ENABLING ASYNCHRONOUS INTERACTION AND COUPLING FOR PARALLEL SCIENTIFIC APPLICATIONS USING DECENTRALIZED SEMANTICALLY-SPECIALIZED SHARED SPACES

BY LI ZHANG

A Dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Manish Parashar

and approved by

____

____

____

____

____

New Brunswick, New Jersey

October, 2006

ABSTRACT OF THE DISSERTATION

Enabling Asynchronous Interaction and Coupling for Parallel Scientific Applications using Decentralized Semantically-Specialized Shared Spaces

by Li Zhang

Dissertation Director: Professor Manish Parashar

While large-scale parallel/distributed simulations are rapidly becoming critical research modalities in academia and industry, their efficient and scalable implementations continue to present many challenges. Key challenges are the dynamic and complex communication, coordination and coupling requirements of these applications, which depend on state of the phenomenon being modeled, are determined by the specific numerical formulation, the domain decomposition and/or sub-domain refinement algorithms used, etc., and are known only at runtime. Most existing solutions addressing these requirements are domain-specific and are typically tightly integrated with individual applications. Further, these solutions tend to be static and are not very flexible due to their underlying approaches.

This dissertation builds on the Tuple Space model and customizes and extends it to support parallel scientific applications. Specifically, this research develops a semantically-specialized shared space abstraction infrastructure to support asynchronous interaction and coupling of parallel scientific applications. The key contribution of this thesis is a conceptual model and implementation architecture for realizing interaction and coupling frameworks that provide application-oriented abstraction for flexible, efficient, scalable, and asynchronous interaction and coupling. The approach emphasizes flexibility, efficiency, and scalability through the use of a tuple space-based abstraction which is customized based on application domain semantics. The approach and resulting systems complement and can be used in conjunction with existing

parallel programming systems such as MPI and OpenMP.

Three prototype systems have been implemented based on this approach. The first prototype, Seine-Geo, provides a dynamic geometry-based shared space interaction framework to enable adaptive multi-physics multi-block multi-scale oil reservoir simulation and addresses the dynamic and complex communication/coordination patterns between multiple blocks in the simulation. The second prototype, Seine-Coupe, applies the geometry-based shared space concept to support MxN parallel data redistribution required for coupling multiple independent parallel simulations. The third prototype, Seine-Salsa, constructs a decentralized temperature shared space to support dynamic pairwise communication. The implementation is used to enable an asynchronous formulation of the replica exchange algorithm for molecular dynamics applications. Experimental evaluations of these prototype implementations demonstrate the flexibility, efficiency, and effectiveness of the approach and these systems, as well as their ability to support complex interaction and coupling requirements of these applications.

Acknowledgements

Thanks go to my thesis advisor, Dr. Manish Parashar, for his guidance and support during my Ph.D study, for showing me passion and spirit of a true researcher and scientist, and for the constant encouragements that help me achieve my goal step by step.

Thanks go to my beloved parents, Chengqiao Zhang and Juying Xu, and my brother and his wife, Jian Zhang and Xiaohong Shi, for their support and encouragement through the years, for showing me virtue and spirit from which my strength and belief originate, and for loving me with absolutely no condition.

Thanks go to my husband, Xifan Wu, for drawing every color of life with we.

Thanks go to my friends for always being there for me.

Table of Contents

Abstract ii

Acknowledgements iv

List of Tables ix

List of Figures x

1. Introduction 1

1.1. Motivation 1

1.1.1. Illustrative Applications 1

1.1.2. Requirements and Challenges 3

1.2. Overview of Seine 5

1.2.1. Seine Prototype Systems 7

Seine-Geo 7

Seine-Coupe 8

Seine-Salsa 8

1.2.2. Contributions 8

1.3. Outline 9

2. Seine: A Dynamic Shared Space Interaction Framework for Parallel Scientific Applications 11

2.1. Problem Description 11

2.2. Parallel Programming Models 11

2.2.1. The Message-Passing Paradigm 11

2.2.2. The Shared-Memory Paradigm 12

2.2.3. The Tuple Space Model 13

2.3. Overview of Seine 14

2.4. Seine Architecture 16

2.4.1. The Distributed Directory Layer 16
2.4.2. The Storage Layer 18
2.4.3. The Communication Layer 19
2.5. Overview of the Prototypes 19

3. Seine-Geo: A Seine-based Intra-Coupling Framework for Parallel Scientific Applications 20
3.1. An Illustrative Application: Interface Coupling in Parallel Adaptive Multi-block Oil Reservoir Simulation 20
3.2. Seine-Geo: A Dynamic Geometry-based Shared Space Interaction Framework for Parallel Scientific Applications 22
3.2.1. The Seine-Geo Architecture 23
The Seine-Geo Directory Layer 23
The Seine-Geo Storage Layer 26
The Seine-Geo Communication Layer 26
3.2.2. Seine-Geo Programming Interface 27
3.2.3. Seine-Geo Operation 28
3.3. Prototype Implementation and Performance Evaluation 30
3.3.1. Prototype Implementation 30
3.3.2. Experimental Evaluation 30
Experiments on DataStar 32
Experiments on a Beowulf Cluster 41
3.4. Related Work 42
3.5. Conclusion 43

4. Seine-Coupe: Seine-based Inter-Coupling for Parallel Scientific Applications . . 44
4.1. Overview 44
4.2. Background and Related Work 45
4.2.1. Component-based Models and the Common Component Architecture Standard 45

4.2.2. Code Coupling and Parallel Data Redistribution 45
Problem Description and Challenges 45
Support for Parallel Data Redistribution 46
4.3. Seine-Coupe: The Seine-based Coupling Framework 50
4.3.1. Design of Seine-Coupe Coupling Framework 51
4.3.2. Coupling Parallel Scientific Applications using Seine-Coupe 52
Building coupled simulations using the Seine abstractions: 53
Computation of communication schedules: 53
Data transfer: 54
4.3.3. Prototype Implementation and Performance Evaluation 55
A CCA-based Prototype Implementation 55
Experiment with different data redistribution scenarios: 57
Experiment with different array sizes: 59
Scalability of the Seine-Coupe directory layer: 59
4.3.4. Conclusion 60
4.4. Data Coupling in the SciDAC Fusion Project 61
4.4.1. An Overview of the Fusion Simulation Project 61
4.4.2. Data Coupling Requirements in the Fusion Simulation Project 61
Challenges and Requirements 62
4.4.3. A Prototype Coupled Fusion Simulation using Seine-Coupe 63
Domain decompositions: 63
Coupled fusion simulations using Seine-Coupe Shared Spaces 64
4.4.4. Prototype Implementation and Performance Evaluation 65
Preliminary Test and Result Analysis 66
Experiments with Local-Area Coupling using the Prototype Seine-Coupe based Fusion Simulation 66
Experiments with Wide-Area Coupling using the Prototype Seine-Coupe based Fusion Simulation 71
4.4.5. Conclusion and Future Work 75

5. Seine-Salsa: Seine-based Salable Asynchronous Replica Exchange for Parallel Molecular Dynamics Applications . . . 76
5.1. Overview . . . 76
5.2. Parallel Replica Exchange for Structural Biology and Drug Design . . . 79
5.2.1. The Replica Exchange Algorithm . . . 79
5.2.2. Existing Parallel Implementations of Replica Exchange-based Simulations . . . 80
5.3. Seine-Salsa: A Framework for Parallel Asynchronous Replica Exchange . . . 81
5.3.1. The Seine-Salsa Architecture . . . 81
The Seine-Salsa Distributed Directory Layer . . . 82
The Seine-Salsa Communication Layer . . . 82
5.3.2. Seine-Salsa Programming Interface . . . 83
5.3.3. Parallel Asynchronous Replica Exchange using Seine-Salsa . . . 83
5.4. Seine-Salsa Implementation and Experimental Evaluation . . . 86
5.4.1. Initial Test of correctness for Salsa-based Replica Exchange . . . 87
5.4.2. Salsa-based vs. MPI-based Replica Exchange . . . 87
5.4.3. Scalability of Seine-Salsa . . . 89
5.4.4. Effect of Posted Temperature Ranges on Cross-Walks . . . 91
5.5. Related Work in Parallel Replica Exchange . . . 92
5.6. Conclusion . . . 93

6. Summary, Conclusion, and Future Work . . . 94
6.1. Summary . . . 94
6.2. Conclusion . . . 95
6.3. Directions For Future Work . . . 96
References . . . 98
Curriculum Vitae . . . 102

List of Tables

3.1. Seine Programming Interface. 27

3.2. Application pseudocode using Seine-Geo API. 31

3.3. Breakdown of Seine-Geo operation costs. 35

4.1. Cost in seconds of computing registration and data transfer for different data redistribution scenarios. 58

4.2. Cost in seconds of registration and data transfer for different array sizes. 59

4.3. Scalability of the directory layer of the Seine-Coupe coupling component. . . . 60

5.1. Seine-Salsa application programming interface. 83

5.2. Number of temperature cross-walk events. 88

List of Figures

2.1. Seine architecture 17

3.1. (a) Various aspects of the multi-block formulation (left: multi-numerics, center: multi-physics, right: multi-scale); (b) 2-D view of decomposed multi-block domains and mortar grids [44]; (c) Illustration of key communication between multi-block domains and mortar grids. 21

3.2. Overview of Seine-Geo space (a 2-D example); (a) Phase 1: *register* a geometric region; [operator: *register*(x_low_coord, y_low_coord, x_high_coord, y_high_coord)]; (b) Phase 2: *put*/*get* geometric objects; [operator: *put*/*get*(x_low_coord, y_low_coord, x_high_coord, y_high_coord, data_handle)] 24

3.3. Seine directory structure using the Hilbert SFC [44]. 25

3.4. Operation of the Seine-Geo framework. 29

3.5. 3-D and 2-D views of the multi-block grid structure for the parallel oil reservoir application used in the experimental evaluation. 32

3.6. Average system initialization and bootstrapping costs and standard deviation on DataStar. 33

3.7. 2-D view of the different interaction patterns used in the experiments (Test cases I, II & III). 34

3.8. (a) Average shared object sizes for the different interaction patterns; (b) Total sizes of registered regions. 34

3.9. (a) Average size of Seine-Geo shared space; the error bars show the sizes of the largest and smallest spaces in each case; (b) Cost per unit region size of the register operation cost for different interaction patterns (Test cases I, II & III). . 36

3.10. (a) Number of co-existing Seine-Geo shared spaces; (b) Average number of objects per Seine-Geo shared space; (c) *Get* operation cost per unit region size for different interaction patterns (Test cases I, II & III). 37

3.11. (a) Total number of processors associated with Seine-Geo shared spaces; (b) Average number of processors associated with a Seine-Geo shared space; (c) *Put* operation cost for different interaction patterns (Test cases I, II & III). . . . 39

3.12. (a) Average size of Seine-Geo shared space; (b) Average execution time for *register* operations for different domain and object sizes on DataStar. 40

3.13. Average execution time for *get* (a) and *put* (b) operations for different domain and object sizes on DataStar. 41

3.14. Average execution time for *init*,*register*, *get* and *put* operations on the 64-processor Beowulf cluster. 42

4.1. 54

4.2. Operations of coupling and parallel data redistribution using Seine-Coupe. . . . 56

4.3. MxN parallel data redistribution between CCA framework instances using the Seine-Coupe coupling component. 57

4.4. Workflow between XGC and MHD. 62

4.5. Mock simulation configuration for data coupling in Fusion. 64

4.6. MxN data redistribution between parallel applications using the Seine-Coupe coupling framework. 65

4.7. *register* operation time cost. 68

4.8. *put* operation time cost. 69

4.9. *get* operation time cost. 70

4.10. operation cost and normalized operation cost for *register* and *put* at XGC site. . 72

4.11. XGC site per processor throughput (plotted in (a)) and estimated effective system throughput assuming data transfer overlap of 35% (plotted in (b)) and 50% (plotted in (c)). 73

4.12. *put* operation cost and XGC site per processor throughput with respect to different data generation rates 75

5.1. An schematic of the Seine-Salsa architecture. 82

5.2. Pseudo-code illustrating the implementation of replica exchange using Seine-Salsa. 84

5.3. An example parallel asynchronous replica exchange implementation using Seine-Salsa. 85
5.4. (a) Average wall-clock execution time and standard deviation with increasing number of processes (walkers); (b) Normalized execution time with increasing number of processes (walkers). 90
5.5. Influence of different posted temperature ranges on the number of cross-walk events. 91

Chapter 1

Introduction

1.1 Motivation

Large-scale parallel/distributed simulations are playing an increasingly important role in science and engineering, and are rapidly becoming critical research modalities in academia and industry. With the increasing scale of parallel systems and sophistication of application formulations and numerical techniques, emerging applications offer the potential for providing dramatic insights into complex phenomena. However, the phenomena being modeled by these applications and their implementations are inherently dynamic and heterogeneous in time, space, and state. Furthermore, coupled physics models and associated parallel codes provide the individual models with a more realistic simulation environment, allowing them to interact with other physics models in the coupled system and to react to dynamically changing boundary conditions. Combined with the complexity and scale of the underlying parallel/distributed system, efficient and scalable implementations of these independent or coupled simulations continue to present many challenges.

1.1.1 Illustrative Applications

Block coupling in adaptive multi-block oil reservoir simulations [42]: Subsurface simulation models consist of a complex interaction of fluid and rock properties that evolves with time. To achieve the desired efficiency and accuracy in the representation of the different phenomena taking place in the subsurface, these simulations provide support for different scales (multiscale), processes (single phase, oil-water, air-water, three-phases, compositional), and algorithms or formulations (IMPES, fully implicit) through a multiblock approach. In these simulations, the oil reservoir is discretized as a series of blocks and interfaces between blocks. The underlying numerical formulation consists of a coupled system of highly nonlinear transient partial differential equations. Its geometrical and geological features induce a multiblock decomposition so that each block is discretized by cell-centered finite differences on

logically rectangular grids. Flux matching conditions are imposed on the interfaces and a non-overlapping domain decomposition algorithm is exploited so that solving the interface problem only requires in-block solves and an exchange of interface values between neighboring blocks. Communication between blocks in this formulation is localized to the interfaces between neighboring blocks. Because the grids are different on the two sides of the interfaces, a different boundary space, called mortar space, is needed. Mortar grids result from the discretization of the mortar space. The challenge in implementing the communication/interaction patterns in such a scenario is that when the decomposed sub-blocks are distributed across the processors in a parallel system, locating the processor assigned to a neighboring block and generating the communication schedules required by typical message passing systems is non-trivial, especially when dynamic load-balancing is used and sub-blocks can move from one processor to another to balance load. Further complexity is introduced when the grid block and/or the mortar grids at the interface are dynamically refined.

Code coupling in the SciDAC Fusion project [40, 41]: The SciDAC Fusion project is developing a new integrated predictive plasma edge simulation code package that is applicable to the plasma edge region relevant to both existing magnetic fusion facilities and next-generation burning plasma experiments, such as the International Thermonuclear Experimental Reactor (ITER). The plasma edge includes the region from the top of the pedestal to the scape-off layer and divertor region bounded by a material wall. A multitude of non-equilibrium physical processes on different spatio-temporal scales present in the edge region demands a large scale integrated simulation. The low collisionality of the pedestal plasma, magnetic X-point geometry, spatially sensitive velocity-hole boundary, non-Maxwellian nature of the particle distribution function, and particle source from neutrals, combine to require the development of a special kinetic transport code for kinetic transport code for kinetic transport physics, using a particle-in-cell (PIC) approach on a massively parallel computing platform. To study the large scale MHD phenomena, such as Edge Localized Modes (ELMs), a fluid code is more efficient in terms of computing time, and such an event is separable since its time scale is much shorter than the transport time. However, the kinetic and MHD codes must be integrated together for a self-consistent simulation as a whole. Consequently, the edge turbulence PIC code XGC-ET will be connected with the microscopic M3D code to study the dynamical pedestal-ELM cycle.

The coupling is based on common grids at the spatial interface. XGC-ET and M3D are two distinct parallel simulations and will be run on different number of processors and on different platforms. To couple the two codes, their data need to be transferred back and forth between the codes, which is essentially a MxN parallel data redistribution problem. The MxN problem refers to the transfer of parallel data structures from one parallel application running on M processors to another parallel application running on N processors. To realize MxN parallel data redistribution, communication schedules need to be computed for every processor to describe the corresponding source or destination of each data transfer. Furthermore, to satisfy application performance and efficiency requirements, the data transfer during data redistribution should be high throughput and low latency.

Replica exchange algorithm for Molecular Dynamics applications [3]: Replica exchange is a powerful sampling algorithm that preserves canonical distributions and allows for efficient crossing of high energy barriers that separate thermodynamically stable states. In this formulation, several copies, or replicas, of the system of interest are simulated in parallel at different temperatures using “walkers”. These walkers occasionally swap temperatures to allow them to bypass enthalpic barriers by moving to a higher temperature. The challenge arises from the fact that the exchange happens between a random pair of walkers and is determined by the current states or physics parameters of the walkers. Such dynamic and unpredictable interaction behaviors cannot be hard coded into an implementation as exactly required by the application without sacrificing or limiting its dynamism. As a result, to the best of our knowledge, all the current parallel/distributed implementations of replica exchange simulations in use by the structural biology community are based on a simplified formulation of the algorithm that limits the potential power of the technique in two important ways: (1) the only parameter exchanged between the replicas is the temperature of each replica, and (2) the exchanges occur in a centralized and totally synchronous manner, and only between replicas with adjacent temperatures. The former limits the effectiveness of the method, while the latter limits its scalability to at most tens of homogeneous and relatively tightly coupled processors.

1.1.2 Requirements and Challenges

As illustrated by the applications described above, emerging applications present significant

challenges and requirements. These include:

Increasing application scale and complexity: Accurate simulations require high resolution domains. This leads to increasing computation and storage requirements and hence increasing system scales and data volumes. At the same time, more sophisticated techniques used by these simulations such as adaptive mesh refinement algorithms, together with the coupling of multiple models and codes, lead to increasing complexity.

Increasing system scale and complexity Emerging parallel systems are becoming increasingly complex due to processor architecture as well as interaction and communication complexities, such as model and code coupling that involve interactions between multiple models and codes. Furthermore, grid environments that combine geographically distributed applications add an additional level of complexity.

Constant performance requirement To achieve accurate or close emulation of natural phenomena, scientific simulations are constantly trying to catch the highest level of resolution, which usually leads to a longer simulation time. Therefore, there is a constant performance requirement for parallel scientific simulations to achieve highest performance possible based on available computational power so as to get simulation results in the shortest possible time.

These requirements introduced significant challenges. A key challenges is the dynamic and complex communication/coordination and coupling requirements. The increasing size of application and runtime system leads to a larger number of individual processes as well as the interaction and coordination among these processes. Furthermore, the increase in complexity of application, the models and the numeric techniques adopted, as well as the additional sophistication due to coupling, make simulations and their implementations inherently more dynamic and heterogeneous in time, space, and state. This causes the interaction among individual tasks to be increasingly dynamic and complex. These communication/coordination requirements are introduced by the interaction and/or coupling between entities in a single or coupled simulation and can be categorized into two types: “intra-coupling” and “inter-coupling”. “Intra-coupling” refers to the interaction between entities within a parallel simulation and “inter-coupling” refers to the coupling across multiple distinct parallel simulations in a coupled simulation system. It is essential to provide: (1) high level abstractions oriented to the application layer to simplify the development of applications with complex and unpredictable communication/coordination

patterns; (2) the support for efficient and scalable realizations to achieve critical performance requirement.

In the past, the complex communication/coordination requirements have usually been handled by individual application developers. More recently, a number of coupling frameworks and software supports have been developed to specifically address parallel data redistribution in the inter-coupling requirement. Message passing frameworks such as MPI, which are the most widely used paradigm, require matching sends and receives to be explicitly defined for each interaction. This approach can be very tedious and highly error-prone. Programming frameworks based on shared address spaces provide higher-level abstractions that can support dynamic interactions. However, the general high level abstraction is achieved at the cost of performance and scalability.

Tuple space is a generic communication model that provides a very powerful and flexible communication scheme and supports asynchronous and decoupled communications. However, a general tuple space implementation is not suitable for parallel scientific applications due to its relatively low performance and poor scalability. The overheads of such general tuple spaces implementations originate from the generality of the semantics supported and the global sharing behavior required by the model. By formatting a model that limits the sharing behavior of the general tuple space model based on domain-specific requirements, overheads imposed by these requirements will be reduced and therefore it is possible to achieve performance need of most parallel scientific applications. In other words, through domain-specific customization, scalable and efficient implementations of domain-specific semantic-specialized tuple spaces can be realized. These specialized tuple spaces conceal the complexity of coupling requirements from the application layer and provide the required performance. We believe that these domain-specific customizations provide an attractive trade-off: It has a flexible and powerful abstraction for specifying interactions and couplings so that users can concentrate on the algorithmic and numerical aspects of the application, while at the same time enabling scalable and efficient implementations.

1.2 Overview of Seine

This goal of this dissertation is to provide a high level abstraction to the application layer to ease the development of parallel scientific applications, while still enabling its efficient and scalable implementations. The work achieves this goal by formulating a semantically specialized shared space model. The shared space abstraction provides the application with flexibility to support extremely dynamic and complex communication and coordination patterns. To enable scalable and efficient implementation, the shared space is specially customized to specific parallel scientific applications/domains.

The key contribution of this work is that it lays out a theoretical foundation and provides a practical implementation of an interaction framework that facilitates flexible and efficient coupling within or between parallel scientific applications. This approach emphasizes flexibility, efficiency, and scalability. The system complements and can be used in conjunction with existing parallel programming systems such as MPI and OpenMP.

The research develops Seine, a dynamic semantically specialized shared space-based interaction framework for parallel scientific applications which is intended to bridge the gap between the requirements of emerging parallel scientific applications and the capabilities of existing parallel programming paradigms. The framework comprises three layers: directory layer, storage layer, and communication layer. The directory layer essentially supports the dynamic generation of communication schedules that masks the complexity of handling the communication/coordination requirements and presents to the application developer an application-oriented interface for interacting with other nodes in the system. Due to diversity of parallel scientific applications, the shared space directory service is customized towards different scientific applications. In other words, the layer provides a domain-specific shared space abstraction and because it is oriented to specific fields, customizations can be done to achieve system efficiency and other desirable properties. In this work, two different shared space abstractions are proposed and applied to applications in diverse scientific disciplines. One is the abstraction of a dynamic geometry-based shared space, which can be applied to system modeled with geometric problem domains. The other is the abstraction of a shared temperature space, which can be applied to molecular dynamics applications. The storage layer is the storage for the shared space. The communication layer provides efficient data transfer and possibly supports application specific communication protocols. In summary, Seine proposes an architecture that

consists of discovery service to manage the dynamic and complex communication patterns, the storage and communication service needed for realizing efficient interaction and the shared space based abstraction. Seine-based systems can consist some or all of the layers according to specific application requirements.

1.2.1 Seine Prototype Systems

Three prototype implementations of the Seine model have been developed.

Seine-Geo

Seine-Geo is a prototype system that presents a geometry-based shared space abstraction. It consists of all three layers described above. The system supports geometry-based object sharing semantics, space dynamism, and scalable realizations. The Seine model builds on two key observations: (a) formulations of most scientific and engineering applications are based on multi-dimensional geometric domains (e.g., a grid or a mesh) and (b) interactions in these applications are typically between entities that are geometrically close in this domain (e.g., neighboring cells, nodes or elements). Rather than implementing a general and global associative space, Seine defines geometry-based transient interaction spaces, which are dynamically created at runtime, and are localized to specific sub-regions of the global geometric domain. Each transient interaction space is defined to cover a closed region of the application domain described by an interval of coordinates in each dimension. The interaction space can then be used to share objects between processors whose computational sub-domains geometrically intersect with that region. To share an object using the interaction space, processors do not have to know of, or synchronize with each other at the application layer. Sharing objects in the Seine model is similar to that in a tuple space model. Furthermore, multiple shared spaces can co-exist simultaneously in the application domain. The framework complements existing interaction frameworks (e.g., MPI [2, 52, 53], OpenMP [54, 55]) and provides scalable geometry-based shared spaces for dynamic runtime coordination and localized communication is presented. The framework uses the Hilbert Space Filling Curve (SFC) [45], a locality preserving recursive mapping from a multi-dimensional coordinate space to a 1-dimensional index space, to construct a distributed directory structure that enables efficient registration of geometric shared

spaces and lookup of objects in the shared space. It is applied to intra-coupling problem of an adaptive oil reservoir simulation.

Seine-Coupe

Seine-Coupe is a prototype system that builds on Seine-Geo. It still presents the geometry-based shared space abstraction and has the same architecture as the Seine-Geo prototype system. The difference is that the system focuses on solving the parallel data redistribution problem, an important issue at the computer science aspect of the general code coupling problem. The foundation of Seine-Coupe is the same as Seine-Geo, i.e., couplings occur at shared boundary, interface, common volume at application problem domains. However, for parallel data redistribution, we additionally propose an abstract array index space when the domain definition and the data to be redistributed do not change over the distribution. The abstract array index space can be used when the geometry of the problem domain is not readily available or as a general way to handle the redistribution despite of the domain definition and decomposition used in the application. Seine-Coupe is applied to the inter-coupling problem of a mocked Plasma Science Simulation to couple two codes executed independently and in parallel.

Seine-Salsa

Seine-Salsa is a prototype system that presents a shared temperature space abstraction to molecular dynamics applications. It consists of a directory layer and a communication layer imbedded with an application-specific communication protocol. The system is applied to improving the Replica Exchange algorithm in molecular dynamics field. It is able to support an asynchronous and scalable implementation of the algorithm and significantly improve the quality of the simulation results.

1.2.2 Contributions

In this thesis, we

- Outline the communication and interaction requirements of emerging large-scale parallel scientific applications and analyze the limitations of existing parallel programming models in addressing these requirements.

- Formulate the semantically specialized shared space model that encapsulates domain specific information to enable scalable implementations and complements existing parallel programming models.
- Apply the framework prototype to three different classes of applications and evaluate its efficiency and flexibility in its support to addressing communication requirements in these parallel scientific applications.

1.3 Outline

The rest of the thesis is organized as follows.

Chapter 2 first defines problem space of this dissertation. It then surveys existing parallel programming paradigms and discusses their limitations with respect to the emerging communication requirements. Further, it describes the Seine system, a dynamic geometry-based shared space interaction framework, including its architecture, operation semantic, and interface.

Chapter 3 presents how Seine framework addresses the intra-coupling problem. A motivation application of an adaptive oil reservoir simulation is given and preliminary test results from a mock simulation are listed and analyzed to evaluate efficiency and scalability of the coupling system.

Chapter 4 presents how Seine framework addresses the inter-coupling problem for coupled system composed from independent parallel scientific applications. It specifically investigates into the parallel data redistribution problem The prototype is used in a mocked SciDAC Plasma Edge Simulation as a proof of concept experiment to prove the efficiency and effectiveness in supporting parallel data redistribution in large-scale coupled system. It also discusses related work of coupling support that have been designed for coupled simulations, focusing on the recent work on parallel data redistribution.

Chapter 5 describes generalizing Seine approach to meet specific communication requirements of the Replica Exchange algorithm in Molecular Dynamics applications. The prototype system presents a shared temperature space abstraction to the molecular dynamics applications based on which scalable and asynchronous replica exchange can be implemented. Experimental result shows the effectiveness of Seine-based Replica Exchange in improving the result

quality.

Chapter 6 concludes our work and gives out future directions.

Chapter 2

Seine: A Dynamic Shared Space Interaction Framework for Parallel Scientific Applications

2.1 Problem Description

As discussed, supporting the dynamic and complex communication/coordination patterns presented by emerging parallel scientific applications is an important and challenging requirement. In this research, we have encountered three types of interaction/coupling problems in various scientific research fields, including multi-block coupling within a parallel scientific application, code or model coupling between multiple parallel scientific applications, and asynchronous pair wise interactions in a parallel scientific application.

In the rest of this chapter, we will first describe existing solutions to these problems and discuss their limitations. We will then introduce the Seine approach as a better solution to these problems.

2.2 Parallel Programming Models

Parallel programming distinguishes itself from sequential programming through the concurrent execution of a division or portion of the entire work. In the ideal case, individually executed tasks are as independent as possible so that little or no synchronization is needed to complete the entire work. However, such an ideal case rarely exists in real application scenarios. Often, communication and synchronization times can dominate the overall execution time of the application. As a result, communication and synchronization must form a core part of parallel program. Based on how communication is carried out, parallel programming models include message-passing, shared-memory, and the tuple space model.

2.2.1 The Message-Passing Paradigm

The message-passing parallel programming model is a generally accepted, well-understood and widely used paradigm. It is oriented toward distributed memory machines and is suited for SPMD applications. The Message-Passing Interface [2] is the *de facto* standard for message passing.

In this paradigm, as its name suggests, inter-process communication is done by passing messages between nodes. Each message originates from a sender and ends at a receiver. It provides a communication interface, and positions itself as a layer specialized for communication that has no knowledge of the application layer. This communication layer is able to provide generic communication support, leaving application-specific coordination/communication scheduling to the upper application layers. Such communication layer-oriented interface, however, complicates the development of upper application layers when the required communication/coordination patterns are dynamic, complex, and unpredictable. Meeting these requirements using message passing is not intuitive and non trivia.

Nevertheless, the message-passing paradigm has had an enormous impact on parallel scientific computing because it provides useful abstraction and efficient implementations for inter-process communication. The existence of a *de facto* standard further enhances its usability and popularity. A significant part of parallel scientific applications to date are based on this paradigm.

2.2.2 The Shared-Memory Paradigm

In the Shared-Memory paradigm, processes communicate by reading/writing to commonly accessible memory areas. It is well accepted that a shared memory abstraction is more desirable from the application programmer's viewpoint than the message-passing abstraction, allowing them to focus on domain-specific or algorithmic development rather than on managing intricate interprocess communications as in message passing. In other words, frameworks based on shared address spaces provide higher-level abstractions that can more effectively support dynamic and complex interactions. Unfortunately, shared memory systems are usually expensive and manufacture-customized. There are efforts focused on using distributed memory systems to emulate the shared memory abstraction. However, these virtual distributed shared memory systems provide acceptable performance for only a limited class of applications. Implementing

scalable shared memory systems remains a challenge.

While message-passing and shared-memory are the dominant parallel programming paradigms, a generative communication paradigm was proposed in 80's by David Gelernter as the basis for a new distributed programming language called Linda [12]. Linda is fully distributed both in space and in time, and provides a simpler and more expressive alternative for many application domains.

2.2.3 The Tuple Space Model

An abstract computation environment called "tuple space" is the basis for Linda's model of communication [12]. The tuple space model provides a very flexible and powerful mechanism for extremely dynamic communication and coordination patterns. In the model, processes interact using an associative shared tuple space. A tuple is a sequence of fields, each of which has a type and contains a value. The producer of a message formulates the message as a tuple and places it into the tuple space. The consumer(s) can associatively look up relevant tuples using pattern matching on the tuple fields. The tuple space model provides two fundamental advantages: simplicity and flexibility. The communicating nodes need not care about who produced or will consume a tuple. Furthermore, the communicating processes do not have to be temporally or spatially synchronized. This decoupling feature automatically supports dynamic communication/coordination. However, scalable implementation of tuple spaces remains a challenge. In a pure tuple space environment, all the communication passes through a central tuple space with relatively slow associative lookup mechanisms [43], which is an inherent bottleneck impeding scalability and efficiency.

Several research projects have addressed this model and its implementation. These include commercial products and research prototypes with varied foci, such as JavaSpaces [15], TSpaces [16, 47], XMLSpaces [17], Lime [18], PeerWare [19], PeerSpace [20], and Comet [21]. The conceptual model underlying this research is based on the Tuple Space model, with a special customization towards the application domain, which allows it to avoid some of the overheads and consistency issues of a general shared space.

Above is a general description about the backgrounds related to our target problem. Related works which are specific to different application domains will be discussed in future chapters

when we discuss the Seine approach to these domains of applications.

2.3 Overview of Seine

Seine is a dynamic semantically specialized shared space interaction framework, based on the tuple space conceptual model. As in the tuple space model, communicating entities in Seine interact with each other by sharing objects in a logically shared space. However, Seine differs from the tuple space in that, by defining application oriented interface primitives, it naturally exploits application-/domain-specific knowledge to provide efficient and scalable support for dynamic and complex interactions and coordinations. The motivation of the inclusion of certain amount of application-specific knowledge is the following.

The tuple space, a generative communication model, is simpler and expressive as the model is formulated to the application layer, instead of the communication layer. However, with very little or no knowledge about the target applications, it is very difficult to realize a generic globally shared tuple space with efficiency because the design not only needs to provide the generality to cover the entire scope of the application domains in various fields but also needs to provide global shared space transparency to every process in the execution environment. On one hand, it needs to provide a generic tuple matching semantic that is applicable to every application; On the other hand, the tuple space needs to maintain a global consistent view across all the process in the execution environment.

Seine is based on the Tuple Space model and inherits its expressiveness, flexibility, and simplicity. Furthermore, Seine also incorporates application-specific information. The derivation is two-fold. Firstly, we survey existing parallel scientific applications and summarize common characteristics of this type of applications. This information is used as application-/domain-specific information in the customization of the Tuple Space to application-/domain-specific shared space. To this end, generic associative tuple matching semantic in the Tuple Space model is replaced by application-specific semantics, avoiding the overheads in handling generic associative semantics. Secondly, the global space sharing in the Tuple Space model is replaced by localized space sharing: sub-spaces are defined which dynamically cover only a portion the entire application domain and provides transparent access to only a dynamic subset of nodes in

the application execution environment. These customizations are feasible due to two important observations about parallel scientific applications. First, scientific applications are typically formulated on a discretization of the problem domain (e.g., mesh, grid, molecules dividing a temperature space); second, interactions in these applications are based on this discretization and are typically localized to portions or sub-domains of the discretized domain, e.g., neighboring particles, blocks, cells or elements, walkers with neighboring temperatures in molecular dynamics systems, etc. Seine shared spaces cover these localized interaction portions of the application domain and only span the processes to which the portion is mapped, which is typically a small subset of the entire process group in the execution environment.

As discussed above, processes in the Seine model interact by reading/writing data or objects from/to the virtual shared spaces. Typically interactions in parallel scientific applications are regular in the sense that processes in the applications have well-defined role as reader or writer due to the application formulation. As a simple example, imagine a parallel application running on two processors and consist of a loop. At each loop, one processor (say processorA) first writes an object (say objectA) to a geometric region (say regionA) in the geometry-based shared space then reads an object (say objectB) associated with another region (say regionB) from the shared space while the other processor (say processorB) first reads objectA from regionA in the shared space and then writes objectB to regionB in the shared space. If irregular interactions occurs, e.g., a third processor writes objectA to regionA in shared space at the same time as processorA writes to the region, the data that processorB will get when it tries to read object from regionA in the shared space are indeterministic. However, since we are targeting specifically at parallel scientific applications, which have very well-defined regular access to the shared space, we believe that seine defines a sufficient model for addressing communication/coordination patterns in this type of applications.

In scientific computing, the specific domains or fields of applications can differ significantly. Nevertheless, we observe that, despite of their different domains and fields, applications have notion of “space”. For example, many scientific simulations address realistic models of natural phenomena or systems by formulating the problem domain using geometric information such as (relative) physical locations and/or positions of the domain. Other scientific simulations investigate systems with different interests other than physical location and position, such as

temperature, energy, etc. In this case, the space concept is broadened from the geometry-based one to cover more of these application specific interests, such as temperature space or energy space, etc. Seine essentially abstracts the broadened space concept and provides applications with a dynamic shared space interaction framework, with the space being customized towards each specific domain of applications as necessary. This work investigates the shared space abstraction of two application-specific types, a geometry-based shared space abstraction and a shared temperature space abstraction. The former is applied to intra-coupling of a parallel scientific simulation (chapter 3) and inter-coupling of two independent simulation systems (chapter 4). The latter is applied to a molecular dynamics simulation to overcome the limitation of existing replica exchange algorithms.

In summary, Seine Provides a semantically specialized dynamic shared space abstraction. The space is dynamic in the sense that it is created and destroyed at runtime based on the changing communication/coordination requirements of the application. Further, the processors in a space can change (possibly due to dynamic redistribution or load balancing). Seine facilitates building dynamic and complex communication and interaction patterns through the de-coupling and self-organizing capabilities of the shared space paradigm. Further, Seine alleviates programmers from manually wiring communication patterns for each process during application development. Finally, Seine maintains scalability and efficiency by exploiting the locality of communications and interactions in the application domain.

2.4 Seine Architecture

A schematic of the architecture of the Seine shared space interaction framework is presented in Figure 5.1. Seine design comprises three main components: directory layer, storage layer, and communication layer.

2.4.1 The Distributed Directory Layer

The directory layer is implemented as a distributed hash table (DHT). The index of the DHT is semantically specialized and retrieved from the application domain. This is possible because,

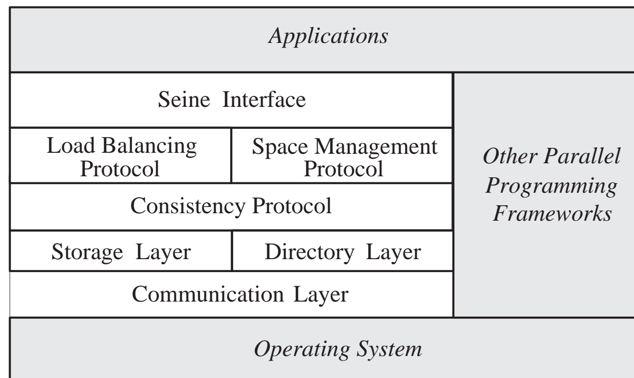


Figure 2.1: Seine architecture

as we mentioned, most scientific applications have the notion of “space”, such as geometry-based space, temperature space, energy space, etc. Based on this “space” notion, it is possible to create domain-specific shared space abstraction. The abstraction helps conceal the complication of explicitly building communication patterns by letting distributed processes interact with the abstract shared space when needed, instead of tediously determining which node(s) to send/receive message to/from for each interaction as required by MPI paradigm. The shared space abstraction is mainly realized by the Seine distributed directory, which provides the discovery service to the distributed processes so that they are able to identify efficiently and locally the nodes that should communicate with based on current communication requirement.

The index of the DHT is constructed from 1/n-dimensional abstract shared space based on application domain using the Hilbert space filling curve (SFC). Space-filling curves [45, 46] are a class of locality preserving mappings from d-dimensional space to 1-dimensional space, i.e. $N^d \rightarrow N^1$, such that each point in N^d is mapped to a unique point or index in N^1. Using this mapping, a point in the N^d can be described by its spatial or d-dimensional coordinates, or by the length along the 1-dimensional index measured from one of its ends. The construction of SFCs is recursive and the mapping functions are computationally inexpensive, which consist of bit level interleaving operations and logical manipulations of the coordinates of a point in multi-dimensional space. SFCs are locality preserving in that points that are close together in the 1-dimensional space are mapped from points that are close together in the d-dimensional space.

The distributed directory layer is constructed in the following way. Given the application

domain, the entire scope of the domain of interest of the application is decomposed into sub-domains, each of which is assigned to a directory service node. A directory service node is a normal computing node in the application execution environment which, aside from computation task given by the application, will also be responsible for handling shared space relevant requests whose operations are on sub-domains that overlap with the sub-domain(s) assigned to this directory service node. By using the DHT, a shared space operation on a specific portion/sub-domain of the application will always be routed to those directory service nodes whose assigned portions/sub-domains overlap with this specific portion/sub-domain.

Depending on the fields of the application as well as the dimension of application domain, the directory layer is built in different ways, which can be grouped into two categories. For applications with 1-dimensional shared space abstraction, the directory layer is constructed by indexing the global dimension of the 1-dimensional shared space and dividing them into a number of intervals, each of which is assigned to a directory service node. For applications with multi-dimensional shared space abstraction, the multi-dimensional space is first mapped to 1-dimensional index space using Hilbert Space Filling Curve (SFC) and then the 1-dimensional index space divided into a number of intervals and assigned to each directory service node. More details about the distributed directory layer is given in the following chapters when the designs and implementations of several Seine-based prototype systems are discussed.

It is this layer that actually distinguishes Seine approach from the MPI-based approach since it sits between the communication layer and the application layer and presents to the application a domain-specific shared space abstraction that is intuitive to the application layer, concealing the complexity of explicitly building the communication/coordination patterns.

2.4.2 The Storage Layer

The Seine storage layer is used to store objects in registered shared spaces. The storage for a shared space is locally maintained at each of the processes that have registered the space. Shared objects are stored at the processors that own them and are not replicated. The layer is created where necessary. It is designed as a list structure, each item of the list being a registered space. Each space on the list further consists of shared objects within the space, which is also organized in a list structure.

2.4.3 The Communication Layer

In Seine, communication between processes is handled by the communication layer and occurs in a peer-to-peer manner, based on the communication wiring information provided by the distributed directory layer. The layer provides basic data transfer support. Its design also includes buffer management to enable efficient and concurrent data transfer in the case of large volume data transfer requirements. Further, application-specific communication protocols can be embedded into this layer to meet the specific need of applications.

2.5 Overview of the Prototypes

In the next three chapters, we will discuss three prototype implementations of the Seine approach. They are:

1. Seine-Geo, which realizes a geometry-based shared space abstraction and is applied to an adaptive multi-block oil reservoir simulation to ease the complexity in building coordination patterns between processes assigned to different blocks. It is achieved by building geometry-based shared space around interface regions between the blocks in the realistic model of the application.

2. Seine-Coupe, which uses Seine-Geo abstraction and design to provide the MxN parallel data redistribution support for code/data coupling of parallel scientific simulations. The system is applied to a simulation to emulate the code coupling scenario in SciDAC Fusion project.

3. Seine-Salsa, which realizes a shared temperature space for molecular dynamics applications. Based on the abstraction, it is able to realize the “replica exchange” algorithm in a decentralized and asynchronous manner. The “replica exchange” algorithm simulates the structure, function, folding, and dynamics of proteins. Seine-Salsa provides a scalable communication and interaction substrate that presents a virtual shared space abstraction and enables the dynamic and asynchronous interactions required by the simulations to be simply and efficiently implemented.

Chapter 3

Seine-Geo: A Seine-based Intra-Coupling Framework for Parallel Scientific Applications

Seine-Geo is a Seine-based prototype system, which provides a dynamic geometry-based shared space abstraction to the application layer. It focuses on the coupling problem within a parallel scientific application and facilitates building complex and dynamic communication patterns by presenting the shared space abstraction to the application while still supporting peer-to-peer direct communication underneath.

3.1 An Illustrative Application: Interface Coupling in Parallel Adaptive Multi-block Oil Reservoir Simulation

The parallel multi-block oil reservoir simulation is used as an illustrative driving application to motivate Seine-Geo and to derive its requirements. The application aims to accurately simulate fluid flows in porous media by using appropriate algorithms to simulate the physical processes, which may include multi-phase, multi-component flow and transport, geomechanics, bio-geochemistry and geophysics and are described by nonlinear elliptic, parabolic, wave and differential-algebraic equations possibly coupled within and across sub-domains [50, 51]. Specifically, the driving application is a subsurface simulation model, which consists of a complex interaction of fluid and rock properties that evolves with time. To achieve the desired efficiency and accuracy in the representation of the different phenomena taking place in the subsurface, the simulation provides the support for different scales (multi-scale), processes (single phase, oil-water, air-water, three-phases, compositional), and algorithms or formulations (IMPES, fully implicit) through a multi-block approach [48, 49]. In the simulation, the oil reservoir is discretized as a series of blocks and interfaces between blocks. The numerical formulation consists of a coupled system of highly nonlinear transient partial differential equations. Its geometrical and geological features induce a multi-block decomposition so that each block is

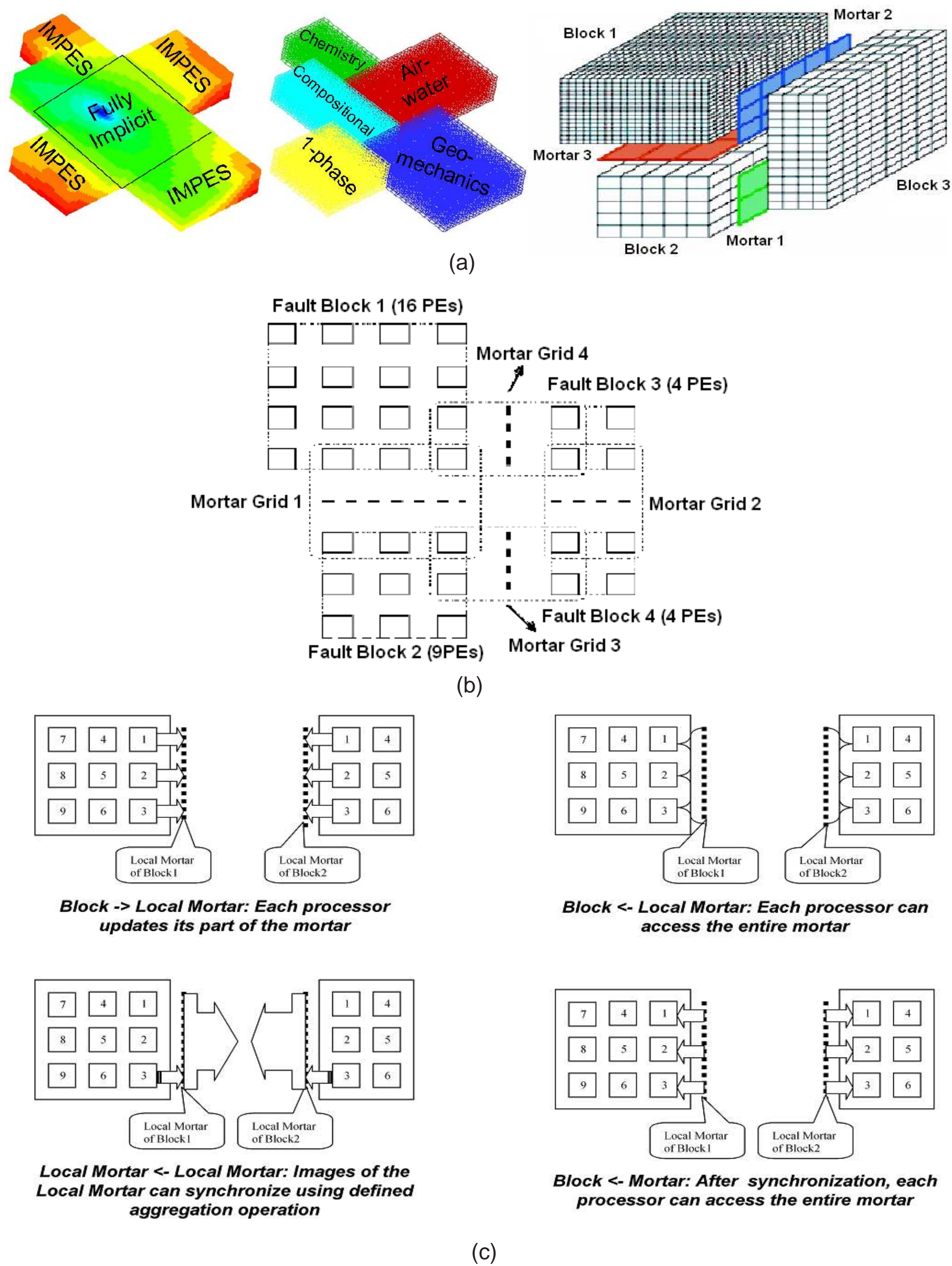


Figure 3.1: (a) Various aspects of the multi-block formulation (left: multi-numerics, center: multi-physics, right: multi-scale); (b) 2-D view of decomposed multi-block domains and mortar grids [44]; (c) Illustration of key communication between multi-block domains and mortar grids.

discretized by cell-centered finite differences on logically rectangular grids. Flux matching conditions are imposed on the interfaces and a non-overlapping domain decomposition algorithm is exploited so that solving the interface problem only requires in-block solves and an exchange of interface values between neighboring blocks [44]. Figure 3.1 (a) shows various aspects of multi-block coupling. Figure 3.1 (b) shows a 2-D view of a decomposed multi-block domain and mortar grids. From the figure we can observe that communication between blocks in this formulation is localized to the interfaces between neighboring blocks. Some of the key multi-block communications using mortar grids are illustrated in Figure 3.1 (c). As the figure shows, a processor on one side of the block interface needs to locate which processor on the other side of the block interface it should communicate with The challenge in implementing these communication/interaction patterns is that when the decomposed sub-blocks are distributed across the processors in a parallel system, locating the processor assigned to a neighboring block and generating the communication schedules required by typical message passing systems is non-trivial, especially when dynamic load-balancing is used and sub-blocks can move from one processor to another to balance load. Further complexity is introduced when the grid block and/or the mortar grids are dynamically refined. Due to the complexity, the MPI-based implementation uses a master node at both sides of the block interface to collect data from all the processors on the side. The master node then sends the collected data to the master node of the other side of the block interface, which then broadcast the data to all the processors at that side of the block interface. As seen, the complexity is handled by using synchronous or collective operations that are essentially not necessary, sacrificing overall system efficiency.

3.2 Seine-Geo: A Dynamic Geometry-based Shared Space Interaction Framework for Parallel Scientific Applications

A Seine-Geo shared space supports interactions corresponding to mortar grid that are associated with its geometric region. The shared space abstraction mediates the interaction between processors so that they need only to put/get geometric object to/from the shared space without delving into the details such as which processor(s) it should communicate with. Multiple non-intersecting shared spaces can co-exist in Seine-Geo. The geometry-based shared spaces are

dynamically created as needed, used for communications and interactions, and destroyed when no longer needed (e.g., application enters a new phase).

Objects shared using Seine-Geo are geometry-based, i.e, each object is associated with a region of the discretized application domain. In addition to data and data descriptors, each geometry-based object is annotated with a geometry descriptor specifying the regions that the object is associated with, and a tag. The geometry descriptor is a box function specifying the coordinates along each dimension. The region specified by this geometry descriptor is used to store and retrieve object to/from the space. The tag is a symbolic name used to identify different objects associated with the same geometric region, for example, different grid functions defined on the same grid.

Note that unlike the general shared object model, where an object is completely shared or not at all, in Seine-Geo an object may be partially shared. For example, if regions associated with two processors are not identical but do intersect, sharing may be limited to the portions of the objects that correspond to the region of intersection. If the object is an array, only the part of the array corresponding to the region of intersection would be read or written in this case. Similarly, if two array objects, *A* and *B*, in the space are associated with regions *regionA* and *regionB* with intersection *regionAB*, when object *A* is written into the space, only the part of the object *A* that corresponds to *regionAB* is propagated to object *B*. Note that this may not be meaningful or possible for all types of objects.

An overview of Seine-Geo space is presented in Figure 3.2. The figure shows a 2-dimensional Seine-Geo shared space and illustrates how 2-dimensional geometric objects are shared using the space. Note that Seine-Geo can support 1, 2 and 3 dimensional spaces and objects and can be extended to support spaces with even higher dimensional.

3.2.1 The Seine-Geo Architecture

The Seine-Geo Directory Layer

The Seine-Geo distributed directory is used to (1) detect relationship between registered objects, (2) manage the creation of shared spaces based on the geometric relationship detected so that all objects associated with intersecting geometric regions are part of a single shared space,

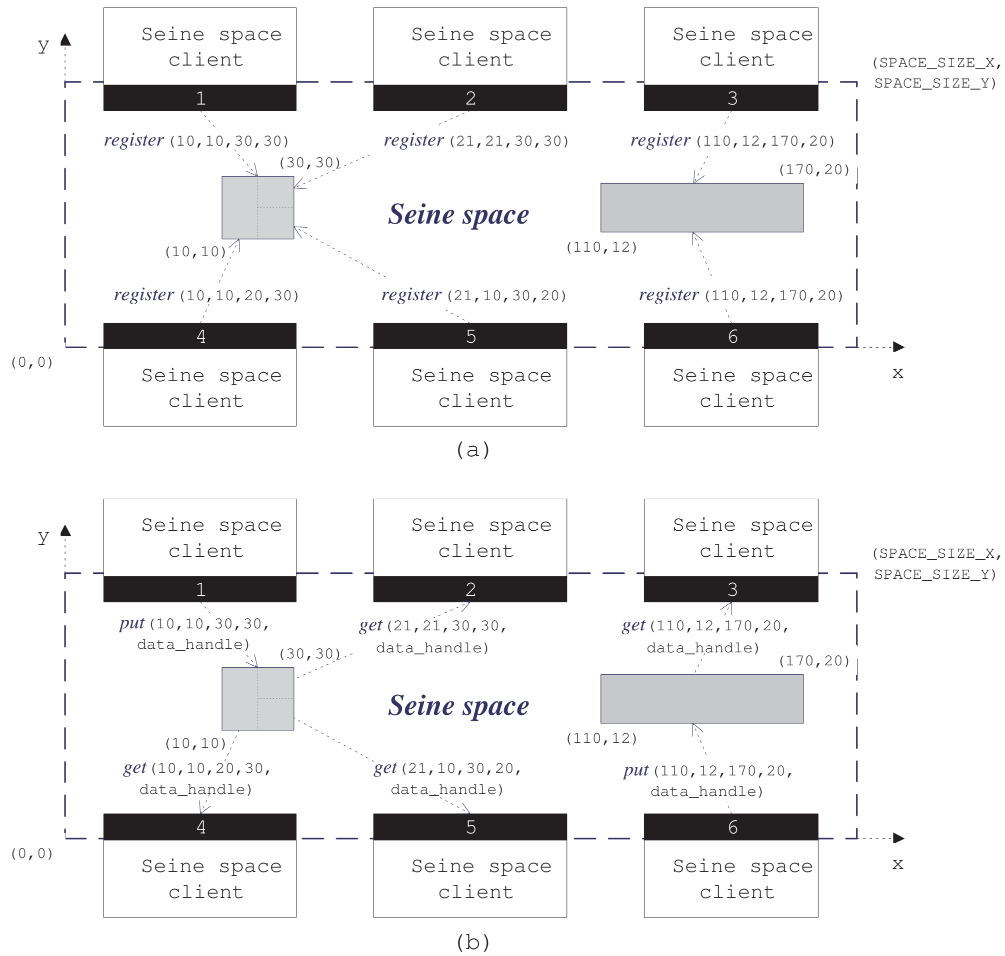


Figure 3.2: Overview of Seine-Geo space (a 2-D example); (a) Phase 1: *register* a geometric region; [operator: *register*(x_low_coord, y_low_coord, x_high_coord, y_high_coord)]; (b) Phase 2: *put*/*get* geometric objects; [operator: *put*/*get*(x_low_coord, y_low_coord, x_high_coord, y_high_coord, data_handle)]

(3) manage the operation of the shared spaces during their lifetimes including the merging of multiple spaces into a single space and the splitting of a space into multiple spaces, and (4) manage destruction of a shared space when it is no longer needed.

The index space of the DHT is directly constructed from the geometry of the discretized computational domain using the Hilbert space filling curve (SFC). The Hilbert SFC is used to map the d-dimensional coordinate space of the computational domain to the 1-dimensional index space of the hash table. The index space is then partitioned and distributed to the processors in the system. As a result, each processor stores a span of the index space and is responsible for the corresponding region of the d-dimensional application domain. The processor manages the operation of the shared space in that region, including space creation, merges, splits, memberships and deletions. As mentioned above, object sharing in Seine-Geo is based on their geometric relationships. To share object corresponding to a specific region in the domain, a processor must first register the region of interest with the Seine-Geo runtime. A directory service daemon at each processor serves registration requests for regions that overlap with the geometric region and corresponding index span mapped to that processor. Note that the registered shared spaces may not be uniformly distributed in the domain and as a result, registration load must be balanced while mapping and possibly re-mapping index spans to processors. The mapping of a 2-dimensional domain using the Hilbert SFC and the structure of the resulting Seine distributed directory layer are illustrated in Figure 3.3. To register a geometric region,

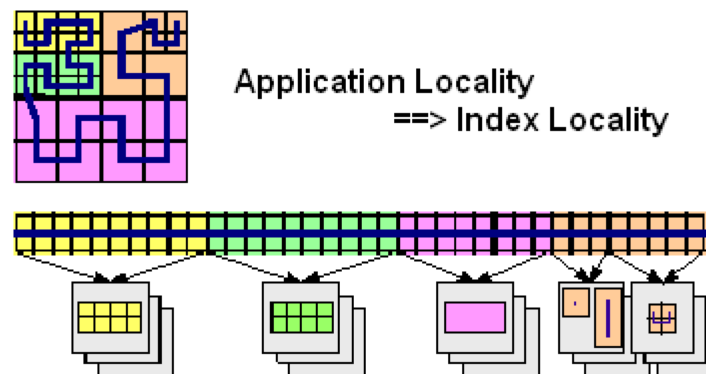


Figure 3.3: Seine directory structure using the Hilbert SFC [44].

the Seine-Geo runtime system first maps the region in the d-dimensional coordinate space to

a set of intervals in the 1-dimensional index space using the Hilbert SFC. The index intervals are then used to locate the processor(s) to which they are mapped. The process of locating corresponding directory processors is efficient and only requires local computation. The directory service daemon at each processor maintains information about currently registered shared spaces and associated regions at the processor. Index intervals corresponding to registered spaces at a processor are maintained in an interval tree.

A new registration request is directed to the appropriate directory service daemon(s). The request is compared with existing spaces using the interval tree. If overlapping regions exist, a union of these regions is computed and the existing shared spaces are updated to cover the union. Note that this might cause previously separate spaces to be merged. If no overlapping regions exist, a new space is created.

The Seine-Geo Storage Layer

The Seine-Geo storage layer consists of the local storage associated with registered shared spaces. The storage for a shared space is maintained at each of the processors that have registered the space. Shared objects are stored at the processors that own them and are not replicated. When an object is written into the space, the update has to be reflected to all processors with objects whose geometric regions overlap with that of the object being inserted. This is achieved by propagating the object (or possibly corresponding parts of the object) to the processors that have registered overlapping geometric regions. Such an update propagation mechanism is used to maintain consistency of the shared space. As each shared space only spans a local communication region, it typically maps to a small number of processors and as a result update propagation does not result in significant overheads. Further, unique tags are used to enable multiple distinct objects to be associated with the same geometric region.

The Seine-Geo Communication Layer

Since coupling and parallel data redistribution for scientific application typically involves communicating relatively large amounts of data, efficient communication and buffer management is critical. Further, this communication has to be directly between individual processors. Currently Seine-Coupe maintains the communication buffers at each processors as a queue, and

Table 3.1: Seine Programming Interface.

Interface Operators	Function Description	Corresponding Linda Operators
init	*init* uses a bootstrap mechanism to initialized the Seine runtime system.	n/a
register	*register* registers a region with the Seine-Geo framework. Based on the geometric descriptor registered, a reference to an existing space or a newly created space is returned.	n/a
put	*put* inserts an object into the shared space.	out
get	*get* removes an object from the shared space. The *get* operator is blocking.	in
rd	*rd* copies an object from the shared space without removing it from the space. Multiple *rd* can be simultaneously invoked on an object.	rd

multiple sends are overlapped to better utilize available bandwidth [22]. Adaptive buffer management strategies described in [22] are being integrated.

3.2.2 Seine-Geo Programming Interface

The access operators provided by the Seine-Geo programming interfaces are similar to those provided by general tuple space based systems such as Linda [12]. The exception is the *eval* function, which is not supported by Seine-Geo. The Seine-Geo programming interface is listed in Table 3.1. It includes operators to initialize the Seine-Geo runtime, to allow processors to join and leave a space, and to access the space. The Seine-Geo runtime is initialized using the *init* operator. The creation/destruction of a space does not require global synchronization and processors can individually and dynamically join or leave a space at runtime. A processor joins a space by registering its region of interaction. A processor leaves a space by de-registering the relevant region. When the last processor associated with a space de-registers, the space is destroyed. A processor inserts an object into the shared space using the *put* operator, which is functionally similar to *out* in Linda. A processor can retrieve an object using the *get* operator, which is functionally similar to *in* in Linda. The *get* operator is blocking and will wait until a matching object is written into the space. The *rd* operator is similar to *get*, except that unlike *get*, the object is not removed from the space. Arguments to these operators include a geometry descriptor to identify the space of interest and a tag to identify the object of interest.

3.2.3 Seine-Geo Operation

The operation of the Seine-Geo framework is illustrated in Figure 3.4 using a 2-dimensional application domain. The initialization of the Seine-Geo runtime using the *init* operator is shown in Figure 3.4 (a). During the initialization process, the directory structure is constructed by mapping the 2-dimensional coordinate space to a 1-dimensional index using the Hilbert SFC and distributing index intervals across the processors.

In Figure 3.4 (b), processor 1 registers an interaction region R1 (shown using a lighter shade in the figure) in the center of the domain. Since this region maps to index intervals that spans all four processors, the registration request is sent to the directory service daemon at each of these processors. Each daemon services the request and records the relevant registered interval in its local interval tree. Once the registration is complete, a shared space corresponding to the registered region is created at processor 1 (shown as a cloud on the right in the figure).

In Figure 3.4 (c), another processor, processor 0, registers region R2 (shown using a darker shade in the figure). Once again, the region is translated into index intervals and corresponding registration request are forwarded to appropriate directory service daemons. Using the existing intervals in its local interval tree, the directory service daemons detect that the newly registered region overlaps with an existing space. As a result, processor 0 joins the existing space and the region associated with the space is updated to become the union of the two registered regions. The shared space also grows to span both processors. As more regions are registered, the space is expanded if these regions overlap with the existing region, or new spaces are created if the regions do not overlap.

Once the shared space is created, processors can share geometry-based objects using the space. This is illustrated in Figures 3.4 (d) and (e). In Figure 3.4 (d), processor 0 and processor 1 use the *put* operation to insert object 2 and object 1 respectively into the shared space. As there is an overlap between the regions registered by processors 0 and 1, the update to object 1 is propagated from processor 0 to processor 1. Similarly, the update to object 2 is propagated from processor 1 to processor 0. The propagated update may only consist of the data corresponding to the region of overlap, e.g., a sub-array if the object is an array. In Figure 3.4 (e), processor 1 and processor 0 retrieve object 1 and object 2 respectively using a local *get* operation.

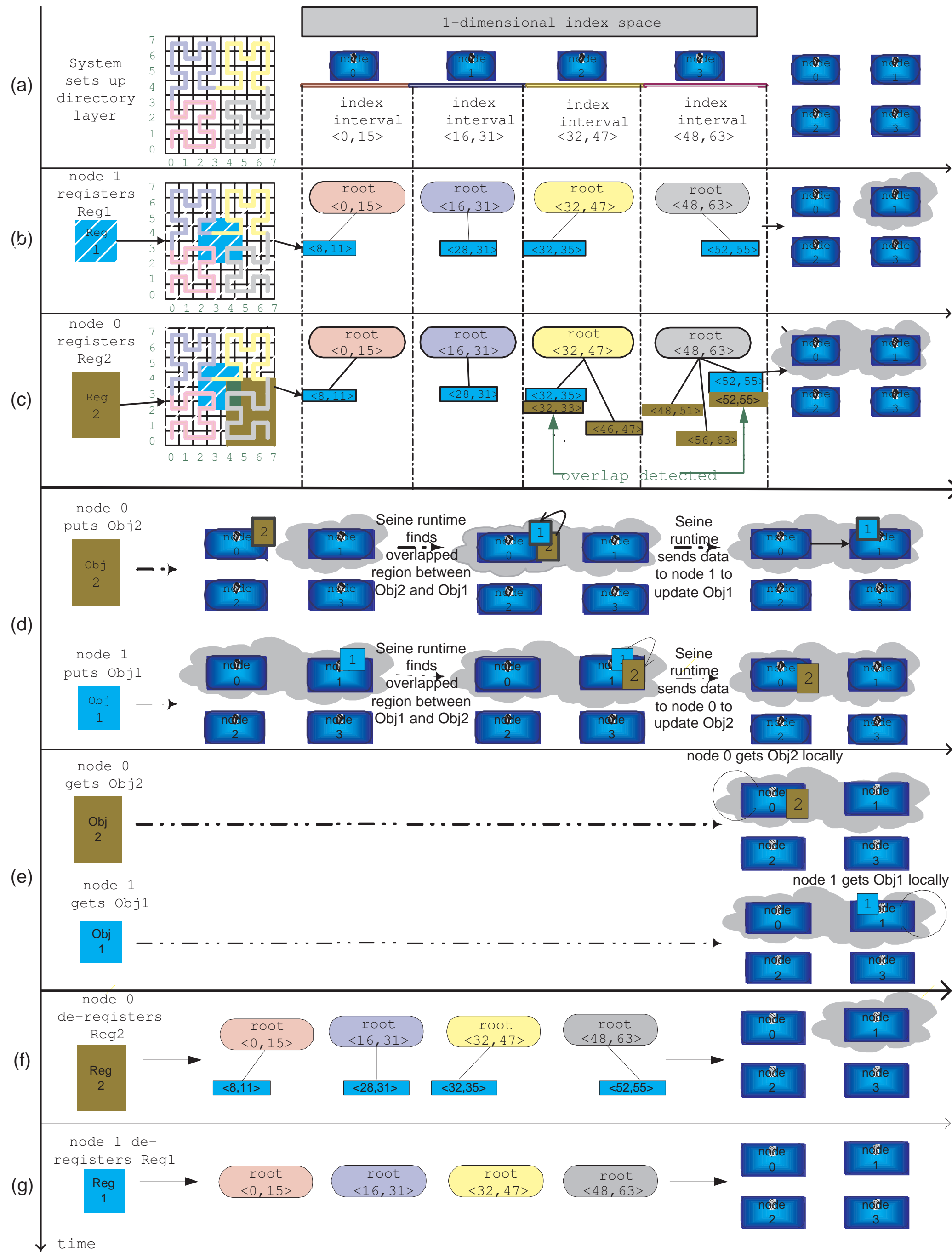


Figure 3.4: Operation of the Seine-Geo framework.

Finally, processor 0 and processor 1 respectively de-register region R2 and region R1 from the shared space, as shown in Figure 3.4 (f) and (g).

3.3 Prototype Implementation and Performance Evaluation

3.3.1 Prototype Implementation

A thread-based prototype implementation of the Seine-Geo interaction framework has been developed. The prototype is targeted to Unix systems including Linux workstations and the IBM SP, and works in tandem with MPI or any other communication substrate. A key component of the implementation is the *Seine-Geo-daemon* thread, which is created at application startup within the user application process on each processor. This daemon handles registration requests by retrieving and updating local directory entries. In the current implementation, *Seine-Geo-daemon* instances bootstrap using a statically defined startup server, which is known a priori to all processors. Besides the *Seine-Geo-daemon*, the other key component is *Seine-Geo-storage*, which stores shared objects. To create a shared space at runtime, a processor registers its region of interest with the distributed directory layer, which is then forwarded to the *Seine-Geo-daemon* at each processor. On receiving the registration request, the *Seine-Geo-daemon* retrieves its local directory to determine whether the registered region intersects with an existing space or if a new space should be created. The *Seine-Geo-daemon* returns a pointer to the existing or a new space, which can then be used by the applications for interactions. The storage itself is a table of objects. Table 3.2 lists sample application pseudo code for initializing Seine-Geo, registering a region, using the space for sharing objects and interacting, and deregistering the region.

3.3.2 Experimental Evaluation

The performance of the Seine-Geo framework has been evaluated using a parallel multi-block oil reservoir simulation. The Seine-Geo geometry-based shared spaces were used to share data on *mortar grid* objects at the interfaces between blocks. Note that the application used Seine-Geo for the couplings between the blocks and MPI for all other communications. The

Table 3.2: Application pseudocode using Seine-Geo API.

```
/* In the pseudo code the Seine-Geo runtime system is initialized up by calling *
 * the system initiate function. A region is then registered with the framework *
 * and an object associated with this region is inserted into the shared space. *
 * The main loop consists of getting the object from the shared space, performing *
 * local computations on the objects, and putting the object back into the space. *
 * When the loop completes, the region is deregistered and the shared space is deleted */
/* Start up the Seine-Geo runtime system by calling system initiate function */
Seine-Geo* Seine-Geo=Seine-Geo_sys→init(processor id, shared space bootstrap server ip);
Create and initialize local object;
/* Register a region with the framework */
Seine-Geo→register(object geometry descriptor);
while(number of iterations is smaller than maximum number of iterations) {
/* If the object is on block face 1 or 3 or 5, first get the object from the shared *
 * space, perform local computations and update the object and then put it back into *
 * the shared space; else first perform local computation and update the object, put *
 * the object into the shared space, and then get the object from the shared space.*/
      if(object.face modular 2){
            Seine-Geo→get(object geometry descriptor, object data descriptor, tag);
            Perform local computation and update object;
            Seine-Geo→put(object geometry descriptor, object data descriptor, tag);
      }else{
            perform local computation and update object;
            Seine-Geo→put(object geometry descriptor, object data descriptor, tag);
            Seine-Geo→get(object geometry descriptor, object data descriptor, tag);
      }
}
/* Deregister the region from the shared space */
Seine-Geo→deregister(object geometry descriptor);
```

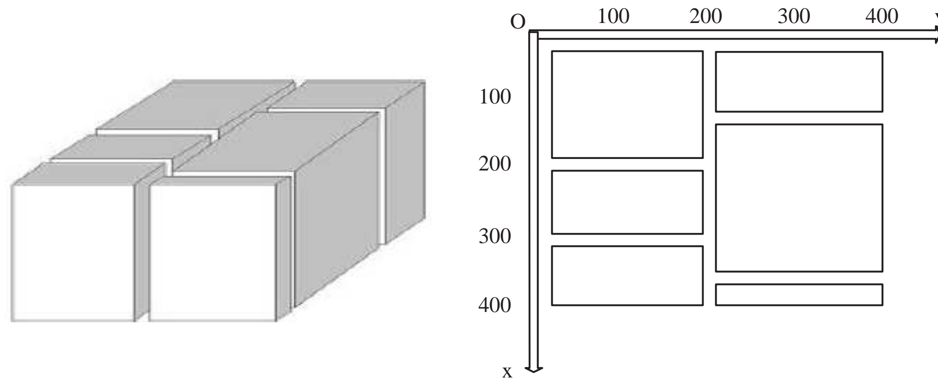


Figure 3.5: 3-D and 2-D views of the multi-block grid structure for the parallel oil reservoir application used in the experimental evaluation.

experiments were conducted on DataStar, the IBM terascale system at the San Diego Supercomputer Center, and on a 64 processor Beowulf cluster. Datastar has 176 (8-way) P655+ and 7 (32-way) P690 compute processors, 16GB memory for 8-way processors and 128GB memory for 32-way processors, and a nominal theoretical peak performance of 10.4 TFlops. The machines uses IBM's AIX 5L 5.2 OS on the processors and has an IBM Federation network interconnect. The Beowulf cluster has 64 Linux-based computers connected by 100 Mbps full-duplex switches. Each processor has an Intel(R) Pentium-4 1.70GHz CPU with 512MB RAM and runs Linux 2.4.20-8 (kernel version).

Experiments on DataStar

The experiments on DataStar used the parallel oil reservoir simulation described above and analyzed the performance of Seine-Geo in detail. The problem domain basically consisted of 6 3-dimensional grid blocks and 5 2-dimensional mortar-grids at the interfaces of the blocks, as illustrated in Figure 3.5. The grid blocks were decomposed and distributed across the processors to balance load. The storage associated with each mortar grid object was a 2-dimensional array of type *double*. In these experiments, however, the interaction patterns between blocks and the size of the shared objects were varied. Different system sizes from 8 to 512 processors were used. The experiments are described below.

Initialization and bootstrapping costs:

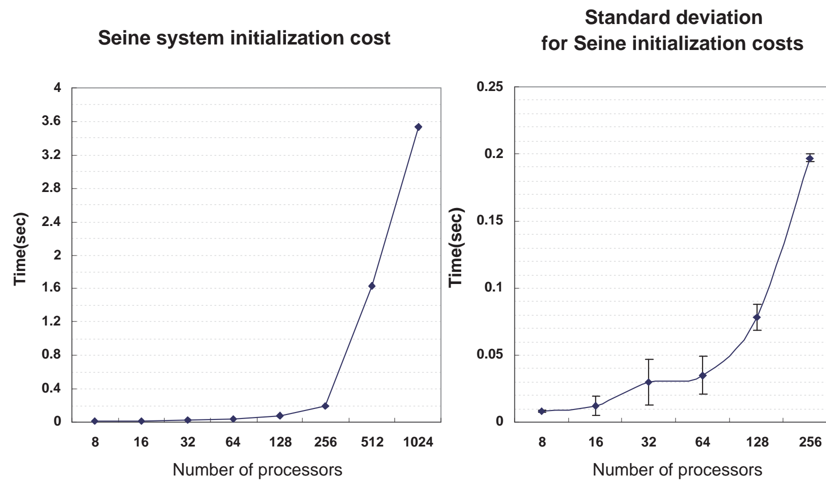


Figure 3.6: Average system initialization and bootstrapping costs and standard deviation on DataStar.

This experiment measured the initialization and bootstrapping costs for different system sizes. Both the average cost and standard deviation for different system sizes are plotted in Figure 3.6. As seen in the figure, the initilization/bootstrapping costs increase as the system size increases. This increase is due to the fact that in the current implementation, all the processors use a single processor to bootstrap causing the bootstrap server to become a bottleneck. We are currently modifying the implementation to distribute this step. Note that this is a one-time cost.

Experiments with different interaction patterns:

This experiment used three different block layouts and correspondingly, three different interaction patterns, as shown in Figure 3.7. Figure 3.8 (a) shows that the average size of shared objects decreases as the system size increases for all the test cases in the experiment. This is because, as the system size increases, each block will be mapped to a larger number of processors and the size of the sub-block (and corresponding shared interface) at each processor will be smaller. Since the average sizes of the shared objects can vary and are different for each test case, we use operation cost per unit region size as the metric. The results are plotted in Figures 3.9, 3.10, and 3.11 for the *register*, *get*, and *put* operations respectively.

An interesting observation from Figure 3.8 (b) is that in test case I & III, the total size of the regions registered increased as the system size increased. This behavior is caused by two factors: (1) the location of the interfaces in the test cases, and (2) the block decomposition

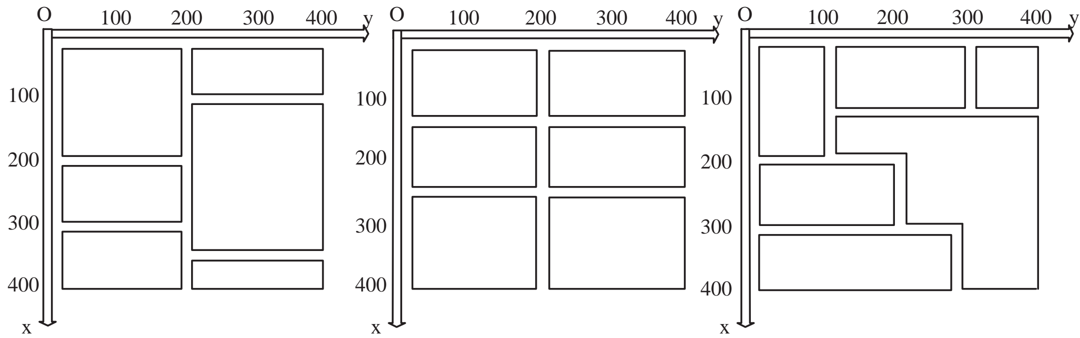
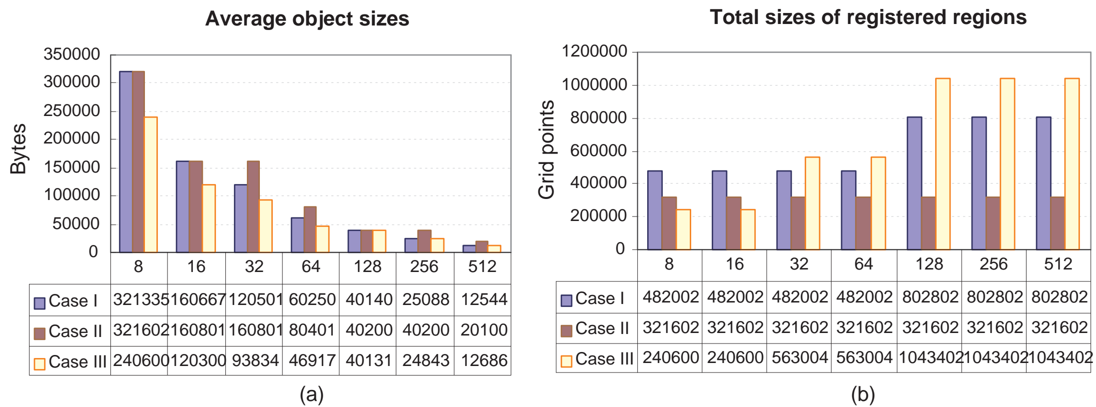


Figure 3.7: 2-D view of the different interaction patterns used in the experiments (Test cases I, II & III).



(a)	8	16	32	64	128	256	512
Case I	321335	160667	120501	60250	40140	25088	12544
Case II	321602	160801	160801	80401	40200	40200	20100
Case III	240600	120300	93834	46917	40131	24843	12686

(b)	8	16	32	64	128	256	512
Case I	482002	482002	482002	482002	802802	802802	802802
Case II	321602	321602	321602	321602	321602	321602	321602
Case III	240600	240600	563004	563004	1043402	1043402	1043402

Figure 3.8: (a) Average shared object sizes for the different interaction patterns; (b) Total sizes of registered regions.

algorithm used. These factors may cause multiple interfaces to reside on the same processors in some cases so that the interfaces can be shared directly rather than through Seine-Geo. On the other hand, if they cause the interfaces to be across processors, the total size of the registered regions and shared objects will increase with system size. These two factors also cause the number of co-existing Seine-Geo shared spaces to increase with the size of the system, as seen in Figure 3.10 (a). Different test cases may also have different numbers of shared spaces. The breakdown of the costs measured in this experiment are presented in Table 3.3. The results are discussed below.

***Register* operation cost:** To analyze the register operation costs plotted in Figure 3.9, consider the breakdown of the cost presented in Table 3.3. The cost of the *register* operation

Table 3.3: Breakdown of Seine-Geo operation costs.

Operation	Cost breakdown	Contributing factors
register	message transfer time	* number of processors associated with Seine-Geo spaces
	blocked waiting time	* size of regions registered with the Seine-Geo space
	local request processing time	* size of regions to be registered
get	local object search time	* number of objects in the shared space at the processor
		* number of shared spaces at the processor
	memory copy time	* size of objects shared
put	local object search time	* number of objects in the shared space at the processor
		* number of shared spaces at the processor
	data transfer time	* number of processors associated with the space
		* size of objects shared

can be decomposed into three main components. The first is the communication cost associated with the *register* request message, the *register* acknowledge message, and possibly a space merge notification message. This cost is essentially a communication cost and is affected by the total number of processors that are involved in the registration. The second is the blocked waiting time between when the request arrives at the directory service daemon and when it is serviced. This cost is affected by the size of regions associated with the Seine-Geo space that the region being registered intersects. The third is the time required to service the request, which is primarily the time required to traverse and update the local interval tree. This cost is affected by the size of region being registered. Among the three factors, the second factor is the most dominant factor because *register* requests are sequentially handled at the daemon.

As seen in Figure 3.9 (b), the average cost per unit region size of the *register* operation increases with system size for all the three cases. This is due to the fact that the number of processors associated with a space increases as the system size increases and this increases the messaging component of the registration cost.

As mentioned, the blocked waiting time is the dominant factor for *register* operation cost and is affected by the size of region associated with the Seine-Geo space that the region being registered intersects with. Since multiple spaces may exist simultaneously in Seine-Geo (see

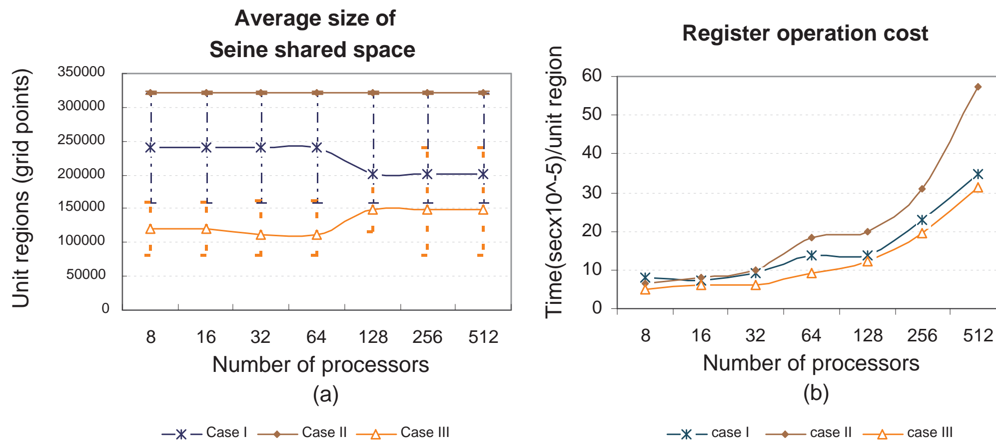


Figure 3.9: (a) Average size of Seine-Geo shared space; the error bars show the sizes of the largest and smallest spaces in each case; (b) Cost per unit region size of the register operation cost for different interaction patterns (Test cases I, II & III).

Figure 3.10 (a)), and the *register* cost for each individual space may differ based on the size of region associated with the space, the *register* cost plotted in Figure 3.9 is actually an average of the *register* cost for all co-existing spaces. The average size of shared spaces for the test cases is shown in Figure 3.9 (a). Figure 3.9 (b) shows that cost of the *register* operation is different for the three test cases and corresponds to the average size of Seine-Geo spaces or the regions being registered for the three cases (Figure 3.9 (a)).

While registration costs are relatively high, these are one-time costs and are only required when a shared space is created. Also note that, in the above discussion, operation time cost per unit region is used as the metric. However, the size of the registered region is the most dominant factor contributing to an operation cost, and as a result, while the cost per unit size of the registered region increases, the overall *register* cost for a processor decreases as the system size increases since each processor needs to register for a smaller region of interest. We will further demonstrate this using the experiments conducted on the Beowulf cluster.

***Get* operation costs:** The operation cost per unit size for the *get* operation is plotted in Figure 3.10 (c). In Seine-Geo, the *get* operation is local to the processor - it searches for the object in the local storage. Further, its operation cost consists of two primary components (see Table 3.3): (1) the time spent for the local search and (2) the time spent to copy the object. The object copy time is determined by the object size and therefore is not a factor in plots as the metric used is cost per unit size. In the current Seine-Geo prototype, shared spaces are locally

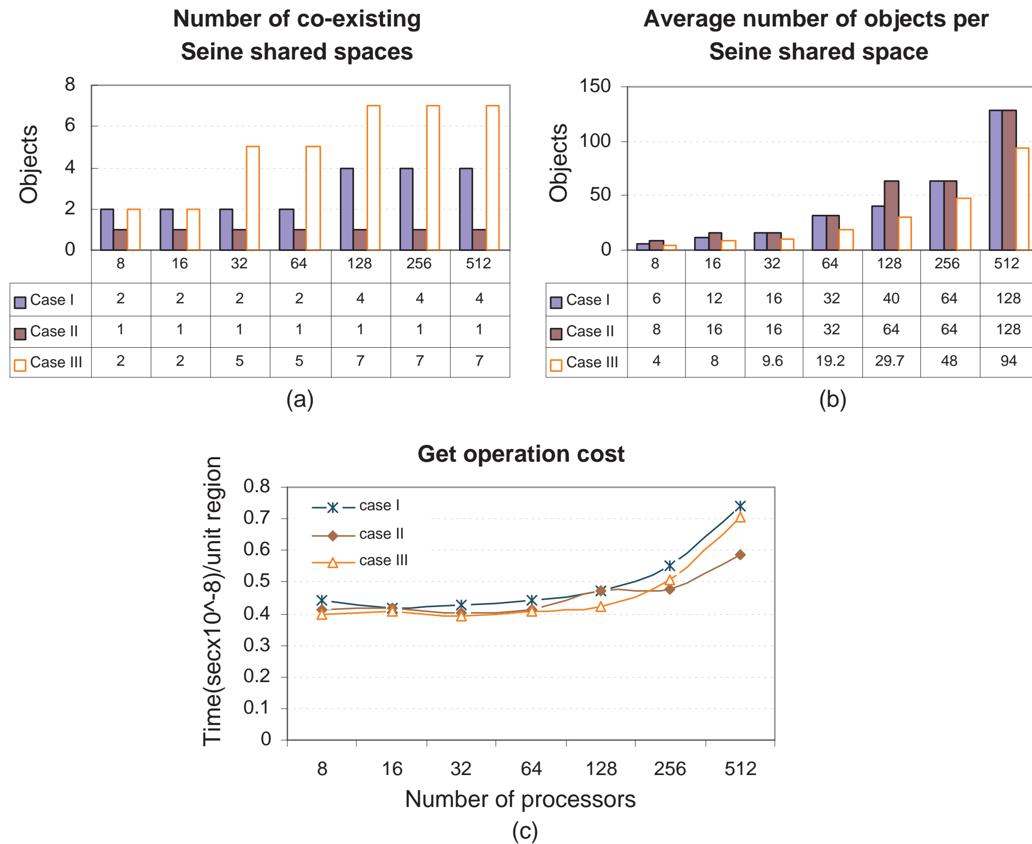


Figure 3.10: (a) Number of co-existing Seine-Geo shared spaces; (b) Average number of objects per Seine-Geo shared space; (c) *Get* operation cost per unit region size for different interaction patterns (Test cases I, II & III).

managed using a list structure and object in a space are also stored using a list. As a result, the local object search operation consists of first locating the appropriate space in the list of spaces, and then searching for the object in the associated object list, i.e., the cost of the *get* operation is $O(S+T)$, where S is the size of the list of spaces and T is size of the object list associated with the space.

In the three test cases, the number of spaces is much smaller than the size of the object lists. As a result, the local object list search dominates the overall *get* operation cost in these test cases. This is reflected in Figure 3.10 where the *get* operation cost per unit object size plotted in Figure 3.10 (c) approximately follows the average number of objects per space plotted in Figure 3.10 (b). However, for system sizes of 256 and 512 processors, test case II has the least cost, which is inconsistent with the average number of objects per space plotted Figure 3.10 (b). A possible reason for this is that test case II has only one shared space, while in test case III the number of spaces increases with system size, causing test case II to have exhibited better performance. Note however that the average cost of a *get* operation per unit object size for all the three test cases is very small.

***Put* operation costs:** The cost of a *put* operation consists of two components: (1) time required to search the local space for objects with regions that overlap with the region of the object being *put* and (2) the time required to propagate the object over the network to remote processors that share the space. As listed in Table 3.3, the *put* cost depends on the number of processors associated with the space, the total size of the objects that are *put*, the number of objects in the shared space at the processor, and the number of shared spaces at the processor. As in the case of the *get* operation, since the metric used in the plots is the cost per unit region, the size of object is not a factor here. Among the other factors, communications cost dominates and largely depends on the number of processors associated with the spaces. The other factor is the number of processors per space, which defines the size of communicating processor groups and indicates how balanced the communications are. When the total number of processors associated with shared spaces are the same, a smaller average processor group size indicates more balanced communication distributions. Note that communication are limited to these groups and there is no communication between the groups. This factor becomes particularly significant for larger system sizes.

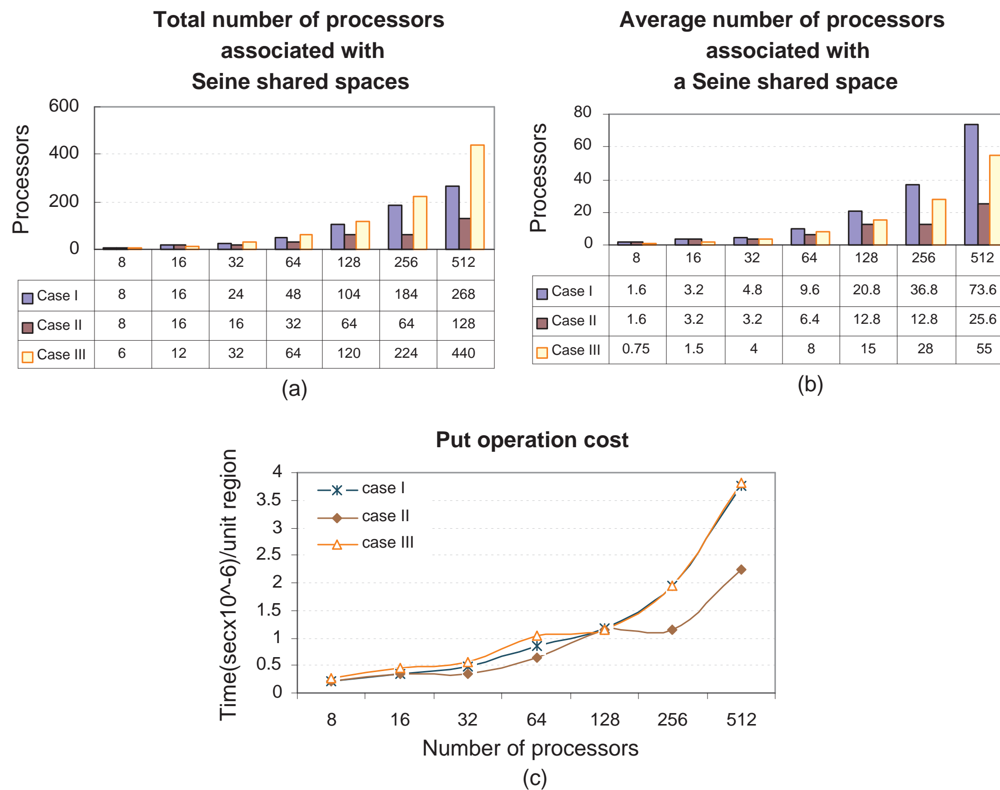


Figure 3.11: (a) Total number of processors associated with Seine-Geo shared spaces; (b) Average number of processors associated with a Seine-Geo shared space; (c) *Put* operation cost for different interaction patterns (Test cases I, II & III).

These effects can be seen in Figure 3.11. For system sizes smaller than 128 processors, the average *put* operation cost per unit object size (plotted in Figure 3.11 (c)) approximately follows the total number of processors associated with Seine-Geo spaces (plotted in Figure 3.11 (a)). For system sizes of 128 processors or larger, the average number of processors associated with a space is more significant. For system sizes larger than 64 processors, test case I has a smaller total number of processors associated with shared spaces as compared to test case III. However, it has a larger average number of processors per space. This results in a less balanced communications, and as a result, for these larger system sizes, even though test case I has a smaller total number of processors, the cost of the *put* operation is close to that of test case III.

Experiments with different object sizes:

In this experiment, the size of problem domain in Figure 3.5 was varied, which resulted in different sizes of the shared interfaces and corresponding shared objects. The costs of *register*, *get*

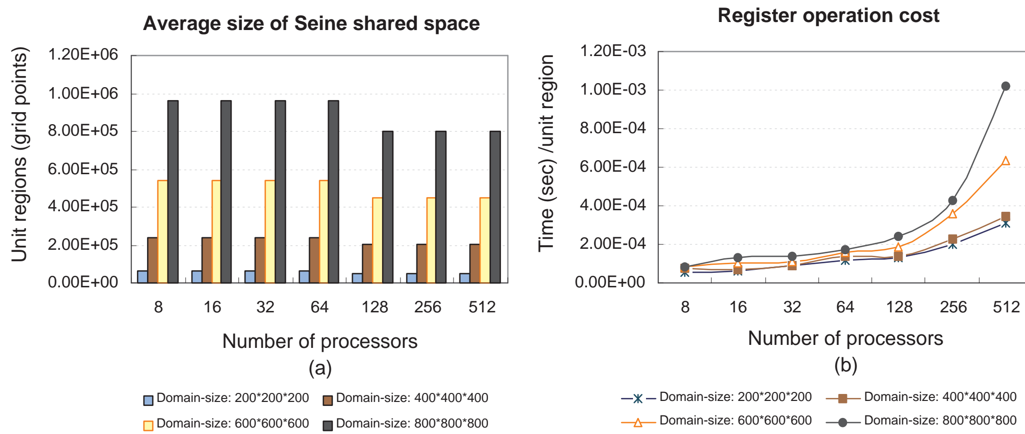


Figure 3.12: (a) Average size of Seine-Geo shared space; (b) Average execution time for *register* operations for different domain and object sizes on DataStar.

and *put* operations were measured and average values per unit region are plotted in Figure 3.12 and Figure 3.13.

Figure 3.12 (a) plots the average size of Seine-Geo shared space for each domain size. This plot is consistent with the previous results, and shows that the cost of the *register* operation increases as the average size of the shared space increases. Note that in this experiment only the domain/object sizes are varied and the interaction patterns remain the same. As a result, the characteristics and distributions of the shared spaces, including the number of co-existing spaces, the average number of objects per space, the total number of processors associated with the spaces, and the average number of processors associated with a space remain the same for the four cases for a given system size (these values are the ones for test case I plotted in Figure 3.10 (a) and (b) and Figure 3.11 (a) and (b) respectively). As a result, in the case of the *get* operation, the four cases have essentially the same object search time cost. As before, the memory copy cost is not a factor here because the metric used is the time per unit size and the total measured cost is divided by the region/object size. However, the search time is also divided by the region/object size and for large object sizes, this results in a lower cost per unit size. This is reflected in the plot of the average *get* operation cost per unit size shown in Figure 3.13 (a). The four cases also have the same communication overheads for the four cases (as they have the same total number of processors associated with the spaces and average number of processors associated with a space). However, once again, in computing the metric,

the communication time is divided by the region/object size and results in a smaller value for large region/object sizes. As a result, the *put* operation cost per unit size decreases for large domain sizes as seen in Figure 3.13 (b).

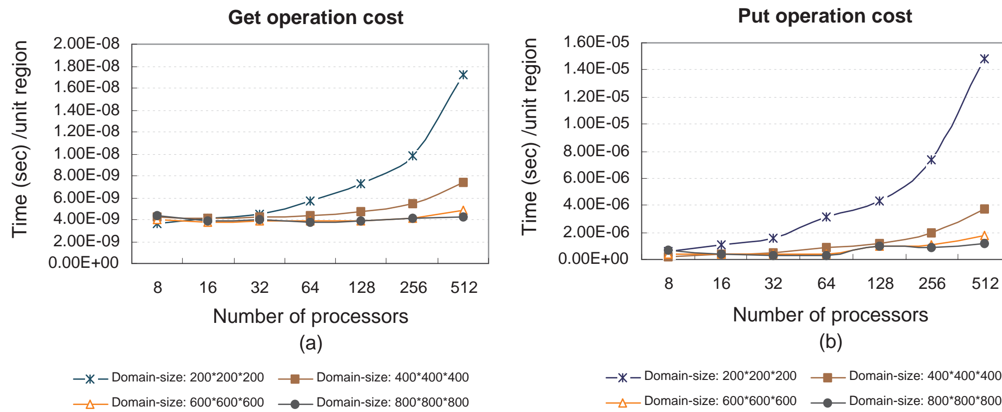


Figure 3.13: Average execution time for *get* (a) and *put* (b) operations for different domain and object sizes on DataStar.

Experiments on a Beowulf Cluster

The experiments on the Beowulf cluster also used the application formulation and problem domain shown in Figure 3.5, consisting of 6 3-dimensional grid blocks and 5 2-dimensional mortar-grids at the interfaces of the blocks. The experiments consisted of measuring the time for *register*, *get* and *put* operations for a range of system sizes, from 8 to 64 processors. In each case, the time for each operation was averaged across the processors. The results are plotted in Figure 3.14. Note that metric used here is the overall average cost rather than the average cost per unit size.

As seen in the figures, the system startup time increases as system size increases, while the times for *register*, *get* and *put* operations decrease. As explained above, the increase in startup time is due to the client-server nature of bootstrapping in the current implementation. However, the decrease in the other three costs is in contrast to the results presented above. The reason for this is the metric used in this experiment. As the system size increases, the average size of objects and corresponding regions decreases as seen in Figure 3.8. The reason is that, as mentioned before, as the system size increases, each block will be mapped to a larger number of

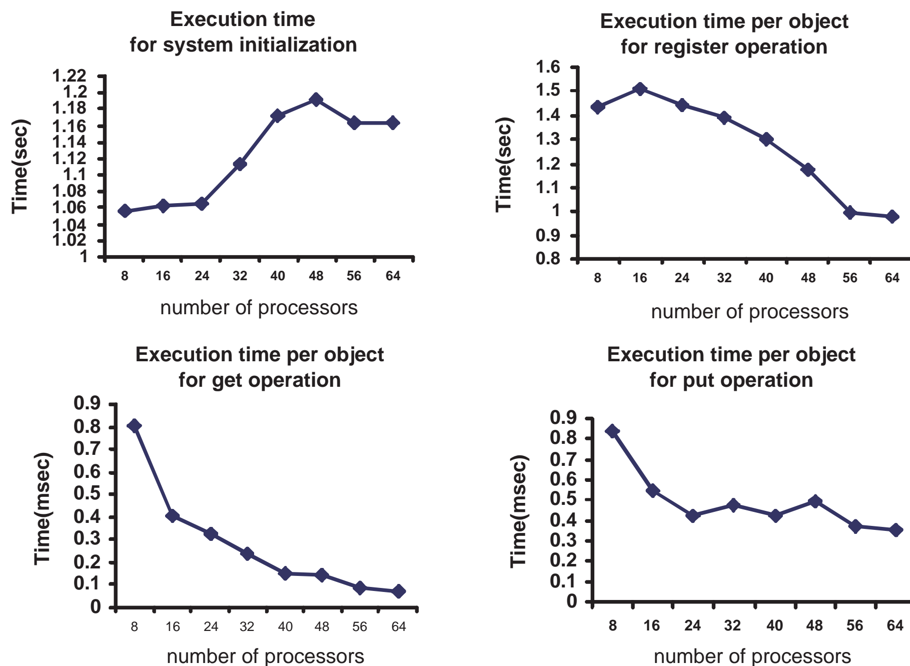


Figure 3.14: Average execution time for *init*,*register*, *get* and *put* operations on the 64-processor Beowulf cluster.

processors and the size of the sub-block (and corresponding shared interface) at each processor will be smaller. Since the size of the registered region is the dominant factor contributing to the cost of an operation, the overall cost decreases as the system size increases but the cost per unit size of the registered region increases.

3.4 Related Work

Several other projects have also based their interaction frameworks on the tuple space model, such as JavaSpaces from Sun [15] and Tspace from IBM [16]. JavaSpaces combines Java with tuple spaces while Tspace emphasizes the integration of tuple space with database systems. These systems are quite complex and tend to be relatively heavy-weight for high performance scientific computing. The lightweight Java taskspaces framework for scientific computing on computational Grids [43] is similar in concept to the Seine-Geo framework presented in this paper. This framework constructs lightweight shared taskspaces for processor pairs that are assigned tasks with neighbor-neighbor inter-task communications. As this system targets very

specific types of applications, the shared space is supported using a direct communication channels between the processor pair. In contrast, the Seine-Geo framework supports more general communication/interaction patters using its shared spaces.

3.5 Conclusion

The Seine-Geo interaction model and framework is a prototype implementation of the Seine approach that supports complex and dynamic interaction patterns required by emerging scientific applications. Specifically, it supports coupling within parallel scientific applications by constructing a geometry-based shared space abstraction. It is based on the observations that most scientific application use a geometric discretization of the problem domain and that communication/interactions in these applications are between entities that are local in this discretized domain. It provides the flexility of the Tuple Space model, extending it to support geometry-based access operators, and enabling scalable implementations. Seine-Geo also supports dynamic creation and deletion of shared spaces. The design and implementation of the Seine-Geo interaction framework was also presented. The framework complements and can be used in conjunction with existing parallel programming systems such as MPI and OpenMP. An experimental evaluation using an adaptive multi-block oil reservoir simulation demonstrated that the performance and scalability of the framework.

Chapter 4

Seine-Coupe: Seine-based Inter-Coupling for Parallel Scientific Applications

4.1 Overview

Coupled simulation system provides the individual models in the system with a more realistic simulation environment, allowing them to be interdependent on and interact with other physics models in the coupled system and to react to dynamically changing boundary conditions. These simulations have the ability to more accurately model the targeted phenomena. For example, in plasma science, an integrated predictive plasma edge simulation couples an edge turbulence code with a core turbulence code through common grids at the spatial interface [40, 41]. These coupled simulation systems are presenting challenging algorithmic, numerical and computational requirements. From the computational point of view, the coupled simulations, each typically running on a distinct parallel system or set of processors with independent (and possibly dynamic) distributions, need to periodically exchange information or data with other parallel systems or set of processors. Specifically, this requires that: (1) interaction/communication schedules between individual processors executing each of the coupled simulations need to be computed efficiently, locally, and on-the-fly, without requiring synchronization or gathering global information, and without incurring significant overheads on the simulations themselves; and (2) data transfers should also be efficient and should happen directly between the individual processors of each simulation. Furthermore, specifying these coupling behaviors between the simulations codes using popular message-passing abstractions can be cumbersome and often inefficient, as these systems require matching sends and receives to be explicitly defined for each interaction. As the individual simulations become larger, more dynamic and heterogeneous and their couplings more complex, implementations using message passing abstractions can quickly become unmanageable. Clearly, realizing coupled simulations requires an efficient, flexible and scalable coupling framework and simple high-level programming abstractions.

4.2 Background and Related Work

4.2.1 Component-based Models and the Common Component Architecture Standard

The component-based methodology has been successfully applied to the business software domain. Examples include CORBA [58], DCOM [59] and Enterprise JavaBeans[60]. Common Component Architecture (CCA)[29] is an effort led by a group of national laboratories and academic institutions in the US that aims at applying this methodology to high performance computing.

The CCA architecture standard defines a set of interfaces that each scientific application components should abide to in order to assemble into a complete system. The focus of the architecture is placed on promoting inter-operability between independently developed components. The basic concepts in this standard include *components*, *ports* and *frameworks*. Components interact through *port*s. There are two types of *port*s: *Provides port*s and *Uses port*s. A *provides port* is a public interface that other components can reference and use. A *uses port* is the connection end point that is connected to *Provides port* of the same type. A *framework* is a runtime system in which components live and interact. CCA uses the Scientific Interface Definition Language SIDL [30] to describe port interface. SIDL is object oriented, language independent, and focuses on scientific computing. An important model in CCA is the Single component multiple data (SCMD) model. It is a component analog of widely used SPMD model. In this model, each process loaded with the same set of components wired the same way. Different components in same process "talk to each other via ports and the framework. Same component in different processes talk to each other through their favorite communications layer (i.e. MPI, PVM, etc.)

4.2.2 Code Coupling and Parallel Data Redistribution

Problem Description and Challenges

CCA terms the parallel data redistribution problem between components as the MxN problem,

which addresses the problem of transferring data from a parallel program running on M processors to another parallel program running on N processors. In CCA, the MxN problem can arise in two contexts: distributed frameworks and direct-connected framework.

In distributed frameworks, since components may run on different number of processors, when parallel data structure is moved from one parallel component to another parallel component, the data must be correctly redistributed according to the data distributions of the components involved. Further, since any communication in CCA is done via port invocation, the RMI needs to be generalized to cope with cases in which there is mismatch of the number of processes at each side.

In direct-connected frameworks, the MxN problem arises when data movement is needed between two separate framework instances. In this context, since port invocation cannot be across framework instances, the “MxN components” will need to be instantiated in both framework instances to mediate the data movement.

Support for Parallel Data Redistribution

Parallel data redistribution (or the MxN problem) is a key aspect of the coupling problem. A number of recent projects have investigated the problem. These include Model Coupling Toolkit [31], InterComm [28], PAWS [27], CUMULVS [26], DCA [33] and SciRun2 [35], each with different foci and approach. In the context of component-based systems such as CCA, parallel data redistribution support is typically encapsulated into a standard component, which can then be composed with other components to realize different coupling scenarios. An alternate approach embeds the parallel data redistribution support into a Parallel Remote Method Invocation (PRMI) [34] mechanism. This PRMI-based approach addresses issues other than just data redistributions, such as remote method invocation semantic for non-uniformly distributed components.

Projects based on these two approaches are summarized as follows. Projects such as Model Coupling Toolkit (MCT), InterComm, PAWS (Parallel Application Workspace), CUMULVS (Collaborative User Migration, User Library for Visualization and Steering), and DDB (Distributed Data Broker) use the component-based approach. As listed in the table, some of these systems have only partial or implicit support for parallel data redistribution, and can support

only a limited set of data redistribution patterns. Projects such as PAWS and InterComm fully address the parallel data redistribution problem. These systems can support random data redistribution patterns. PRMI-based projects include SciRun2, DCA and XCAT.

While all these existing coupling frameworks address the parallel data redistribution problem, they differ in the approaches they use to compute communication schedules, the data redistribution patterns that they support, and the abstractions they provide to the application layer. Some of the existing systems gather distribution information from all the coupled models at each processor and then locally compute data redistribution schedules. This implies a global synchronization across all the coupled systems, which can be expensive and limit scalability. Further, abstractions provided by most existing systems are based on message passing, which require explicitly matching sends and receives and sometimes synchronous data transfers. Moreover, expressing very general redistribution patterns using message passing type abstractions can be quite cumbersome. Existing systems are briefly described below.

InterComm [28]

InterComm is a framework that couples parallel components and enables efficient communication in the presence of complex data distributions for multidimensional array data structures. It provides the support for direct data transfer between different parallel programs. InterComm supports parallel data redistribution in several steps: (1) locating the data to be transferred within the local memories of each program, (2) generating communication schedules (the patterns of inter-processor communication) for all processes, and (3) transferring the data using the schedules. Note that these are three common steps in various data coupling frameworks. But different frameworks may have different ways to compute communication schedules. Furthermore, some have support for only limited data redistribution patterns. InterComm supports data distributions with any complex patterns. It uses a linearization space to find the mapping between two parallel data objects. The Seine-Coupe approach in addressing the parallel data redistribution problem has similarity to InterComm in that Seine-Coupe also uses a linearization space to find the mapping between data objects. The difference between them include: (1) how the linearization is done; (2) how the communication schedule is computed; (3) the abstractions they present to the application.

PAWS [27]

PAWS (Parallel Application WorkSpace) is a framework for coupling parallel applications within a component-like model. It supports redistribution of scalar values and parallel multi-dimensional array data structures. The shape of a PAWS sub-domain can be arbitrarily defined by an application. Multi-dimensional arrays in PAWS can be partitioned and distributed completely generally. It uses a process as a central controller to link applications and parallel data structures in the applications. PAWS controller establishes a connection between those data structures using information in its registry. The controller organizes and registers each of the applications participating in the framework. Through the controller, component applications ("tasks") register the data structures that should be shared with other components. Tasks are created and connections are established between registered data structures via the script interface of the controller. The controller provides for dynamic data connection and disconnection, so that applications can be launched independently of both one another and the controller. PAWS uses point-to-point transfers to move segment of data on one node to remote nodes directly and in parallel. PAWS uses Nexus to permit communication across heterogeneous architectures.

MCT [31]

Model Coupling Toolkit (MCT) is a system developed for the Earth System Modeling Framework (ESMF). The system addresses the general model coupling problem, with the parallel data redistribution tightly integrated into the system. MCT defines a globalSegmentMap to describe each continuous chunk of memory for the data structure in each process. Using globalSegmentMaps, MCT can generate a router or a communication scheduler , which tells processes how to transfer data elements between a simulation component and the flux coupler. Note that this indicates the transfers between two physics components are executed through the flux coupler.

CUMULVS [26]

CUMULVS is a middleware library aimed to provide support for remote visualization and steering of parallel applications and sharing parallel data structures between programs. It supports mult-dimensional arrays like PAWS, however arrays cannot be distributed in a fully general way. A receiver program in CUMULVS is not a parallel program. It specifies the data it requires in a request that is sent to the parallel sender program. After receiving the request, the sender program generates a sequence of connection calls to transfer the data. CUMULVS

essentially provide Mx1 parallel data redistribution support, not full MxN support.

DDB [37]

Distributed Data Broker (DDB) Handles distributed data exchanges between ESM (Earth Science Model) components. DDB is a general purpose tool for coupling multiple, possibly heterogeneous, parallel models. It is implemented as a library used by all participating elements, one of which serves as a distinguished process during a startup phase preceding the main computation. This "registration broker" process correlates offers to produce quantities with requests to consume them, forwards the list of intersections to each of the producers, and informs each consumer of how many pieces to expect. After the initial phase, the registration broker may participate as a regular member of the computation. A library of data translation routines is included in the DDB to support exchanges of data between models using different computational grids. Having each producer send directly to each consumer conserves bandwidth, reduces memory requirements, and minimizes the delay that would otherwise occur if a centralized element were to reassemble each of the fields and retransmit them.

RedGRID [61]

RedGRID is a software environment dedicated to the coupling of parallel codes, and more specifically to the parallel data redistribution of complex parallel data objects. It supports the redistribution of scalar, (dense/sparse) field, particles, points, meshes.

CISM Code Coupling [1]

The CISM code coupling system targets the general code coupling problem, instead of only focusing on parallel data redistribution. In other words, parallel data redistribution is usually one of its functionality imbed in the system. The Center for Integrated Space Weather Modeling (CISM) work on improving a series of comprehensive scientific models describing the Solar Terrestrial environment for the solar surface to the upper atmosphere of earth. The system needs to couple core global codes which address the corona, heliosphere, the earth's magnetosphere, and ionosphere, and codes which model important local processes such as magnetic reconnection. They pointed out that coupling these codes requires four separate functions: (1) efficient transmission of information among codes, (2) interpolation of grid quantities, (3) translation of physical variables between codes with differing physical models, and (4) control mechanisms to synchronize the interaction of the codes. They also pointed out that the characteristics of

these codes dictate an approach involving loosely coupled groups of independently running programs.

SciRun2 [35]

SciRun2 defines PRMI and data redistribution as extensions to the SIDL [30] language. It supports all-to-all or one-to-one process participation in the parallel RMI. It provides limited support for data redistribution patterns.

DCA [33]

DCA is a prototype distributed CCA framework built on top of MPI. It adopts many MPI concept in defining process participation, MxN data redistribution, and argument passing in Parallel RMI. It provides limited support for data redistribution patterns.

XCAT [39]

XCAT is a distributed CCA framework based on Globus that uses RMI over XSOAP. It supports parallel data redistribution for the case of M=N.

4.3 Seine-Coupe: The Seine-based Coupling Framework

To achieve the goal of efficient, flexible and scalable parallel data redistribution, we apply the Seine approach to address the problem. The coupling framework based on Seine is named Seine-Coupe.

Seine-Coupe is a prototype system derived from Seine-Geo that targets the parallel data redistribution problem. It builds on the same concept and model as Seine-Geo, however, provides an execution environment to support the coupling of two or more independent parallel programs. Seine-Coupe coupling framework is based on the observation that the coupling between two independent applications is always on a shared boundary, interface, or common volume in the domains of the applications to be coupled. As a result, the coupling can be realized by creating geometry-based shared space around the coupled entities so that the target applications can communicate with each other by sharing datao/bjects associated with the space. It enables efficient computation of communication schedules, supports low-overheads processor-to-processor data streaming, and provides high-level abstraction to the application layer.

The Seine geometry-based coupling framework differs from existing approaches in several ways. First, it provides a simple but powerful abstraction for coupling in the form of the virtual geometry-based shared space. Processes register geometric regions of interest, and associatively read and write data associated with the registered region from/to the space in a decoupled manner. Second, it supports efficient local computation of communication schedules using lookups into directory implemented as a distributed hash table.

The index space of the hash table is once again, directly constructed from the geometry of the application using Hilbert space filling curves [45]. Processes register their regions of interest with the directory layer, and the directory layer automatically computes communications schedules based on overlaps between the registered geometric regions. Registering processes do not need to know of or explicitly synchronize with other processes during registration and the computation of communication schedules. Finally, it supports efficient and low-overhead processor-to-processor socket-based data streaming and adaptive buffer management. All interactions are completely decoupled and can be synchronous or asynchronous. Moreover, Seine-Coupe, like other Seine-based systems, can work in tandem with systems such as MPI, PVM and OpenMP.

The design, implementation and experimental evaluation of the Seine-Coupe framework are presented next. The implementation is based on the DoE Common Component Architecture (CCA) [29] and enables coupling within and across CCA-based simulations. The experimental evaluation measures the performance of the framework for various data redistribution patterns and different data sizes. The results demonstrate the performance and overheads of the framework.

4.3.1 Design of Seine-Coupe Coupling Framework

The Seine-Coupe coupling framework builds on the geometry-based shared space abstraction and has the same system architecture as the Seine-Geo system presented in the previous chapter. Therefore, it has the same key components:

- A distributed directory layer that enables the registration of spaces and the efficient lookup of objects using their geometry descriptors.

- A storage layer consisting of local storage at each processor associated with a shared space and used to store its shared objects.

- A communication layer that provides efficient data transfer between processors.

4.3.2 Coupling Parallel Scientific Applications using Seine-Coupe

A simple parallel data redistribution scenario shown in Figure 4.1(a) is used to illustrate the operation of the Seine-Coupe coupling framework. In this scenario, data associated with 2-dimensional computational domain of size 120*120 is coupled between two parallel simulations running on 4 and 9 processors respectively. The data decomposition for each simulation and the required parallel data redistribution are shown in the figure. Simulation M is decomposed into blocks M.1 - M.4 and simulation N is decomposed into blocks N.1 - N.9. The Seine-Coupe coupling framework coexists with these simulations and is also distributed across 4 processors in this example. This is shown in the top portion of Figure 4.2. Note that these processors may or may not overlap with the simulation processors. Figure 4.2 also illustrates the steps involved in coupling and parallel data redistribution using Seine-Coupe, which are described below.

The initialization of the Seine-Coupe runtime using the *init* operator is shown in Figure 4.2 (a). During the initialization process, the directory structure is constructed by mapping the 2-dimensional coordinate space to a 1-dimensional index using the Hilbert SFC and distributing index intervals across the processors.

In Figure 4.2 (b), processor 1 registers an interaction region R1 (shown using a lighter shade in the figure) in the center of the domain. Since this region maps to index intervals that spans all four processors, the registration request is sent to the directory service daemon at each of these processors. Each daemon services the request and records the relevant registered interval in its local interval tree. Once the registration is complete, a shared space corresponding to the registered region is created at processor 1 (shown as a cloud on the right in the figure).

In Figure 4.2 (c), another processor, processor 0, registers region R2 (shown using a darker shade in the figure). Once again, the region is translated into index intervals and corresponding registration request is forwarded to appropriate directory service daemons. Using the existing

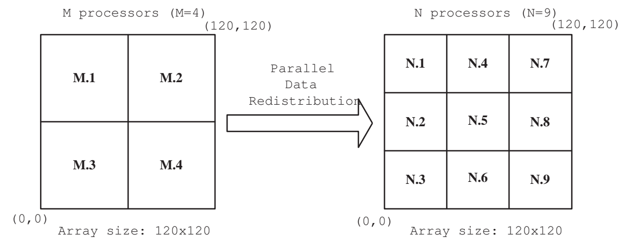
intervals in its local interval tree, the directory service daemons detect that the newly registered region overlaps with an existing space. As a result, processor 0 joins the existing space and the region associated with the space is updated to become the union of the two registered regions. The shared space also grows to span both processors. As more regions are registered, the space is expanded if these regions overlap with the existing region, or new spaces are created if the regions do not overlap.

Once the shared space is created, processors can share geometry-based objects using the space. This is illustrated in Figures 4.2 (d) and (e). In Figure 4.2 (d), processor 0 and processor 1 use the *put* operation to insert object 2 and object 1 respectively into the shared space. As there is an overlap between the regions registered by processors 0 and 1, the update to object 1 is propagated from processor 0 to processor 1. Similarly, the update to object 2 is propagated from processor 1 to processor 0. The propagated update may only consist of the data corresponding to the region of overlap, e.g., a sub-array if the object is an array. In Figure 4.2 (e), processor 1 and processor 0 retrieve object 1 and object 2 respectively using a local *get* operation.

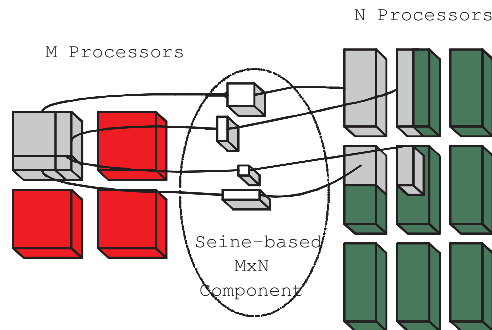
Building coupled simulations using the Seine abstractions:

Seine-Coupe provides a virtual dynamic geometry-based shared space abstraction to the parallel scientific applications. Developing coupled simulations using this abstraction consists of the following steps. First, the coupled simulations register their geometric regions of interests with Seine-Coupe. The registration phase detects geometric relationships between registered regions and results in the creation of a virtual shared space localized to the region and the derivation of associated communication schedules. Coupling data between simulations consists of one simulation writing the data into the space, along with a geometric descriptor describing the region that it belongs to; and the other simulation independently reading data from the space with an appropriate geometric descriptor. The communication schedule associated with the space and the Seine-Coupe communication layer is used to set up parallel point-to-point communication channels for direct data transfer between source and destination processors. The associated parallel data redistribution is conceptually illustrated in Figure 4.1(b).

Computation of communication schedules:



(a) An illustrative example



(b) Abstraction of Seine-Coupe based parallel data redistribution

Figure 4.1:

Communication schedules in the context of coupling and parallel data redistribution refer to the sequence of messages required to correctly move data among coupled processes [34]. As mentioned above, these schedules are computed in Seine-Coupe during registration using the Hilbert SFC-based linearization of the multidimensional application domain coordinate space. When a region is registered, the Seine directory layer uses the distributed hash table to route the registration request to corresponding directory service node(s). The directory service node is responsible for detecting overlaps or geometric relationships between registered regions efficiently. This is done by detecting overlaps in corresponding 1-dimensional index intervals using the local interval tree. Note that all registration requests that are within a particular region of the application domain are directed to the same Seine directory service node(s), and as a result, the node(s) can correctly compute the required schedules. This is in contrast to most existing systems which require information about the distributions of all the coupled processes to be gathered.

Data transfer:

When an object is written into a space, Seine propagates the object (or possibly the appropriate part of the object, e.g., if the object is an array) to update remote objects based on the relationships between registered geometric regions, using the communication layer.

4.3.3 Prototype Implementation and Performance Evaluation

A CCA-based Prototype Implementation

The current prototype implementation of the Seine-Coupe coupling framework is based on the DoE Common Component Architecture (CCA) [29] and enables coupling within and across CCA-based simulations. In CCA, components communicate with each other through *ports*. There are two basic types of *ports*, the *provides* port and the *uses* port. Connections between components are achieved by wiring between a *provides* port on one component and a *uses* port on the other component. The component can invoke methods on the *uses* port once it is connected to a *provides* port. CCA ports are specified using the Scientific Interface Definition Language (SIDL) [30]. CCA frameworks can be distributed or direct-connected. In a direct-connected framework all components in one process live in the same address space. Communication between components, or the port invocation, is local to the process.

The Seine-Coupe coupling framework was encapsulated as a CCA compliant component within the CCAFFEINE direct-connected CCA framework, and is used to support coupling and parallel data redistribution between multiple instances of the framework, each executing as an independent simulation, as well as within a single framework executing in the multiple component-multiple data (MCMD) mode. The former setup is illustrated in Figure 4.3. In the figure, cohorts of component A1 need to redistribute data to cohorts of component B1; similarly, cohorts of component B2 need to redistribute data to cohorts of component A2. To enable data redistribution between framework instances A and B (executing on M and N processors respectively) a third framework instance, C, containing the Seine-Coupe coupling component is first instantiated. This framework executes on X processors, which may or may not overlap with the M and N processors of frameworks A and B. The Seine-Coupe coupling component handles registrations and maintains the Seine directory. Frameworks A and B initiate Seine stubs, which register with the Seine-Coupe coupling component. The ports defined by the

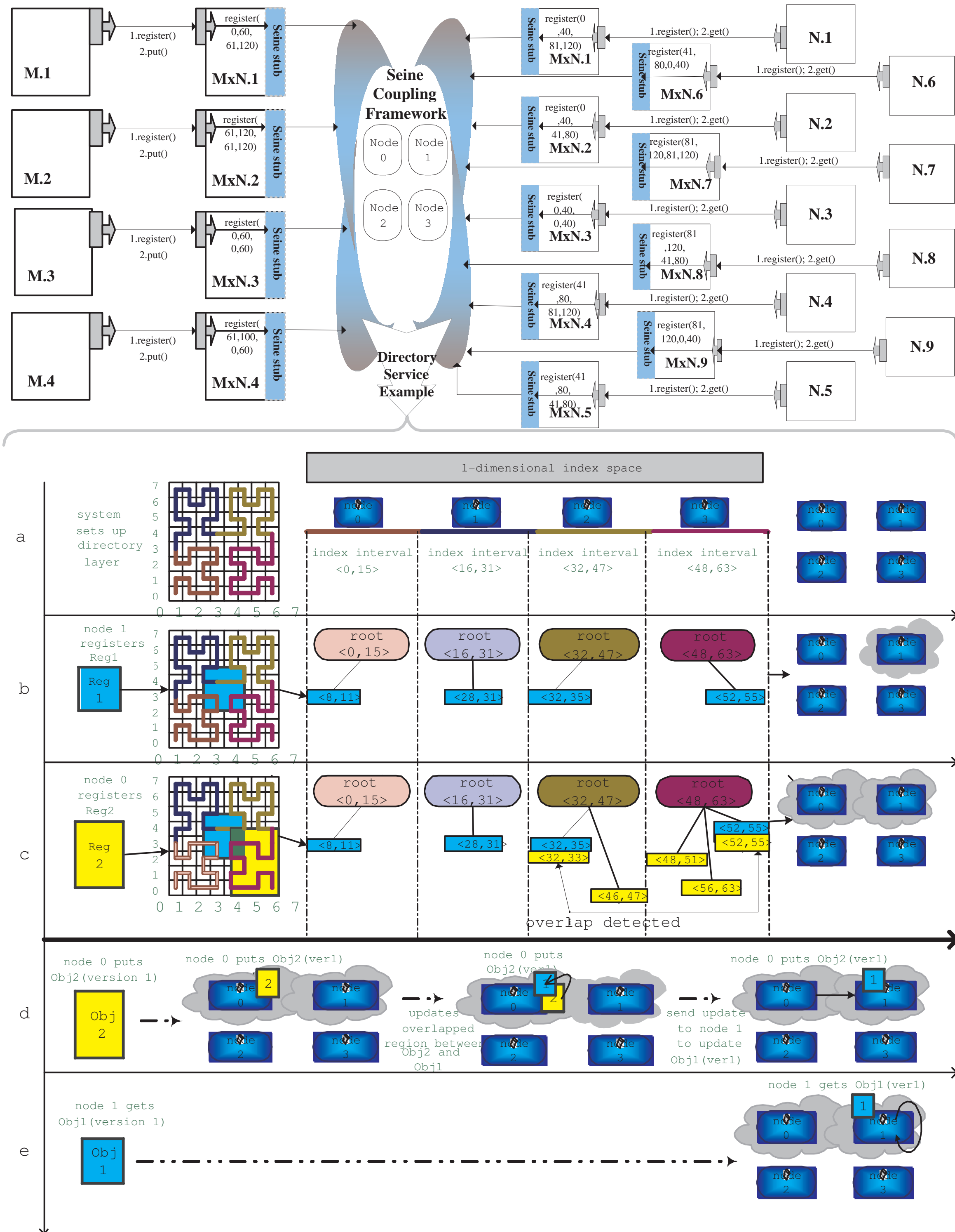


Figure 4.2: Operations of coupling and parallel data redistribution using Seine-Coupe.

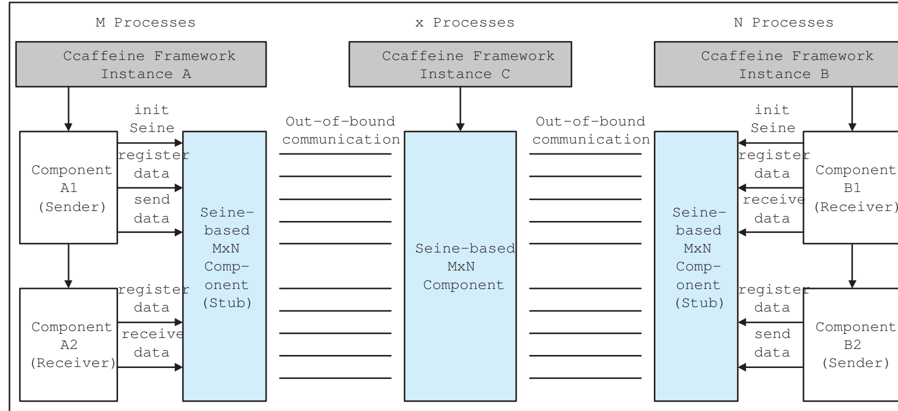


Figure 4.3: MxN parallel data redistribution between CCA framework instances using the Seine-Coupe coupling component.

Seine-Coupe coupling and stub components are *register*, *put* and *get*. The operation is as follows. To achieve A1 - B1 coupling, component A1 registers its region of interest using the *register* port of its stub and invokes the *put* port to write data. Similarly component B1 independently registers its region of interest using the *register* port of its stub and invokes the *get* port. The *register* request is forwarded to the Seine-Coupe coupling component, and if there is an overlap between their registered regions, at the *put* request from A1, the data written by A1 are directly and appropriately forwarded to Seine-Coupe stubs at B1 by Seine-Coupe stubs at A1. At the *get* request from B1, the data received by Seine-Coupe stubs at B1 are copied from Seine-Coupe's buffer. Note that A1 does not have to be aware of B1 or its data distribution.

Experiment with different data redistribution scenarios:

In this experiment, a 3-dimensional array of size 120*120*120 is redistributed to 27 processors from 2, 4, 8, and 16 processors respectively, i.e., M = 2, 4, 8 and 16, and N = 27. Data in the array are of type double. The distribution of the array is (Block, Block, Block) [1] on the x-, y-, z-axis respectively on both the sender and receiver ends, except that for case I and II, (Block, Block, Collapsed) [2] is used at the sender end. The registration, data transfer, and send (put) and receive (get) costs are listed in Table 4.1. The application components do not

[1] Block distribution along each dimension. This notation for distribution patterns is borrowed from High Performance Fortran [38] and is described in [24].

[2] The first two dimension are distributed using a Block distribution and the third dimension is not distributed.

Table 4.1: Cost in seconds of computing registration and data transfer for different data redistribution scenarios.

Array size: 120x120x120
Data type: double
Distribution type (x-y-z):
M side: block-block-collapsed (Cases I & II)
or block-block-block (Cases III & IV)
N side: block-block-block

MxN	Case I 2x27	Case II 4x27	Case III 8x27	Case IV 16x27
Registration	5.6725	3.6197	2.7962	2.273
Data transfer	0.6971	0.3381	0.1636	0.1045
M side (put)	0.6971	0.3381	0.1636	0.1045
N side (get)	0.0012	0.0012	0.0012	0.0012

explicitly compute communication schedules when using Seine-Coupe. However, from their perspective, the time spent on computing communication schedule is equivalent to the time it takes to register their region of interest, i.e., the cost of the *register* operation, which is a one time cost. Since this cost depends on the region and its overlap, it will be different for different components, and the cost listed in Table 4.1 is the average cost. As the table shows, for a given total data size to be redistributed, the average registration cost decreases as the number of processors involved increases. The reason for this decrease is that as the number of processors involved in registration increases, each processor is assigned a correspondingly smaller portion of the array. As a result, each processor registers a smaller region. Since processing a *register* request involves computing intersections with registered regions, a smaller region will result in lower registration cost. In this experiment, as the number of processors increases, i.e., 2+27, 4+27, 8+27, and 16+27, and the size of the overall array remains constant, the sizes of regions registered by each processor decrease and the average registration cost decreases correspondingly.

This experiment also measured the cost in data send (put) and receive (get) operations respectively. The Seine model decouples sends and receives. In Seine, a push model is used to asynchronously propagate data. As a result, the cost of a data send consists of data marshalling, establishing a remote connection, and sending the data, while the cost of data receive consists of only a local memory copy from the Seine buffer. Consequently, the total data transfer cost is

Table 4.2: Cost in seconds of registration and data transfer for different array sizes.

MxN: 16x27 *Data type: double* *Distribution(x-y-z): block-block-cyclic* *M side: number of cycles at z direction=3* *N side: number of cycles at z direction=4*				
Array size	60^3	120^3	180^3	240^3
Registration	0.0920	0.9989	6.31	9.987
Data transfer	0.0423	0.1117	0.271	0.8388
M side (put)	0.0423	0.1117	0.271	0.8388
N side (get)	0.0001	0.0008	0.004	0.0136

essentially the data send cost, and is relatively higher than the data receive cost. We explicitly list the data transfer cost in the table since this metric has been used to measure and report the performance of other coupling frameworks.

Experiment with different array sizes:

In this experiment, the size of the array and consequently, the size of the data redistribution is varied. The distribution of the array is (Block, Block, Cyclic) on the x-, y-, z-axis respectively on both the sender and receiver ends. The registration, data transfer, and send (put) and receive (get) costs are listed in Table 4.2. As these measurements show, the registration cost increases with array size. As explained for the experiment above, this is because the registered regions of interest are correspondingly smaller and the computation of intersections is quicker for smaller array sizes. As expected, the costs for send and receive operations also increase with array size.

Scalability of the Seine-Coupe directory layer:

In this experiment, the number of processors over which the Seine-Coupe coupling and parallel data redistribution component is distributed is varied. Distributing this component also distributes the registration process, and registrations for different regions can proceed in parallel. This leads to a reduction in registration times as seen in Table 4.3. The improvement, however, seems to saturate around 4 processors for this experiment, and the improvement from 4 to 8 processors is not significant. Note that the actual data transfer is directly between the processors and is not effected by the distribution of the Seine-Coupe component, and remains

Table 4.3: Scalability of the directory layer of the Seine-Coupe coupling component.

MxN: 16x27 *Array size: 120x120x120* *Data type: double* *Distribution(x-y-z): block-block-block*			
Number of processors running Seine-Coupe coupling component	1	4	8
Registration	4.2	2.273	2.089
Data transfer	0.112	0.1045	0.1172
M side (put)	0.112	0.1045	0.1172
N side (get)	0.0012	0.0012	0.0012

almost constant.

From the evaluation presented above, we can see that the registration cost ranges from less than a second to a few seconds, and is the most expensive aspect of Seine's performance. However, registration is a one-time cost for each region, which can be used for multiple *get* and *put* operations. Note that registration cost also includes the cost of computing communication schedules, which do not have to be computed repeatedly. Data transfers take place directly between the source and destination processors using socket-based streaming, which does not incur significant overheads as demonstrated by the evaluation. Overall, we believe that the costs of Seine operations are reasonable and not significant when compared to per iteration computational time of the targeted scientific simulations.

4.3.4 Conclusion

The Seine-Coupe framework is a prototype system based on Seine. It is intended to provide support for data coupling and parallel data redistribution. Seine-Coupe addresses the model/code coupling requirements of emerging scientific and engineering simulations, enables efficient computation of communication schedules, supports low-overheads processor-to-processor data streaming, and provides high-level abstraction for application developers. Communications using Seine-Coupe are decoupled and the communicating entities do not need to know about each other or about each other's distributions. A key component of Seine-Coupe is a distributed directory layer that is constructed as a distributed hash table from the application domain and

enables decentralized computation of communication schedules. A component-based prototype implementation of Seine-Coupe using the CAFFEINE CCA framework, and its experimental evaluation are also presented. The Seine-Coupe CCA coupling component enables coupling and parallel data redistribution between multiple CCA-based applications. The evaluation demonstrates the performance and low overheads of Seine-Coupe. However, there remain several issues that still need investigation. For example, optimization of memory usage during redistribution, especially when data sizes are large.

4.4 Data Coupling in the SciDAC Fusion Project

4.4.1 An Overview of the Fusion Simulation Project

The DoE SciDAC Fusion project is developing a new integrated predictive plasma edge simulation code package that is applicable to the plasma edge region relevant to both existing magnetic fusion facilities and next-generation burning plasma experiments, such as the International Thermonuclear Experimental Reactor (ITER). The plasma edge includes the region from the top of the pedestal to the scape-off layer and divertor region bounded by a material wall. A multitude of non-equilibrium physical processes on different spatio-temporal scales present in the edge region demand a large scale integrated simulation. The low collisionality of the pedestal plasma, magnetic X-point geometry, spatially sensitive velocity-hole boundary, non-Maxwellian nature of the particle distribution function, and particle source from neutrals, combine to require the development of a special massively parallel kinetic transport code for kinetic transport code for kinetic transport physics, using a particle-in-cell (PIC) approach. To study the large scale MHD phenomena, such as Edge Localized Modes (ELMs), a fluid code is more efficient in terms of computing time, and such an event is separable since its time scale is much shorter than the transport time. However, the kinetic and MHD codes must be integrated together for a self-consistent simulation as a whole. Consequently, the edge turbulence PIC code (XGC) will be connected with the microscopic MHD code, based on common grids at the spatial interface, to study the dynamical pedestal-ELM cycle.

4.4.2 Data Coupling Requirements in the Fusion Simulation Project

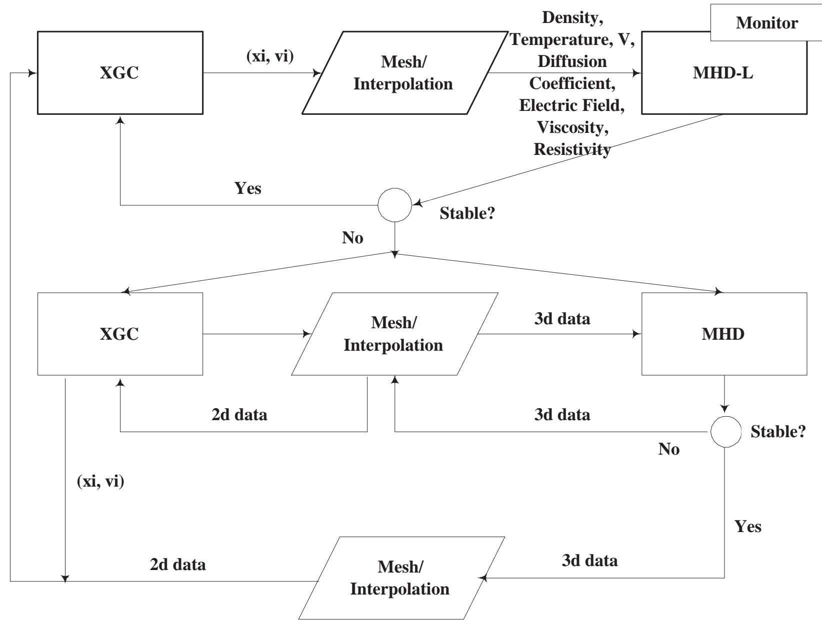


Figure 4.4: Workflow between XGC and MHD.

In this project, two distinct parallel simulation codes, XGC and MHD, will be run on different numbers of processors on different platforms. The overall workflow illustrating the coupling between XGC and MHD code is shown in Figure 4.4. The coupling begins with the generation of a common spatial grid generation. XGC then calculates two dimensional density, temperature, bootstrap current, and viscosity profiles in accordance with neoclassical and turbulence transport, and send these to MHD. The input pressure tensor and current information are used by MHD to evolve the equilibrium magnetic field configuration, which it then sends back to XGC to enable it to update its magnetic equilibrium, and check for stability. During and after the ELM crash, the pressure, density, magnetic field, and current will be toroidally averaged and sent to XGC. During the ELM calculation, XGC will evaluate the kinetic closure information and kinetic E_r evolution and pass them to MHD for a more consistent simulation of ELM dynamics. Note that most probably XGC and MHD will use a different formulation and domain configuration and decomposition. As a result, a mesh interpolation module (referred to as MI afterwards) is needed to translate between the mesh/data used in the two codes, as shown in Figure 4.4.

Challenges and Requirements

XGC is a scalable code and will run on a large number of processors. MHD, on the other hand, is not as scalable due to high inter-processor communications, and runs on relatively smaller numbers of processors. As a result, coupling these codes will require MxN parallel data redistribution. In this project, the transfer is MxPxN, where the XGC code runs on M processors, the interpolation module runs on P processors, and the MHD code runs on N processors.

The fusion simulation application imposes strict constraints on the performance and overheads of data redistribution and transfer between the codes. Since the integrated system is constructed in a way that overlaps the execution of XGC with stability check by MHD, it is essential that the result of the stability check is available by the time it is needed by XGC, otherwise the large number (1000s) of processors running XGC will remain idle offsetting any benefit of a coupled simulation. Another constraints is the overhead of the coupling framework imposed on the simulations. This requires effective utilization of memory and communication buffers, specially when the volume of the coupled data is large. Finally, the data transfer has to be robust and ensure that no data are lost.

4.4.3 A Prototype Coupled Fusion Simulation using Seine-Coupe

Since the Fusion project is at a very early stage, the scientists involved in the project are still investigating the underlying physics and numerics, and the XGC and MHD codes are still under development. However, the overall coupling behaviors of the codes is reasonably understood. As a result, we use synthetic codes, which emulate the coupling behaviors of the actual codes but perform dummy computation, to develop and evaluate the coupling framework. The goal is to have the coupling framework ready when the project moves to production runs. The configuration of the mock simulation using the synthetic codes is shown in Figure 4.5. In the figure, the coupling consists of two parts, the coupling between XGC and MI and the coupling between MI and MHD.

Domain decompositions:

The entire problem domain in the coupled fusion simulation is a 3D toroidal ring. The 3D toroidal ring is then sliced to get a number of 2D poloidal planes as the computation domains. Each plane contains a large number of particles, each of which is described by its physical

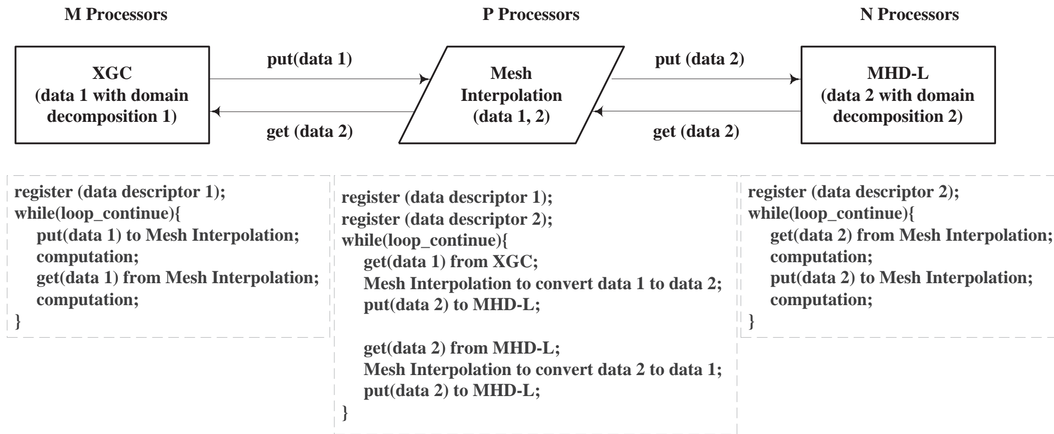


Figure 4.5: Mock simulation configuration for data coupling in Fusion.

location information via coordinates and a set of physics variables. Each 2D poloidal plane is assigned to and replicated on a group of processors. Since XGC and MHD use different domain decompositions, the numbers of planes in the two codes are different, and MI is used to map the XGC domain decomposition to the MHD domain decomposition.

Coupled fusion simulations using Seine-Coupe Shared Spaces

Recall that coupling in Seine-Coupe is based on a spatial domain that is shared between the entities that are coupled. This may be the geometric discretization of the application domain or may be an abstract multi-dimensional domain defined exclusively for coupling purposes. In the prototype described here, we use the latter.

Given that the first phase of coupling between XGC and MHD is essentially based on the 2D poloidal plane, a 3D abstract domain can be constructed as follows.

- The X-axis represents particles on a plane and is the dimension that is distributed across processors.
- The Y-axis represents the plane id. Each processor has exactly one plane id and may have some or all the particles in the plane.
- The Z-axis represents application variables associated with each particle. Each processor has all the variables associated with each particle that is mapped to it.

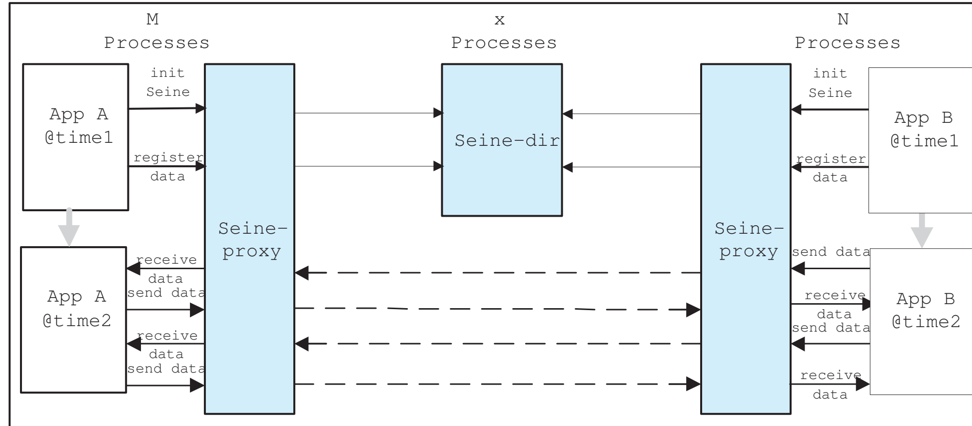


Figure 4.6: MxN data redistribution between parallel applications using the Seine-Coupe coupling framework.

Using this abstract domain, coupling using Seine-Coupe is achieved as follows. Each XGC processor registers its region in the 3D abstract domain based on the 2D poloidal plane and the particles assigned to it, and the variable associated with each particle. The registered region is specified as a 6-field tuple and represents a 2D plane in the 3D abstract domain since each processor is assigned with particles on only one poloidal plane. Each processor running MI similarly registers its region in the 3D abstract domain. Note that since MI acts as the "coupler" between XGC and MHD, these processors register regions twice - once corresponding to the XGC domain decomposition and the second time corresponding MHD domain decomposition. Once the registration is complete, the simulations can use the operators provided by Seine-Coupe, i.e., *put* and *get*, to achieve coupling.

4.4.4 Prototype Implementation and Performance Evaluation

The Seine-Coupe coupling framework is used to support coupling and parallel data redistribution between multiple parallel applications, each executing as an independent simulation. A schematic of the prototype implementation of the Seine-Coupe based coupled fusion simulation is shown in Figure 4.6. The Seine-Coupe framework has two key components. The first is Seine-dir, the Seine-Coupe distributed directory layer that deterministically maps the shared abstract domain onto the Seine-dir infrastructure processors. The second is the Seine-proxy, which is a local daemon that resides on each processor that uses Seine-Coupe. Note that the Seine-dir infrastructure runs on X processors, which may or may not overlap with the M, P and

N processors running XGC, MI and MHD respectively. The Seine-proxy at each processor is initialized by calling *i*nit within the application code. Once the Seine-proxy is initialized, it handles all the processor interaction with Seine-Coupe including registration and *put* and *get* operations. Once Seine-proxy is initialized, each processor registers its region(s) of interest. Registration request are used to construct the Seine-Coupe distributed directory and to detect overlaps between regions of interest of different processors. Using this overlap, data *put* by one processor is automatically forwarded to all processors that have overlapping regions of interest, and can be read locally using the *get* operation. All interaction are completely decoupled and asynchronous. Details about the implementation and operation of Seine have been described previously.

Preliminary Test and Result Analysis

We have conducted two series of experiments. One runs the mock simulations on two local clusters connected by 10/100M interconnection while the other runs the mock simulations across wide area interconnection. In both experiments, the XGC domain was decomposed into 8 2D poloidal planes, while the MHD problem domain was decomposed into 6 2D poloidal planes. The number of particles in each plane was varied in the different experiments. Each particle is associated with 9 variables.

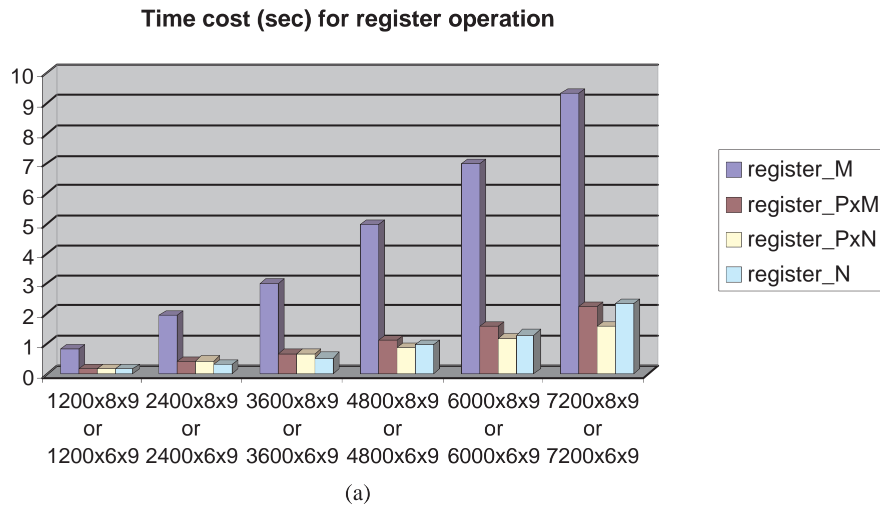
Experiments with Local-Area Coupling using the Prototype Seine-Coupe based Fusion Simulation

In this experiment, the mock simulation is run on two local clusters connected by a 10/100M interconnection. Since XGC will be running on massively parallel computing platforms with I/O node separate from compute nodes, to redistribute data from XGC to the MI, the data will need first to be ported to I/O nodes. The Seine-Coupe framework will run on those I/O nodes and performs data redistribution as required. Due to the small number of I/O nodes compared to the number of nodes running the MI module and the MHD code, we construct the mock simulation in the following way: the mock XGC code runs on 8 processors, the MI code runs on 24 processors, and MHD runs on 18 processors. The Seine-Coupe coupling framework is run on one of the clusters to couple between the mocked XGC and MI and between MI and

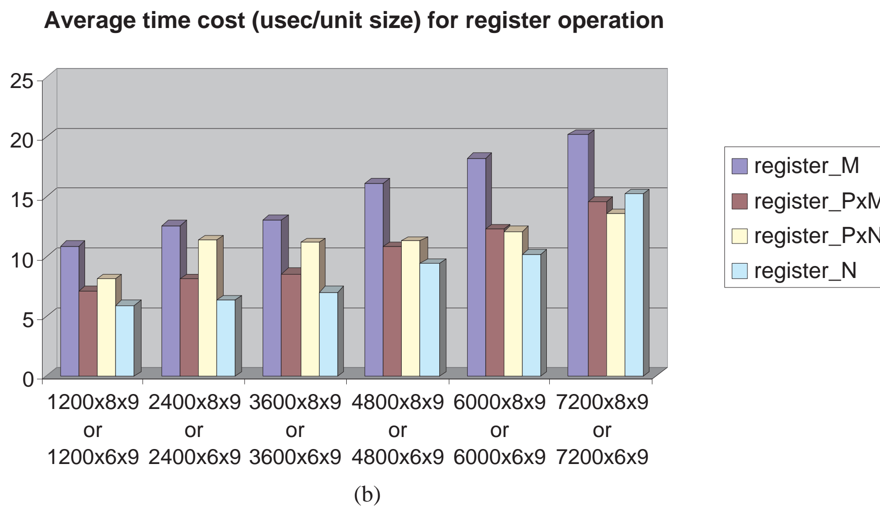
MHD. The cluster running XGC has lower performance than the cluster running MI, MHD, and the coupling frameworks. The communications between XGC and coupling frameworks as well as MI are inter-cluster communication. All the other communications are intra-cluster communication.

The experimental results are illustrated in Figure 4.7, 4.8, and 4.9. In these figures, the data redistribution sites are labeled as M, P, and N, with M corresponding to XGC, P corresponding to MI and N corresponding to MHD. The notation PxM represents the operations on site P (i.e., MI), which is related to data redistribution between site M (i.e., XGC) and site P. Similarly, the notation PxN represents the operations on site P, which is related to data redistribution between site N (i.e., MHD) and site Ps.

***register* time cost:** The *register* cost is measured for the three application components. As stated before, since the site P, or the MI module, needs to be coupled with both site M(XGC) and site N (MHD), it needs to register twice, onec based on the domain definition of site M and the other time based on the domain definition of site N. As a result, the *register* cost in the whole simulation system has four components: the cost for processors running XGC, the cost for processors running MHD, and the costs on processors running MI. The results are plotted in Figure 4.7. Figure 4.7 (a) shows the *register* operation cost on one processor. We can observe that the operation cost increases as the size of the region to be registered increases. The reason to this has been discussed in detail in 3.3.2. Another observation from the figure is that processors on site M has much higher *register* cost compared to the other sites. This is caused by several factors. First, since site M is run on only 8 processors, each processor is assigned a larger region in the abstract array space. As a result, it needs to register for larger regions than the processors at other two sites. Since the cost of the *register* operation increases as the size of region to be registered increases, this leads to a higher cost. Second, the communication between site M and the coupling framework is inter-cluster while the communication between other sites and coupling frameworks are intra-cluster. We believe this also contributes to a higher registration cost. Third, the cluster running M has lower performance than the cluster running P and N. To better understand the result, we compute the *register* cost per unit size so that the size of the registered region is no longer a factor. Figure 4.7(b) plots the averaged cost. This plot shows that after eliminating the size factor, the difference between the *register*



(a)



(b)

Figure 4.7: *register* operation time cost.

costs is less dramatic than the one in Figure 4.7 (a). The average *register* costs on P site and N site are similar but there are still some differences and variations. For example, for array size 2400x8x8 (or 2400x6x8), PxN has the highest average cost while for array size 7200x8x8 (or 7200x6x8), PxN has the lowest average cost. Such variations can result from the order in which registration is invoked. Usually the site that invokes the registration request first will have lower average cost because the service daemons does not have to retrieve its local interval trees to detect overlaps, since the tree is still empty. The service daemon only needs to record registered intervals by inserting them into the tree. In contrast, when a subsequent site registers, the service daemons will need to traverse its local interval tree, detect overlaps, manipulate

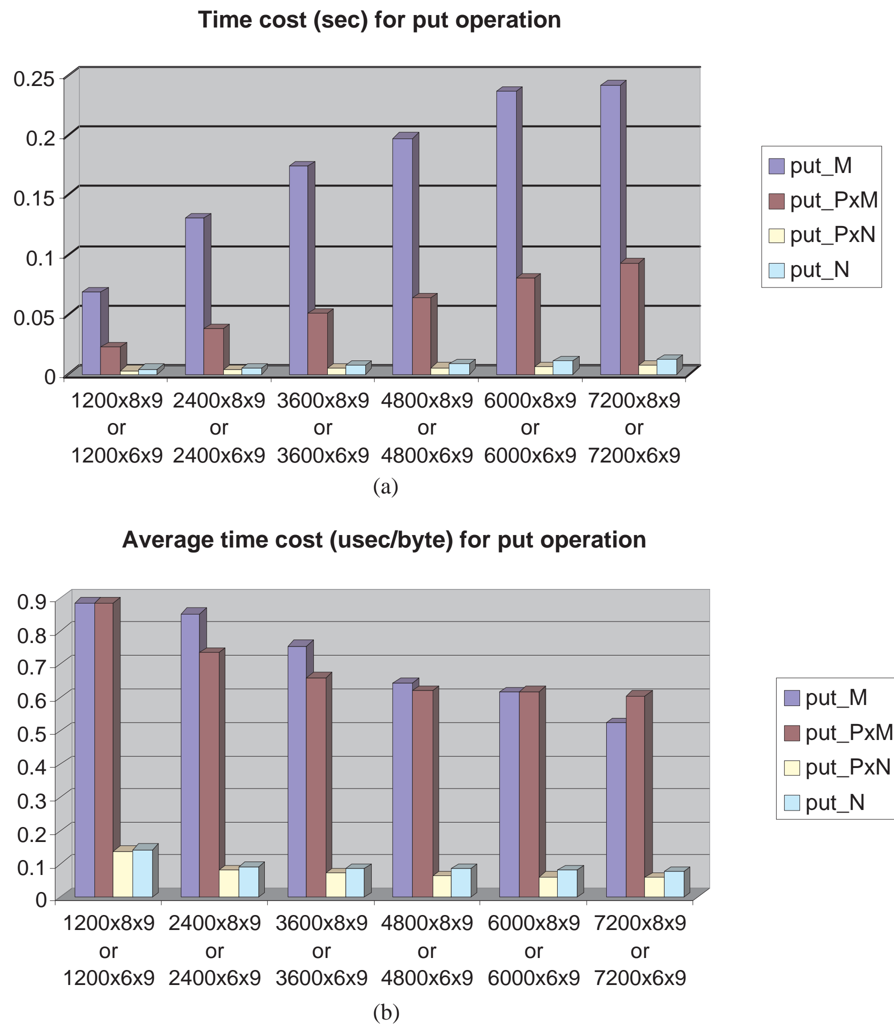


Figure 4.8: *put* operation time cost.

existing intervals if necessary, and insert nonexisting intervals. These result in extra registration costs in Figure 4.7(b). As we mentioned in 3.3.2, while registration costs are relatively high, these are one-time costs. The communication patterns created during the *register* operation are reusable as long as the domain decompositions of the simulations/applications does not change.

Data transfer time cost: As the measurement for the *register* operation, this experiment measures *put* cost from four perspectives. Figure 4.8 (a) shows *put* operation cost on one processor and Figure 4.8 (b) shows the average cost per byte for the operation. Figure 4.8 (a) shows the operation cost increases with the increasing data size on each mock system. The detailed explanations for this phenomenon has been previously presented in 3.3.2. Also shown

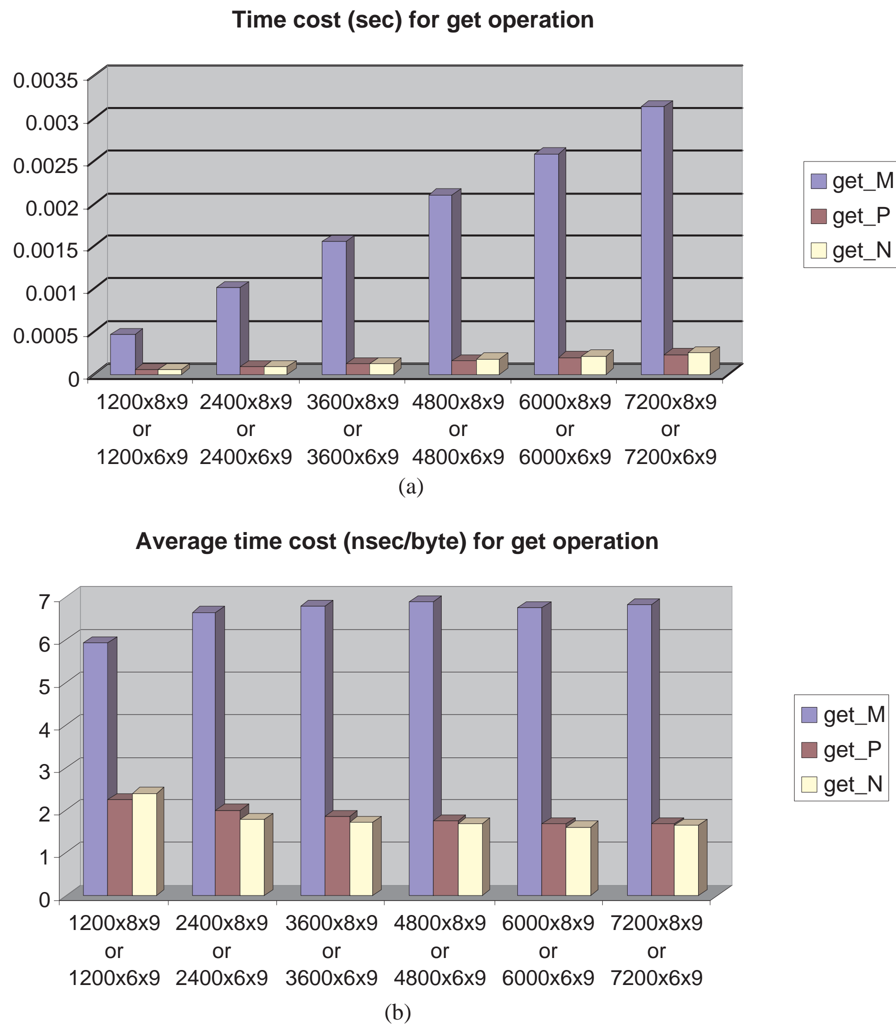


Figure 4.9: *get* operation time cost.

in the figure is the higher cost for *put* operation at site M, followed by PxM site. The reason is the same that for the *register* operation. Figure 4.8 shows that when the array size increases, the average time cost for *put* operation actually has an decreasing trend. The reason is that these overheads imposed by Seine-Coupe are not affected by the data sizes. Therefore, when this overhead is averaged over larger data sizes, the cost per byte decreases. Figure 4.8 (b) also shows that average data transfer cost between site M and site P is much higher than the one between site P and site N. The reason is that it uses inter-cluster communication while the communication between P and N uses intra-cluster communication. Moreover, the cluster running M has lower performance than the cluster running N and P.

Figure 4.9 shows the cost on a process. *get* essentially invokes a local memory copy operation. Figure 4.9 (a) shows that as the size of memory to be copied increases, the operation cost increases. Once again, since M is running on a cluster with lower performance, it shows comparatively higher costs. After eliminating the data size factor, Figure 4.9(b) shows the average *get* operation costs are almost constant from each application point of view.

Experiments with Wide-Area Coupling using the Prototype Seine-Coupe based Fusion Simulation

The experiments presented in this section were conducted between two sites; a 80 nodes cluster with 2 processors per node at Oak Ridge National Laboratory (ORNL) in TN, and 64 node cluster at the CAIP Center at Rutgers University in NJ. The synthetic XGC code is ran on the ORNL cluster, and the MI module and the synthetic MHD code ran on the CAIP cluster. That is, site M was at ORNL and sites P and N were are CAIP. The two clusters had different processors, memory and interconnects. Due to security restrictions at ORNL, these experiments were only able to evaluate the performance of data transfer from ORNL to CAIP, i.e., XGC pushing data to the MI module, which then pushes the data to MHD.

Since the *g*et operation in Seine-Coupe is local and does not involve data communication, the evaluations presented below focus on the *p*ut operation, which pushes data over the network. The experiments evaluate the operation cost and throughput achieved by the *p*ut operation.

Operation cost: This experiment evaluated the cost of Seine coupling operations. In this experiment, 7,200 particles were used in each poloidal plane resulting in an abstract domain of size 7,200x8x9 between XGC and MI and 7,200x6x9 between MI and MHD. The number of processors at site M, which ran the XGC code, was varied and the costs of the *r*egister and *p*ut operations were measured and plotted in figure 4.10. The plot shows the operation cost and the normalized operation cost, i.e., the cost per unit region in the abstract domain. The figure shows that as the number of processors at site M increases, the absolute time cost of the *r*egister and *p*ut operations decrease while the normalized time costs for them increases. The decrease in absolute time cost is because the size of the abstract domain is fixed, and as the number of processor increases, each processor registers/puts a smaller portion of this domain. Since the operation costs are directly affected by the size of the region, these costs decrease

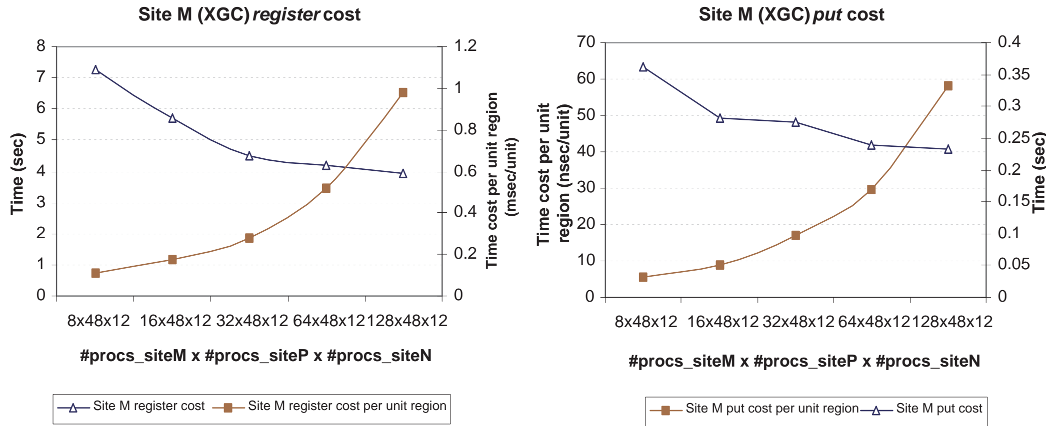


Figure 4.10: operation cost and normalized operation cost for *register* and *put* at XGC site.

accordingly. The increase in the normalized time cost is due to the increasing communication overhead imposed on each unit region being sent as system size increases. Detailed analysis of this behavior has been discussed in 3.3.2.

Throughput achieved: The goal of this experiment is to measure the per processor throughput that can be achieved during wide-area data coupling for different system and abstract domain sizes. In the experiment, the number of particles per poloidal plane was varied to be 7,200, 14,400, and 28,800, and the number of processors running XGC at site M were varied to be 8, 16, 32, 64 and 128. Throughput per processor in this experiment was calculated as the ratio of the average data size used by a *p*ut operation to the average cost of a *p*ut operation. Note that data transfers from the processors at site M occur in parallel and the effective application level throughput is much higher. The per processor throughput on site M is plotted in Figure 4.11(a) and the estimated effective system throughput are computed assuming different concurrency levels of the data transfer and plotted in Figure 4.11(b) and (c). Two observations can be made from the Figure 4.11(a). First, the per processor throughput at site M decreases with the number of processors used at site M, for all abstract domain sizes tested. This is because when the number of processors increases, the bandwidth available to each processor decreases, resulting in a lower throughput on each processor. Second, for the same number of processors at site M, in most cases the per processor throughput for smaller abstract domain sizes are higher than the throughput for larger abstract domain sizes. This is because for larger abstract domain sizes,

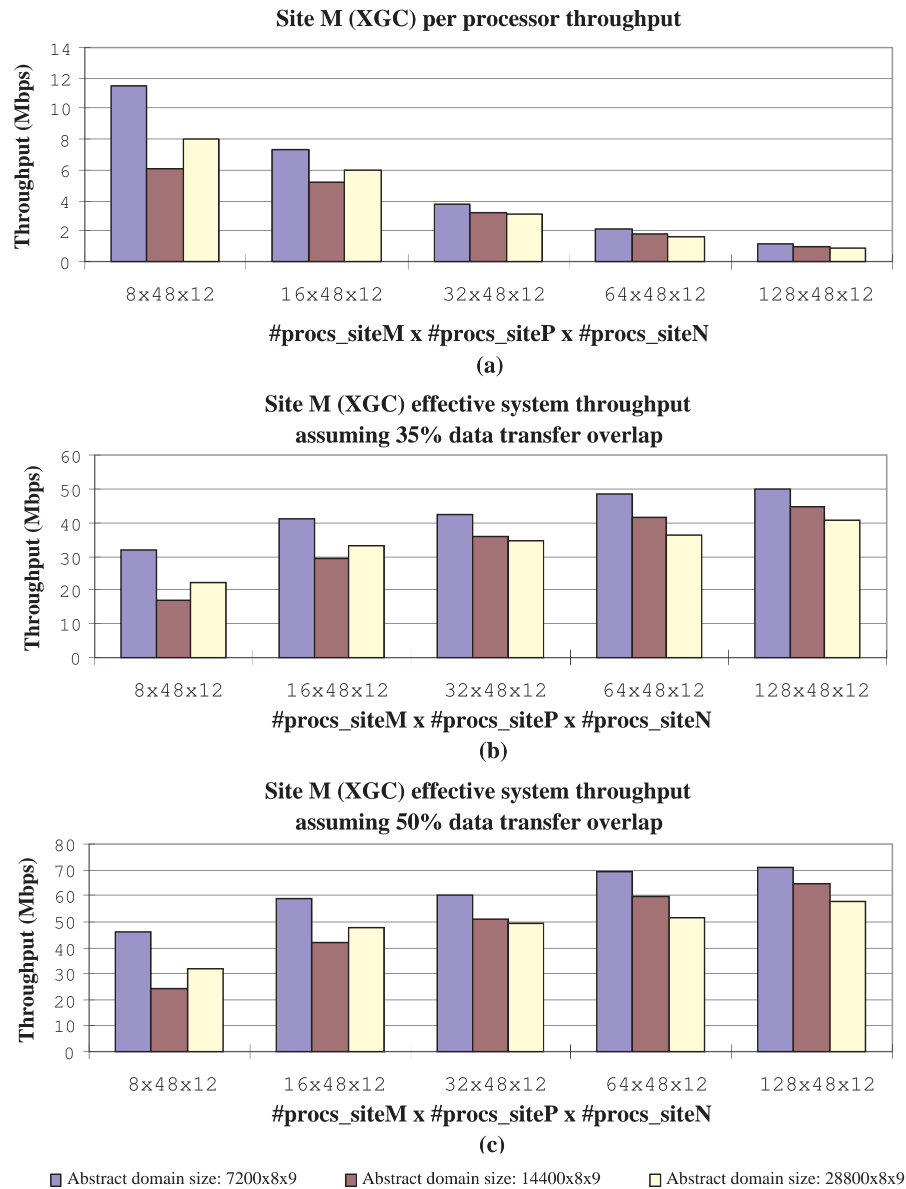


Figure 4.11: XGC site per processor throughput (plotted in (a)) and estimated effective system throughput assuming data transfer overlap of 35% (plotted in (b)) and 50% (plotted in (c)).

the size of data to be redistributed is correspondingly larger, resulting in more congested network. Further, the processors at site P are connected to the processors at both site M and site N. Consequently, site M processors have to compete with site N for connections with site P, which further causes the throughput to decrease for larger abstract domain sizes.

In the Fusion project, XGC site throughput is a key requirement that must be meet by the coupling framework. An initial conjecture of the transfer rate from XGC to MI is 120Mbps. By running test in a wide area environment as in the experiments presented above, one can analyze the feasibility of Seine-Coupe meeting the requirement. We compute the effective system bandwidth based on the per processor bandwidth measured above. The estimation assumes 35% and 50% overlap in the per processor data transfer respectively and the results are plotted in Figure 4.11(b) and (c). Since in real Fusion production XGC will be running on a large-scale platform with 40 I/O nodes, we look at the results measured for 32 processors in the figures. The estimated effective system throughputs are around 34 - 42Mbps assuming 35% transfer overlap and 50 - 60Mbps assuming 50% transfer overlap. While these figures are still not close to the Fusion throughput requirement, we believe that Seine-Coupe can meet such requirement when used in the real production run for two reasons: (1) the experiment conducted here is in a wide area environment while in real production XGC will be connected with MI with a private and extremely fast interconnection; (2) these experiments assumed an extreme case where data was continuously generated by XGC, which was pushed to MI and MHD, and therefore provides a conservative measure of the throughput that can be achieved. Experiment with a more realistic scenario will be presented next.

Effect of data generation rate: This experiment evaluated the effect of varying the rate at which data was generated by XGC at site M. In this experiment, XGC generates data at regular intervals, between which, it computes. It is estimated by the physicists that on average, XGC requires 3 times the computation as compared to MI and MHD. As a result, we experiment computes times for XGC, MI and MHD of (1) 0, 0 and 0 seconds (corresponding to the previous cases), (2) 30, 10 and 10 seconds, and (3) 60, 20 and 20 seconds respectively. The results are plotted in Figure 4.12. As seen from the plots, as the data generation rate reduces from case (1) to (2) to (3), the cost of the *p*ut operation and the throughput per processor improves, since the network is less congested. The jump is more significant from case (1) to (2) and less significant

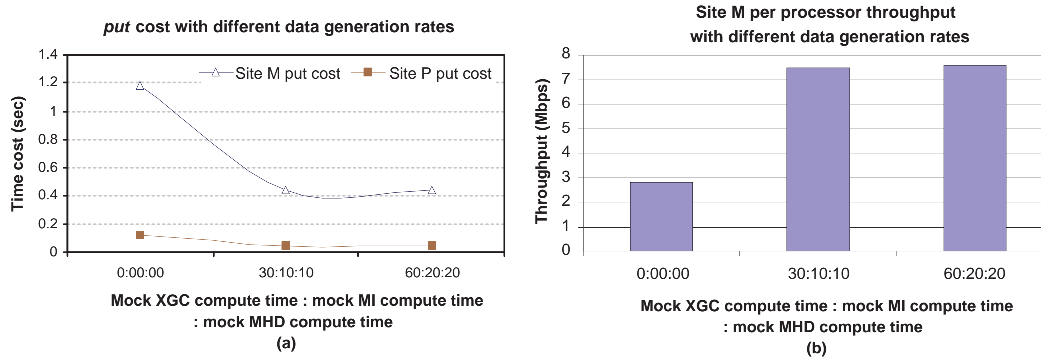


Figure 4.12: *put* operation cost and XGC site per processor throughput with respect to different data generation rates

from case (2) to (3).

4.4.5 Conclusion and Future Work

The mock simulation presented here is intended to be a proof of concept experiment. As more details about the data coupling scenario become available, specific requirements and relevant information regarding coupling criteria will be used to customize and optimize Seine-Coupe to address data coupling in the Fusion project. For example, in the future the 2D poloidal plane might be decomposed into circular portions, each of which wil be assigned to processors as a computational sub-domain on the processor. In this case, the data redistribution/data coupling cannot be simply based on plane id. We might need to extend the 3d abstract array index space to more dimensions to support extra dimensions introduced by more refined domain decomposition.

Chapter 5

Seine-Salsa: Seine-based Salable Asynchronous Replica Exchange for Parallel Molecular Dynamics Applications

5.1 Overview

Seine-Salsa is another prototype system based on Seine with a focus on asynchronous and decentralized replica exchange formulations for parallel molecular dynamics applications. Seine-Salsa provides a scalable communication and interaction substrate that presents a virtual shared temperature space abstraction and enables the dynamic and asynchronous interactions required by molecular simulations to be simply and efficiently implemented.

The "replica exchange" algorithm simulates the structure, function, folding, and dynamics of proteins. It is a powerful sampling algorithm that preserves canonical distributions and allows for efficient crossing of high energy barriers that separate thermodynamically stable states. In this formulation, several copies, or replicas, of the system of interest are simulated in parallel at different temperatures using "walkers". These walkers occasionally swap temperatures to allow them to bypass enthalpic barriers by moving to a higher temperature. The replica exchange algorithm has several advantages over formulations based on constant temperature, and has the potential for significantly impacting the fields of structural biology and drug design - specifically, the problems of structure based drug design and the study of the molecular basis of human diseases associated with protein misfolding, which are the applications currently targeted by this research.

However, efficient and scalable parallel implementations of general formulations of the replica exchange algorithm present significant challenges. These challenges are primarily due to the dynamic and complex coordination and communication patterns between the walkers. These communication/coordination patterns depend on state of the simulation at the replicas and are known only at runtime, and consequently, implementing these simulations using commonly used parallel programming frameworks is non-trivial. Message passing frameworks

such as MPI [2] require matching sends and receives to be explicitly defined for each interaction. Programming frameworks based on shared address spaces provide higher-level abstractions that can support more dynamic interactions. However, scalable implementation of global shared address spaces remains a challenge.

As a result, to the best of our knowledge, all the current parallel/distributed implementations of replica exchange simulations in use by the structural biology community are based on a simplified formulation of the algorithm that limits the potential power of the technique in two important ways: (1) the only parameter exchanged between the replicas is the temperature of each replica, and (2) the exchanges occur in a centralized and totally synchronous manner, and only between replicas with adjacent temperatures. The former limits the effectiveness of the method, while the latter limits its scalability to at most tens of homogeneous and relatively tightly coupled processors.

Enabling larger scale implementations of the general replica exchange formulation thus requires a communication substrate that can support decentralized management of exchange schedules, and asynchronous and decoupled communications, eliminating synchronization overheads and allowing implementations to be scalable. The chapter describes the design and implementation of Seine-Salsa, a communication and interaction substrate that meets these requirements, and enables a novel, decentralized and asynchronous realization of the replica exchange algorithm for simulating the structure, function, folding, and dynamics of proteins. Seine-Salsa enables arbitrary walkers to dynamically exchange target temperatures and other information in a pair-wise manner based on an exchange probability condition that ensures detailed balance.

The Salsa-based replica exchange realization distinguishes itself from existing implementations in multiple aspects. As mentioned above, existing replica exchange implementations use a simplification of the replica exchange algorithm where the walkers that can exchange temperatures is limited to those with neighboring target temperatures. In these implementations, the pairs of walkers that exchange temperatures is centrally determined at a single node by periodically gathering temperatures from all the walkers at this node. While such a scheme is acceptable for simulations with a small number of walkers, as the number of walkers increases, the possibility of successful non-nearest neighbor temperature exchange also increases significantly, and temperature mixing is severely impeded when exchanges can only occur between

neighboring temperatures.

Seine-Salsa essentially provides a virtual shared space abstraction that is specifically customized for replica exchange. The pairs of walkers that exchange information are dynamically and asynchronously determined using this virtual shared space abstraction. Walkers periodically post temperature ranges that are of current interest for exchange to the shared space. If this range overlaps with the range of interest posted by another walker, an exchange can occur. The actual exchange is then negotiated and completed by the individual walkers in a peer-to-peer manner. As a result, the exchanges are decoupled, dynamically and asynchronously determined, and not limited to neighboring temperatures. Seine-Salsa provides simple tuple-space-like [12] abstractions for accessing the virtual space. Walkers use the *post* operator to post temperature ranges of interest, and use either the blocking *get* or the non-blocking *getp* operator to retrieves a new temperature if the attempted exchange is successful, or the old one if the attempted exchange is unsuccessful. Further, since exchanges are decoupled and asynchronous, communications and computation at the walkers can be overlapped to improve overall simulation performance.

The design and implementation of Seine-Salsa is based on the following observations about the replica exchange algorithm: (1) the overall temperature range of the simulation and the set of temperatures that are assigned to walkers are determined at the beginning of the simulation and are known to all the walkers; (2) the temperature assigned to a walker only changes when the walker performs an exchange; and (3) exchanges occur between pairs of nodes. The first two observations allows individual walkers to locally determine temperature ranges of interest and exchange decisions to be made in a decentralized and decoupled manner. The third observation allows actual exchanges to occur between pairs of walkers in parallel. Seine-Salsa, like the other Seine architecture, consists of two main components: a directory layer that is implemented as a distributed hash table (DHT) where walkers can post temperature ranges of exchange interest and discover potential exchange partners in a decentralized and asynchronous manner; and a communication layer that manages the actual exchange of data in an efficient and peer-to-peer manner.

Seine-Salsa has been implemented within the IMPACT (Integrated Modeling Program, Applied Chemical Theory) molecular mechanics program [3] to enable two specific applications,

which require large scale distributed replica exchange implementations: (1) simulations of the binding of ligands to the cytochrome P450 class of enzymes responsible for cellular detoxification and drug metabolism, and (2) simulations of the misfolding of naturally occurring human and mutated forms of the protein synuclein associated with Parkinson's disease. It is not possible to carry out these studies using standard molecular dynamics simulation techniques.

The rest of the chapter is organized as follows. Section 5.2 describes the problem domain and gives an overview of the replica exchange algorithm. The section also describes current MPI-based implementations of the method. Section 5.3 describes the design of Seine-Salsa and presents the implementation of the Salsa-based decoupled and asynchronous replica exchange algorithm. Section 5.4 describes the implementation and the experimental evaluation of Salsa-based simulation. Section 5.5 reviews related work. Section 5.6 concludes the chapter and outlines future research directions.

5.2 Parallel Replica Exchange for Structural Biology and Drug Design

The sequencing of the human genome, in conjunction with rapidly increasing efforts in structural genomics, is producing an explosion in the number of available high resolution protein structures. Molecular simulations of protein structural changes and drug binding to proteins depend critically on the design of highly efficient algorithms to search over the very rough energy landscapes that govern protein folding and binding. Scalable parallel replica exchange implementations can potentially address these molecular search problems and can significantly impact structure based drug design applications.

5.2.1 The Replica Exchange Algorithm

Replica exchange is an advanced canonical conformational sampling algorithm designed to help overcome the sampling problem encountered in biomolecular simulations. The method had been proposed independently on several occasions in various disciplines [5, 6, 7, 8]. In this method, several copies, or replicas, of the system of interest are simulated in parallel at different temperatures using walkers. These walkers occasionally swap temperatures based on a proposal probability that maintains detailed balance [4]. These exchanges allow individual

replicas to bypass enthalpic barriers by moving to high temperatures. A parallel version of this algorithm was proposed by Hukushima & Nemoto [8]. The replica exchange algorithm is easy to implement and does not require time-consuming preparatory procedures. Further, it can decrease the sampling time by factors of 20 or more, as compared to constant temperature molecular dynamics when applied to peptides at room temperature [9]. Details of the algorithm [4] as well as application examples [10, 11] are available elsewhere.

The MD replica exchange canonical sampling method has been implemented in IMPACT, the molecular simulation package used in this work, following the approach proposed by Sugita & Okamoto [10]. The method consists of running a series of simulations at fixed specified temperatures. Each replica corresponds to a temperature. An exchange of temperatures between replicas i and j at temperatures T_m and T_n is attempted periodically and is accepted according to the following Metropolis transition probability [10]:

$$W = \min\{1, \exp[-(\beta_m - \beta_n)(E_j - E_i)]\} \tag{5.1}$$

where $\beta = 1/kT$ and E_i and E_j are the potential energies of replicas i and j, respectively. After a successful exchange, the velocities of replicas i and j are rescaled at the new temperature.

5.2.2 Existing Parallel Implementations of Replica Exchange-based Simulations

Molecular dynamics programs are essentially loops over a large number of integration steps, each of which advances the time forward for one step. Replica exchange is attempted periodically at a chosen interval of steps. As mentioned before, existing MPI-based parallel implementations of replica exchange are centralized and synchronous. For example, in the existing implementation in IMPACT, a central master node collects temperature data about all the replicas from the walker nodes, and then broadcasts the collected data array to the walkers. Each walker node receives this data array and sorts the array locally. Neighboring temperatures in the sorted array are potential partners for temperature exchange. The master node randomly selects between two modes of exchange. One is to exchange with upper neighboring temperature and the other is to exchange with lower neighboring temperature. The master notifies the walkers about the selected mode, and walkers can then mutually exchange temperatures based on this

information. During the actual exchange, one of the two walker nodes with neighboring temperatures in the sorted array that are paired up for temperature exchange, acts as a temporary server. This walker collects temperature and potential energy data from the other node, determines whether the exchange is feasible based on the transition probability given in Eq. (5.1), and replies with either the new temperature, if the exchange is successful, or with a notice of denial otherwise.

The parallel replica exchange implementation described above has several limitations. First, the scheme limits the exchange to only neighboring temperatures. This limitation is not a concern when the number of replicas is small and there is a small chance of exchange between non-nearest temperatures. However, as the number of processors (and correspondingly walkers) increases, the difference between target temperatures becomes small enough to allow exchanges between non-nearest neighbor replicas. In such cases, more flexible schemes which allows non-nearest neighbor temperature exchange are desirable. Second, the implementation is based on a centralized master that gathers and scatters data system wide. Gathering data from all the nodes on a single node may be infeasible in large systems, and a centralized master can quickly become a bottleneck. Further, gather and scatter operations are synchronous and expensive. Also, since the master node also participates in the simulation as a walker, there is a load imbalance which can lead to additional synchronization overheads.

To overcome the above limitations, we propose a scalable decentralized and asynchronous realization of the replica exchange algorithm using the Seine conceptual model and the Seine-Salsa communication substrate, which is presented in the following section.

5.3 Seine-Salsa: A Framework for Parallel Asynchronous Replica Exchange

5.3.1 The Seine-Salsa Architecture

Seine-Salsa provides abstractions and underlying mechanisms to support efficient and scalable parallel implementations of the general replica exchange formulation, where walkers can exchange non-nearest neighbor temperatures in a decoupled, decentralized, and asynchronous manner. It essentially provides the abstraction of a virtual shared space that is specifically customized for replica exchange. The shared space supports tuple-space-like abstractions and is

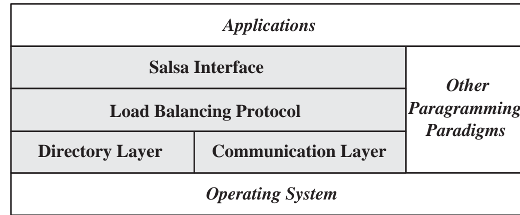


Figure 5.1: An schematic of the Seine-Salsa architecture.

used by walkers to post temperature ranges of exchange interest and discover candidate walkers that it can potentially exchange temperature with. The underlying mechanisms support negotiation and efficient peer-to-peer data exchanges between appropriate walkers. A schematic of the Seine-Salsa architecture is presented in Figure 5.1. The framework consists of two main components: (1) a distributed directory layer; and (2) a communication layer. These components are described below.

The Seine-Salsa Distributed Directory Layer

The distributed directory layer provides the Seine-Salsa virtual shared space abstraction and supports its associative access operators. It is implemented as a distributed hash table (DHT) where the index of the hash table is derived from the overall temperature range used by the simulation using a simple hashing function. This index is then dynamically partitioned and distributed across the participating nodes. Each node is thus responsible for its portion of the index and the corresponding temperature range, and essentially serves as the rendezvous point for exchange interest posting that intersects with its range. A Seine-Salsa service daemon running at each node is responsible for handling these exchange interest postings and storing them locally, and for detecting matches with existing postings of exchange interest at the node.

The Seine-Salsa Communication Layer

The communication layer provides the mechanisms for negotiations and efficient data transfers. The negotiation mechanisms enable walkers to mutually agree to exchange data, while the data transfer mechanisms support low latency peer-to-peer data exchanges [22] between the walkers. Note that multiple data exchanges between different pairs of walkers can proceed in parallel.

Table 5.1: Seine-Salsa application programming interface.

Operation	Description
init(global-temperature-range)	Initialize the shared space.
post(exchange-temperature-lower-bound, exchange-temperature-upper-bound)	Post a temperature range of exchange interest to the space.
get(?temperature, energy)	Get the exchanged temperature from the space. This is a blocking call and the calling process blocks until a matching temperature is available. The retrieved temperature is removed from the space.
getp(?temperature, energy)	Get the exchanged temperature from the space. This is a non-blocking call and the calling process continues if no matching temperature is available. The retrieved temperature is removed from the space.

5.3.2 Seine-Salsa Programming Interface

The operators provided by the Seine-Salsa application programming interface (API) are listed in Table 5.1. These operations provide associative access to the virtual shared space and are similar to the operators provided by the tuple space model [12]. The interface includes operators to initialize the Seine-Salsa runtime (*init*), to allow processes to post (*post*) temperature range of exchange interest and retrieve available exchanged temperatures (*get*/*getp*). Note that the retrieved temperatures are removed from the space.

The replica exchange algorithm can be simply implemented using the Seine-Salsa API as illustrated by the psuedo-code presented in Figure 5.2. The MPI-based implementation tends to be significantly longer and more complex. Note that as it is based on the Seine conceptual model, these Seine-Salsa operators can be easily extended to support "temperature plus potential parameter replica exchange" formulation that facilitates barrier crossing by "flattening" the energy barriers in addition to kinetically activating the crossing by heating the system. Specifically, by extending the 1D shared temperature space to multi-dimensional domain-specific spaces, it is straightforward to implement the "temperature plus potential parameter replica exchange".

5.3.3 Parallel Asynchronous Replica Exchange using Seine-Salsa

```
if (seineinitflag .eq. 0) then
      call init_salsa (global_temperature_lowbound,
                        global_temperature_upperbound)
      seineinitflag = 1
      timestamp = 0
else
      timestamp = timestamp + 1
endif

if (timestamp .eq. (timestamp/exchange_rate)*exchage_rate)
then
      call post (tempt(nspec+1) - GUESSRANGE,
                 tempt(nspec+1) + GUESSRANGE)
endif
call getp (newtemp, epot, accepted)
```

Figure 5.2: Pseudo-code illustrating the implementation of replica exchange using Seine-Salsa.

The overall operation of Seine-Salsa is as follows. When a walker attempts to exchange its current target temperature, it computes a temperature range that it is willing to exchange with, and posts this range to the shared space using the *post* operator. Based on the temperature range posted, the request is routed to all the service daemons whose index ranges overlap with the hashed posted range. Note that a posted temperature range may be unevenly distributed across the directory nodes resulting in load balancing issues. Currently Seine-Salsa addresses the issue using a simple load balancing protocol and plans to further optimize the protocol. When a remote *post* request is received by a service daemon, the daemon first checks its storage for potential exchange partners. If a candidate exists (say $walker_2$), the requesting walker (say $walker_1$) is notified. Otherwise, the incoming request is stored.

Since a *post* request typically maps to multiple directory nodes and is therefore sent to multiple service daemons, it is possible that a requesting walker is notified of multiple candidates, if more than one daemons find a potential exchange partner. In this case, the first notification that reaches the requesting walker is accepted. However, the exchanging walkers must mutually agree to exchange data with each other. This requires a two-way handshake. In the example above, $walker_1$ will send out a query message to ask $walker_2$ whether it is available for an exchange. On receiving this query from $walker_1$, $walker_2$ checks it local state. This state can be “free”, “onhold”, or “finished”. The walker is available for an exchange only if it is in the “free” state. The “onhold” state indicates that the walkers has already agreed to exchange with another walker but exchange has not yet occurred. The “finished” state indicates the walker

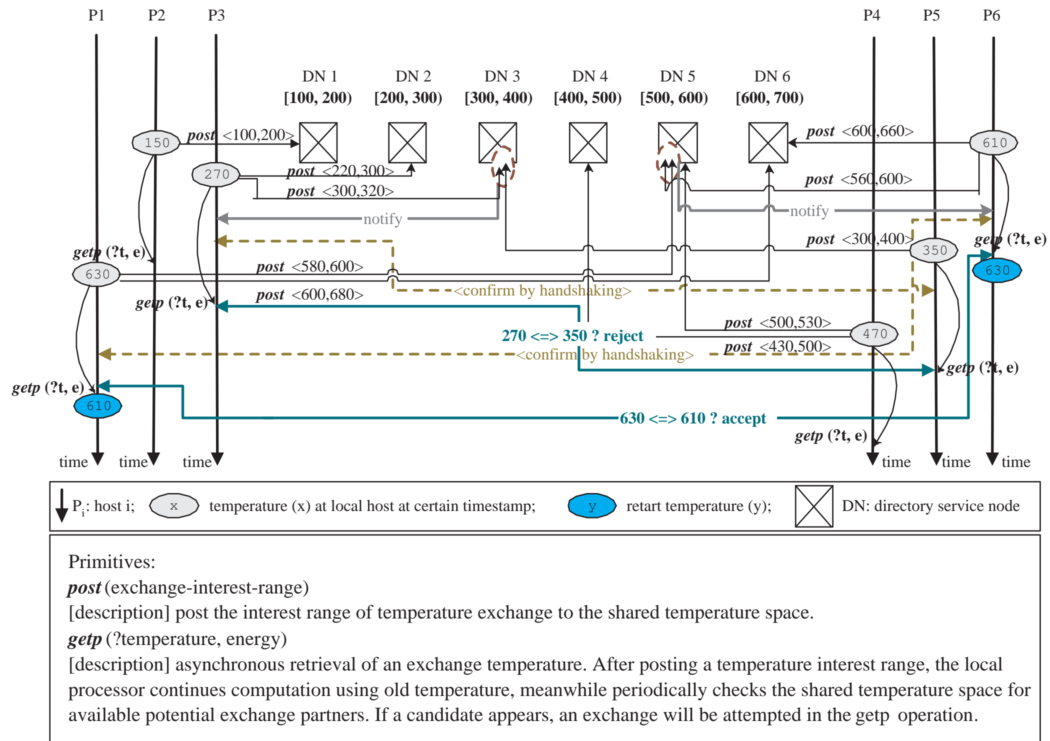


Figure 5.3: An example parallel asynchronous replica exchange implementation using Seine-Salsa.

has already finished an exchange with another walker and its posted interest to exchange is no longer valid. In the example, if $walker_2$ is in the "free" state it will respond positively to $walker_1$. At this point both walkers confirm their intent to exchange data with each other and change their state to "onhold". If $walker_2$ had responded negatively, $walker_1$ would have continued to negotiate with other walkers that it had been notified of until it finds a willing exchange partner or it has no more walkers to negotiate with. In the latter case, it just gives up and continues simulation with its current data until the next exchange cycle.

Once a pair of walkers agree to exchange data, they initiate the actual exchange by invoking the *get* or *getp* operator. The exchange is performed using the mechanisms provided by the communication layer. The exchange proceeds as follows. One of the walkers sends its current data (e.g. temperature and energy) to its potential partner. The potential partner determines whether they can exchange based on data it receives and its own data. This step is necessary since the exchange happens asynchronously and in parallel with the computation, and a walker's data (i.e., energy) may have changed since it posted its exchange interest. If

the walker decides to continue with the exchange, it will notify its partner of the decision and sends its current local data to complete the exchange. Note that an exchange is between a pair of walkers and multiple exchanges between different pairs of walkers can proceed in parallel.

An implementation of a parallel asynchronous replica exchange using Seine-Salsa is illustrated in Figure 5.3. As described above and shown in the figure, the scheme consists of two phases. In the *post* phase, candidate exchange partners are identified and notified. These walkers negotiate with each other to create potential exchange partnerships. In the *get* or *getp* phase, these potential partners then attempt to exchange data.

In the Salsa-based replica exchange algorithm, a walker specifies the temperature range that it is interested in exchanging with as a parameter of the Seine-Salsa *post* operator. Usually, the larger the range, the higher is the probability of finding an exchange partner and will result in better solution quality. However, a larger range will also map to a large number of directory nodes and the *post* request will be forwarded to a large number of service daemons. This in turn increases communication overheads. The service daemons are also more loaded in this case, reducing their performance. Consequently, the temperature range posted must reflect the best tradeoff between solution quality and simulation performance.

5.4 Seine-Salsa Implementation and Experimental Evaluation

A prototype of Seine-Salsa has been implemented using multi-threading and TCP sockets. When Seine-Salsa is initialized by the application, a Seine-Salsa thread is spawned at each node in the system, which co-exists in the application address space. This thread acts as a Seine-Salsa service daemon and handles post request. Seine-Salsa service daemons discover and coordinate with each other to construct the directory layer DHT structure using a bootstrap server.

The correctness, effectiveness and scalability of the Salsa-based replica exchange algorithm is evaluated using the alanine tripeptide molecule. The tests are conducted on up to 68 processors (due to availability) using two Beowulf cluster, with 64 processors and 8 processors respectively, which consists of Linux-based computers connected by 100 Mbps full-duplex switches. Each processor has an Intel(R) Pentium-4 1.70GHz CPU with 512MB RAM and

runs Linux 2.4.20-8 (kernel version). All the tests are configured to run for 10 ns total simulation time using the Hybrid Monte Carlo (HMC) [23] molecular dynamics sampling algorithm. Each run is composed of 250,000 HMC cycles, each including 10 molecular dynamics integration steps with a 4 fs time-step. Replica exchange is attempted every 25 HMC cycles. Replica exchange temperatures are distributed exponentially within the 200-700 K range. The experiments compare the efficiency and performance of the Salsa-based implementation with the original MPI-based implementation in Impact. These experiments are described below.

5.4.1 Initial Test of correctness for Salsa-based Replica Exchange

First of all, we want to show that the Salsa-based asynchronous algorithm gives the same result as the synchronous results in cases when both can be tested reliably (i.e. using few walkers on a homogeneous computing cluster). An alpha conformation is defined as one in which a ”hydrogen bond” exists between the first aminoacid and the third aminoacid of the tripeptide molecule. For both the asynchronous and synchronous algorithm the simulation length was 10 nanoseconds (250,000 HMC cycles each made up of 10 MD steps with a 4 femtosecond time step). For the synchronous calculation the rate of attempted temperature exchange rate was once every 25 HMC cycles. For the asynchronous algorithm the exchange rate was chosen randomly and between once every 25 and 200 HMC cycles.

We have computed the melting curve for the alpha conformation of the tripeptide molecule using the Salsa in Impact with 8 walkers and both the synchronous and asynchronous algorithms. The results are as follows. The table shows that the predicted population of the alpha conformation is the same within the precision of the calculation for the synchronous and asynchronous algorithm, which proves the consistency between the results computed by using the Salsa-based asynchronous Replica Exchange and using the synchronous one.

5.4.2 Salsa-based vs. MPI-based Replica Exchange

An important feature of the Salsa-based replica exchange implementation is its ability to support non-nearest neighbor temperature exchanges. This feature is essential for ensuring proper

[1]When the system size is large, *post* requests are only sent to a subset of the service daemons that correspond to the requests to reduce communication overheads.

Temperature (K)	Asynchronous	Synchronous
200	0.8893	0.8883
239	0.8761	0.8645
286	0.8361	0.8263
342	0.7604	0.7755
409	0.7117	0.6813
489	0.5617	0.5420
585	0.4122	0.3740
700	0.1711	0.2093

Table 5.2: Number of temperature cross-walk events.

Number of walkers	8	16	32	60	120	136
Temperature range (in Kelvin) for measuring the number of cross-walks (write rate: per 250 steps)	250=> 650=> 250	250=> 650=> 250	250=> 650=> 250	250=> 650=> 250	250=> 650=> 250	250=> 650=> 250
Posted temperature range for replica exchange	Salsa-based simulation					
[-80K, 80K]	106	120	249	347	798 [1]	941 [1]
	MPI-based simulation					
	100	101	94	44	6	3

mixing of temperatures across the walkers, especially when the simulation includes a large number of walkers. At equilibrium each walker visits each temperature with equal probability. The rate of temperature equilibration is measured by the number of "cross-walks", whereby a walker originally within the low temperature range (200 K $\leq$ T $\leq$ 250 K) reaches the upper temperature range (650 K $\leq$ T $\leq$ 700 K) and then returns to the lower temperature range. As shown in Table 5.2, the Salsa-based replica exchange implementation achieves a larger number of cross-walks as compared to the synchronous MPI-based replica exchange implementation. This is because the synchronous MPI implementation only supports nearest neighbor temperature exchanges, and the walkers have to travel through every temperature in order to complete a cross-walk in the temperature space. This diffusion process is particularly slow when the replica exchange simulation includes many temperatures. Since Seine-Salsa supports non-nearest neighbor temperature exchanges, a walker can "jump" through temperature space, resulting in a faster rate of temperature equilibration.

Table 5.2 shows the number of temperature cross-walks measured for Salsa-based replica exchange simulations with 8, 16, 32, 60, 120 and 136 walkers, compared with the corresponding number of cross-walks obtained using the MPI-based synchronous implementation. In these experiments, the temperature range posted by walkers in the Salsa-base replica exchange implementation was set to a window of 300 K around its target temperature, i.e., as [temp - 80 K, temp + 80 K]. In this configuration, a walker can exchange temperature with any other walker whose target temperature is less than 160K away, i.e., the ranges that the two walkers post intersect. The temperature ranges that defined a cross-walk in the experiments are listed in the Table 5.2. As seen from the results listed in the table, the number of observed temperature cross-walks is larger for the Salsa-based simulation, and it increases as the number of walkers increases. In contrast, the number of cross-walks observed for the MPI-based implementation decreases as the number of walkers increases. With 136 walkers there are only 3 temperature cross-walks observed in the case of the MPI-based implementation. In comparison 941 cross-walks are observed for the Salsa-based implementation. These results demonstrate that using non-nearest neighbors replica exchange algorithm enabled by Seine-Salsa provides significant benefits, especially when the density of the temperature distribution is large.

5.4.3 Scalability of Seine-Salsa

The decentralized design of Seine-Salsa enables it to scale well with increasing number of processors. This experiment measures the wall-clock execution time for the Salsa-based and MPI-based implementations of the replica exchange simulation with 8, 16, 32, 60, 120, and 136 walkers. The results are plotted in Figure 5.4. Note that for the simulation with the cases of 120 walkers and 136 walkers, 2 walkers were mapped to each processor since the combined system used had only 68 processors available. The execution time for both implementations are appropriately scaled for this case. Also note that, since Salsa-based replica exchange executes in a decentralized asynchronous manner, different walkers or processes may proceed at different speeds and therefore have different execution times.

The measurements plotted in Figure 5.4 correspond to the average execution time across all the walkers/processes. A large portion of the execution time is due to local potential energy evaluations necessary to propagate each walker in time. This component of the execution time

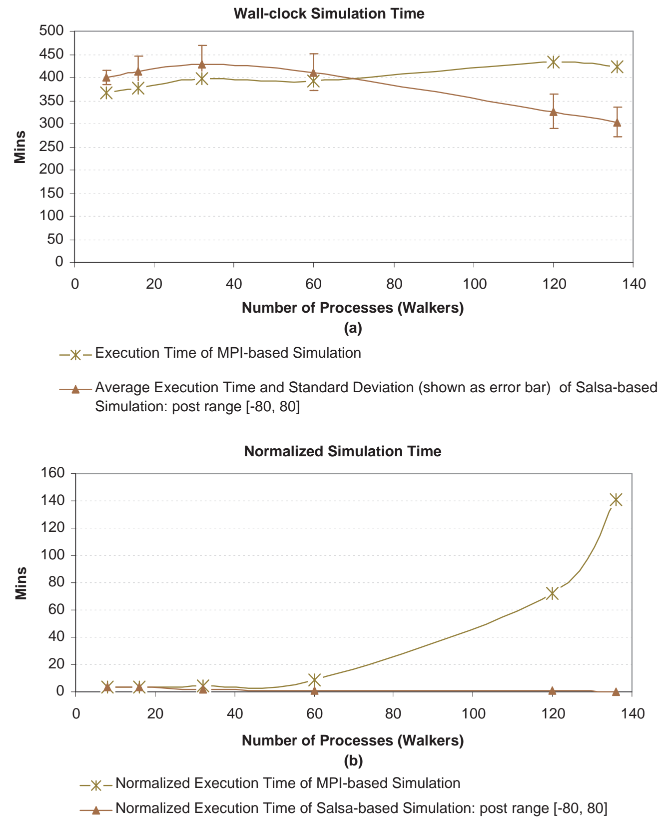


Figure 5.4: (a) Average wall-clock execution time and standard deviation with increasing number of processes (walkers); (b) Normalized execution time with increasing number of processes (walkers).

is the same for both implementations. The remaining portion of the execution time is due to exchange communications and includes both communication times and the synchronization overheads. This component of the execution time is proportional to the number of exchanges and is different for the two implementations.

Figure 5.4(a) plots the average run-times for the two implementations for different numbers of processors and correspondingly, walkers. This plot reflects a combination of factors. First, since in the MPI-based implementation, exchanges are centralized and synchronous, they have high probability to be successful. However, in the case of Seine-Salsa, exchanges are decentralized and asynchronous, and many initiated exchanges may not succeed. As a result, there is some "wasted" communication in Seine-Salsa. However since this communication occurs in parallel and is overlapped with computations its impact is not as significant. Second, the

Salsa-based simulations result in a larger number of cross-walks and thus perform a significantly larger number of exchange communications. The combined result of these two factors results in higher execution times for the Salsa-based implementations for up to 60 walkers. For a larger number of walkers, the bottleneck caused by centralization and synchronization in the MPI-based implementation becomes significant as apparent for the 120 and 136 walker cases in the plot. We believe that the impact of this bottleneck will be even more pronounced for larger systems and for distributed systems where gather/scatter operations are expensive.

It is also useful to consider the relative effective performance of the two implementations. As shown in Table 5.2, the Salsa-based implementation produces more temperature cross-walks that in turn are reflected in a smaller convergence time, i.e., the time required to obtain a particular thermodynamic quantity of the molecular system within a given statistical uncertainty. To evaluate the effective gain in performance achieved using Seine-Salsa, Figure 5.4(b) plots the normalized execution time for both implementations, obtained by dividing the wall clock execution time by the number of cross-walks. This gives the average time required to achieve one temperature cross-walk. As this figure shows, for small number of processes the Seine-Salsa and MPI implementations have similar effective performance. However as the number of processes and correspondingly, the number of walkers increases, the performance of the Salsa-based simulation increases noticeably.

5.4.4 Effect of Posted Temperature Ranges on Cross-Walks

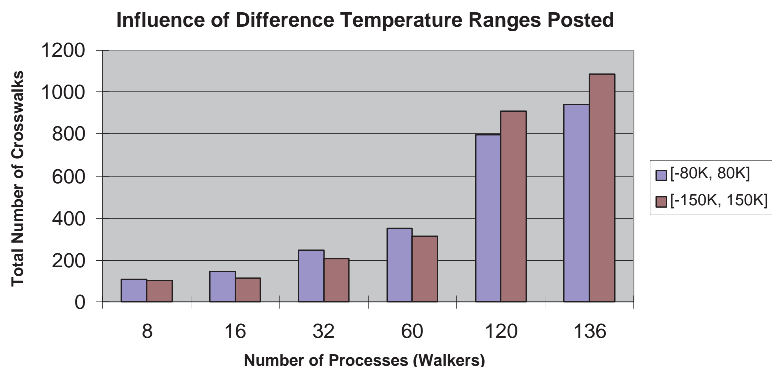


Figure 5.5: Influence of different posted temperature ranges on the number of cross-walk events.

The temperature range posted by a walker ([-80 K, 80 K] in the experiment presented above) is an adjustable parameter that can be optimized for a specific system and number of walkers, to achieve the fastest convergence. To illustrate the effect of the posted temperature range on the number of temperature cross-walks, Figure 5.5 plots the number of cross-walks obtained using the ranges [-150 K, 150 K] and [-80 K, 80 K]. These results show that the [-80 K, 80 K] temperature range gives better results than the [-150 K, 150 K] range in the cases of 8, 16, 32, and 60 walkers while [-150 K, 150 K] gives better result in the case of 120 and 136 walkers.

5.5 Related Work in Parallel Replica Exchange

Existing parallel implementations of replica exchange are based on a simplified formulation of the replica exchange algorithm as described previously, and use a centralized master to periodically schedule and manage exchanges. These implementations are either directly build on sockets, as in [4], or use message passing libraries such as MPI or PVM, as in [14]. In [14], H. J. C. Berendsen et al present a parallel replica exchange implementation developed specially for a ring topology. The implementation is suitable for systems where the processors can be configured as a (logical) ring and which support blocking send and receive calls. In this implementation, each processor only communicates with its two nearest neighbors on the ring. Like the MPI-based implementations, this implementation also supports only nearest neighbor temperature exchanges.

The Folding@home [13] project at Stanford University has proposed a multiplexed replica exchange algorithm. The algorithm uses multiplexed-replicas with a number of independent molecular dynamics runs at each temperature, and attempts exchanges of configurations between these multiplexed-replicas. In this formulation, the efficiency of the simulation is enhanced as a number of independent molecular dynamics simulation replicas are run at each temperature and there are a larger number of potential exchange partners available. Further, the multiplexing between replicas is arranged in such a way that the discrepancy between exchange partners is reduced. In contrast, Seine-Salsa improves simulation efficiency by eliminating the limitation of nearest neighbor exchanges, instead of introducing redundant computations. Both algorithms, however, use parallelism to improve the efficiency of the simulation.

To the best of our knowledge, the work presented in this paper is the first to address the decentralized and asynchronous parallel implementation of replica exchange. This not only improves scalability but also improves efficiency by enabling non-nearest neighbor temperature exchanges, which is desirable for simulations with a large number of replicas.

5.6 Conclusion

The chapter presented a novel implementation of replica exchange algorithm using Seine-Salsa, a Seine-based framework that presents a shared space abstraction to applications. The Salsa-based replica exchange supports exchange between non-nearest neighboring temperatures, asynchronous communication to enable overlapping communication with computation, decentralized communication paradigm to avoid the central bottleneck in a client/server version of the implementation. The approach helps improving the overall efficiency and scalability of the simulation.

The overall goal of the project is to enable large-scale Grid-based parallel and distributed molecular simulations of protein structural changes and drug binding to proteins. Specific tasks include (1) implementing a prototype interaction and coordination framework, based on Seine-Salsa, for wide-area distributed replica exchange simulations, (2) developing, deploying and evaluating the Grid-based Impact implementation, and (3) using the grid-based Impact implementation to provide scientific insights .

Chapter 6

Summary, Conclusion, and Future Work

6.1 Summary

The primary objective of the research presented in this thesis is to investigate programming abstraction and middleware mechanisms that addresses the dynamic and complex communication/coordination requirements of parallel scientific applications. The Seine approach supports the flexible and efficient intra-coupling and inter-coupling of parallel scientific applications and presents an application-layer interface that can dramatically improve programmability, offer flexility to the application layer, and ease the coupling between independent applications.

This thesis presented the Seine interaction framework. Seine builds on the Tuple Space model and is semantically specialized to specific application fields. This research (1) highlights the absence of support for addressing the dynamic and complex communication and coordination patterns required by parallel scientific applications and proposes the use of a semantically specialized shared space abstraction to address these requirements; (2) enables flexible and efficient intra- and inter-coupling using the developed semantically specialized shared space, and (3) provides a runtime interaction framework that can be adapted to different intra-/inter-coupling scenarios.

Three Seine prototype systems implement domain-specific shared space abstractions. Seine-Geo enables scalable, efficient, and flexible intra-coupling of parallel scientific applications. Seine-Geo is based on the observations that most scientific application use a geometric discretization of the problem domain and that communication/interactions in these applications are between entities that are local to this discretized domain. It builds on the tuple space model, providing the flexibility of this model, extending it to support geometry-based access operators, and enabling scalable implementations. Seine-Geo also supports dynamic creation and deletion of shared spaces. The effectiveness and flexibility of the framework supporting dynamic and complex communication requirements is evaluated using an adaptive multi-block oil reservoir simulation. The experimental evaluation demonstrated the performance and scalability of the

framework.

The inter-coupling framework, Seine-Coupe, specifically addresses the problem of data coupling and parallel data redistribution. Seine-Coupe enables efficient computation of communication schedules, supports low-overheads processor-to-processor data streaming, and provides high-level abstractions to application developers. A component-based prototype implementation of Seine-Coupe using the CAFFEINE CCA framework is implemented and evaluated. The Seine-Coupe CCA coupling component enables parallel data redistribution between multiple CCA-based applications. The evaluation demonstrates the performance and low overheads of Seine-Coupe. Seine-Coupe is also being used in the undergoing SciDAC Fusion Project, to couple two independent code packages, XGC and M3D, in the Fusion Plasma Edge simulation. A mock runtime execution environment is set up to simulate the real program execution environment and to evaluate Seine-Coupe in this context.

Seine-Salsa is another adaptation of Seine that addresses the specific communication and coordination requirement of the replica exchange algorithm in Molecular Dynamics applications. Seine-Salsa enables a novel implementation of replica exchange based on the abstraction of a shared temperature space in Molecular Dynamics Applications. The flexibility provided by the Seine-Salsa shared space abstraction enables the Salsa-based implementation of the replica exchange algorithm to support exchange between non-nearest neighboring temperatures, asynchronous communication to enable overlapping communication with computation, decentralized communication paradigm to avoid the central bottleneck in a client/server version of the implementation, and is able to break through the limitation imposed by the MPI-based approach. The approach helps improving the overall efficiency and scalability of the simulation.

6.2 Conclusion

Emerging large-scale parallel scientific applications and the code, model, and data coupling requirements by these applications pose challenging requirements. These requirements warrant middleware supports that feature powerful coordination mechanism, efficient runtime, and an application oriented interface.

The lack of support for these emerging communication requirements of current parallel

scientific applications has, in most cases, lead to application developers having to handle all complexity. Moreover, since these communication/coordination requirements depend on the current state of program execution and change dynamically at runtime, they can not be specified at the development phase. As a result, programming these interaction patterns is extremely difficult, and application developers resort to static approaches which can lead to more collective operations or inefficient implementations. Existing approaches which address coupling of parallel simulations either use a central controller to manage the coupling between two models/systems/packages, or are based on the message-passing paradigm. These coupling systems either lack scalability due to the introduction of the central controller or an application-oriented higher level abstraction to ensure programmability and flexibility of the system.

The Seine interaction framework adopts a distributed directory layer to provide a discovery service for setting up arbitrarily complex and dynamic communication/coordination patterns, a storage layer to present a local shared space abstraction, and a communication layer to facilitate efficient data transfer, adaptive buffer management and other necessary application-specific communication protocols in a peer-to-peer manner. The combination of the three layers enables a flexible, efficient, scalable, and application-friendly intra-/inter-coupling support for parallel scientific applications. The effectiveness of the support is demonstrated by three prototype implementations and experimental evaluations of these systems.

6.3 Directions For Future Work

We envision the following directions for future extensions of the research presented in this thesis:

- Formalize the definition of domain-specific descriptors, such as geometric descriptors that can express various objects/structures commonly used in parallel scientific applications. The directory layer can be extended to address a broader range of application-/domain-specific semantics based on different requirements from different application domains. This extension will essentially make Seine applicable to a broader range of applications.

- Extend the Seine shared space to encapsulate numeric/mathematic operations to support

data interpolation, transformation, aggregation, etc.

- Further optimize the communication layer to support efficient and adaptive buffer management and data transfer on different architectures.

- Survey parallel scientific applications in various fields, summarize and appropriately classify them into categories based on common requirements, and propose a generic code coupling template for each category to provide desirable coupling supports to different types of applications.

References

[1] C.C. Goodrich, A.L. Sussman, J.G. Lyon, M.A. Shay, P.A. Cassak. The CISM code coupling strategy. Journal of Atmospheric and Solar-Terrestrial Physics 66 (2004) 1469-1479.

[2] The Message Passing Interface (MPI) Standard. http://www-unix.mcs.anl.gov/mpi/

[3] J.L. Banks, H.S. Beard, Y. Cao, A.E. Cho, W. Damm, R. Farid, A.K. Felts, T.A. Halgren, D.T. Mainz, J.R. Maple, R. Murphy, D.M. Philipp, M.P. Repasky, L.Y. Zhang, B.J. Berne, R.A. Friesner, E. Gallicchio, and R.M. Levy. Integrated Modeling Program, Applied Chemical Theory (IMPACT), J. Comp. Chem., 26, 1752-1780 (2005).

[4] H. Nymeyer, S. Gnanakaran, and A. E. Garcia. Atomic SImulations of Protein Folding, Using the Replica Exchange Algorithm. Methods in Enzymology. Vol. 383. page 119-149. (2004)

[5] R. Swendsen and J. Wang. Phys. Rev. Lett. 57, 2607 (1986)

[6] C. Geyer and E. Thompson. J. Am. Stat. Assoc. 90, 909 (1995)

[7] E. Marinari and G. Parisi. Europhys. Lett. 19, 451 (1992)

[8] K. Hukushima and K. Nemoto. J. Phys. Soc. Jpn. 65, 1604 (1996)

[9] K. Y. Sanbonmatsu and A. E. Garcia. Proteins 46, 225 (2002)

[10] Y. Sugita and Y. Okamoto. Replica-exchange molecular dynamics method for protein folding. Chemical Physics Letters 314 (1999) page141-151.

[11] A. K. Felts, Y. Harano, E. Gallicchio, and R. M. Levy. Free energy surfaces of β -Hairpin and α -Helical peptides generated by replica exchange molecuar dynamics with AGBNP implicit solvent model. Proteins: Stucture, Function, and Bioinformatics 56: 310-321 (2004).

[12] N. Carriero, D. Gelernter. Linda in context. Communications of the ACM, Volume 32, Issue 4, pp.444-458, April 1989, ISSN:0001-0782.

[13] Y. M. Rhee and V. S. Pande. Multiplexed-replica exchanged molecular dynamics method for protein folding simulation. Biophysical Journal. Volume 84. February 2003. Page 775-786.

[14] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: a message-passing parallel molecular dynamics implementation. Computer Physics Communications 91 (1995) 43-56.

[15] J. et al. JavaSpace Specification 1.0. Technical report, Sun Microsystems, 1998.

[16] T. J. Lehman, S. W. McLaughry, and P. Wycko. TSpaces: The Next Wave. In Proceedings of Hawaii International Conference on System Sciences, 1999.

[17] R. Tolksdorf and D. Glaubitz. CoordinatingWeb-based Systems with Documents in XMLSpaces. In Proceedings of the Sixth IFCIS International Conference on Cooperative Information Systems, 2001.

[18] A. Murphy, G. Picco, and G.-C. Roman. Lime: A Middleware for Physical and Logical Mobility. In Proceedings of the 21st International Conference on Distributed Computing Systems, pages 524C536, 2001.

[19] G. Cugola and G. Picco. PeerWare: Core Middleware Support for Peer-To-Peer and Mobile Systems. Technical report, Politecnico di Milano, 2001.

[20] N. Busi, C. Manfredini, A. Montresor, and G. Zavattaro. PeerSpaces: Data-driven Coordination in Peer-to-Peer Networks. In Proceedings of the 2003 ACM symposium on Applied computing, pages 380C386. ACM Press, 2003.

[21] Z. Li and M. Parashar. Comet: A Scalable Coordination Space in Decentralized Distributed Environments. In Proceedings of the 2nd International Workshop on Hot Topics in Peer-to-Peer Systems (HOT-P2P 2005), San Diego, CA, USA, IEEE Computer Society Press, July 2005.

[22] V. Bhat, S. Klasky, S. Atchley, M. Beck, D. McCune, and M. Parashar. High Performance Threaded Data Streaming for Large Scale Simulations. 5th IEEE/ACM International Workshop on Grid Computing, Pittsburgh, PA, USA, November, 2004.

[23] R. Zhou and B. J. Berne, Smart walking: A new method for Boltzmann sampling of protein conformations. J. Chem. Phys. 107 (1997) 9185-9196.

[24] Distributed Data Descriptor. http://www.cs.indiana.edu/ febertra/mxn/parallel-data/index.html

[25] F. Bertrand, Y. Yuan, K. Chiu, R. Bramley: ”An Approach to Parallel MxN Communication”, 2003, LACSI Conference, Sante Fe, NM.

[26] J.A. Kohl, G.A. Geist. Monitoring and steering of large-scale distributed simulations, In IASTED International Conference on Applied Modeling and Simulation, Cairns, Queensland, Australia, September 1999.

[27] K. Keahey, P. Fasel, S. Mniszewski. PAWS: Collective interactions and data transfers. In Proceedings of the High Performance Distributed Computing Conference, San Francisco, CA, August 2001.

[28] J.-Y. Lee and A. Sussman. High performance communication between parallel programs. In proceedings of 2005 Joint Workshop on High-Performance Grid Computing and High Level parallel Programming Models (HIPS-HPGC 2005). IEEE Computer Society Press. Apr. 2005. To appear with the Proceedings of IPDPS 2005.

[29] CCA Forum. http://www.cca-forum.org

[30] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. Divorcing language dependencies from a scientific software library. In Proceedings of the Eleventh SIAM Conference on Parallel Processing for Scientific Computing. SIAM, Mar. 2001.

[31] J.W. Larson, R.L. Jacob, I.T. Foster, and J. Guo. The Model Coupling Toolkit. In V.N.Alexandrov, J.J.Dongarra, B.A.Juliano, R.S.Renner, and C.J.K.Tan, editors, Proceedings of the International Conference on Computational Science (ICCS) 2001, volume 2073 of Lecture Notes in Computer Science, pages 185-194, Berlin, 2001. Springer-Verlag.

[32] S. Zhou. Coupling earth system models: An ESMF-CCA prototype. http://webserv.gsfc.nasa.gov/ESS/esmf_tasc/, 2003.

[33] F. Bertrand and R. Bramley. DCA: A distributed CCA framework based on MPI. In Proceedings of HIPS 2004, 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments, Santa Fe, NM, April 2004. IEEE Press.

[34] F. Bertrand, D. Bernholdt, R. Bramley, K. Damevski, J. Kohl, J. Larson, A. Sussman. Data Redistribution and Remote Method Invocation in Parallel Component Architectures. In Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS'05). April, 2005.

[35] K. Zhang, K. Damevski, V. Venkatachalapathy, and S. Parker. SCIRun2: A CCA framework for high perfomance computing. In Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004), Santa Fe, NM, April 2004. IEEE Press.

[36] L. Zhang, M. Parashar: A Dynamic Geometry-Based Shared Space Interaction Framework for Parallel Scientific Applications. High Performance Computing - HiPC 2004: 11th International Conference, Bangalore, India, December 19-22, 2004. Proceedings p.189-199.

[37] L.A. Drummond, J. Demmel, C.R. Mechoso, H. Robinson, K. Sklower, and J.A. Spahr. A data broker for distributed computing environments. In Proceedings of the International Conference on Computational Science, pages 31-40, 2001.

[38] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., M. Zosel. The High Performance Fortran Handbook. MIT Press, Cambridge, MA. 1994.

[39] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley. XCAT 2.0: A component-based programming model for grid web services. Technical Report TR562, Department of Computer Science, Indiana University, June 2002.

[40] Y. Chen and S.E. Parker. Phys. Plasmas 8, 2095 (2001).

[41] I. Manuilskiy and W.W. Lee. Phys. Plasmas 7, 1381 (2000).

[42] Q. Lu, M. Peszynska, and M.F. Wheeler. A parallel multi-block black-oil model in multi-model implementation. In 2001 SPE Reservoir Simulation Symposium, Houston, Texas, 2001. SPE 66359.

[43] H.D. Sterck, R.S. Markel, Pohl T, Rude U. A lightweight Java Taskspaces framework for scientific computing on computational grids. *The eighteenth annual ACM symposium on applied computing*, March 2003, Melbourne, Florida, USA. pp.1024-1030, 2003, ISBN:1-58113-624-2

[44] M. Parashar, I. Yotov. An Environment for Parallel Multi-Block, Multi-Resolution Reservoir Simulations. *Proceedings of the 11th International Conference on Parallel and Distributed Computing Systems* (PDCS 98), Chicago, IL, International Society for Computers and their Applications (ISCA), pp.230-235, September 1998.

[45] T. Bially. A class of dimension changing mapping and its application to bandwidth compression. PhD thesis, Polytechnic Institute of Brooklyn, June 1967.

[46] G. Breinholt. and C. Schierz. 1998. Algorithm 781: generating Hilbert's space-filling curve by recursion. ACM Transactions on Mathematical Software (TOMS), Vol 24, Issue 2 (June 1998), pp.184-189.

[47] TSpaces: intelligent connectionware. http://www.almaden.ibm.com/cs/TSpaces/

[48] M. Wheeler, T. Arbogast, S. Bryant, J. Eaton, Q. Lu, M. Peszynska, and I. Yotov. Parallel Multiblock/Multidomain Approach for Reservoir Simulation, SPE 51885, 15th SPE Reservoir Simulation Symposium, Houston, TX, pp.14-17, February. 1999.

[49] T. Arbogast, L.C. Cowsar, M.F. Wheeler, and I. Yotov. Mixed finite element methods on non-matching multiblock grids. SIAM Journal of Numerical Analysis, 37:1295-1315, 2000.

[50] J. Bear. Dynamics of Fluids in Porous Media. Elsevier. New York, 1972.

[51] D.W. Peaceman. Fundamentals of Numerical Reservoir Simulation. Elsevier. New York, 1977.

[52] Message Passing Interface Forum. MPI: A message-passing interface standard. International Journal of Supercomputer Applications, 8(3/4), 1994. Special issue on MPI.

[53] D. Walker. Standards for message passing in a distributed memory environment. Technical Report TM-12147, Oak Ridge National Laboratory, August 1992.

[54] The OpenMP specification. http://www.openmp.org/specs/

[55] An OpenMP tutorial. http://www.llnl.gov/computing/tutorials/openMP/

[56] H. Abbasi, M. Wolf, K. Schwan, G. Eisenhauer, and A. Hilton. XChange: Coupling parallel applications in a dynamic environment. In IEEE International Conference on Cluster Computing, September 2004.

[57] K. Keahey and D. Gannon. PARDIS: CORBA-based architecture for application-level parallel distributed computation. In Proceedings of the 1997 International Conference on Supercomputing, San Jose, CA, November 1997.

[58] Object Management Group. CORBA component model. http://www.omg.org/technology/documents/formal/components.htm, 2002.

[59] Microsoft Corporation. Distributed Component Object Model.

[60] Sun Microsystems. Enterprise JavaBeans downloads and specifications. http://java.sun.com/products/ejb/docs.html, 2004

[61] The RedGRID Parallel Data Redistribution Library. http://www.labri.fr/perso/esnard/RedGRID/

Curriculum Vitae

Li Zhang

2006 Ph.D. in Computer Engineering; Rutgers University, NJ, USA.

2001 MS in Information Engineering; Beijing University of Posts & Telecoms, Beijing, China.

1998 BS in Telecommunication Engineering; Nanjing University of Posts & Telecoms, Nanjing, China.

2006 Graduate Assistant, The Applied Software System Laboratory, Rutgers University, NJ, U. S .A.

2001-2006 Teaching Assistant, Electrical and Computer Engineering Department, Rutgers University, NJ, U. S. A.

1998-2001 Research Assistant, Center for Information Processing and Artificial Intelligence, Beijing University of Posts & Telecoms, Beijing, China.

Publications

L. Zhang and M. Parashar. A Dynamic Geometry-based Shared Space Interaction Framework for Parallel Scientific Applications. In the Proceedings of the 11th Annual International Conference on High Performance Computing (HiPC 2004). Bangalore, India. Vol 3296, Page 189-199

L. Zhang and M. Parashar. Shared Memory Multiprocessors. Encyclopedia of Computer Science and Engineering. Editor: B. Wah, John Wiley and Sons, Inc, 2005.

L. Zhang and M. Parashar. Seine: A Dynamic Geometry-based Shared Space Interaction Framework for Parallel Scientific Applications. Concurrency and Computations: Practice and Experience. John Wiley and Sons. 2006.

L. Zhang and M. Parashar. Enabling Efficient and Flexible Coupling of Parallel Scientific Applications. In the Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006). Rhodes Island, Greece. IEEE Computer Society Press.

L. Zhang, M. Parashar, E. Gallicchio and R.M. Levy. Salsa: Scalable Asynchronous Replica Exchange for Parallel Molecular Dynamics Applications. In the proceedings of the 2006 International Conference on Parallel Processing (ICPP-06). August 14-18, 2006. Columbus, Ohio, USA.

L. Zhang and M. Parashar. Experiment with Wide Area Data Coupling Using the Seine Framework. Submitted to the 13th Annual International Conference on High Performance Computing (HiPC 2006). Bangalore, India. Dec. 2006.