and 4.

The results are shown in the Figure 6.3. Consider the 8 replica configuration. Within error bars, we don't observe a change in T, . The absolute changes in T are in the range of 10 to 20 s from 1 to 2 and 2 to 4 machines. Given that the experiments were repeated only 5 times and that the big-jobs became active within 30 s of each other, we can only say that there is no statistical difference between T for 1, 2 or 4 machines. This is aslo true for 16 and 32 replica configurations.

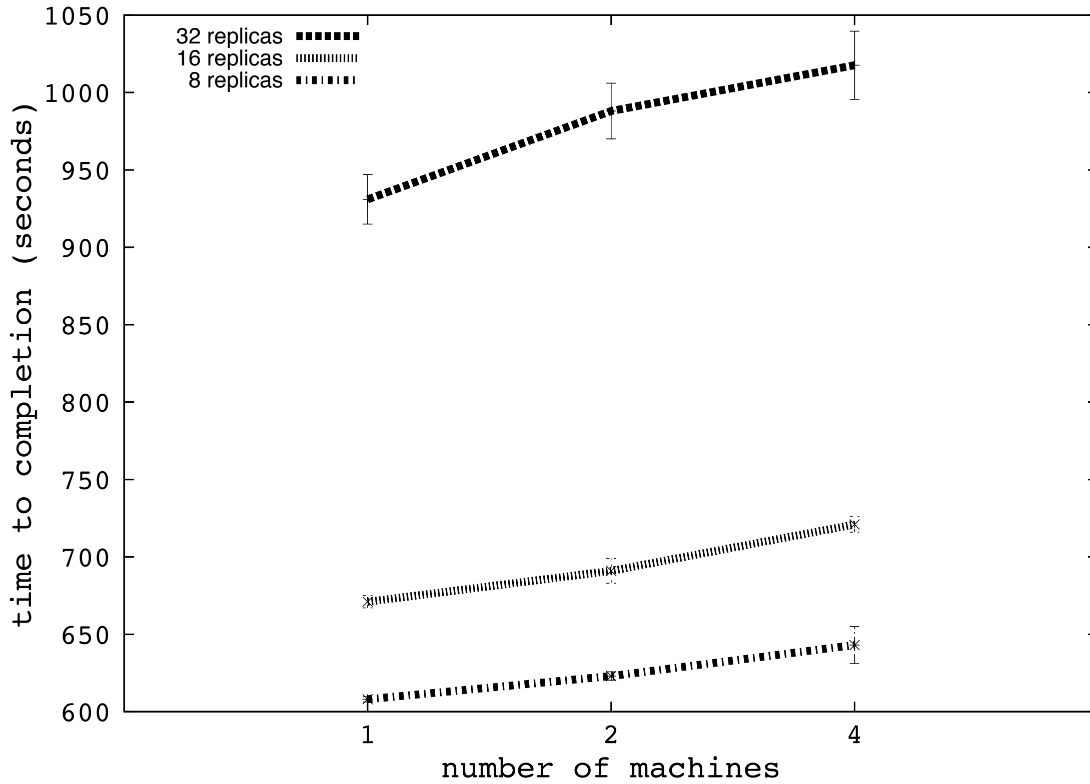

Figure 6.2: **Scale-out performance for 8, 16 and 32 replicas, synchronous:** The experiments were done on LONI resources and repeated at least 5 times. The error bars denote standard error. As the number of machines increases, the time-to-completion increases in general mainly due to higher exchange costs (T_{EX}) caused by e. g. remote file copies and additional synchronization costs.

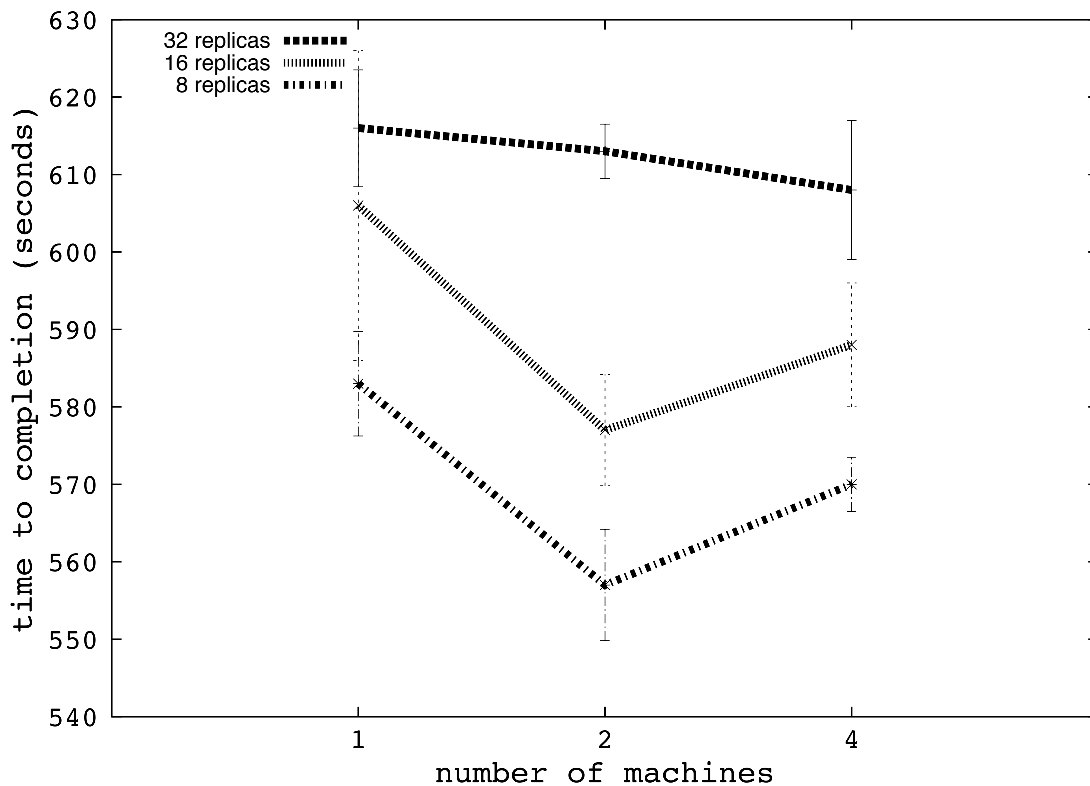

Figure 6.3: **Scale-out performance for 8, 16 and 32 replicas, asynchronous (decentralized):** The experiments were done on LONI resources and repeated at least 5 times. The error bars denote standard error. As the number of machines increases, the time-to-completion remains constant as there are no remote file copies or synchronization costs.

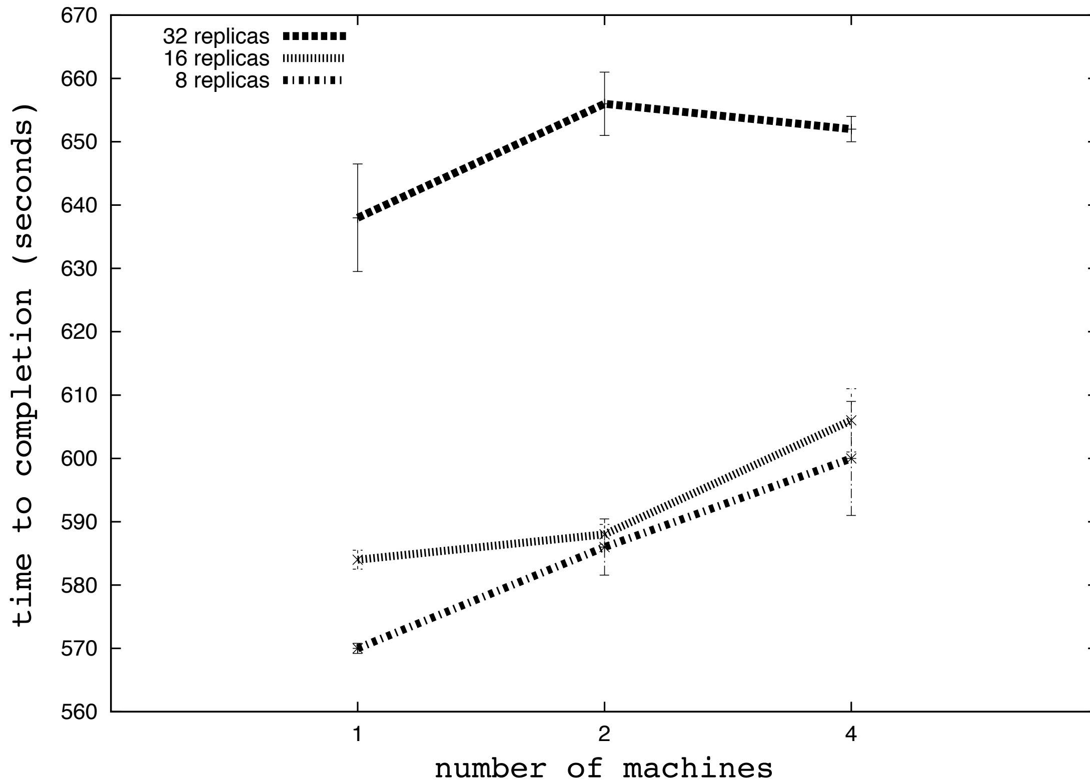

Figure 6.4: **Scale-out performance for 8, 16 and 32 replicas, asynchronous (centralized):** The experiments were done on LONI resources and repeated at least 5 times. The error bars denote standard error. As the number of machines increases, the time-to-completion increases in general mainly due to higher exchange costs (T_{EX}) caused by e. g. remote file copies.

Asynchronous - Centralized

Even though in a distributed scenario, T_{MD} can be different on different machines, the resources used in this experiment are homogeneous in performance and T_{MD} is not effected. Also, depending on the physical location and other network related issues, T_{EX} can be effected. Because T_W is 0 s in asynchronous RE, the main component that influences T is T_{EX}, which is comprised of T_{find}, T_{file} and T_{state}. Finding exchange partners and bookkeeping involves frequent communication with the advert service and some local processing. As the variation in these communication times is very small, the impact on T can be neglected. We observed that T_{file} – i. e. the time to transfer the configuration file to a replica – has a bigger impact on T.

The results obtained from the scale-out experiments are plotted in Figure 6.4; note that the scale of y-axis is much smaller than in Figure 6.1. The variations in T are small, ranging from 10 to 20 s. The fluctuations are all within the 30 s tolerance imposed by the experimental method.

For 8 replicas conducting 32 exchanges on 1 machine, T is 570 s; if these replicas are distributed equally over 2 machines, T increases slightly to 586 s. In total the observed overhead of distribution is quite small and essentially acceptable (with at worst 5 % of the overall runtime). As explained earlier, the main source of these overheads are the necessary remote file transfers. Each remote file transfer takes 0.38 s while a local transfer requires only 0.01 s, i. e. 0.37 s less. For the 8 replicas, 32 exchanges and 2 machines configuration, half of the replicas are located on a different resource than the RE-Manager, i. e. the input files must be transferred to the remote resource in these cases. On average when 2 machines are used, one of the necessary input files for an exchange requires a remote copy; for the 8 replica, 32 exchange configuration, 32 remote transfers are necessary. Thus, the additional overhead in T_{file} can be approximated to $0.37 \times 32 = 12\,s$. In Figure 6.4, this accounts for the difference, to within error bars, between the T values of 1 and 2 machines (for the 8 replica configuration).

As mentioned, we keep the ratio of the number of replicas to the number of exchanges a constant, thus as the number of replicas increases the time spent in the exchanges as a fraction of the overall time becomes less; thus lessening further, any effects of the small increase due to distributed exchange. This explains why at low replica counts there is a small (5%) increase in T as the number of distributed resources increases from 1 to 4, but effectively there is no change in T when going from 1 to 4 machines when N_R is 32.

Note that the fluctuations are in the range of 10 to 20 s. As mentioned earlier, the big-jobs became active within 30 s of each other, thus the fluctuations are within the tolerance of the experimental design. Additionally, for the 32 replica configuration, we see larger fluctuation in T on 1 machine than on 2 and 4 machines. There are several reasons that can be attributed to the decrease in the fluctuation in going from 1 to 2 (and 4) machines; however the dominant reason for the high-fluctuation for 1 machine is likely to be finite sampling – recall that experiments were repeated only 5 times. We plan to further the number of experimental samples, and at larger scales to analyze their behavior accurately. Nonetheless, from these results, we can say that our RE Framework is capable of scaling-out without acutely effecting the performance.

Our model and data from Figure 6.4 might only suggest additional file transfer costs, but

in a typical distributed scenario, with machines on different networks, physical locations, cost of distributed exchanges and heterogeneity may also play a role in effecting the performance. We intend to examine these factors in the future; obviously a comparison with other RE implementations will also be performed.

6.3 Heterogeneity in Resources

When running large scale simulations, heterogeneity in the resources used can quickly become a bottleneck. Heterogeneity could come in different varieties – different processor power, different communication times, etc., could all effect the performance of a distributed simulation. Especially, in the synchronous RE, there are periodic synchronization steps followed by exchange steps. If there are a large number of exchange steps involved, these synchronization could add up to substantially effect the total time to completion.

In this section, we only consider the effect of heterogeneous processors on the performance of the different RE algorithms. We do this by artificially introducing heterogeneity by running half the replicas with 16 processors and the other half with 8 processors. We did this not due to a lack of heterogeneous resources, but because we wanted all the replicas to start running at the same time. We cannot control the start times of jobs on different resources and we might potentially end up with only one BigJob becoming active.

We conducted the heterogeneous experiments with the following configuration: 32 replicas making 128 exchanges, where 16 replicas were assigned 16 processors and 16 replicas were assigned 8 processors. Each replica was run for 500 time steps. In the homogeneous experiments, all 32 replicas were launched with 16 processors each. The runtime of a replica launched with 16 processors is $\approx 71\ s$ while the runtime when 8 processors are used is $\approx 112\ s$, which is a difference of 41 s. We did these experiments with all the implementations and the machine used was *QueenBee*. Each experiment was repeated 5 times.

We observed that in the synchronous RE experiments, the fastest replicas were made to wait for the slowest replicas to finish. This added 41 s at every exchange step and over 8 exchange steps, this added $\approx 328\ s$. This can be seen in Figure 6.5. In the asynchronous (centralized) RE experiments, we can easily observe that asynchronous exchange helped keep the increase in T low. The faster replicas made more exchanges than the slower replicas. In the asynchronous (decentralized) RE experiments too, it is the same situation. It can be observed from the graph that the asynchronous (centralized) implementation had a smaller absolute increase in T when compared to the asynchronous (decentralized) RE implementation. This is because in the centralized implementation, the demands on the Bigjob-Agent were reduced as smaller number of replicas now were starting and ending at the same time. Thus, the synchronous algorithm is most adversely effected by the heterogeneity in the processors. The asynchronous algorithm can at least retain the performance advantage of the faster processors.

There are many different ways in which heterogeneity may present itself. The heterogeneity could be in the processors, network/communication related or due to other factors such as software. We only consider different processing powers in this section but other factors could also effect the performance.

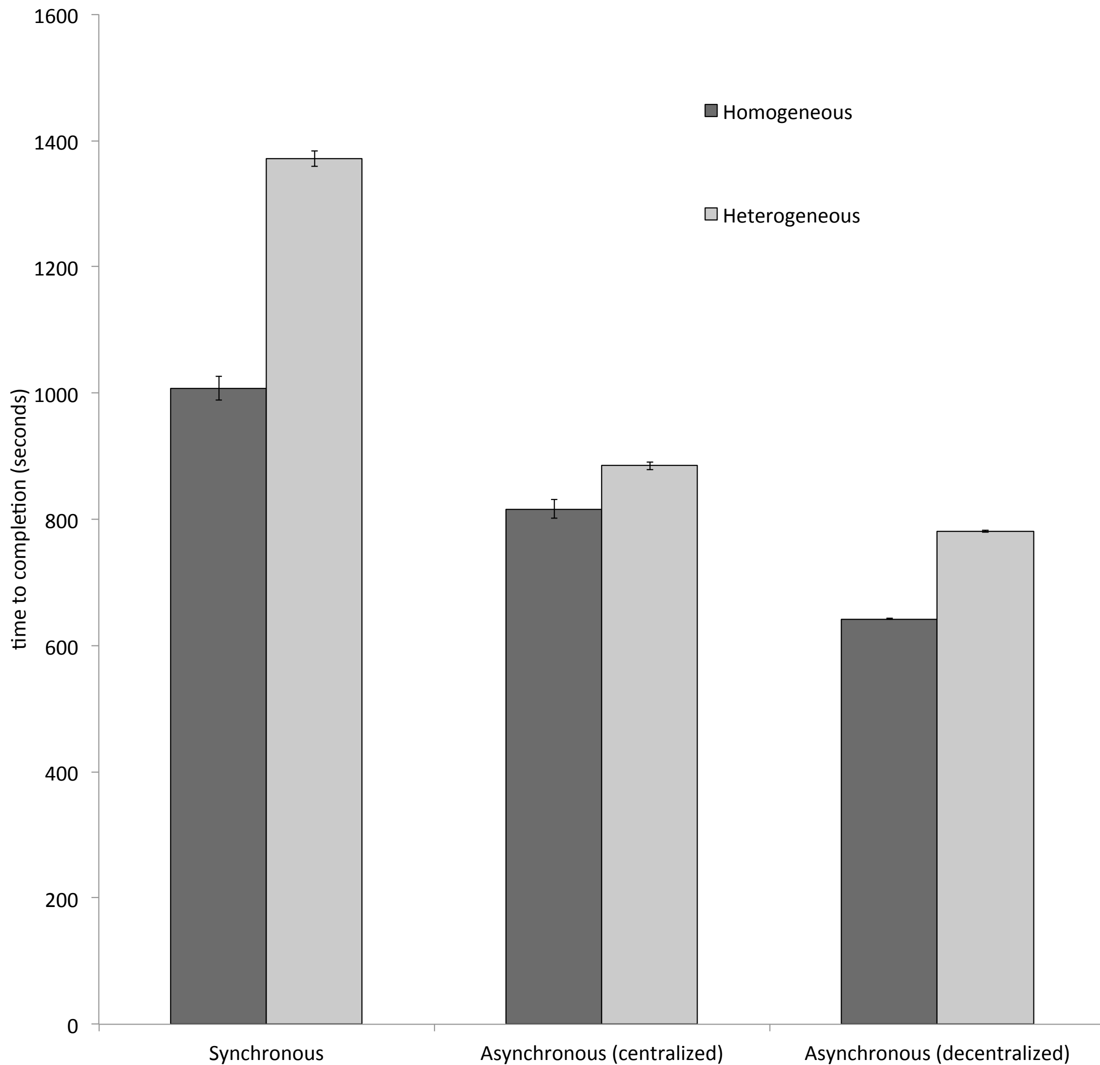

Figure 6.5: **Different replicas were launched with different number of processors:** The experiments were done on *QueenBee* and repeated 5 times. The error bars denote standard error. The heterogeneous experiments were done by launching 16 replicas with 16 processors and 16 replicas with 8 processors. In the homogeneous experiments all the replicas were launched with 16 processors. The synchronous RE is most adversely effected by the slower replicas. The other two at least take some advantage of the the faster replicas.

6.4 Crosswalks and Temperature Mixing

6.4.1 Crosswalks

Measuring the number of crosswalks and the extent of temperature mixing among the replicas are important means to evaluate the scientific performance of a RE experiment. A crosswalk is said to occur when a replica transitions from a low temperature to a high temperature and back [8]. According to [8, 18, 9], at lower replica counts, the number of crosswalks is higher for synchronous RE, but as the number of replicas in the ensemble is increased, asynchronous RE shows better crosswalk performance. We did a sample analysis, where we counted the number of crosswalks in a 256 replica experiment and the result supports this statement. We found that there were no crosswalks in the synchronous RE experiment, where as there were many crosswalks in the asynchronous RE experiment. In synchronous RE, we observed that at large replica counts, a replica is very likely to remain within a temperature range. The replica has to find an exchange partner with a neighboring temperature, as exchanges are nearest-neighbor only. This makes it very improbable for a replica to reach the both the extreme regions of temperature in the ensemble. But at smaller replica counts, a replica can traverse all the other temperatures one after the other, as the number of replicas (and the range temperatures) is small.

6.4.2 Temperature Mixing

Another measure of success of the simulation is the extent of temperature mixing among the replicas. In other words, it is better to have the replicas exchange with as many unique replicas and transition through as many temperatures as possible.

We primarily want to evaluate whether the two algorithms: (i) make the same number of unique exchanges and (ii) make the same number of total exchanges. We modeled the synchronous and asynchronous RE algorithms to see if one of them displays a better performance. We used the following configuration: probability of a successful exchange (p) was set at $\frac{1}{3}$. 4, 8, 16, 32, 64, 128 and 256 replicas were launched to perform 256, 512, 1024, 2048, 4096, 8192 and 16384 exchanges, respectively, keeping the ratio between N_R and N_X constant. The temperatures assigned to the replicas were in the range of $300 \pm 128K$. The difference in temperatures of any two neighboring replicas is $1K$. In the synchronous RE, each replica can exchange with any other replica with a neighboring temperature. In the asynchronous RE, any two replicas can attempt to exchange.

By keeping the ratio between N_R and N_X constant, we are keeping the average number of exchanges possible per replica constant. We are doing this to see if the synchronous and asynchronous algorithms perform differently as we increase N_R. According to the current ratio between N_R and N_X, each replica gets approximately 128 chances to attempt an exchange. But since p is set as $\frac{1}{3}$, the number of successful exchanges per replica would be $\approx \frac{128}{3} \approx 42$.

From Figure 6.6, for the asynchronous algorithm, the average total number of exchanges happening per replica is ≈ 42. But for the synchronous algorithm, the value is closer to 35. This is the case because in synchronous algorithm the exchanges happen with nearest neighbors only and the number of possible exchange partners to any replica is limited to

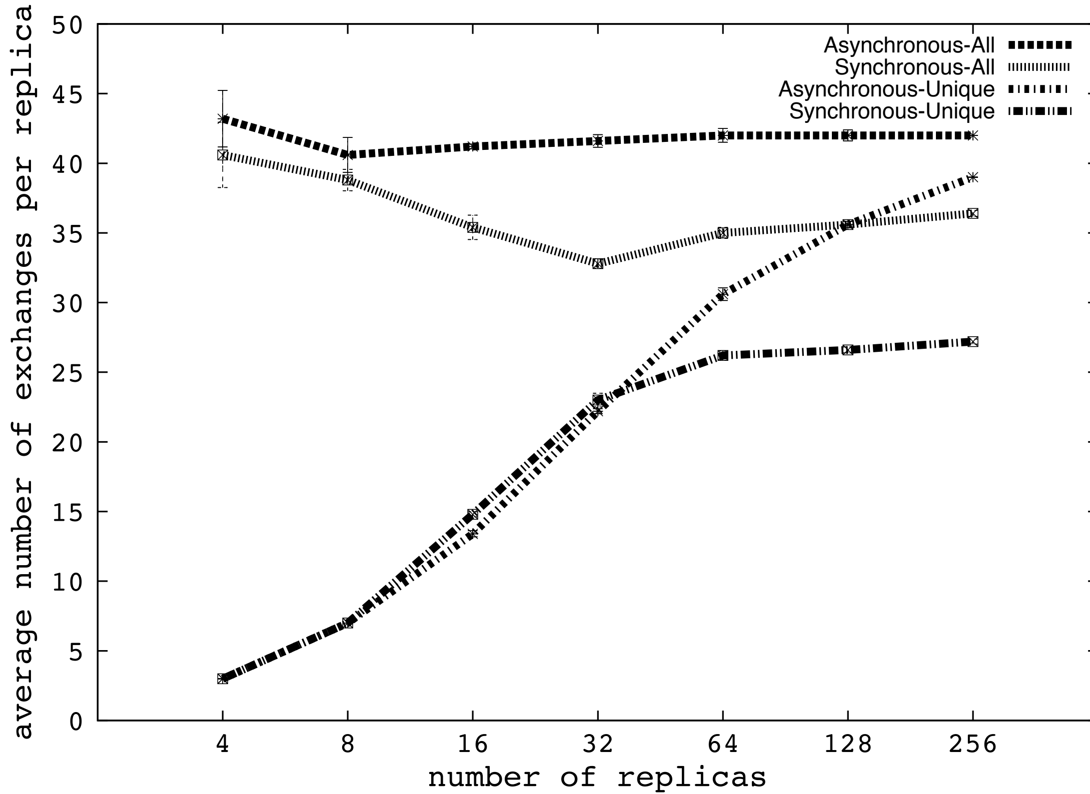

Figure 6.6: **Average number of unique exchanges and total exchanges per replica:** These exchanges were modeled according to the synchronous and asynchronous RE algorithms. The top two lines show the average number of total exchanges per replica for asynchronous and synchronous RE, respectively. It shows that asynchronous RE makes more number of exchanges per replica. The bottom two lines show the average number unique exchanges per replica. Given that the maximum possible exchanges is ≈ 42, for smaller N_R, both algorithms perform similarly. But with larger number of replicas, the asynchronous RE shows better temperature mixing/sampling.

two. Not only does this limit mixing, but this also implies that in some instances, a replica might not find a partner. This happens when both the neighbors of a replica exchange-with other replicas. This reduces the number of total exchanges per replica and the total number of successful exchanges in general.

In synchronous RE, we can see from Figure 6.6, that after a certain point, as the number of replicas increases, the number of unique partners does not increase. But on the other hand, in asynchronous RE, the number of unique partners keeps on increasing with increasing number of replicas. Note that the number of exchange possible per replica is still ≈ 42 and when the total number of replicas is small, both the algorithms seem to have the same number of unique exchanges. But with larger N_R, the number of unique partners per replica in the asynchronous RE keeps increasing and approaches 42, while in synchronous RE it stagnates. From this, we can say that, after a point, with increasing N_R, the number of unique partners per replica in the synchronous RE remains the same, while for asynchronous RE it increases. This indicates that the synchronous RE's usefulness tends to come down as the N_R increases. The reason for this could be that it is just difficult in general to continuously traverse the temperature range given that only exchanges with neighboring temperatures are allowed.

6.5 Experience using LONI and Teragrid Resources

In this section, we want to describe our experience using different large scale production computing resources. We present to the reader our experience using the LONI [4] and Teragrid [3] resources. It is by no means a trivial thing to do, especially if the user wants to use large number of processors and has applications that have special requirements, like RE does. We have done hundreds of simulations to collect the data presented in this work and feel that we are in a good position to say a few words about the issues involved. But first, a few words describing the machines used.

LONI, which stands for Louisiana Optical Network Initiative, provides fiber-optic connectivity around half-a-dozen supercomputing centers in the state of Louisiana. *QueenBee* is the biggest and fastest of those resources and has a peak performance of 50 Teraflops. There are nearly a dozen other supercomputers in the network. We have used about half-a-dozen of the LONI machines. Teragrid is nation-wide network of supercomputing resources. We used *Ranger*, which is a very large resource and has a peak performance of 579 Teraflops.

6.5.1 SAGA-BigJob Setup

It is actually quite simple to submit individual/single jobs. For every machine, there is a manual which describes the process to follow. But our use case is different. We have a workload of parallel coupled jobs and require a large number of processors. That is the reason we use the SAGA-BigJob Framework, to efficiently request and manage resources.

Firstly, installing SAGA is not a trivial thing to do for a beginner. Many biologists and physicists who were not familiar with grid computing technologies and tried to use SAGA have expressed this. But we can confidently say that once the user becomes familiar with the general concepts and SAGA, it becomes easier. That does not mean everything is always going to work. SAGA needs other software to be installed to work and SAGA-BigJob has

even more requirements. SAGA has two hard requirements - a recent C++ compiler and recent version of Boost Libraries [30]. SAGA-BigJob additionally requires Python [31] and PostgreSQL [32]. We also need to install additional SAGA Adaptors to to be able to submit jobs. In particular, we need the SAGA Globus Adaptor to submit jobs using GRAM [33]. Most often, the SAGA Globus adaptor turns out to be the weakest link in the whole setup process.

The SAGA Globus adaptor compiles against the Globus Toolkit [34] installation on the HPC machine. In many cases, we found that there are many Globus installations on the same machine. But only one of the installations usually works. Generally, after trying a few of the versions installed, one of them works out. But if everything fails, we would have to install Globus ourselves. But this may also not work.

It could be that the trouble making Globus installations are missing some libraries that are needed by the SAGA Globus Adaptor. But since the standalone Globus installation is functional, the system administrators most often cannot help to solve the problem. One way around this is to use other means to submit jobs - that is using the batch queueing systems. SAGA has adaptors for some versions of the Portable Batch System (PBS) [35] queuing system, so it is not a complete show stopper.

SAGA-BigJob makes use of the advert service, which runs PostgreSQL. The advert service could be physically anywhere but must be accessible via the internet. In our experiments, the server running the advert service was located at Louisiana State University. But since the HPC machines we used included *Ranger* and other LONI machines, which are on different networks, it is possible that some of the machines might not allow the compute nodes to access the advert service. But we did not face any trouble with the machines that we used. Other machines might have different policies regarding this.

6.5.2 Running Simulations

As mentioned earlier, SAGA-BigJob submits jobs to the resource manager using Globus. Users need a grid certificate, issued by the administrators to be able to submit jobs using Globus. The problems we faced could be divided into different groups.

The first set of problems are application errors that we encountered. The first problem that we faced was with the NAMD [29] version that we were using on *QueenBee*. There were again many versions installed on each machine. For some reason, the version of NAMD that we were using was giving us trouble. The individual NAMD jobs were frequently failing. After putting in considerable time and effort to find the problem and after talking to the system administrators, we found that one of the output files generated was saying that the NAMD version we were using was meant to be used with AMD processors. But *QueenBee's* processors are Intel manufactured. So we switched to a different version and the NAMD job failures went down. But they did not completely go away.

We are now moving onto the second set of errors, which are due to hardware failure. After further looking into this, we found that the errors were recurring on the same nodes. It basically turned out that *QueenBee* had multiple faulty nodes. The issue was fixed by the administrators. We saw these errors from time to time, but it was just that different nodes were failing randomly. We did not see this kind of failures on any other LONI or Teragrid machine.

Other issues that we faced include not being able to submit jobs due to GRAM services not coming online when machines are restart, grid certificates expiring, resource manager crashing etc., which can be categorized as middleware issues.

While we used LONI resources for most of our simulations, we also used *Ranger*, a Teragrid resource. We faced some similar issues, mostly related to GRAM.

We also had access and intended to use two other Teragrid resources, *Abe* and *Kraken*. But we were not able to use these machines due to GRAM issues. We were not able to submit jobs using GRAM on these two resources using SAGA-BigJob. But now SAGA is providing additional adaptors for PBS and similar batch systems.

Chapter 7

Conclusion and Future work

Following on from earlier theoretical explorations [8, 9], in this work, we investigate *traditional* and *advanced* replica-exchange algorithms at unprecedented scales. An important motivation for this work has been to implement a framework which supports multiple RE algorithms; it is the aim that the RE framework use general purpose and standard capabilities available on production infrastructure, such as, the Teragrid and LONI. Additionally, our framework uses a flexible pilot-job implementation, which enables effective resource allocation for multiple replicas.

Results shown in figures 6.1, 6.2, 6.3 and 6.4 indicate that using algorithmic formulations that impose less tight coordination constraints enables both good scale-up and scale-out behavior. Algorithms based on asynchronous coordination are typically more difficult to implement than synchronous ones; however, we find that even with a simple, non-optimized prototype of the RE Framework, the advantages of asynchronous formulations soon outweigh the synchronous formulations, i.e., as N_R increases the performance of asynchronous RE beats that of the synchronous RE.

Figure 6.5 indicates that asynchronous algorithm performs better on heterogeneous resources. Figure 6.6 shows that asynchronous RE helps produce better scientific results by enabling better mixing of temperatures among the replicas and consequently, better sampling. It has also been indicated that asynchronous RE enables larger number of crosswalks.

Our analysis shows that a fundamental trade-off is between the lower cost of replica synchronization at the exchange stage that asynchronous formulations provide, versus the higher cost of permitting dynamic replica pairing. In an attempt to investigate an optimal trade-off between these factors and to demonstrate the advantages of asynchronous RE, we implemented a centralized version of the asynchronous RE with a lower cost of dynamical pairing than in the decentralized implementation.

Our initial results show better scale-up behavior than the synchronous algorithm. The scale-out results are not conclusive, and more work is required to separate and understand fundamental algorithmic advantages from implementation specific issues. But in general, asynchronous RE tends to perform better than the synchronous RE in the scale-out experiments as well. Therefore, we think that with this work, we are able to say that asynchronous RE is more flexible than synchronous RE algorithms and also performs better on a wide range of machines and produces better science.

We would like to extend this work to include built in fault tolerance and recovery. In

our experiments, we have seen replicas failing due a range of reasons. Capacity to restart or resubmit jobs in such cases would improve the efficiency of the simulation.

Bibliography

[1] U. Hansmann, "Parallel Tempering Algorithm for Conformational Studies of Biological Molecules," in *Chemical Physics Letters*, vol. 281, 1997, pp. 140–150.

[2] Y. Sugita and Y. Okamoto, "Replica-Exchange Molecular Dynamics Method for Protein Folding," in *Chemical Physics Letters*, vol. 314, 1999, pp. 141–151.

[3] "TeraGrid – American Grid Infrastructure," http://www.teragrid.org/, 2008.

[4] "LONI – Louisiana Optical Network Initiative," http://www.loni.org, 2008.

[5] Woods, C. et al., "Grid Computing and Biomolecular Simulation," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 363, no. 1833, pp. 2017–2035, 2005.

[6] A. Luckow, L. Lacinski, and S. Jha, "Saga bigjob: An extensible and interoperable pilot-job abstraction for distributed applications and systems," in *The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2010.

[7] A. Luckow, S. Jha, J. Kim, A. Merzky, and B. Schnor, "Adaptive Replica-Exchange Simulations," *Royal Society Philosophical Transactions A*, 2009.

[8] Z. Li and M. Parashar, "Grid-based asynchronous replica exchange," in *GRID '07: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing.* Washington, DC, USA: IEEE Computer Society, 2007, pp. 201–208.

[9] E. Gallicchio, R. M. Levy, and M. Parashar, "Asynchronous replica exchange for molecular simulations," *Journal of Computational Chemistry*, vol. 29, no. 5, pp. 788–794, 2008.

[10] A. Thota, A. Luckow and S. Jha, *Efficient Large-Scale Replica-Exchange Simulations on Production Infrastructure*, submitted to Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences; draft at http://saga.cct.lsu.edu/publications/papers/confpapers/erelpri.

[11] D. E. Shaw, R. O. Dror, J. K. Salmon, J. P. Grossman, K. M. Mackenzie, J. A. Bank, C. Young, M. M. Deneroff, B. Batson, K. J. Bowers, E. Chow, M. P. Eastwood, D. J. Ierardi, J. L. Klepeis, J. S. Kuskin, R. H. Larson, K. Lindorff-Larsen, P. Maragakis, M. A. Moraes, S. Piana, Y. Shan, and B. Towles, "Millisecond-scale molecular dynamics simulations on anton," in *Proceedings of the Conference on High Performance*

Computing Networking, Storage and Analysis, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 65:1–65:11. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654126

[12] B. W. Project, 2011. [Online]. Available: http://www.ncsa.illinois.edu/BlueWaters/

[13] "Folding@home Distributed Computing," http://folding.stanford.edu/.

[14] "Genome@home Distributed Computing," http://genomeathome.stanford.edu/.

[15] G. Bussi, "A simple asynchronous replica-exchange implementation," *Nuovo Cimento C Geophysics Space Physics C*, vol. 32, no. 2, pp. 020 000–65, Mar. 2009.

[16] *MPI: A Message-Passing Interface Standard.* Message Passing Interface Forum, 1995. [Online]. Available: http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[17] T. Sterling, E. Lusk, and W. Gropp, Eds., *Beowulf Cluster Computing with Linux*, 2nd ed. Cambridge, MA, USA: MIT Press, 2003.

[18] L. Zhang, M. Parashar, E. Gallicchio, and R. M. Levy, "Salsa: Scalable asynchronous replica exchange for parallel molecular dynamics applications," *Parallel Processing, International Conference on*, vol. 0, pp. 127–134, 2006.

[19] I. Buch, M. J. Harvey, T. Giorgino, D. P. Anderson, and G. De Fabritiis, "High-throughput all-atom molecular dynamics simulations using distributed computing," *Journal of Chemical Information and Modeling*, vol. 50, no. 3, pp. 397–403, 2010, pMID: 20199097. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/ci900455r

[20] Y. M. Rhee and V. S. Pande, "Multiplexed-replica exchange molecular dynamics method for protein folding simulation," *Biophysical journal*, vol. 84, no. 2, pp. 775–786, 02 2003/02/01. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0006349503748978

[21] "EGEE – European Grid Infrastructure," http://public.eu-egee.org/, 2008.

[22] N. Jacq, V. Breton, H.-Y. Chen, L.-Y. Ho, M. Hofmann, V. Kasam, H.-C. Lee, Y. Legr, S. C. Lin, A. Maa, E. Medernach, I. Merelli, L. Milanesi, G. Rastelli, M. Reichstadt, J. Salzemann, H. Schwichtenberg, Y.-T. Wu, and M. Zimmermann, "Virtual screening on large scale grids," *Parallel Computing*, vol. 33, no. 4-5, pp. 289 – 301, 2007, large Scale Grids. [Online]. Available: http://www.sciencedirect.com/science/article/B6V12-4N3WYF2-1/2/e5c8d221738d976fc4596e62b703d027

[23] H.-C. Lee, J. Salzemann, N. Jacq, H.-Y. Chen, L.-Y. Ho, I. Merelli, L. Milanesi, V. Breton, S. Lin, and Y.-T. Wu, "Grid-enabled high-throughput in silico screening against influenza a neuraminidase," *NanoBioscience, IEEE Transactions on*, vol. 5, no. 4, pp. 288 –295, dec. 2006.

[24] "WISDOM – Initiative for Grid-Enabled Drug Discovery Against Neglected and Emergent Diseases," http://wisdom.eu-egee.fr/.

[25] M. Eleftheriou, A. Rayshubski, J. Pitera, B. Fitch, R. Zhou, and R. Germain, "Parallel implementation of the replica exchange molecular dynamics algorithm on blue gene/l," *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 281, 2006.

[26] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. 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.

[27] SAGA, 2011. [Online]. Available: http://saga.cct.lsu.edu

[28] Open Grid Forum. [Online]. Available: http://www.ogf.org/

[29] J. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. Skeel, L. Kale, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, pp. 1781–1802, 2005.

[30] "Boost C++ Libraries," http://www.boost.org, 2008.

[31] "Python Programming Language," http://www.python.org, 2008.

[32] "PostgreSQL Database System," http://www.postgresql.org, 2008.

[33] M. Feller, I. Foster, and S. Martin, "GT4 GRAM: A Functionality and Performance Study," in *Proceedings of the Teragrid 2007 Conference*, Madison, WI, USA, 2007.

[34] I. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems," in *Proceedings of IFIP International Conference on Network and Parallel Computing*. Springer, 2006, pp. 2–13.

[35] R. Henderson and D. Tweten, "Portable Batch System: External Reference Specification," NASA Ames Research Center, Tech. Rep., 1996.

Vita

Abhinav Thota is a computer science graduate student in the Louisiana State University. His interests lie in the high-performance, grid and supercomputing fields. He previously received a bachelor's degree from Osmania University, Hyderabad, India.