Adaptive Grid Computing

Dissertation Presented

by

Juemin Zhang

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the field of

Computer Engineering

Northeastern University

Boston, Massachusetts

Contents

1 Introduction **1**

1.1 Message passing model and MPI 6

1.2 Resource management system 8

1.3 Grid computing 11

1.3.1 Computing grid 11

1.3.2 Characteristics of a computing grid 13

1.4 Workflow execution 15

1.5 Problem statement 18

1.5.1 Assumptions 20

1.6 Related work 21

1.6.1 Grid computing infrastructure 22

1.6.2 Globus-enabled MPI approaches 24

1.6.3 Resource management of computing grids 26

2 Resource Allocation Model **30**

2.1 Model description 30

2.2 Functional specification 34

2.2.1 Site selection . . . 34
2.2.2 Job management . . . 36
2.2.3 Resource assessment . . . 37
2.2.4 Resource binding . . . 39
2.2.5 Resource synchronization . . . 41
2.3 Model simulation . . . 43
2.3.1 Workload generation . . . 44
2.3.2 Simulation of a computing site . . . 46
2.3.3 Global job scheduler . . . 47
2.4 Simulation result analysis . . . 52
2.4.1 Centralized and distributed comparison . . . 52
2.4.2 Distributed scheduling comparison . . . 54
2.4.3 Job duplication . . . 56
2.4.4 Resource co-allocation using the stop-and-wait method . . . 59
2.4.5 Resource co-allocation using hibernation . . . 61
2.4.6 Simulation of heterogeneous grid . . . 63
3 Framework Design **65**
3.1 Workflow scheduler . . . 66
3.1.1 Petri net . . . 67
3.1.2 Scheduler design . . . 69
3.1.3 Parallelization of the ranking procedure . . . 73
3.2 Task grouping . . . 76
3.2.1 Task binding procedure . . . 79

3.2.2 Task coordination 82

3.3 Message Relay 83

3.3.1 Inter-site communication analysis 84

3.3.2 3-hop message relay 87

3.4 Framework integration 89

3.5 SGR framework implementation 93

4 Motivating Application: Tomosynthesis Mammography 98

4.1 Introduction to Tomosynthesis Mammography 99

4.2 Parallelization of Tomosynthesis 101

4.2.1 Non-intercommunication 106

4.2.2 Overlap with intercommunication 107

4.2.3 Non-overlap 108

4.3 Parallelization performance comparison 109

4.4 Workflow execution of Tomosynthesis 113

5 Experimental Evaluation of the SGR system 117

5.1 Communication experiments 118

5.1.1 Computing the communication lower bound 120

5.1.2 Point-to-point communication experiments 122

5.1.3 Collective communication experiments 125

5.1.3.1 Cross-site MPI barrier 125

5.1.3.2 Cross-site MPI broadcast 126

5.1.3.3 Cross-site MPI scatter 128

5.1.3.4 Cross-site MPI gather 129

5.1.3.5 Cross-site MPI allgather 131
5.1.3.6 Cross-site MPI alltoall 134
5.1.3.7 Cross-site ring test 136
5.1.3.8 Summary of communication tests 137
5.2 SGR synthetic workload experiments 138
5.2.1 Comparing experiment and simulation 141
5.3 Experiments using Tomosynthesis 143
5.3.1 Speedup estimation 143
5.3.2 Experiments on the Joulian-Keys grid 147
6 Summary 150

List of Tables

3.1 The binding procedure performed on the client and server side 80

3.2 Comparisons of cross-site communication methods 84

4.1 Processor and interconnection network specifications of testing platforms. 110

5.1 Cluster and network configuration of the testing environment 118

List of Figures

1.1 The message passing computing model 6

1.2 The physical structure of a computing grid 12

1.3 The multi-layer structure of a computing grid 13

2.1 Comparison of centralized and distributed schedulers 53

2.2 Normalized average queuing times of global jobs scheduled by distributed schedulers using different ranking criteria. 55

2.3 Normalized average queuing times of global jobs using a range of duplication factors 56

2.4 Job duplication's impact on local jobs' queuing times. 58

2.5 The average co-allocation time normalized to the Ideal Min using stop-and-wait synchronization. 60

2.6 The average co-allocation time using stop-and-wait synchronization. . 61

2.7 Reduction in average queuing time for all jobs when compared to stop-and-wait with no duplication. 62

2.8 The average co-allocation time normalized to the Ideal Min, while using hibernation. 63

2.9 The reduction on average global job queuing time by using 1 duplication on heterogeneous grids. 64

3.1 Syntax symbols of Petri nets. 67
3.2 Examples of the three fundamental structures of Petri nets. 68
3.3 A Petri net description of the Tomosynthesis workflow application . . 68
3.4 The function diagram of the workflow scheduler 69
3.5 The overview of launching a workflow application using the SGR framework scheduler 72
3.6 The parallelization of the ranking procedure. 74
3.7 A single application task's execution specified in a Petri net. 75
3.8 The task scheduling workflow for an application task. 75
3.9 The parallel client-server approach used in task grouping. 78
3.10 Illustration of 3-hop message relay 87
3.11 The framework integration of three functional components in the client-server structure. 90
3.12 The deployment of the framework with workflow scheduling, task grouping, and message relay on a multi-site computing grid. 91
3.13 The structure of the integrated framework and its execution environment 92
3.14 The structure of the SGR framework implementation 94

4.1 Tomosynthesis image reconstruction algorithm. 101
4.2 Data dependencies among volxels of the 3D volume and the corresponding pixel on the detector panel. 103
4.3 Image acquisition while X-ray rotates on the pivot point 104

4.4 3D volume segmentation method viewed at the Y-axial direction. . . 104

4.5 The extended region of a segment. 105

4.6 The non-intercommunication parallelization method 106

4.7 The overlap with intercommunication parallelization method. 107

4.8 The non-overlap parallelization method 108

4.9 Performance of three implementations on number of processors, using UIUC NCSA's Titan cluster. 111

4.10 Profiling result comparison of three parallelization methods tested on 32 process on the NCSA Titan cluster at UIUC. 112

4.11 System comparison using the non-overlap parallelization running on 32 processors. 113

4.12 Performance test of three parallelization methods using 32 nodes on five platforms. 114

4.13 Illustration of the three Tomosynthesis tasks executed in a workflow . 115

5.1 Evaluation of the communication latency of the cross-site point-to-point communication. 122

5.2 Estimation of the SGR-introduced service overhead 123

5.3 Throughput of the cross-site point-to-point communication operation 125

5.4 Latency of the cross-site MPI barrier operation 126

5.5 Latency of cross-site broadcasting among 16 processes over the two-cluster grid 127

5.6 Latency of cross-site broadcasting 256 KB message 128

5.7 Latency of cross-site MPI scatter among 16 processes 129

5.8 Latency of cross-site MPI scatter of 256KB message blocks 130
5.9 Latency of cross-site MPI gather among 16 processes 131
5.10 Latency of cross-site MPI gather of 256KB message blocks 131
5.11 Latency of cross-site MPI allgather among 16 processes 132
5.12 Latency of cross-site MPI allgather of 256KB message blocks 133
5.13 Proportion of cross-site communication in the overall cross-site MPI allgather among 16 processes 134
5.14 Throughput of the cross-site MPI alltoall operation 135
5.15 Latency of cross-site MPI alltoall on 16 processes 136
5.16 Time to complete the cross-site ring communication 136
5.17 Average queuing time for global jobs submitted to the 2-cluster grid . 139
5.18 Results of single-site executions of the parallel Tomosynthesis image reconstruction using 8 processes on Joulian and Keys 147
5.19 Results of cross-site executions of the parallel Tomosynthesis image reconstruction using 16 processes on the Joulian-Keys grid 147

Abstract

Many innovative scientific applications rely on high-performance computing to perform large computations. Tomosynthesis Mammography, which performs high-resolution image reconstruction, is one such computation-intensive application. Currently, it is difficult to launch MPI applications on multiple distributed and heterogeneous computing platforms, or computing grids, since application developers and users must address problems such as application scheduling, resource allocation and co-allocation, and inter-process communication. Our objective is to provide location-, topology-, and administrative-transparent grid computing for MPI applications, while hiding the physical details of computing platforms, and heterogeneous networks from the application developers and users.

In this dissertation, we introduced a resource allocation model, workflow structures to specify MPI applications involving multiple tasks, and message relay to enable communication across different networks. We developed the SGR framework, which integrates workflow scheduling, task grouping, and message relay services, while hiding resource allocation, heterogeneous networks, and decentralized resource management systems from application developers and users. The SGR system has been implemented on a Globus-enabled computing grid.

To investigate the effectiveness of our resource allocation model and framework design, we created a simulation environment for a computing grid and task schedulers. The simulation results show that the new dynamic task duplication approach can allow simple task scheduling algorithms to achieve performance similar to what

would be achieved using a more sophisticated scheduling algorithm with accurate predictions of queuing times and the job preemption technique. Over 40% performance improvement is obtained by simple task schedulers using two duplicated requests.

We tested our SGR framework by conducting detailed experiments on a two-cluster grid, and observed that duplication can improve performance by more than 15%. These results validate our model. Moreover, we experimentally evaluated our new message relay service for cross-site message passing. The test results indicate that although the SGR's message relay service has some communication overhead, the system is scalable with respect to the number of processes and the message size.

Chapter 1

Introduction

Computational applications rely on high-performance computing. Computers are becoming faster, closely following Moore's law [38]. Parallel and distributed computing has made it possible to aggregate computing power from numerous computing sources. However, these advances cannot match continuously increasing computational demands.

Scientific computations frequently have non-polynomial complexity and process large quantities of data that can easily exceed the computing capacity of modern computers. Emerging applications, such as high resolution protein structure prediction [11, 45] used in protein folding, and molecular dynamic simulation [43] used in computational physics, continuously demand faster processing capability. However, a single parallel system, such as a cluster, has limited resources, and may not be capable of providing sufficient computing power for those computation-intensive applications. One solution to this problem is to collect and utilize distributed computing resources that are owned and maintained by different institutions, labs or computing centers.

Grid computing [21] has emerged as a new approach to transparently harness distributed computing power to meet the demand of computation-intensive applications. When computing resources are distributed across multiple organizations, such a computing environment is expected to hide institutional and system-level differences from application users and developers as much as possible. Therefore, an application's portability would not increase software complexity and implementation efforts when it targets computing grids. A computing grid is constructed on multiple parallel computing systems, each of which can be in a different administrative domain and separated geographically. As a result, a major objective of grid computing is to provide location-, topology- and administration-transparent computational services for users and computational applications. Similar to the electric power grid, a computing grid is designed to provide computing power for parallel and distributed applications, while hiding the location where the power is generated.

Computing grids are characterized as loosely coupled systems, because of heterogeneity of platforms and networks, geographically distributed resources, and the autonomy of each participant. In contrast to traditional parallel and distributed computing, grid computing can employ multiple private networks and parallel computing systems that are owned by different institutions. Heterogeneous networks can use a range of network protocols, and may differ in communication bandwidth. Individual administration and management of computing systems may not be able to converge to one centralized resource management.

Globus [20] and Legion [26] are the first software infrastructures to support grid computing. These platforms provide security, inter-operability, authentication and authorization. However, many challenges remain to implement truly transparent par-

allel and distributed computation that supports heterogeneous resources, and allows them to be used in an efficient and secured manner. Many applications experience difficulties in adapting to computing grids. In this study, we targeted message passing interface (MPI) [2] applications and focused on how to execute MPI applications on computing grids, while hiding the heterogeneity of the computing environment.

In the field of parallel and distributed computing, *degree of parallelism* is defined as the number of a parallel program's processes that can be executed concurrently. The granularity is used to describe the computation workload that is executed between two inter-process communication operations. The unit of granularity (or grain size) can be the execution time, the number of instructions, or the number of floating-point operations. Therefore, *fine-grained* parallelization can be defined as the parallelization that has a small granularity, while *coarse-grained* parallelization has a large granularity [30]. While MPI can be used for both fine-grained and coarse-grained parallelization, many applications use the coarse-grained approach to execute on computing grids because of the large inter-site communication overhead of grids.

On a Globus-enabled computing grid, source code modifications or system-level alterations may be required to run legacy MPI programs, since heterogeneous networks and resource allocation across multiple resource administrative domains are not transparent for application developers and users. Our objective is to enable network topology-transparent execution for MPI applications, including legacy programs, to run on a computing grid, while hiding decentralized resource management and resource allocation process from applications.

To achieve this objective, we extended the conventional message passing executions to workflows. Targeting workflow applications, we specified an adaptive grid

computing framework, consisting of task scheduling, task grouping, and message relay services. This framework, which is based on the client-server paradigm, includes application-independent function specifications on the server side and an MPI-compatible application programming interface on the client side. The task scheduling service is designed to submit tasks of a multi-task application to a computing grid, while trying to minimize each task's resource queuing time. The task grouping service allows running tasks to exchange information about each running task's allocated resources and to perform dynamic resource selection. The message relay service enables intra- and inter-cluster communication between workflow tasks.

We introduced dynamic resource allocation in the adaptive grid computing framework which is based on multiple duplicated submissions of an application task, targeting computing grids. Based on this approach, we can locate the earliest resources available for the task, and allow the scheduler to dynamically adapt to changes in the availability of resources, which usually cannot be accurately predicted at the task submission time. We used a new modeling environment to investigate scheduling algorithms with and without dynamic resource allocation. Our modeling results demonstrate that when dynamic resource allocation is used, both simple and sophisticated scheduling algorithms yield similar performance [1].

We developed the SGR system, a portable implementation of the adaptive grid computing framework. The system is based on Globus-enabled computing grids and allows legacy MPI programs to execute without source code modification. We used synthetic workloads to verify our framework design and to evaluate performance of

[1] In this dissertation, the *performance* of a scheduler is defined as the average queuing time of a list of job requests which are scheduled by the scheduler to a computing grid.

the SGR system in a two-cluster grid environment. Experimental results demonstrate distinct advantages of the framework, validating our finding in the modeling analysis.

This thesis makes the following contributions. First, we define the workflow structures for specifying the execution process of a multi-task MPI workflow application running on a computing grid, while hiding locations of computing resources of the computing grid. Second, based on the workflow structure, we develop a framework integrating task scheduling, grouping and message relay services. Third, building on this framework, we implement a portable SGR system which allows multi-task workflow applications and conventional MPI programs to execute on a Globus-enabled computing grid. Fourth, we demonstrate the SGR system is capable of delivering better performance in task scheduling by using task duplication, and message relay can be efficient and scalable with respect to the size of messages and number of processes. Overall, we introduce a new resource allocation model, message relay and the framework integration to address several challenges of grid computing for MPI applications. Using the SGR system, we can achieve location-, topology- and administrative-transparent grid computing for conventional MPI programs and those MPI applications that involve complex, computation-intensive and multi-task workflows.

In this thesis, we first give an introduction to parallel and distributed computing, and define our problems. We then review research related to our work. Chapter 2 presents a new dynamic execution model for MPI applications. Based on this model, we describe a framework targeting legacy MPI applications and MPI workflow applications. To evaluate our model and the framework design, we use a computing grid simulation environment to assess the model and schedulers. We present the implemen-

tation of our framework and the SGR system in Chapter 3. Chapter 4 describes our motivating application, Tomosynthesis Mammography, and its parallelization work. Finally, we discuss experimental results for the SGR system and the Tomosynthesis application on a two-cluster grid.

1.1 Message passing model and MPI

Many parallel computing models have been studied for a specific application domain using a specific problem-solving approach. Some applications are parallelized at the instruction level by using the single instruction multiple data (SIMD) model [19]; some are parallelized at the programming level by using the client-server, thread, or message passing model. The message passing model has been successfully applied to many parallel applications due to its simplicity and applicability. This model consists of context-independent and cooperating processes. Each process sends data to, or receives data from, one another in the form of messages, as shown in Figure 1.1. Application developers can adopt this model by partitioning a given problem and assigning partitions to multiple processes, while allowing communication among all processes.

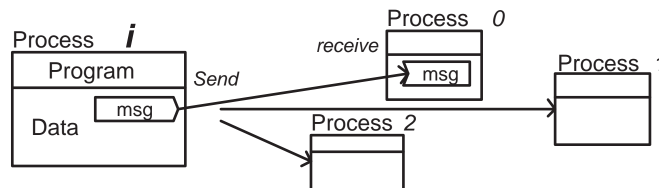

Figure 1.1: The message passing computing model

Message passing interface (MPI) [3], which is defined by a public consortium, has become the de facto industrial standard in parallel and distributed computing. Based on the message passing model, MPI specifies hardware-transparent functions and interface for inter-process communication and logical execution topology manipulation. The high-level description of MPI hides physical network and system-level details from applications. Therefore, application developers can focus on the problem-solving process, and MPI-compliant programs are portable to a wide range of diverse platforms.

In this dissertation, we define a *computing resource* as the computing hardware used by a single MPI process. An MPI program can be executed on multiple computers, and each computer is assigned with one process. In this case, computer resources used in the execution are considered as computers. In different scenarios, computer resources can be represented by CPUs, processors or computing nodes of a cluster.

The MPI standard is designed to support applications that run on multiple interconnected computing resources. Processes of an MPI execution running on multiple computing resources can send messages to and receive messages from one another. Since MPI is platform- and hardware-independent, it relies on two presumptions: computing resources are available, and networks are fully connected at the network layer (of the 5-layer OSI model).

An MPI execution can have more than one process and thus requires multiple computing resources. Before starting a parallel execution, all computing resources required by the MPI execution must be allocated and available to use. The process of acquiring the application-specified number of computing resources for the MPI execution is referred to as *resource allocation*.

Conventionally, resource allocation has not been considered in the message passing model, the MPI specification, or MPI implementations. Instead, it is the application user's responsibility to make sure that requested resources are available for executions. All allocated resources are assumed to remain available throughout an MPI parallel execution.

Complying with the message passing model and the MPI standard, MPI implementations must support communication between any pair of processes. MPI implementations, including open-source OpenMP [12], MPICH [27], and vendor-provided MPI libraries, rely on the underlying communication network layer of their host systems. In a homogeneous environment, such as a cluster or a shared-memory system, the communication topology is fully connected at the application layer, which is provided either by direct physical links or via the IP network layer. Typically, an MPI implementation assumes that its host system provides a fully-connected communication topology among all processes. However, if the network layer imposes restrictions on connections between some computing resources, e.g., in heterogeneous networks, then it would require that the application layer participate in routing messages.

1.2 Resource management system

In a resource-sharing environment, such as a cluster or a mainframe computer that has a large number of computing resources, a privileged software system is designated to manage how these resources are used. Such a system is called *resource management system*(RMS), also known as job queuing system or batch job system. To run an application in a resource-sharing environment, application users submit job requests

to the resource management system. A job request specifies the resource requirement, the application execution code, and input and output files. In general, a *resource requirement* describes all resources required by the application, such as the number of computing resources, memory size, disk space and network bandwidth. However, we assume that the resource requirement only specifies the number of computing resources required by the application.

A resource management system determines how and which resources are assigned to meet resource requirements of all job requests, while trying to balance workloads and maximize the system throughput. Since computing resources inevitably cannot satisfy all the resource requirements at the same time, the resource management system is responsible for resolving resource request conflicts by queuing all job requests and properly allocating resources for each of them. PBS [28], LoadLeveler [31] and LSF [52] are among the most commonly deployed resource management systems in a resource-sharing environment.

A typical resource management system is composed of three major function modules: a resource monitor, a job queue, and a job scheduler. The resource monitor collects information about workloads on all managed computing resources and the status of all running jobs. This information is used to balance workloads during job scheduling. After the system receives a job request with its resource requirement, the job request is put into the job queue. Within the queue, all queued requests are processed based on a queuing policy, and the most commonly used policy is known as first-come-first-serve (FCFS). For a given job request, the scheduler locates resources by comparing the job's resource requirement with the current availability of resources. If the requirement is met, the job is scheduled to launch on the allocated resources.

Otherwise the job request is kept in the job queue.

When the scheduler is unable to schedule the job request with the earliest arrival time in the job queue due to the lack of sufficient resources, the FCFS queuing policy will stop scheduling and wait for more resources to become available. In contrast, backfill policies, such as backfill EASY [18] and backfill conservative [39], allow out-of-order scheduling. The backfill EASY and the backfill conservative policies, which are based on FCFS, schedule jobs that arrive after unscheduled jobs. Given that each job request specifies the maximum execution time, the estimated start time of each queued job can be calculated by the scheduler. The backfill conservative policy guarantees that scheduling jobs that are out-of-order does not delay the estimated start time of unscheduled jobs that arrived earlier. The backfill EASY policy only guarantees that the unscheduled job having the earliest arrival time will not be delayed.

Besides FCFS and backfill, there are many other queuing policies used for prioritized jobs, and various techniques to balance workloads, such as job migration and job preemption. Job migration stops a running job and resumes its execution on different computing resources while keeping the same execution context (such as register values). Job preemption replaces a running job with a different job on the same resources, and resumes the execution of the replaced one later. Although these techniques enhance management capability and can balance workloads dynamically, they lead to higher system complexity, more system overhead, and less adaptability for jobs. To support these techniques, applications must have checkpoints inserted within their execution code, and such a feature is only allowed by some specific applications.

1.3 Grid computing

In the last few decades, new technologies have made computer systems faster and more affordable. A growing number of labs, departments, and institutions have installed high-performance parallel systems, such as clusters, for individual research purposes. Meanwhile, communication bandwidths over metropolitan area networks (MANs) and wide area networks (WANs) have been increasing dramatically.

As technologies advance, applications' computational demands also increase. Many parallel and distributed applications must be deployed on more computers as users pursue higher performance to solve larger problems. Considering that a single computing system has limited computing resources, one way to satisfy the increasing computational demands is to utilize abundant computing resources across multiple distributed computing systems.

When computing resources are owned and managed by different institutions that are geographically separated from one another, such an environment differs from the conventional parallel and distributed computing environment, creating new challenges in software portability and compatibility, resource availability and accessibility, and security and authorization. These challenges are addressed in a newly emerging field called grid computing [22].

1.3.1 Computing grid

Grid computing introduces large-scale resource sharing among dynamic, multi-institution organizations. A computing grid is constructed on multiple systems, each of which is assumed to be a conventional parallel or distributed system that performs its own

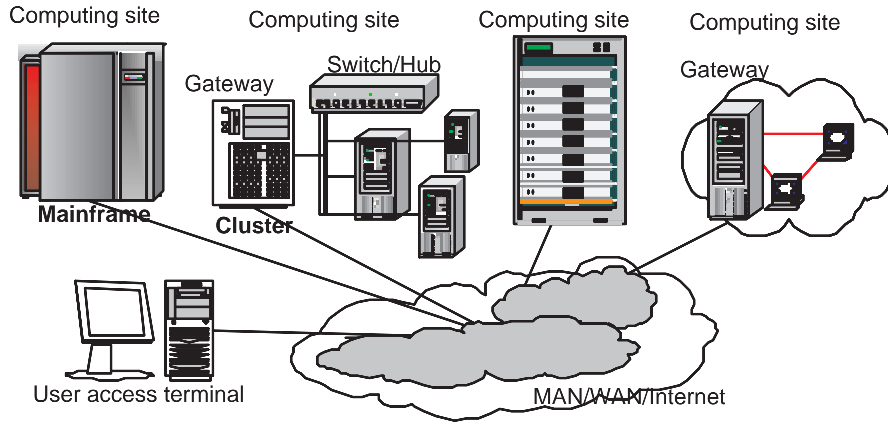

Figure 1.2: The physical structure of a computing grid

resource management. Systems participating in a computing grid could be mainframe computers, workstations, or clusters. As shown in Figure 1.2, components of a computing grid could be separated geographically, and interconnected by a public network. Administration of each participating system remains highly autonomous. Unlike conventional parallel and distributed systems, there is no assumption of a centralized administration over all participating systems in a computing grid environment. An important objective of grid computing is to enable topology- and administration-transparent computational services for parallel and distributed applications.

In this thesis, we use the following definition of a computing site. A *Computing site* is a collection of computing resources which are connected by a private or local area network within a single administrative domain. A single *administrative domain* means that all computing resources within a computing site are managed by one resource management system, such as OpenPBS, Loadleveler or LSF. For example, a cluster with an OpenPBS system installed is considered as a computing site. Based on

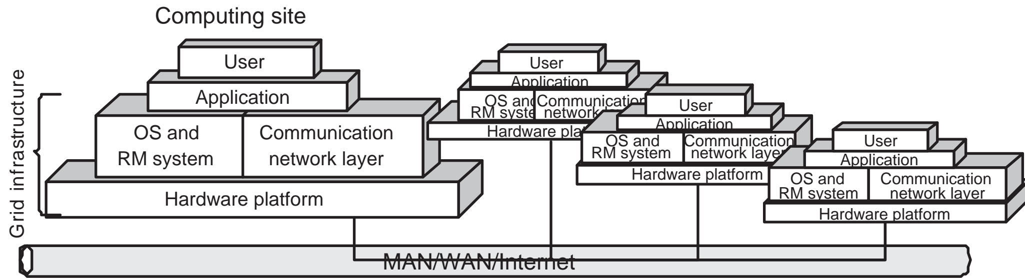

Figure 1.3: The multi-layer structure of a computing grid

the definition of computing site, *computing grid* can be defined as multiple computing sites interconnected by a public network, and software infrastructure enabling resource sharing among these computing sites.

1.3.2 Characteristics of a computing grid

To pursue the objective of topology- and administration-transparent computing, grid computing requires collaboration at the hardware, system, and application levels, as shown in Figure 1.3. A computing grid is constructed on multiple computing sites; however, it also relies on infrastructural system software to enable secured resource sharing over a public network and across different administrative domains in an efficient and fair manner. As for applications, they need to adapt to various computing environments, resource management systems, and heterogeneous networks.

At the hardware level, grid computing involves multiple computing sites' participation in a loosely-coupled fashion, since participating sites are owned by different institutions that may not consent to centralized management. Even if all sites can reach an agreement, the centralized system may not be scalable or capable of managing the massively distributed resources of a computing grid. Furthermore, continuous

availability of every site is difficult to achieve. A computing site may reserve some computing time for its local users and temporarily withdraw from the grid. Components of a computing grid may be changeable, and this characteristic is different from conventional parallel and distributed computing which has dedicated computing resources. Therefore, grid computing is subject to lower resource reliability than conventional parallel computing.

Grid computing consists of hierarchical and heterogeneous networks, while conventional parallel and distributed computing uses homogeneous networks. A computing grid is built on a public network to interconnect all its participating sites, and private networks to connect computing resources within each site. The external public network, such as the Internet, is not as reliable as internal private networks or local area networks (LANs). In addition, the communication bandwidth and latency of the external network are worse than those of LANs.

At the system level, grid computing demands a higher level of security, authorization and authentication, as well as requiring software to enable resource sharing across the boundary of administrative domains in a secured and efficient manner. Since inter-site communication is exposed on a public network, strong data encryption must be used. Resource sharing across multiple resource management systems may demand new queuing policies. Furthermore, information such as workloads, resource utilization, and workload status of computing sites must be available to all participating sites and users. This information plays an important role in balancing workloads among computing sites.

Given that a computing grid may be composed of different types of processors and operating systems, applications must be more portable. For example, applica-

tions must handle architectural endianness correctly if they will execute on X86 and RISC architectures. The differences in processors' performance could vary over a larger range in a computing grid than within a single computing site, making it more difficult for application developers and users to adapt to such an environment. Furthermore, since intra-site and inter-site communication may have different bandwidth and latency, which is uncommon within a parallel computing platform, application developers and users may also need to balance communication among processes to optimize performance. The heterogeneity of grid systems presents more difficulties for application developers than homogeneous systems.

Since a computing grid can provide much more computing resources than a single computing site, a grid usually serves as a computing platform to solve more computation-intensive and complex applications. Such applications may need more computing resources that may exceed a single computing site's capacity, involve specific computing resources from different computing sites, or consist of multiple parallelized procedures with different degrees of parallelism. These applications could be different from conventional, single program applications. The execution of such an application could involve multiple parallel executions that are carried out sequentially or concurrently. Therefore, we need a model to represent the above application. In the next section, we will describe this model.

1.4 Workflow execution

As we discussed in the previous section, the MP model and the MPI standard assume that a parallel execution runs on computing resources which are allocated throughout

the execution. Many time-consuming parallel applications demand a large amount of computation, which may require tens or hundreds of processors for hours. An MPI parallel program may be divided into sequential and parallel segments, or the program's parallelism can vary. If one segment of the parallel program executes sequentially on one processor, then the remaining computing resources will be unused. To improve efficiency and utilization, we need to separate sequential and parallel segments of a program (or segments with different degrees of parallelism), and execute each of them only using the resources they require. Although this process introduces additional overhead, the benefits from improvements in efficiency will overcome the costs when it is applied in large and computation-intensive applications.

Furthermore, some parallel applications employ multiple parallel executions, each of which requires a different number of computing resources. These executions are specified in sequence or in parallel, exhibiting dependency or concurrency. Such applications are no longer single-execution parallel applications, and we therefore need a model to represent them. In this thesis, we use the workflow to specify how a multi-task MPI application is executed.

A *workflow*, which contains a sequence of related activities or operations, represent how these activities are scheduled within a simple flow chart. The concept of a workflow is commonly used in the field of operations research. A workflow application contains a list of tasks that are executed with specified dependence concurrence among tasks. In this study, we define *task* as an execution of a specified program, which can be either a serial execution on one computing resource or a parallel execution using multiple computing resources within a computing site. Accordingly, a task is only allowed to use computing resources within one administrative domain. The workflow

specifies the execution procedure of all the tasks, and interactions among them as well.

In this thesis, we use three fundamental structures in a workflow: dependent, parallel forking, and optional forking structures. The dependent structure contains dependent tasks that are executed sequentially and one-by-one in a flow. One task that depends on other tasks is executed only after its depending tasks complete. Tasks in a parallel forking structure are concurrent tasks. The parallel forking structure divides a flow into two or more flows, and each flow is allowed to have a task executed concurrently with other flows. The optional forking structure divides a flow into two or more candidate flows, and only one of the candidate flows that satisfy the user-specified condition will be executed at runtime. Tasks within the candidate flows are called optional tasks.

The workflow structures give application users the capability to manage how a complex multi-task application is executed. While application developers only focus on individual tasks of the application, application users are allowed to manipulate the sequence of tasks' executions to achieve different objectives. In Section 4.4, we will demonstrate that our motivating application requires multiple image processing procedures to complete a single 3D reconstruction

Using workflow to represent applications gives us several benefits. First, it can reduce the complexity of application development by decomposing a complex problem into multiple smaller tasks. Second, it can improve resource utilization and efficiency when applications exhibit different degrees of parallelism. Finally, the workflow structure can be applied to many applications, while requiring little programming effort.

1.5 Problem statement

Our objective is to provide location-, topology- and administrative-transparent grid computing for MPI applications. Location-transparency means that application developers and users do not need complete knowledge about available computing resources and their physical locations. Topology-transparency means that the network heterogeneity of a computing grid is hidden from application developers and users; applications can assume that allocated computing resources are fully connected at the network layer, and therefore, processes can communicate with one another. Administrative transparency means that application developers and users do not have to deal with various resource management systems for job submission, resource allocation, or resource coordination. To achieve these objectives for MPI applications, we must overcome three major challenges: application scheduling, resource allocation and co-allocation, and cross-site message passing.

The application scheduler performs site selection and application job submission in a computing grid environment. Application users do not participate in the resource allocation process. This is similar to how an MPI program is launched in a cluster environment. Besides providing services to interact with host resource management systems, the application scheduler is responsible for balancing workloads among computing sites. Given a job request, the queuing time is the amount of time for the job request to wait for its requested resources in a job queue before it can be executed. Given multiple resource management systems and their job queues, it is important to determine which computing site will yield the shortest queuing time for a given job request. In general, computing sites that are heavily loaded with queuing jobs should

be avoided. However, queuing times is difficult to predict, making it hard to develop effective site selection algorithms.

The second challenge is how to allocate resources for a cross-site execution, and more specifically, resource co-allocation on a computing grid. We define *cross-site execution* (or two-site execution) as a parallel execution that uses computing resources from two different computing sites. When resources from two computing sites are *synchronized*, all the resources are available and can be used for a cross-site execution. Therefore, *resource co-allocation* is the process of allocating and synchronizing computing resources from two computing sites so that the resources can be used to perform a cross-site execution.

To conduct resource co-allocation, two questions must be addressed: which sites should be selected in a cross-site execution and how resources from different sites are coordinated. Since there is no centralized resource management system within a computing grid, site selection, resource allocation, and resource coordination cannot be performed in the conventional way.

The third challenge is how to enable inter-site communication. In a grid environment, seamless communication across different computing sites for MPI applications may not be feasible at the network layer. Considering that each computing site is constructed on its own private network, data communication across different networks requires network reconfiguration by either adding ports or changing routing tables. In addition, communication traffic between external and internal computers of a computing site is routed through a common gateway node or an access point. Many computing sites adopt a single access point which allows monitoring network traffic and filtering suspicious or virus-contaminated data packages. Inter-site com-

munication data must be routed through the access points explicitly. However, communication re-routing should be transparent for application developers. Therefore, we must provide network-transparent and adaptive cross-site message passing to enable conventional MPI applications to execute on multiple computing sites.

1.5.1 Assumptions

Since we are targeting MPI-based applications and pursuing a portable solution for existing computing platforms, we make the following assumptions. First, we assume that users of a grid environment have no higher privileges than a common local user of any computing site. A job request submitted by a computing grid user is not entitled to a higher priority in a job queue, nor can it alter the queuing order. In addition, a computing site is assumed to have one job queue, and job priorities are not considered. These assumptions allow us to focus on the most basic resource-sharing environment. Although a computing site, such as a cluster, usually has multiple job queues and job priorities, its behavior can be modeled by a single job queue. Our simplification reduces the complexity of our analysis and can help us find a general and effective solution.

Second, some advanced resource management features, such as job preemption, resource reservation, or advance reservation are not considered. These features are designed to improve resource utilization and enable system interoperability. However, they are not supported or available on many conventional computing systems, and some of these features require privileged user permissions. Furthermore, various constraints imposed on applications and system-level modifications to support these

features could limit our solution's portability. Therefore, we exclude these features from our work.

Finally, we assume that each participating computing site within a grid is highly autonomous. The resource management system of each computing site is operated on its own, and resource coordination across administrative boundaries is unavailable. Thus, we avoid any hardware and network reconfiguration to deploy our solution software; neither extra network ports nor modifications on network routing tables are needed.

1.6 Related work

Many researchers have worked to develop software to enable and facilitate grid computing. These software tools provide a range of services, including interoperability among different systems, data transfer, authentication and authorization, data encryption and secured communication, and meta-scheduling. Since MPI is the de facto standard in parallel programming, a grid-enabled MPI implementation allows scientific applications to use massive, diverse and distributed computing resources on grids. However, most implementations of grid-enabled MPI do not allow existing MPI applications to be easily ported to heterogeneous environments. Moreover, none of them provides a complete solution to all those problems discussed in the previous section. In this section, we will describe grid computing infrastructures, and compare various implementations of grid-enabled MPI, resource management systems, and grid scheduling tools.

1.6.1 Grid computing infrastructure

A grid computing infrastructure is a collection of common standard protocols, services and APIs that allows applications to access and coordinate geographically distributed data, software, and hardware resources. This infrastructure involves technologies in different domains, such as communication, security, and resource management. These technologies are integrated within an architecture that allows them to interact and to be interoperable. Based on such an infrastructure, programmers can develop services and functions to exploit diverse, heterogeneous and distributed computing resources to meet the application's computational demands. Globus [21] and Legion [26] are the most well-developed grid computing infrastructures.

The Globus Toolkit [20] is a collection of software tools and libraries that serve as the building blocks of a computing grid. The Globus Toolkit is an implementation of key protocols defined by the Open Grid Services Architecture (OGSA) [23]. Each component in the Globus Toolkit performs particular functions or provides particular services defined by OGSA. OGSA and its implementation, the Globus Toolkit, adopt a service-oriented architecture that allows services to be added on demand. The service and functionality of each existing component is well-defined and provides a standard interface, regardless of the underlying host systems. Application developers and grid users rely on one or more components in the Globus Toolkit to achieve the services needed for grid computing, such as secured inter-site communication, user authentication, or application specified resource allocation. Functions and services required by specialized grid systems, or targeting parallel applications, can be built based on the toolkit and deployed within the existing Globus installation. The Globus

Toolkit is portable to many operating systems and collaborates with a wide range of resource management systems. However, the current releases of the Globus Toolkit do not include a fully functional meta-scheduling service. Indeed, the optimized scheduling decisions rely on the requirements of specific applications and systems.

The Legion project's goals are similar to those of Globus. Unlike the Globus Toolkit, which is designed in a top-down manner, Legion uses a bottom-up approach. By analyzing the most basic requirements of grid computing, Legion develops a core object model. This object model provides a virtual machine for application developers and grid users, and defines services and functions as objects and object classes. Based on the object-oriented design, these service, function objects, and classes are encapsulated within an integrated architecture. Interfaces between Legion objects define how objects interact with one another.

The use of the core model allows Legion to scale well with respect to the number of computing sites. Virtualization provides a unified computing programming platform for software developers. On the other hand, the core model and virtualization make it challenging to implement all the functions needed to support a range of applications on different platforms. Since the Globus Toolkit is more flexible and more adaptable to heterogeneous systems, many scientific applications and software tools targeting computing grids have been developed based on the Globus infrastructure [41, 10, 34, 32]. Our work also targets Globus-based grid systems, and we will focus our discussions on this environment.

1.6.2 Globus-enabled MPI approaches

MPICH-G2 [33] is an implementation of Globus-enabled MPI 1.1 that is integrated within the Globus Toolkit. Message passing between computing sites over a public network is based on Globus-IO to provide secured communication, which is a component of the Globus Toolkit. Resources from different computing sites allocated for an MPI execution are coordinated by the Dynamically-Updated Request Online Coallocator (DUROC) library [15], another Globus Toolkit component. MPICH-G2 enables MPI applications to run on multiple computing sites, such as clusters. However, there are restrictions on how computing resources are used within heterogeneous networks.

As we described in a previous section, communication from an external computer to an internal resource of a computing site must be routed through the headnode. When an MPICH-G2 compiled program is launched across multiple computing sites, headnodes of each computing site must participate in the cross-site execution. Any two computing resources in different computing sites cannot communicate with one another directly unless they are assigned with public IP addresses. Inter-site messages must be explicitly routed through the process running on the headnode. As a result, applications compiled using MPICH-G2 must be modified to enable such inter-site communication. MPICH-G2 cannot provide a portable or topology-transparent environment for application developers. The above restriction makes it difficult to port existing MPI applications to computing grids.

MPICH-G2 uses the DUROC library to coordinate processes running on different computing sites. DUROC supports static resource allocation and co-allocation; it can only synchronize processes running on computing sites where the user specifies

computing sites at job submission time. The DUROC runtime control function places a global barrier on every application process within MPI initialization. Therefore, an application cannot change the computing site where the job is executed or the number of processes after its job request is submitted. This limits an MPI application's ability to adapt to unbalanced workloads of computing grids dynamically. Since the Globus Toolkit does not provide centralized resource management, changes in the utilization of different computing sites are hard to predict, and therefore, DUROC cannot be used to balance workloads among sites dynamically.

MPICH-GX [14] takes a different approach than MPICH-G2 in handling inter-site messages. MPICH-GX uses Globus-IO of the Globus Toolkit to send inter-site data securely. System proxies, running on the headnodes, are responsible for forwarding inter-site messages to destination processes. An outgoing inter-site message must be sent directly to the proxy process of the destination computing site. The message is then forwarded to the destination process from the proxy process within the computing site. The advantage of using proxies to forward inter-site messages is that the heterogeneous networks are transparent to application developers, and MPI applications do not need to be modified. However, each proxy must connect to all application processes, introducing large overhead during the initialization routine. Furthermore, application users have to manage job submission and resource co-allocation when they launch applications on a computing grid.

1.6.3 Resource management of computing grids

A resource management system, or a batch job system, monitors and manages computing resources. It assigns computing resources to job requests, and controls how long the jobs are allowed to use the resources. To process requests fairly, most batch job systems process queued job requests based on a first-come-first-served policy, and non-privileged users are not allowed to directly specify when a particular job will be executed.

LoadLeveler [31], PBS [28], LSF [52], NQS [29], Maui [1] and Condor [36] are the most commonly used resource management systems. These systems manage computing resources within a single administrative domain. A resource management system assigns resources according to the resource specifications of job requests and resource availability. Loadleveler, PBS, and LSF can allocate resources in a parallel system running a homogeneous operating system. Condor is designed for a distributed computing environment with non-dedicated resources that can be shared with local users while processing computational requests from remote users. These system tools manage how the computing resources are used within one administrative domain. Some of these systems provide features such as backfill to increase resource utilization for small job requests, while others can support job re-entry, preemption, or job migration. However, none of these management systems are designed to collaborate with one another over multiple administrative domains.

The Globus Toolkit includes a collection of software tools for grid computing; however, it does not offer resource management functions. It defines a standard application programming interface and provides a range of adapters to interact with

various resource management systems. Application users can submit jobs, query a job's status, and manage job execution using the standard tools and the API interface without knowledge of the host resource management systems. On a Globus-based grid platform, job requests can be submitted to one or more computing sites specified by the user. While the Globus infrastructure offers job submission tools to access heterogeneous systems, it is not a centralized resource management system. The grid infrastructure does not queue or collect job requests submitted to the grid system, nor does it function like any known resource management software. Each Globus job request is passed to the local resource management system. When a job targets multiple computing sites of a grid, the key to optimize application and grid system performance is in selecting computing sites and coordinating resource allocations. This is also known as the meta-scheduling problem. Existing grid infrastructures, such as Globus, do not provide the meta-scheduling capability. However, several individual research projects have developed tools with the meta-scheduling capability, while targeting a specific range of applications [9].

The Condor-G system [24] extends the resource management capabilities of Condor from a single administrative domain to multiple domains through integration with the Globus Toolkit. The system can manage application computation and a remote execution environment, while allowing remote resource access across administrative domains using Globus services. Matchmaking techniques [4] are currently under investigation by the Condor-G system to implement meta-scheduling. This technique matches applications to computing sites, taking into account the status of each computing site and the resource requirements of the applications. To launch an application, Condor-G starts daemon processes on selected computing resources,

creating a resource pool for the application. The GlideIn mechanism is then activated to start the application execution on those selected resources. While Condor-G addresses the location- and administration- transparency issues, its execution environment cannot create a network-transparent environment for parallel applications running on multiple computing sites. Furthermore, the daemon processes running at the user level also consume computing resources even though they are not used to execute applications, making it expensive and inefficient to use.

Nimrod/G [13] is a resource management and scheduling system based on the Globus Toolkit. It targets applications that involve a large number of task executions and a range of parameters. The Nimrod/G system provides economic computing by scheduling tasks based on the application-specified cost and deadline. Given application-specified requirements, the scheduling process relies on resource reservation and bidding techniques to achieve optimized results in terms of computing cost. However, in order to do so, the scheduler must know the resource availability and the task execution time (or the estimated execution time). Another concern of using Nimrod/G is that it cannot perform coordinated scheduling for multiple tasks.

The Pegasus framework [16] maps a complex and abstract workflow onto distributed computing resources. The system transforms a user-specified workflow to concrete executions with specific resources allocations, while hiding resource location and multiple administrative domains of a computing grid. The Pegasus scheduler can use the random, round-robin and min-min site selection algorithm to select a computing site for a given workflow task. However, the selection process is based on available and accessible resources, and the known task execution time. Furthermore, the Pegasus system does not address the resource co-allocation issue for multiple

workflow tasks which must be executed concurrently.

Many of above meta-scheduling systems commonly assume that computing resources are available for an applications to use during the scheduling process. However, this is often not the case in practice. In fact, job requests are queued and wait for their requested resources to be allocated by resource management systems. When meta-schedulers are not capable of accurately predicting resource availability, one common approach is to use the advance reservation mechanism to assure application users that all requested resources will be available at a particular time. Therefore, it can allow schedulers to make sure that applications can complete before their deadlines, or to reduce coordination time among resources across different computing sites. An advance reservation-enabled batch system enforces the requested job to be executed at a specified time, instead of following the FIFO order of the job queue or a particular queuing policy. Thus, the resource management system must keep the requested computing resources available for the user, either by job preemption, or by preventing other users from obtaining the resources. However, these management options are not supported in many batch job systems, nor available to the non-privileged users. Furthermore, interruptions of job queues caused by advance reservation can severely impact the system throughput [44]. Because of all the above limitations, we must develop a new resource allocation and coordination mechanism that can allow applications to adapt to a dynamic and heterogeneous environment.

Chapter 2

Resource Allocation Model

To address the challenges discussed in the previous chapter, we created a new resource allocation model. Section 2.1 describes how this model enables dynamic resource allocation in a multi-site computing grid environment. We will present the model's function specifications in Section 2.2. Then, we will evaluate the model's performance by analyzing the results obtained from simulating computing grids.

2.1 Model description

Within a single administrative domain, the conventional way to allocate resources for an MPI parallel execution is to specify the resource requirement in a job request and submit it to the resource management system of the host. The resource requirement, also known as resource specification, contains the number of computing resources required by the application. To simplify our study, we do not consider other resource requirements, such as the disk space, memory size, and network bandwidth. After the requested resources are allocated by the host resource management system, the

application is launched on the resources. When the application program executes the standard MPI initialization routine, all the MPI processes running on the resources are synchronized. From then on, all the processes can perform inter-process communication.

The conventional resource allocation, which is static, cannot be effectively applied to computing grid platforms across multiple administrative domains (see our discussion in Section 1.5). Therefore, we create a new resource allocation model, called the dynamic resource allocation model, which targets multiple administrative domains. This model allows an MPI application to request more resources than needed; however, the application uses only the needed resources and the remaining allocated resources must be released.

For a single site execution targeting a multi-site computing grid, the conventional resource allocating method must determine which site will yield the shortest queuing time for the requested resources. Unlike the conventional approach, our new resource allocation model allows the application to duplicate its job request and submit the request with its duplications to different computing sites so that the application can use those available resources allocated at the earliest time.

In general, accurate prediction of job queuing time is difficult to achieve. Although it is possible to estimate the execution time of a given program based on the profiling information from its previous executions, such information is not always available, nor produces accurate estimations. Given that it is unknown which computing site will allocate the requested resources first, our new model allows application users to submit the job request and its copies to those sites that are most likely to produce shorter queuing times. Therefore, our model relaxes the requirement that the site

with the shortest queuing time be located.

This new model provides dynamical resource allocation by using duplicated job requests. Once the requested resources are allocated and the application's resource requirement is met, the application starts to run. Hence, all resources that are allocated later become redundant, and must be released. The redundant resources could be utilized by other job requests waiting in the queue. How to release redundant resources (i.e., how to invalidate a redundant allocation) is left for the implementation of the model.

A cross-site parallel execution can also use this model to collect computing resources from multiple computing sites dynamically. In a cross-site execution, the resources required for the execution need to be allocated by multiple job requests. [1] These job requests are allowed to be duplicated and submitted to a number of computing sites. When these job requests allocate resources, the application starts on those resources that are allocated at the earliest time. Those earliest allocations that can jointly satisfy the application's resource requirement are used to perform the cross-site execution. The resources that are allocated later become redundant and must be released. However, how to divide the required resources among multiple job requests is determined by application users.

Conventionally, the application resource requirement (or resource specification) is specified by a single number of computing resources. We take a different approach in our model by using two boundary conditions to specify the application resource requirement. The two boundary conditions are: a lower-bound resource requirement

[1] We limit our discussion to the execution using resources from two computing sites, or 2-site execution, in this thesis.

and an upper-bound resource requirement. The *lower-bound resource requirement* is the minimum number of computing resources required for a parallel execution to start. In contrast, the *upper-bound resource requirement* is the maximum number of computing resources that are allowed to participate in the parallel execution. Any additional resource allocation that adds to the total allocated resources above this value is redundant.

For a single site execution, the lower-bound and the upper-bound values are equal. However, for a cross-site execution, the lower-bound and the upper-bound can be different, providing a range of numbers of resources which can be collected from different resource allocations. The advantage of this approach is that it allows an application to adapt to changing load conditions by using varied numbers in the resource specification. After job submissions, the application can use fewer resources when the environment becomes heavily loaded.

Our new model requests more resources than needed, uses only the resources that are required, and releases redundant resources. Given a resource allocation, the key function of the model is to determine if this allocation is insufficient, adequate, or redundant for the application resource requirement. We will discuss this function in Section 2.2.3.

Our new resource allocation model has several advantages over the conventional model. First, it complements the conventional resource allocation methods by allowing request duplication at submission time and invalidation of duplication at runtime. For applications and application users to employ this model, no changes are required in the application source code. Since the importance of having accurate queuing time prediction is reduced, our new model is much easier to implement than other

prediction-based schedulers. The model can adapt to dynamic changes in workloads across different computing sites. Furthermore, the model provides administrative-transparency for applications targeting computing grids.

2.2 Functional specification

Based on our resource allocation model, this section presents its functional specification, which guides our implementation of the adaptive grid computing framework. The specification is divided into five function components: site selection, job management, resource assessment, resource binding, and resource synchronization. These function specifications can complement the existing MPI standard applied for computing grids.

2.2.1 Site selection

Given an MPI application targeting a computing grid, *site selection* is the process of creating a list of computing sites, to which the application's job request and its duplications are submitted. The site selection algorithm is designed to select computing sites that are most likely to have shorter queuing time for the job request among all sites of the targeting grid. This function does not need to rely on accurate queuing time prediction to locate the site that queues the job request in the shortest time on the grid. The site selection function only needs to pick out those sites that are most likely to have the least queuing time.

When application users pick all sites on the grid for submissions, the site with the shortest queuing time will be located. However, submitting a duplicated job request

and releasing a redundant resource allocation inevitably consume system resources and introduce system-wide overhead. If every application has its job request duplicated on every computing site, this flooding may congest network and decrease the throughput of all resource management systems. With fewer duplications for each application, computing platforms will perform better. However, as the number of duplications decreases, it is less likely that the site with the shortest queuing time will be located. The optimal duplication number may depend on the application resource specification, the site selection algorithms, the available computing resources, and the system load. In this thesis, we also refer the number of duplications as *duplication factor*. A duplication factor of 0 means that only one job request is submitted, and a duplication factor of 1 means that the original job request and a single duplicate is submitted.

Since it is difficult to estimate job queuing time, the site selection function does not need to predict the job queuing time. Computing sites are ranked by their expected queuing time. The higher a site's rank, we can expect the shorter the site's queuing time. System workload information, such as the job queue length, the number of unused resources, system throughput and the average resource utilization, can be used as the ranking criteria. Furthermore, the selection algorithm can formulate a probability function to rank each site based on job profiling, processing speed, workload, or submission time. All these factors contribute to the queuing time, and therefore, can be used as the ranking criteria. However, the implementation should consider the availability of the selection criteria on the targeted systems, and allow users to choose the selection criteria and to modify the number of sites selected. Later in this chapter, we will evaluate the performance of various ranking criteria using our

simulation environment, and analyze the impacts of different duplication factors on host systems.

2.2.2 Job management

Given an MPI application, the job management component is responsible for managing the application's job requests and duplications, including three types of functions: job submission, job status query, and job cancellation. All management functions are executed at the user-level, similar to the user-side functions of most resource management systems. Therefore, job management functions can be implemented based on the host's resource management API. To achieve portability, adapters are needed, through which job management can interact with different types of host resource management systems, such as OpenPBS, LSF, and loadlever.

The job submission function is responsible for submitting each application-specified job request and its duplications to local resource management systems (or batch job systems). In addition, the job submission function must be specified with the resource lower- and upper-bound conditions to support the resource assessment process, as to be described in Section 2.2.3.

An application's job request and its duplications are submitted to selected sites independently from one another. Submissions of the job request and its duplications do not need to be synchronized. The query function returns four different results of job submission status, including queuing, execution, complete, and abort. Moreover, there is no need for the query function to differentiate between a job request and its duplications.

When the application's resource requirement is satisfied, any redundant resources must be released. Redundant resources can be released using one of the following two approaches: job cancellation and execution self-termination. The job cancellation function removes redundant resource requests from job queues when these redundant requests are still in the queuing state. If a redundant job request moves to the execution state, which means that the resources have been allocated for an MPI application, the processes running on the redundant resources can conduct self-termination within the MPI initialization routine.

2.2.3 Resource assessment

Our model allows application users to request more computing resources than what the application requires by submitting duplicate job requests to multiple computing sites. However, a job request and its duplications are processed independently from one another, given that they are submitted to different administrative domains. We do not assume that resource management systems can perform intercommunication or have interoperability across the administrative domains. Therefore, when resources are allocated for a job request, the model must determine whether such a resource allocation is needed or redundant for the application. The specification of resource assessment is designed to perform such functionality.

In this research, we define *bound resources* as computing resources that are allocated to satisfy the application resource requirement and participate in the application execution. In our model, when resources are allocated for a job request, this resource allocation is evaluated against the application's resource specification - the

two boundary conditions. Therefore, the resources become either bound or redundant for the application execution. For a single site execution, only one resource allocation is needed to meet the application resource requirement. In comparison, two resource allocations are needed for a cross-site execution,. Given newly acquired resources for a job request, the resource assessment function compares the application resource requirement with the sum of of existing bound resources and the newly acquired resources.

The number of bound resources can be obtained by querying the resource binding service, which will be described next. The computing resources allocated for an MPI execution can be retrieved from its job request, or obtained from a standard MPI library function. The application resource requirement (i.e., the two boundary conditions) is provided by the job management functions.

The total allocated resources is the sum of the existing bound resources and newly acquired computing resources. The state of the newly acquired resources can be determined by the resource assessment as the following:

1. Insufficient: the total allocated resources is below the lower-bound of the boundary condition.

2. Adequate: the total allocated resource is between the lower-bound condition and the upper-bound condition.

3. Redundant: the total allocated resources is above the upper-bound condition.

When newly acquired resources are insufficient, the application resource requirement is not met. The newly acquired resources can join the existing bound resources

and wait for additional resources to be bound, which will be discussed in the specification of the resource synchronization. If newly acquired resources are redundant, the resources must be released. The redundant resources can be released either by the job management function or by application self-termination. When the number of the total allocated resources is above the upper-bound condition, our specification treats the newly acquired resources as redundant resources, even if the existing bound resources do not satisfy the lower-bound of the application resource requirement.

If newly acquired resources are assessed to be adequate, the newly acquired resources will be bound with the existing bound resources, and then all bound resources will proceed together with the application parallel execution. However, when those existing bound resources already satisfy the requirement and the additional acquired resources do not exceed the upper-bound condition, the resource assessment function still consider the additional acquired resources to be adequate. As such, our specification allows the number of allocated computing resources to increase dynamically during MPI execution, which gives applications more flexibility to utilize computing resources efficiently. Application users can increase the upper-bound condition of the application resource requirement. To support this function, the application itself must allow varied degrees of parallelism at runtime.

2.2.4 Resource binding

Based on our model, a cross-site MPI execution on a computing grid consists of multiple MPI executions, and each execution is independently launched on a computing site. The *execution topology*, which specifies an MPI execution on one site, contains

information about computing resources that are used by the parallel execution and process ranks. The *global execution topology*, which is used to specify a cross-site MPI execution, contains the execution topology of each individual MPI execution on a site and the identifier of the site. Thus, *resource binding* is the process of adding the execution topology of an individual MPI execution to the global execution topology.

When using our model, an application's job request and its duplications are submitted to a number of computing sites simultaneously. However, these job requests are processed independently among different administrative domains. There could be more than one binding procedure attempting to modify the global execution topology. The resource binding specification requires that any modification to the global execution topology be guarded with a write lock to prevent the data from being read or written during the modification.

The component of resource binding also includes functions for querying the global execution topology and withdrawing from the bound resources. The query function returns the latest global execution topology. When an MPI application completes its parallel execution, all bound resources must be removed from the global execution topology. The withdrawing function removes the information about the bound resources of an MPI execution from the global execution topology.

All resource binding functions must be transparent to application developers and users so that no source code modification is required for legacy MPI applications. To achieve this transparency, application-side procedures can be implemented using a daemon process, or within the standard MPI initialization routine. The daemon is a designated program, running ahead of the application execution, which performs the binding procedure. This is similar to the approach used by the Condor-G system.

2.2.5 Resource synchronization

Resource synchronization is used to coordinate resources allocated on different computing sites for cross-site MPI executions. This function component specifies how to synchronize the allocated and unallocated resources, when the lower-bound condition of the application resource requirement is not met. For a single-site MPI execution, there is no need to perform resource synchronization because one resource allocation already satisfies the lower-bound condition. For a cross-site MPI execution, our model allows applications to collect computing resources from multiple allocations. Since all the requested resources on different computing sites cannot be assumed to be available at the same time, resource synchronization should block the application's execution on those allocated resources, until there are enough resources to meet the application's resource requirement. Functions for resource synchronization can be implemented within the MPI initialization routine, and therefore, the synchronization process will be transparent to application developers and users. Next, we will specify three synchronization methods, including stop-and-wait, hibernation, and job resubmission.

In the *stop-and-wait* synchronization method, all application processes stop running on the allocated resources and wait until the resource requirement is met. This procedure relies on the query function to obtain the global execution topology, and resource assessment to determine the current state of bound resources. No additional support from host resource management system is required. However, leaving allocated resources unused or idle may reduce utilization of computing resources.

The *hibernation* method provides an alternative way to synchronize bound re-

sources with unallocated ones. This approach puts an MPI execution into the hibernation state and releases its allocated resources. The hibernation method stops a running program at a recovery point and then restart at the same point later, while maintaining the same execution context. This procedure requires that job preemption not only be supported by host resource management system, but also be allowed by applications. When an execution is woken up from the hibernation state, the resource management system must be able to reclaim those resources used by the hibernating execution through forcing other executions that are using the same resources to stop. The hibernation function must be able to interact with resource management systems to wake up hibernating executions when the lower-bound condition is satisfied by a newly bound resource allocation. However, preemption may not be available in some batch job systems, and may not be supported by the queuing policy. Moreover, some jobs may not allow to be preempted during their executions. As such, the hibernation method has limited portability and adaptability.

The third synchronization method is *job resubmission*. In the resubmission procedure, an MPI execution withdraws from bound resources and is terminated. Then, the resource synchronization function resubmits the same job request back to the local resource management system. The limitation of this procedure is that the application must allow re-entry, as its execution may be terminated and restarted again later.

The stop-and-wait approach is the default method, since it is more portable than the other two options. In comparison, the hibernation and the resubmission approaches, which have limited system portability due to preemption and re-entry requirements, can allow users to improve resource utilization. The common advantage of these three synchronization methods is that they can be implemented based on the

existing resource management systems.

The specifications of resource synchronization are designed to handle varying resource availabilities across different computing sites. Using resource binding and synchronization, we can collect those earliest available resources to execute user-defined parallel MPI programs across multiple computing sites. In the next section, we will evaluate the performance of our resource allocation model using a simulation environment.

2.3 Model simulation

In previous sections, we have introduced a new resource allocation model and its functional specifications. In this section, we will describe a simulation environment that is used to evaluate the model's performance. Here, the performance is the average queuing time of a list of job requests that are simulated in our computing grid environment. The simulation imitates the behaviors of our resource allocation model and multiple job queues, given lists of job requests to process. One of our objectives is to quantify the improvement in individual applications' performance and the impact from using duplications on a computing grid, and to find the correlation between them. Different load levels and a range of duplication factors are tested in our simulations to identify those system configurations that can achieve the best performance. Two different implementation approaches for site selection are investigated. The findings of our simulations are used to guide our system design and its implementation to be discussed in the next chapter.

The simulation environment is developed based on CSim [6], which can simulate

multiple concurrent events and running processes. CSim provides C-based functions to create random variables with various distributions. The environment simulating a computing grid consists of three components: workload generation, simulation of a computing site, and a global job scheduler. The global job scheduler simulates site selection, which has been defined in Section 2.2.1.

2.3.1 Workload generation

Before a simulation starts, the workload generation component creates lists of job requests, which are the inputs to the simulation. In an actual resource management system, a job request is the specification of the execution code of an application, how to run it, and what are resources it requires. To simulate how job requests are processed by resource management systems, we assign the following fields to each job request: inter-arrival time, execution time, estimated runtime, and the number of computing resources. These values are generated based on specific random distributions, imitating how resources are used among different users and applications.

In our simulation, two types of job requests are defined: local job requests and global job requests. A *local job request* (also referred to as local job) is only allowed to be processed by one computing site when the job request is created. A *global job request* (also referred to as global job), on the other hand, can be processed by any computing site that is determined by the global job scheduler. A local workload contains a list of local job requests that are submitted to the same computing site, imitating how resources are used by local users of that site. The global workload, containing a list of global job requests, represents applications that can use resources

of any computing site on a computing grid. In our simulation, when a global job is submitted to a computing site, it is treated as a local job without given any higher priority.

For each local or global job, the job inter-arrival time is defined as an exponential random variable, and the job execution time is defined as a Zipf distribution. The job size is generated using a hybrid method, which is based on a modified Poisson distribution. This distribution assigns job sizes of $1, 2, 4, 5, 8, 10, 16, 30$ and 32 with higher probability than others [7, 17].

Given an m-job workload $W = \{j_0, j_1, ..., j_{m-1}\}$, the level of the workload running on N_{comp} computing resources is:

$$load_level = \frac{T_{exe} \times N_{size}}{T_{arr} \times N_{comp}} \tag{2.1}$$

where T_{arr} is the average arrival time, T_{exe} is the average executions time, and N_{size} is the average job size of all m jobs. The level of the workload is the ratio between the computation time, which is needed by the workload's m jobs, and the computation time, which is provided by all N_{comp} computing resources. This value represents the load condition of a computing site. A higher workload level implies a longer average queuing time.

In our simulations, the average job size of a local workload is 7, while the average job size for the global workload is 12. Global jobs are larger than local jobs because global jobs targeting computing grids tend to require more resources than local jobs. The average execution time for all jobs is 60 simulation seconds. Based on Equation 2.1, we can obtain different values of the workload level by modifying the average job inter-arrival time.

During a simulation, every local workload is generated using the same workload level, and is assigned to one computing site. Therefore, all computing sites, overall, have balanced workloads. At a specific time, each site may experience differences in workload and job requests' queuing times. However, the differences tend to be smaller than the variations that occur given different workload levels, allowing us to detect the effectiveness of site selection functions.

2.3.2 Simulation of a computing site

In our simulation of a multi-site computing grid, each computing site is simulated as a batch job system with a single job queue. Three commonly used queuing policies are implemented in our simulation environment including first-come-first-serve(FCFS), backfill EASY and backfill conservative policy. The two backfill policies require that all jobs specify their estimated maximum execution times, which are used to calculate the minimum waiting time for a queued job request.

Our testing environment can simulate a computing grid with 4 to 16 computing sites, and each computing site is configured with 64 computing resources. Our simulation of a computing grid is modeled as a collection of uniform systems; each computing site of the grid is assumed to have the same computing power. We also simulated heterogeneous grids, and the results are presented in Section 2.4.6. Within a heterogeneous grid's simulation, half sites are configured with 32 computing resources and the others are configured with 64 computing resources.

A computing site also publishes the queue length, the number of available computing resources, and the average utilization of all computing resources. We assume

that no simulation time is required to query the above information by the global job scheduler. However, for any job submission, 1 simulation second is needed to process a job request or to cancel a redundant job request.

2.3.3 Global job scheduler

In our simulation environment, the behavior of the global job scheduler complies with the specification of site selection. The global job scheduler assigns each global job request to a batch job queue which represents a computing site. Given a global job, the global job scheduler ranks all computing sites based on a specific ranking criterion. Then, each job request and its duplications are submitted to those top-ranked sites. We investigated different ranking criteria, including the queue length, system throughput, and the number of available resources. We also tested some prediction-based ranking criteria, such as estimated queuing time, which is calculated using the user-specified maximum estimated execution times for job requests.

To evaluate the performance of our proposed global job schedulers, we implemented an ideal global job scheduler, Ideal Min, which can select the computing site with the shortest queuing time for each global job request. In a FCFS queue, the queuing time for a job request can be computed, given that the exact execution times of all jobs are known ahead of a simulation. Therefore, with the actual queuing times for the global job on all computing sites, the Ideal Min scheduler can locate the job queue with the shortest queuing time. However, when managing a queue using the backfill EASY or conservative policies, a job's start time can be delayed beyond its predicted start time, because out-of-order scheduling may allow a job submitted later

to get executed ahead of all queued jobs. Nevertheless, such delays are assumed to be small and therefore, are not taken into account by the Ideal Min scheduler.

We investigated two approaches to implement the global job scheduler: centralized and distributed. The distributed approach can be described as a parallel design, in which the scheduler runs on multiple computers. Therefore, we define the *distributed scheduler* as a parallel program that runs on multiple computing sites and makes scheduling decisions jointly among all processes. In this dissertation, the distributed scheduler is designed to process each global job individually without considering other job requests, or in a decentralized manner. As a result, in the distributed global job scheduler each scheduling decision is made independently, while the objective of the scheduler is to minimize the job queuing time for each individual global job.

Given a global job, the distributed scheduler ranks all computing sites based on a specified ranking criterion. It submits the global job to the top-ranked site without considering other global job requests, incoming or ongoing. As a result, this greedy approach may not produce optimal scheduling results for the overall throughput. We investigated seven different ranking criteria for the distributed job scheduler in the simulation environment:

1. Random value (Random);
2. Queue length (Qlen), which is the number of job requests queued in a job queue;
3. System workload (Workload), which is the number of jobs being processed in the last time period;
4. The number of available resources (Avail);

5. Estimated work time (Est. RT), which is the sum of estimate run times of all queued jobs;

6. Estimated queuing time (Est. QT), which is the waiting time for a specified number of computing resources, calculated based on the estimated run times of queued and running jobs; and

7. Actual queuing time (Ideal Min), which is calculated based on the actual execution times of queued and running jobs. This method can produce the optimal scheduling results, and it is mainly used as the benchmark for comparison.

In contrast to the decentralized decision-making process of the distributed scheduler, the *centralized scheduler* is a single-process scheduler which makes scheduling decisions for all global jobs that are submitted to the scheduler. In our simulation environment, the centralized scheduler makes scheduling decisions, while considering all global jobs which are submitted during a period of time. All global jobs submitted to the centralized scheduler are buffered in a window before they are scheduled together. The buffered job requests are sorted by the job size or by the estimated job completion time. Each job request on the sorted list is submitted to a top-ranked computing site one at a time, which is similar to the distributed approach. The centralized approach can avoid submitting multiple global jobs to one computing site, and eliminate a sudden load surge caused by multiple submissions of global jobs to the same site simultaneously. However, the buffering time is accounted as part of the global job's queuing time.

For the centralized scheduler, we investigated five different ranking criteria in selecting computing sites for a global job:

1. Random value(Random). The scheduler lists job requests in the window randomly, and selects a site for each job randomly. It corresponds to the Random distributed scheduler.

2. Queue length (Qlen). The scheduler sorts job requests in descending order based on their job sizes, and submits each job to the site with the shortest queue length. It corresponds to the Qlen distributed scheduler.

3. Estimated run time (Est. RT). The scheduler sorts job requests in descending order based on job size, and submits each one to the site with the shortest estimated run time of all the queued jobs. It corresponds to the distributed Est. RT scheduler, when the window size is 0.

4. Estimated queuing time (Est. Max-Min). The scheduler uses the user-specified estimated execution times of job requests to compute a job's estimated completion time. Buffered job requests are sorted in descending order based on the maximum completion times among all sites. Each job is then submitted to the site with the shortest estimated completion time. It corresponds to the distributed Est. QT scheduler.

5. Actual queuing time (Max-Min) [37]. The scheduler uses the actual queuing times (which have been defined during the workload generation) to compute a job's completion time. It behaves in the same way as the centralized Est. Max-Min scheduler. This scheduler corresponds to the distributed Ideal Min scheduler.

To simulate job duplication, the global job scheduler submits a global job request

and its duplications to multiple computing sites at the same time. Those sites used for duplicate submissions are selected based on their ranks, and a higher rank indicates a higher likelihood of having a shorter job queuing time. When a duplicate job submission enters the execution stage, it checks the global execution topology and performs resource assessment. If resource assessment determines the duplicate job is redundant, this job is terminated in the next simulation second. As a result, all redundant resources consume 1 simulation second before they are released back to the resource pool.

We also investigated how resource co-allocation is affected by using duplicate submissions. A resource co-allocation requires two resource allocations, and each allocation may specify a different amount of computing resources. How to specify these two allocations is determine by individual applications and application users. However, to simplify our study we specify the resource co-allocation is to allocate the same amount of computing resources from two different sites. A global job request or its duplication only allocates half of the required computing resources. After the request and its duplications are submitted to multiple computing sites, the application resource requirement will be met by collecting two of these resource allocations.

We simulated stop-and-wait synchronization to study the resource co-allocation for cross-site executions. In stop-and-wait synchronization, the first resource allocation is forced into the idle state before the second resource allocation becomes available. Therefore, the synchronization time between the first and the second allocations must be included in the total execution time. If the synchronization time exceeds the time limit of the estimated execution time specified by the global job request, the allocated resources are released and the co-allocation job is marked as having failed. In the next

section, we will investigate how the failure rate of resource co-allocation is decreased by using job duplications.

We also simulated hibernation to compare it with stop-and-wait synchronization. When a global job request goes into the hibernation state, it releases the allocated resources back to the resource pool. When the hibernate global job is woken up, it can reclaim all the allocated resources by preempting those local jobs that are using the resources. In the hibernation method, the job request that provides the first resource allocation among all submissions goes into the hibernation state, and is woken up by another job request, which provides the second resource allocation.

2.4 Simulation result analysis

We use a multi-site grid simulation environment to evaluate the dynamic resource model, various schedulers and two scheduler implementation methods. Each computing site in our simulation environment is given a $50,000$-job local workload, and a $5,000$-job global workload is generated for all sites. Since the backfill conservative and backfill EASY queues yield similar results, we only present the results using the conservative queue in this thesis.

2.4.1 Centralized and distributed comparison

This section first compares the two different implementation methods: centralized and distributed, without job request duplication. The performance of centralized and distributed schedulers is shown in Figure 2.1. The figure shows that the average queuing times of global jobs, which are scheduled by the distributed and centralized

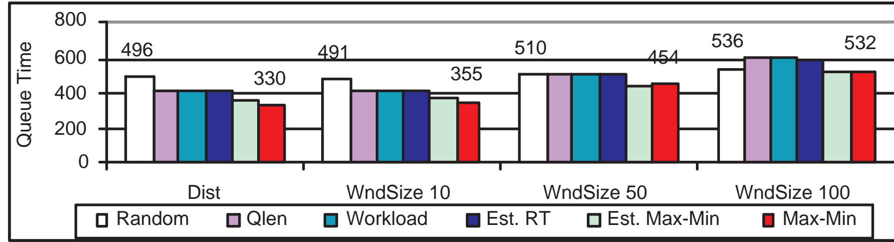

Figure 2.1: Comparison of centralized and distributed schedulers

schedulers using five different ranking criteria. The centralized scheduler is tested using a range of window sizes. When the window size equals to 0, the centralized Est. Max-Min and Max-Min are equivalent to the distributed Est. QT and the Ideal Min schedulers. Test results shown in Figure 2.1 are collected from simulations of an 8-site grid. Each site has a FCFS queue, given a local workload with a level of 0.6, while the level of the global workload is 1.2.

Figure 2.1 shows that the distributed scheduler performs better than the centralized scheduler. For the centralized scheduler, the shorter the window size, the better the observed performance. Our simulation results indicate that most global jobs do not benefit from the centralized scheduling, because the buffering time used by the centralized schedulers adds extra time to the total waiting time for global jobs. When multiple global jobs are submitted at the same time, the centralized approach can balance global workload by assigning these global jobs to different computing sites. However, to make the centralized scheduler most effective in our simulation, it is expected that the workload level of global jobs must be greater than 1.2, which is much higher than our estimation for the global jobs.

Furthermore, the centralized approach requires that all global job requests are

submitted to and subsequently processed by one system. In a real environment, this design may lead to larger system overhead and longer response time as an increasing number of global job requests are processed in such a centralized manner. The disadvantage of the centralized approach, therefore, will limit its scalability in a heavily loaded grid environment. Because of these limitations, we focus on the scheduler using the distributed approach and the distributed scheduler is employed in our framework design and implementation.

2.4.2 Distributed scheduling comparison

Figure 2.2 shows the performance of the distributed schedulers which use different ranking criteria. We simulated grid environments that have different sizes (shown in the left figure) and are assigned with different local workload levels (shown in the right figure). Each computing site is operated by using the backfill conservative policy and the level of the global workload is 0.5. Six ranking criteria of the distributed scheduler are selected from the list in Section 2.3.3. No request duplications are used in these simulations. The y-axis shows the normalized average queuing time of all global jobs over the average queuing time of the Ideal Min scheduler. The Ideal Min scheduler produces the ideal job assignment for each global job request, since each job request is always assigned to the site which yields the shortest queuing time for the global job request.

In Figure 2.2, the Random scheduler, which makes scheduling decision randomly, yields the worst performance when compared to other schedulers. We can also observe that the Est. QT scheduler outperforms other schedulers. However, the Est.

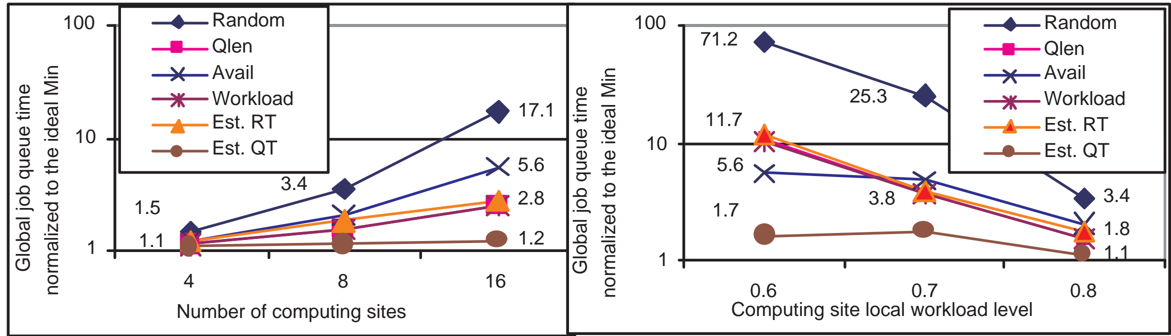

Figure 2.2: Normalized average queuing times of global jobs scheduled by distributed schedulers using different ranking criteria.

QT scheduler requires additional information about every job request, the estimated execution time on all computing sites for both local and global job requests.

The scheduler based on ranking the system throughput (Workload) does not demonstrate similar effectiveness as the scheduler based on ranking the queue length (Qlen), since it cannot respond fast enough to sudden job surges at some computing sites. Figure 2.2 also shows that the Avail scheduler performs well when the local workload level is low; however, it may lose some of its advantage on grids that have more computing sites. In most test cases, the Qlen and the Ext. RT schedulers do not outperform the Est. QT scheduler. However, when the actual queuing time is short (or in a lightly loaded environment) and the computing grid is smaller, the three schedulers yield similar performance.

Based on Figure 2.2, we can observe that the estimated execution times of all job requests can help the global job scheduler achieve better performance. However, we will show that our new resource allocation model can achieve similar performance by using job duplication, without requiring additional information about all jobs'

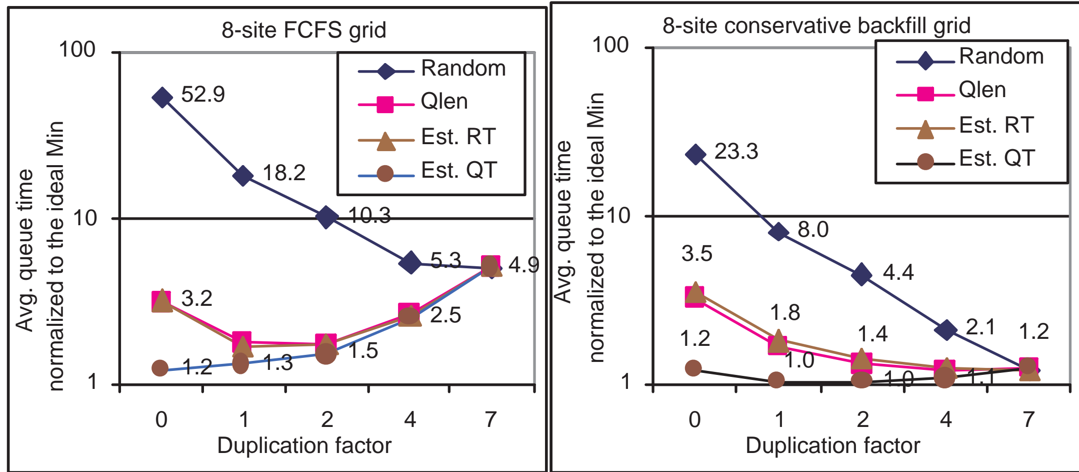

Figure 2.3: Normalized average queuing times of global jobs using a range of duplication factors

execution times.

2.4.3 Job duplication

Figure 2.3 shows the results of duplicating global jobs on an 8-site grid, when each site uses the FCFS queuing policy (shown in the left figure) and the conservative backfill queuing policy (shown in the right figure). Since a conservative backfill queue can process more jobs than a FCFS queue in a given period of time, we assigned a local workload of 0.6 to each FCFS queue and a local workload of 0.7 to each conservative backfill queue. Four schedulers are compared using a range of duplication factors, from 0 to 7, in Figure 2.3. The y-axis shows the normalized average queuing times for all global jobs over the average queuing time of the Ideal Min scheduler. The global workload level is 0.4.

By using job duplication, the Random scheduler achieves the highest performance

improvement, as compared to other schedulers using different ranking criteria. On the other side, job duplication is least effective when applied to the Est. QT scheduler. When the duplication factor is 2, the Qlen scheduler reduces the average global job queuing time to 6.9 and 7.0 simulation seconds on the two computing grids using the FCFS queuing policy and the conservative backfill queuing policy, respectively. This represents a reduction of 44.8% and 59.1%, compared to the scheduling results using no duplication in both grid environments. The best results obtained by the Ideal Min scheduler are 4.0 and 5.2 simulation seconds on the two grids, respectively. The Est. QT scheduler, taking 4.8 and 6.0 simulation seconds without job duplications, still outperforms the Qlen scheduler with a duplication factor of 2. However, the differences between them have been significantly reduced, and are expected to be even less when the local workload level increases.

Our simulations demonstrate that job duplication can help a global job scheduler perform significantly better, when no information about job requests' execution times is available as required by the Est. QT scheduler. Such information may not be available, or may be private in a resource-sharing environment. Furthermore, the accuracy of execution time estimation cannot be guaranteed across different executions,users, or platforms. A global job scheduler based on such estimates may not be able to produce consistent performance. In comparison, a global job scheduler with job duplication can achieve equivalent performance using only commonly available system information and without resorting to any prediction. Our simulation results suggest that the duplication factor of 2 is sufficient to improve performances.

Job duplication increases the workload on computing sites of a computing grid. Those computing sites that process redundant job requests could decrease their overall

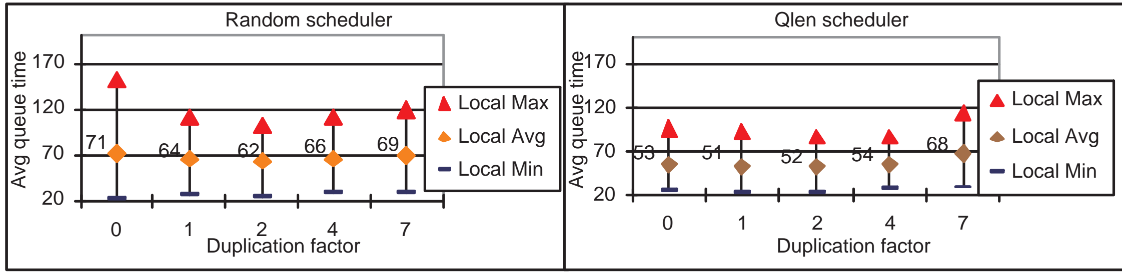

Figure 2.4: Job duplication's impact on local jobs' queuing times.

system performance. In our simulations, it will cost 1 simulation second to terminate one redundant resource allocation. When every redundant job request introduces 1 simulation second delay in a job queue, other job requests, including both local and global jobs, will be affected.

Figure 2.4 shows the effects of job duplication on local jobs' queuing times on a computing grid consisting of eight sites, each of which uses the conservative backfill queuing policy. The queuing times for local jobs on 8 computing sites are indicated by vertical lines, with the largest time on the top (Local Max), average in the middle (Local Avg), and the smallest time at the bottom (Local Min). On the left side of Figure 2.4, all global jobs are scheduled by the Random scheduler. On the right side, all global jobs are scheduled by the Qlen scheduler.

When the duplication factor equals to 1 or 2, we can observe reduced queuing time for local jobs on all eight computing sites. When the global jobs are scheduled by the Random scheduler with the duplication factor of 2, it reduces the average queuing time for local jobs by 13% over non-duplication. This performance improvement is a result of better balanced workloads among computing sites, which can also be identified by the decrease in the largest local job's queuing time (Local Max) and the

smallest local job's queuing time (Local Min).

Figure 2.4 also shows that flooding, the most greedy approach having each global job duplicated on all computing sites, will cause overall system performance degradation. To schedule a global job, the flooding does not need to estimate queuing time or rank all sites. The site providing the earliest available resources will be located, when each site is submitted with a copy of the job. A duplicate copy will not consume computing resources for execution, if the copy turns out to be redundant. However, the time to invalidate redundant jobs takes its toll on all job queues and increases the average queuing times for both global jobs and local jobs. As such, the global job scheduler should limit the number of job duplications, while avoiding flooding so as to balance the performance improvement between global and local jobs.

2.4.4 Resource co-allocation using the stop-and-wait method

Using our simulation environment, we investigated the dynamic allocation model for resource co-allocation, which is used in cross-site parallel executions. To simplify our study, we simulated resource co-allocation which requires two resource allocations, each of which allocates the same amount of computing resources. The *co-allocation time* is the time used to have such two resource allocations available at the same time for a cross-site execution.

When we apply job duplication in a cross-site execution, two global jobs and duplicated jobs, all of which specify the same number of resources, are submitted simultaneously. Only the first two global job requests that allocate resources at the earliest time will be used for the cross-site execution and all the others are redundant

and will be terminated. Using stop-and-wait synchronization, the co-allocation time can be computed by the sum of the queuing time for the first resource allocation and the synchronization time, which is used to wait for another resource allocation.

Figure 2.5 and Figure 2.6 show simulation results of using stop-and-wait synchronization (as specified in Section 2.2.5) to synchronize two resource allocations. We tested the Random, Qlen, Est. RT and Est. QT schedulers for global job scheduling, while applying different duplication factors for global jobs. Since it is presumed that the Ideal Min scheduler selects the computing site that yields the shortest queuing time for any given global job request, we can say that that the Ideal Min also leads the shortest co-allocation time for all resource co-allocation test cases. The co-allocation time obtained by using the Ideal Min scheduler gives a lower bound, and therefore, we used it the comparison benchmark to study the performance of other schedulers. In the simulation, each site of the 8-site grid runs the conservative backfill queuing policy, and is given a local workload of 0.6. Global jobs are generated using 0.4 as its workload level.

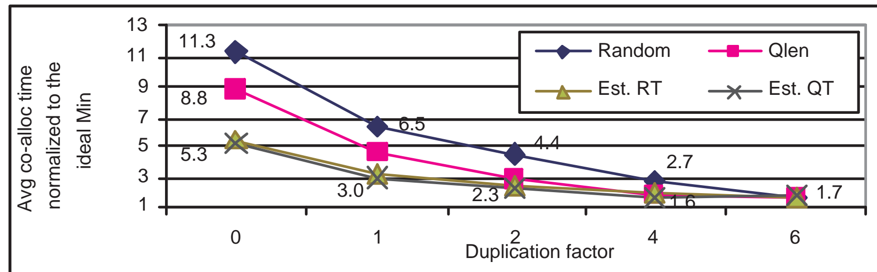

Figure 2.5: The average co-allocation time normalized to the Ideal Min using stop-and-wait synchronization.

Figure 2.5 shows the co-allocation time of global jobs normalized to the Ideal Min scheduler results. Job duplication significantly improves the performance of all global job schedulers tested in our simulation. While the Est. RT and the Est. QT schedulers perform better than the rest, there is only a marginal difference for the Qlen scheduler when using 2 duplicated job requests.

Figure 2.6 shows the average queuing time and synchronization time used during the co-allocation process. The co-allocation time, as presented in columns, consists of the queuing time (Queue) and the synchronization time (sync). In all simulations of the four different global job schedulers, it is found that the synchronization time represents approximately 15% of the total co-allocation time. All schedulers using 4 duplications are able to reduce the co-allocation time to 20% of the time consumed by the schedulers using no duplication.

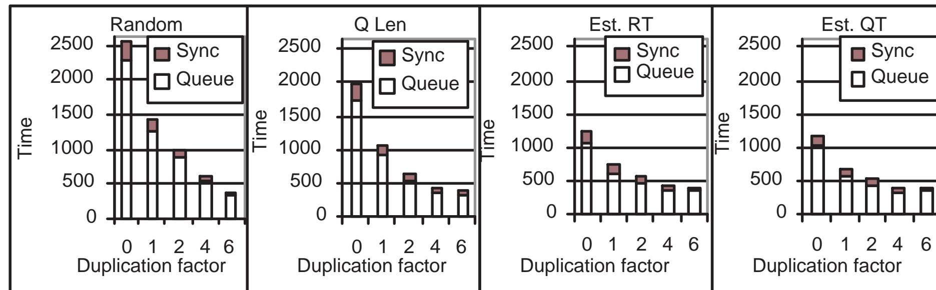

Figure 2.6: The average co-allocation time using stop-and-wait synchronization.

2.4.5 Resource co-allocation using hibernation

We also simulated hibernation-based synchronization as discussed in Section 2.2.5. In contrast to stop-and-wait synchronization, hibernation allows local jobs to use the

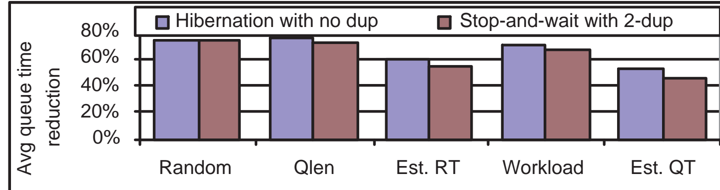

Figure 2.7: Reduction in average queuing time for all jobs when compared to stop-and-wait with no duplication.

resources that have been assigned to the global jobs in the hibernation state. As a result, during the co-allocation synchronization period, more jobs are processed, and therefore, the average queuing time for all job requests is reduced. Figure 2.7 compares the reduction of the average queuing time in the hibernation-enabled environment with that using stop-and-wait synchronization, which uses 2 duplications for global jobs. In the tests, we simulated 8-site grids and each site was configured to use the conservative queuing policy.

As shown in Figure 2.7, when we simulated the five global job schedulers, the stop-and-wait synchronization method using 2-duplication can achieve performance similar to that observed using hibernation. Unlike hibernation, which requires job preemption, our dynamic resource allocation model can achieve similar performance (as indicated in Figure 2.7) without relying on job preemption. This is a distinct advantage of our model.

Within a hibernation-enabled grid environment, we found that job duplication can further improve the overall system performance. Figure 2.8 shows the simulation results from using job duplications in the hibernation-enabled 8-site grid environment. The results indicate that the Qlen, the Est. RT and the Est. QT scheduler lead to

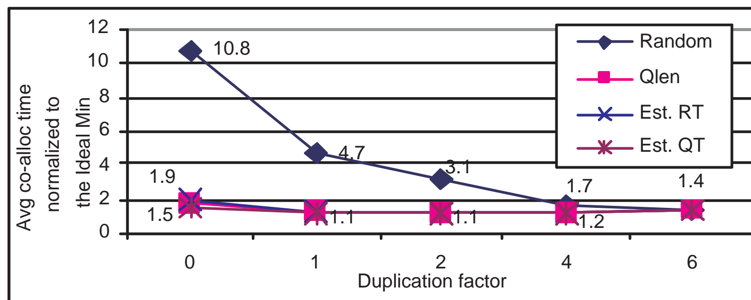

Figure 2.8: The average co-allocation time normalized to the Ideal Min, while using hibernation.

similar performance when using various duplication factors. We found that the performance gain using job duplications is relatively small as compared to stop-and-wait synchronization. However, 1 duplication can reduce the performance gap between the Qlen and the Ideal Min scheduler to 10%.

In conclusion, job duplication allows resource co-allocation using the stop-and-wait method to achieve significant performance improvement, as shown in Figure 2.5. However, a greater duplication factor may not lead to a desired performance gain, as shown from Figure 2.8, when hibernation is used in resource co-allocation.

2.4.6 Simulation of heterogeneous grid

So far, we have used simulation to investigate the dynamic resource allocation model and various global job schedulers in a homogeneous environment, in which all computing sites have the same number of computing resources. In Figure 2.9, we present the simulation results of heterogeneous grid environments. In these environments, half of computing sites of a grid are configured with 64 computing resources, and the remaining sites are configured with 32 computing resources. The smaller computing

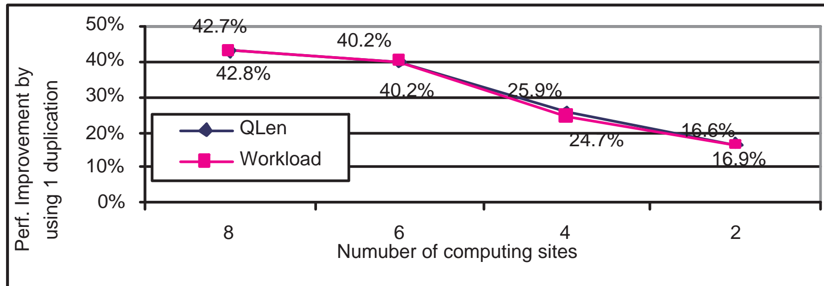

Figure 2.9: The reduction on average global job queuing time by using 1 duplication on heterogeneous grids.

sites are assigned local workloads having a greater number of small local job requests.

Figure 2.9 shows the performance improvement of the Qlen and Workload schedulers with one duplication. Our results show that the two global job schedulers produce very similar performance, and have similar performance improvement when the duplication factor is one. With one duplication for all global jobs, both schedulers reduce the average queuing time of global jobs by approximately 42% on an 8-site heterogeneous grid. This performance improvement is similar to that obtained in the 8-site homogeneous environment, as shown in Figure 2.3. Overall, these results indicate that our dynamic resource allocation model is highly effective in improving the performance of global job schedulers in both heterogeneous and homogeneous environments consisting of computing grids with various sizes.

Chapter 3

Framework Design

In the previous chapter, we introduced a new resource allocation model. Based on this model, we developed a framework, called the Schedule-Group-Relay (SGR) framework, which can hide different administrative domains and heterogeneous networks, and allows MPI workflow applications to execute on computing grids. Our SGR framework design complies with the model's specification, and consists of three service components: a workflow scheduler, task grouping, and message relay. Given an MPI multi-task workflow application, the scheduler conducts task scheduling based on the application's workflow dependency, the task grouping component performs resource binding and coordination for concurrent tasks, and the message relay component enables inter-task communication. In the following sections, we will describe the design of each component, the integration of the three components within the same framework, and how these three components are implemented in our computing environment.

3.1 Workflow scheduler

In Section 1.4, we have described the fundamental structures of a workflow, and workflows are used to specify our MPI applications targeting computing grids. The objective of the workflow scheduler is to launch tasks of a workflow application without user intervention, and at the same time, to achieve the optimized task scheduling result for the application.

In the previous chapter, we have introduced a new resource allocation model. This model achieves dynamic resource allocation across multiple computing sites by duplicating a job request on multiple sites to allocate required resources with the smallest queuing time. To use this model, application users directly map the duplicated jobs onto concurrent tasks in a parallel forking structure of a workflow. Given concurrent tasks in a parallel forking structure, only task resource requirements need to be modified to enable task duplication. When concurrent tasks are submitted to selected computing sites the first task which satisfies the application resource requirement will perform the single-site execution. For a cross-site execution (also referred to as two-site execution), the first two concurrent tasks which satisfy the cross-site execution's resource requirement jointly will perform the application's execution together. Our workflow scheduler in the SGR framework is designed to perform site selection and job submission functions, while resource assessment and resource binding are implemented within task grouping, which will be discussed in the next section.

In this section, we will describe the Petri net syntax that is used to specify a workflow. We then describe our design of the workflow scheduler, and its parallelized ranking procedure.

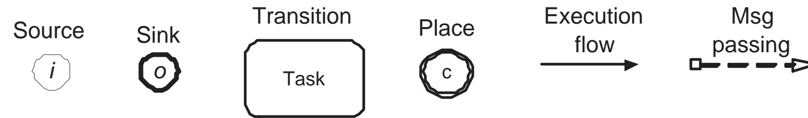

Figure 3.1: Syntax symbols of Petri nets.

3.1.1 Petri net

We first describe how we specify an application workflow using a Petri net. A *Petri net* is a directed bipartite graph, in which discrete events are represented by transition nodes, pre- and post-conditions of events are represented place nodes, and directed arcs connect between transitions and conditions of transitions [40]. Some basic syntax symbols for Petri nets are shown in Figure 3.1. Each transition node represents an application task, which is specified by the task resource requirement. A place node defines the system status after completion of a transition which is connected by a directed link. The directed link between a transition node and a place node is referred to as execution flow. An execution flow is only allowed to connect a transition node and a place node, and continuous execution flows represent the dependence order among tasks. Two special place nodes are the start and the sink used as the beginning and the end of a workflow. A non-solid link connecting two transition tasks represents inter-task communication between tasks.

In Figure 3.2, we illustrate examples of three fundamental workflow structures including the dependent, the parallel and the optional forking structure. The dependent structure requires that *task0* must complete before its dependent *task1* can start. In the parallel forking structure, *task1* and *task2* are started simultaneously. In the optional forking structure, either *task1* or *task2* is chosen to be executed, which is

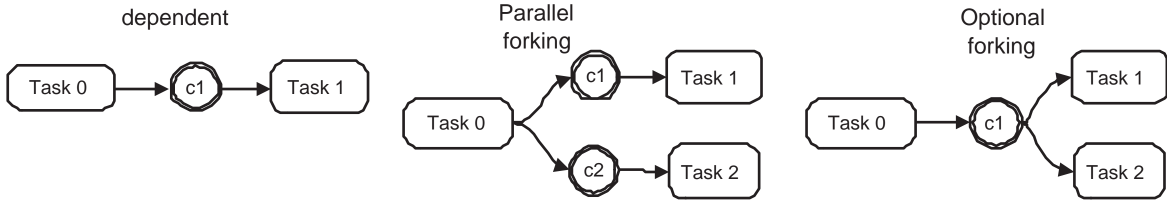

Figure 3.2: Examples of the three fundamental structures of Petri nets.

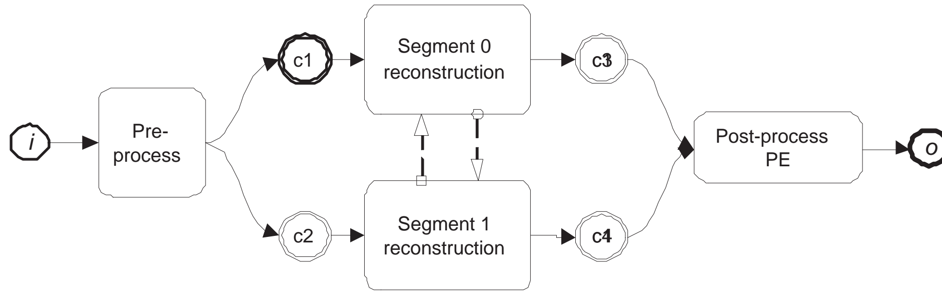

Figure 3.3: A Petri net description of the Tomosynthesis workflow application

determined at run time based on the user-specified condition.

In Section 4.4, we will describe the multi-task workflow of our targeted application, Tomosynthesis Mammography. In this section, we use the Tomosynthesis workflow as an example to illustrate a Petri net, as shown in Figure 3.3. In this figure, the application has four tasks, one pre-processing task, two concurrent tasks that are used for the core image reconstruction, and a post-processing tasks. The two concurrent tasks represent a cross-site parallel execution, while inter-task message passing is enabled.

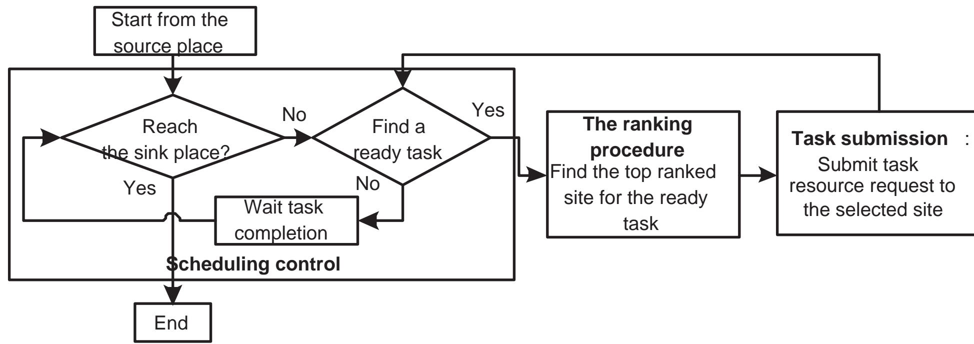

Figure 3.4: The function diagram of the workflow scheduler

3.1.2 Scheduler design

Given a workflow specified by a Petri net, the SGR framework scheduler is designed to submit each application task to the top-ranked site, while scheduling all tasks in the sequence based on the task dependency. We partition the functions of the scheduler into three parts: scheduling control, the ranking procedure, and task submission, as shown in Figure 3.4.

The scheduling control function locates tasks that are ready to be scheduled, while it traverses the Petri net starting from the source. A task is ready only when all tasks on which it depends have finished their executions. The scheduling control function relies on the resource management system (the batch job system) to query the status of submitted tasks.

For a given task, the ranking procedure selects a computing site with the smallest estimated queuing time based on a user-specified ranking criterion. In the ranking procedure, values of the ranking criterion are collected from computing sites. The top-ranked site is then selected for the task. Therefore, the ranking function relies on the query function for the specified ranking criteria, including the queuing length,

system throughput, and available resources.

Given a task specification, the task submission function first creates a job request based on the task's resource requirement. It then submits the job request to the selected site and records the submission's job identification, which is assigned by the resource management system to retrieve the state of the submission on the site.

Since our workflow applications target multiple computing sites of a computing grid, the workflow scheduler must interact with multiple resource management systems and job submission tools. If the scheduler is designed as a single process program running on a stand-alone host, remote queries and remote job submissions must travel over the public network connecting the geographically separated computing sites. These remote operations are slow and communication traffic is exposed to public, requiring additional services to support secured communication, interoperability and remote procedure calls for all the resource management systems. In comparison, local operations generated within the same computing site do not have such limitations. Therefore, the workflow scheduler shall avoid using the remote operations.

To address the above challenge, we designed a parallel scheduler which has multiple processes running in parallel on the headnode or the gateway node of each computing site. Each scheduler process only needs to perform local job queries and job submissions within the same site, while inter-process communication is enabled through encrypted message passing. As a result, the scheduler does not rely on interoperability of resource management systems to implement remote job queries and job submission. While we eliminate the remote operations, we can improve the scheduler's performance and scalability by using this parallelization approach.

To schedule a task, every scheduler process running in parallel must participate

in the scheduling process. We introduce a parallelized ranking procedure to select the computing site for the task, which will be discussed in the next section. After a computing site is selected, the scheduler process running on the headnode of the same site performs job submission and monitors its status until its completion.

The parallel scheduler must be started before scheduling tasks of a workflow application. Therefore, it takes two steps to launch a workflow application on a computing grid. First, application users start the scheduler on the headnodes of selected computing sites of the computing grid. Second, all scheduler processes running in parallel make scheduling decision collectively and submit user-specified workflow tasks. Figure 3.5 gives an overview of the two-step scheduling procedure from the perspectives of the application user and the task scheduler. An application user provides a list of computing sites and other runtime parameters to start the scheduler on a computing grid. The list of computing sites can be pre-selected by the application user. Some heavily-loaded computing sites and those that do not have qualified computing resources can be eliminated in the first step. The application user also provides the Petri net and all tasks' specifications of the application, including each task's input and output, which are used by the the SGR framework scheduler to create and submit job requests. How a task is scheduled and where it will be executed are hidden from application users.

Several requirements must be satisfied before an application task's execution can begin. The executable code of the task must be available on each candidate site, and input files for the task must be accessible within the execution site. The input files do not need to be transmitted to all sites before the user starts the scheduler, since it could be time-consuming to broadcast the input files to all sites. The scheduler can

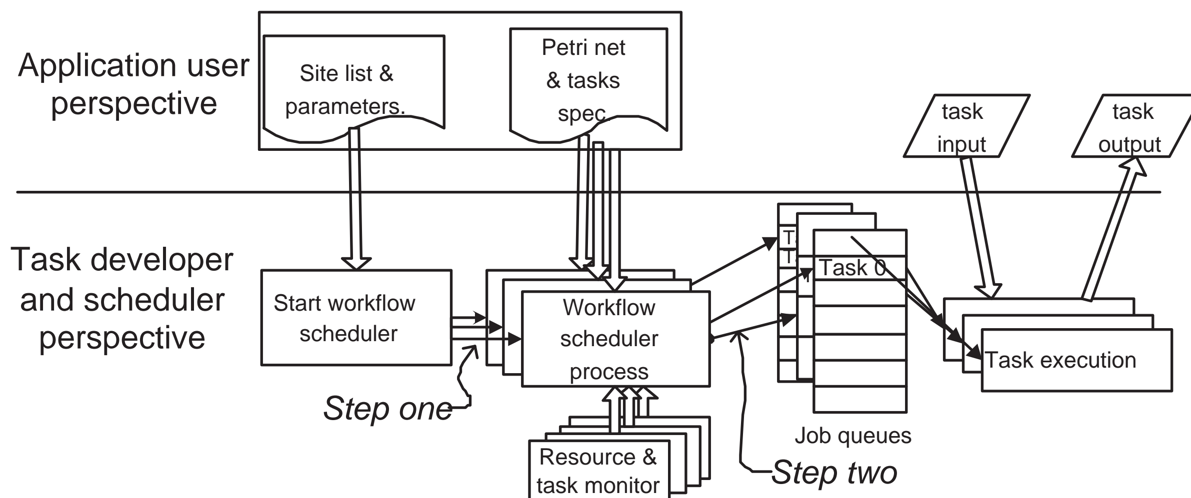

Figure 3.5: The overview of launching a workflow application using the SGR framework scheduler

create a file transmission task and insert this transmission task in the user specified workflow. This transmission task is placed before the application task, which needs the input files.

Our parallel design has several advantages over the conventional single-server approach. Because a workflow application targets massive computing resources on a computing grid, a single-server scheduler may not be able to quickly response to changes in all monitored computing resources, which are distributed over a wide area network. The status information about job queues and running tasks on each computing site must be detected in a timely manner so as to make accurate scheduling decisions. A single-server scheduler could experience large overhead in communication, because the number of connections increases with the number of queries, which could saturate the limited bandwidth rapidly. Furthermore, cross-site job submissions and cross-site queries take more time than a local job submission and a local

query. Therefore, the conventional single-server scheduler may not scale well with the number of tasks and the number of computing sites it monitors.

The parallel approach distributes all resource monitoring, job submission, and task management among multiple scheduler processes. Each scheduler process is responsible for task submission, system load-status querying, and checking the job status on its local computing site. A task's job request is submitted within its local batch job system, and no cross-site job submission is performed. A query of a local batch job system, using system-provided API functions directly, is faster than a cross-site query using web services which requires additional system configurations and may not be available in some computing grid environments.

3.1.3 Parallelization of the ranking procedure

The ranking procedure of the framework scheduler computes a ranking value for each computing site based on a user-specified ranking criterion. The *rank* of a computing site is the position in the ordered list of computing sites, which are graded using a specific ranking criterion. The ranking criterion can be the queuing length, system throughput, or the number of available resources within a computing site. A higher rank indicates that the site is more likely to provide the application-specified resources using less time than a site with a lower rank. After the ranking procedure completes, the scheduler process running within the top-ranked computing site submits the ready task to its local resource management system.

The framework scheduler is designed as a parallel program and runs on the headnodes of selected computing sites of a computing grid. The ranking procedure

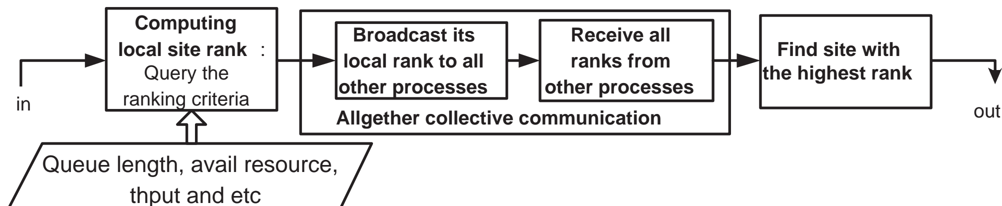

Figure 3.6: The parallelization of the ranking procedure.

performs the site selection for application tasks within the scheduler, and is also parallelized as a collective function, which requires all scheduler processes to participate. We illustrate the parallelized ranking procedure in Figure 3.6. First, each scheduler process queries the ranking criterion of its local resource management system, such as the queuing length or system throughput. An allgather collective communication operation is then performed to allow each process to have ranking values from all other computing sites. When the allgather communication completes, all the scheduler processes can sort all the ranking values to locate the site with the highest one. The scheduler process running on the top-ranked site then submits the application task to its local resource management system.

The input to our framework is a workflow application specified in a Petri net. Given an application task, the execution process of the task can be specified in a simple Petri net graph, as shown in Figure 3.7. We incorporate the task scheduling process, including the ranking procedure and task submission, into the application workflow, as illustrated in Figure 3.8. This method significantly simplifies the structure of our framework. Three system tasks, including the ranking, bypass, and synchronization tasks, are introduced in our framework to create the task scheduling workflow. These system tasks differ from application tasks because they perform

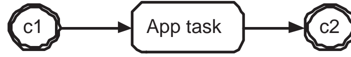

Figure 3.7: A single application task's execution specified in a Petri net.

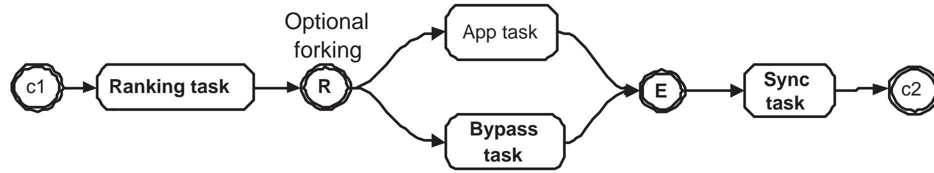

Figure 3.8: The task scheduling workflow for an application task.

specified functions and require no dedicated computing resources. System tasks encapsulate the specified functions, and therefore, they can be reused easily.

In the task scheduling workflow shown in Figure 3.8, the parallelized ranking procedure is executed by all scheduler processes as shown in Figure 3.6. The result of the ranking task determines which execution flow a scheduler process will take in the optional forking structure. The system bypass task, a no-operation function, is used to bypass an application task for those scheduler processes which are not selected to launch the application task. After completing the application task or the bypass task, all scheduler processes are synchronized by executing the synchronization task. This task is designed as a barrier, which makes sure that every scheduler process executes the same synchronization task and completes this task at the same time.

In this thesis, we only consider exclusive submission when scheduling multiple concurrent tasks. The scheduler submits each concurrent task from the same parallel forking structure to a different computing site. Each scheduler process maintains a record of application tasks which are submitted, but have not completed, within its computing site. If a computing site has an unfinished application task, the ranking

procedure will assign the site with the lowest ranking value during scheduling of concurrent tasks. Exclusive submission allows us to implement resource co-allocation and duplicated resource allocation without modifying the scheduler, since the two functions require that tasks are submitted to different computing sites. If we allow two or more concurrent tasks to be submitted to the same computing site, a sudden workload surge for the site could occur when users submit multiple concurrent tasks to the same site at the same time. To prevent this situation, the scheduler must implement additional routines to monitor every global job request from all users. As a result, the overall complexity of the scheduler could be increased dramatically.

3.2 Task grouping

In the previous section, we described the framework scheduler, which is used to schedule multi-task workflow applications and select a computing site for each application task. After a task starts to run, the framework must determine whether the currently allocated computing resources meet the user's resource requirement (i.e., the two resource boundary conditions). We introduce task grouping to implement this functionality. *Task grouping* is the process of acquiring computing resources from one or more tasks' resource allocations to satisfy the two resource boundary conditions. This process relies on the functions defined by the specifications of the resource assessment, and resource binding and coordination as described in Section 2.2.

In this framework, applications are required to participate the task grouping process. However, to achieve better adaptability, our framework must keep the application-side functions transparent to application developers. There are two com-

mon places to implement these functions in an MPI program: within the boot trap or the MPI initialization routine. The boot trap is a piece of code and it is inserted at the start of the application program to invoke multiple application processes. To add our task grouping functions to the boot trap, the host system's MPI compiler must be modified. On the other hand, adding task grouping functions to the MPI initialization routine can be achieved by recompilation of an application's program. We can replace the standard MPI initialization function by our customized initialization function, and use the MPI compiler to link the application code to our library. To achieve better portability, therefore, we implement the task grouping functions within our customized MPI initialization routine.

The process of task grouping may involve one or more concurrent tasks running from different computing sites. However, each task is treated as a conventional job request, and is submitted independently without any knowledge of other concurrent or duplicate tasks. Therefore, we must consider how to connect executions of different tasks.

Multiple executions are typically connected using the the client-server model: a public server is used to inter-connect independent clients. Because our execution environment consists of multiple administrative domains, a single process server may not be allowed to access internal computing resources of different computing sites. Furthermore, the structure of a centralized server does not scale well as the number of tasks increases. We introduce a parallel client-server approach, in which the task grouping server has multiple processes running in parallel and each server process is used to connect a client task running within the same computing site. Each server process must run on the headnode of a computing site, and can communicate with

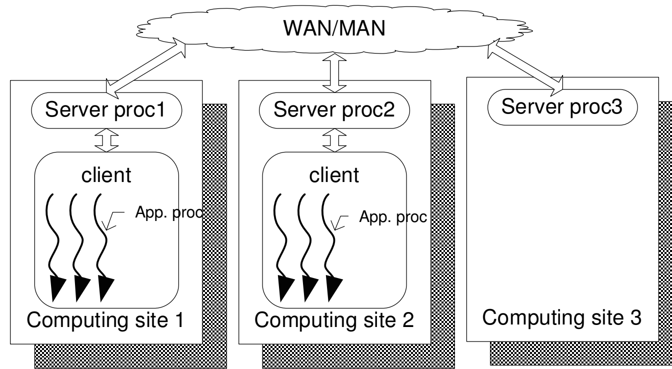

Figure 3.9: The parallel client-server approach used in task grouping.

one another over the public network which connects all computing sites. When a client task starts to run, it first connects to the grouping server process running on the headnode within the same site, as illustrated in Figure 3.9. The task grouping server maintains the global execution topology of all client tasks so that it can support resource assessment, resource binding and resource coordination.

Since a task is an MPI execution with multiple application processes running in parallel, only one process needs to communicate with the task grouping server during task grouping. We select the root process with rank 0 to perform the task grouping functions on the application side, and the remaining processes are stopped at a barrier until task grouping completes. A communication connection between a client process and its server process is created using a host-port connection pair, which is published to the server and the client task before the task is submitted to the local resource management system.

3.2.1 Task binding procedure

After a client task is connected to its local task grouping server process, the client and server start the binding procedure as shown in Table 3.1. The binding procedure uses the resource assessment function (defined in Section 2.2.3) to determine whether newly acquired resources of the client task can satisfy the application resource requirement. Given a client task, the resource assessment returns *Insufficient*, *Adequate* or *Redundant*, indicating more resources are needed, the requirement is met, or no need to use task's resources, respectively.

Before a client task sends the binding request, the task needs to update its local copy of the global execution topology and call the resource assessment function. If the resource assessment determines the task itself is redundant, the task exits the binding procedure and terminates its execution. If the resource assessment returns *Insufficient*, the task starts the task coordination procedure. A simple coordination method is to continue the binding procedure and wait for more resources which will be discussed in the next section. If the resource assessment returns *Adequate*, the task sends the binding request to its server process, and then checks the binding results by searching its local topology information in the latest global topology. In a successful binding, the task will find its local execution topology in the global execution topology. In a failed binding, the task will not find its local execution topology in the latest global topology, because the resource requirement has been satisfied and the server ignores its binding request. When the binding procedure fails, the client task can confirm the result by using the resource assessment function, which returns *Redundant* as the result.

	Client side	Server side
1	Query the global execution topology.	Reply the global topology query.
2	Receive global execution topology.	—
3	Perform resource assessment. * If *Insufficient*: start the task coordination procedure. * If *Adequate*: send the binding request with the task's local execution topology. * If *Redundant*: exit and terminate execution.	Receive task's binding request and its topology.
4	Check the binding result by searching its local topology in the latest global topology. * If found: return success. * If not found: return failure.	Perform resource assessment for the client task: * If not *Redundant*: add the task's local execution topology in the global topology. * If *Redundant*: ignore the request.

Table 3.1: The binding procedure performed on the client and server side

When we use concurrent tasks to duplicate resource allocations on multiple computing sites, there could be more than one concurrent task sending binding requests at the same time. Therefore, the task grouping server must perform the resource assessment function for each requesting task so that it can detect any redundant resource allocation. The grouping server will ignore a request if the resource assessment function determines that the requesting task is redundant.

Because the grouping server has multiple processes and each server process has a copy of the global execution topology, these copies must be consistent. A simple way to achieve this result is to allow one designated process to modify the topology. A single master server process is instructed to manage the global execution topology, including adding and removing tasks' local execution topologies. All binding requests are forwarded to the master server process and the requests are processed sequentially. The master server broadcasts the updated global execution topology to all other server processes after each modification.

Besides processing binding requests, the task grouping server also performs task removals and replies to queries of the global execution topology. After receiving a query request for the global execution topology, the server process running at the same site as the requesting client task replies to the task with its local copy of the global execution topology. The task removal request is processed in the same way as the task binding request. Like the opposite process of task binding, the removal request is forwarded to the master server, and the updated topology is sent back to all server processes. The requesting client task only needs to check the global execution topology to determine whether its request has been processed or not.

3.2.2 Task coordination

Task coordination is the process of synchronizing resources which are allocated by multiple task requests to meet the two resource boundary conditions for a cross-site execution. The SGR framework currently supports the most basic coordination function: stop-and-wait.

In the stop-and-wait coordination procedure, after a client task becomes a bound task, it repeatedly queries the global execution topology until the resource requirement for the cross-site execution is met. We implement this function within the MPI initialization routine and place it after the binding procedure.

The stop-and-wait procedure cannot guarantee success for all cross-site resource allocations. While a client task is waiting for more resources to be bound to satisfy the resource boundary conditions, the running task is also consuming the computing time of its allocated computing resources. If the stop-and-wait procedure uses all the available computing time during synchronization, the task execution will be terminated by the local resource management system. This situation is more likely to occur when computing sites have unbalanced workloads. To avoid this problem, application users can extend the estimated execution time limit for concurrent tasks, and use task duplication to allocate resources from more computing sites.

The problem described above is a result of the fact that resource management systems cannot differentiate synchronization times from task execution times. However, our SGR framework can distinguish them, since it knows the status of the currently allocated resources and the application resource requirement for the cross-site execution. Therefore, the two other task coordination methods, which are resubmission

and hibernation, can be used to solve this problem. If a parallel task cannot bind enough resources after a certain time, the task can either be hibernated and woken up when a concurrent task is bound, or terminated and resubmitted back to the job queue. Thus, it limits the consumption of resource time in waiting.

Task grouping allows our framework to handle varying resource availabilities across computing sites, since system loads cannot be accurately predicted at job submission time. Using binding and coordination, we dynamically group the earliest available resources to execute user-defined applications. Binding and coordination, performed at runtime, allow applications targeting computing grids to allocate the required computing resources in the shortest time without relying on any privileged-management functions across different administrative domains.

3.3 Message Relay

In previous sections, we described a workflow scheduler which launches a multi-task workflow application in a computing grid environment, and task grouping which allows users to acquire computing resources across multiple computing sites at runtime. When an MPI parallel execution is launched across multiple computing sites, messages sent between processes must be allowed to cross between computing sites. Computing sites participating in a computing grid may be owned and managed by different institutions, and each site maintains its own individual private networks or LANs. As we pointed out in Section 1.5, communication between computing resources in two different computing sites must be rerouted through the gateway node or the headnode of each site. Because our framework transforms a multi-site MPI

	Public-IP	NAT	System level	User level
Commu. layer	IP	IP routing	Globus-IO/XIO	MPI
Commu. overhead	Low	Low	Medium	Large
Imp. effort	Low	Moderate	High	Moderate
Hardware cost	High	Low	None	None
Portability	Low	Low	Low	High
Access ctrl	Complex	Complex	Moderate	Least
App. recompile	No	No	Yes	Yes

Table 3.2: Comparisons of cross-site communication methods

execution into multiple concurrent tasks, inter-task message passing must be enabled while hiding the heterogeneous network environment for application developers. This section analyzes different approaches to enable cross-site communication, and we will introduce MPI-compliant message relay.

3.3.1 Inter-site communication analysis

One of our objectives is to develop a portable system for inter-task message passing across different computing sites, while hiding physical details of heterogeneous networks from application developers. In searching for such a highly portable and transparent solution for applications targeting the diverse grid environment, we first compare and analyze some existing approaches, which are designed at different levels. A summary is shown in Table 3.2.

Assigning public-IP addresses to all internal computing resources of all computing sites is a network approach at the physical level that allows external computers to

directly communicate with any internal node. Therefore, each internal computing resource of a computing site can communicate with any internal resource of a different site. However, this approach is often impractical because of security concerns, since all internal resources are exposed to the public network. Furthermore, it is an expensive approach because IP resources are limited. Only a few computing grids, such as TeraGrid [5], adopt this approach.

An alternative way to enable communication cross different computing sites is to enable IP packet-routing across different private networks by using the network address translation (NAT) [46] of computing sites. NAT uses port numbers on the headnode of a computing site to map its internal computing resources. IP packets with a port number of the destination site can be routed through the headnode to the destination computing resources. No modification of system communication libraries is required for this approach. However, the mapping between internal computing resources and port numbers on each site must be maintained by system administrators. Like the public-IP approach, NAT requires stronger security protection since all internal computing resources are exposed to the public Internet through their port numbers on the headnode.

Instead of using the conventional TCP/IP system communication library, a system level approach creates a customized communication system to enable cross-site data communication. Some research projects, such as MPICH-GX [14], are investigating a proxy-like server running on a public host to route communication across computing sites based on Globus-IO/XIO to achieve secured communication over a public network. However, this system level approach requires contributions from three parties to implement the cross-site communication. First, system developers must create

a reliable and robust customized system communication library and communication service at the network layer to enable network routing through the headnode of a computing site. Second, system administrators are required to install such a system and maintain those additional services. Third, applications must be rebuilt to allow the new system library to take effect. Therefore, both its development and deployment are complex and time consuming.

A user level approach employs a dedicated application program running as an agent to route cross-site communication between tasks. Running at the user level, the agent program intercepts and passes cross-site messages. This method requires recompilation of application programs and computing resources on the headnode. The user level agent is portable to many different platforms and can be installed without modifying local system settings or network configurations. Some performance loss is expected, because additional layers of system calls which disassemble and reassemble communication data are used in the routing procedure. When running at the user level on a busy headnode, the system could experience longer delays due to resource contention in a shared environment. On the other hand, a user level approach has a higher portability and is easier to be deployed without system reconfiguration.

Since one of our objectives is to develop a portable and transparent framework for a diverse grid environment while limiting host system and application modification, we adopt the user level approach to enable inter-task message passing. This approach allows parallel scientific applications to run seamlessly on multiple computing sites within a grid environment.

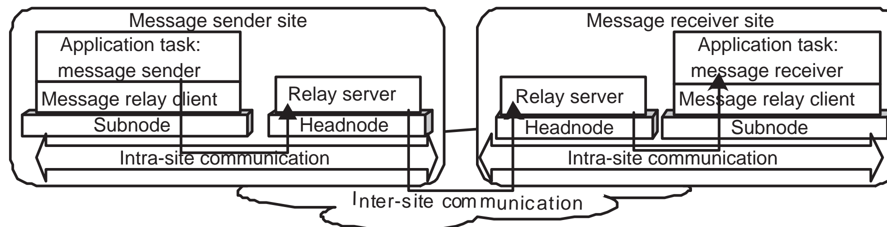

Figure 3.10: Illustration of 3-hop message relay

3.3.2 3-hop message relay

We developed 3-hop message relay to enable message passing between tasks. This method uses the parallel client-server approach, similar to the one used in task grouping, where the client is an application process requesting cross-site communication. The message relay server is a parallel program based on MPI, which is independent from the user application, and runs on each computing site's headnode with a public-IP address. Because all server processes run on headnodes with public-IP addresses, they can communicate with one another directly via the standard MPI communication routines.

During the process of the 3-hop message relay, each inter-site message travels three hops, or three continuous point-to-point communication operations. The first hop is from the sender process to the server process running on the headnode within the message sender site. The second hop is from the headnode of the sender site to the headnode of the message receiver site. The final hop is from the headnode to the receiver process within the receiver site. The three hops of the inter-site message passing between two computing resources across different computing sites are shown in Figure 3.10.

For every inter-site message, the sender client attach a message envelope to the

application's message data. The message envelope consists of the source process rank, the destination process rank, a message tag and a communicator to differentiate the inter-site message from other messages. The inter-site message data and its message envelope are redirected to the sender's local message relay server process running on the headnode. The receiver client also sends a receiving request, including the message envelope and a unique return-tag, to its local message relay server. Communication of all these intra-cluster messages is designed based on the standard point-to-point MPI routines.

Next, the sender's relay server forwards the message data and its message envelope to the receiver's relay server. Because server processes run on the headnode within the same MPI execution, communication between server processes is implemented using non-blocking message passing defined by the MPI standard.

Upon receiving an inter-site message with its message envelope, the receiver's relay server searches the request queue which stores all the inter-site receiving requests. When the inter-site message envelope matches a receiver's message envelope, the server process sends the inter-site message data to the message destination. This last hop of inter-site communication differentiates itself from other inter- or intra-site message heading to the same receiver by using the unique return-tag which is specified in the receiving request.

Our 3-hop message relay requires that all server processes run on the headnode of each computing site with a public-IP address. A disadvantage of this method is that extra resources are needed to run the relay server processes. However, the server is a lightweight process; little computational work is involved in the server program. The server only communicates with local client processes and with other server processes.

Communication between server processes on the public network is encrypted and secured.

Since message relay requires three message passing steps to complete cross-site communication between two tasks, performance loss is expected for small messages. Cross-site communication between computing resources with public-IP addresses, or hardware-enabled rerouting, will have lower communication overhead, but they require system or network support. Since our design adopts the user level approach, it is more portable, and no source code modification is required for applications to run on different networks in a computing grid environment.

3.4 Framework integration

Previous sections describe three functional components: the workflow scheduler, task grouping and 3-hop message relay services. Each of these components is designed for a specified service function: managing workflow execution and scheduling workflow tasks, inter-connecting parallel tasks and resource assessment, and enabling inter-task message passing between concurrent tasks. Separate implementations for these components would increase the overall system complexity, making the system hard to implement and difficult to use. Therefore, we integrate all the components in one framework.

Because the task grouping and message relay services use the same parallel client-server design, they can share the same client and server implementation code. The workflow scheduler, on the other hand, consisting of server-based functions, is only implemented within the server. Figure 3.11 illustrates the integration of the three

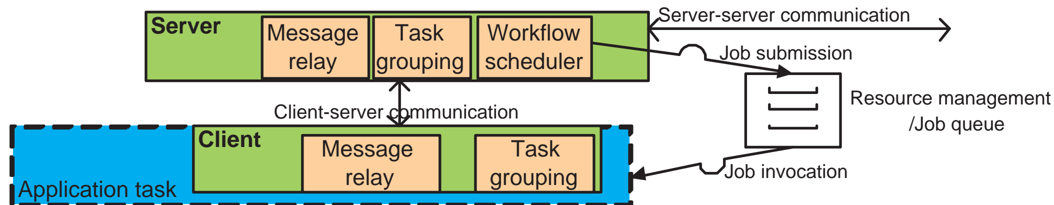

Figure 3.11: The framework integration of three functional components in the client-server structure.

components within the same client-server structure. The server is a parallel program which is dedicated to performing the scheduling, grouping and message relay services. Server-server communication is required for the server processes running in parallel on headnodes of selected computing sites to perform services, including site ranking, task binding and message relay. The client is an application library, supporting message relay and task grouping. This client library, which is built within the application program, provides an interface that complies with the MPI standard, so no source code modifications within application programs are needed.

On the server side, service functions of the three components need to share some information and data structures. For example, the two resource boundary conditions are used in specifications of workflow tasks, and are also used in the resource assessment function during task binding. Task binding defines and modifies the global execution topology which is used during inter-task message routing.

Figure 3.12 shows a deployment of the framework for a workflow application with two concurrent tasks running on a computing grid. The server program is launched on the headnodes of computing sites of the grid so that server processes can communicate based on the public network, while each server process only monitors its local workload

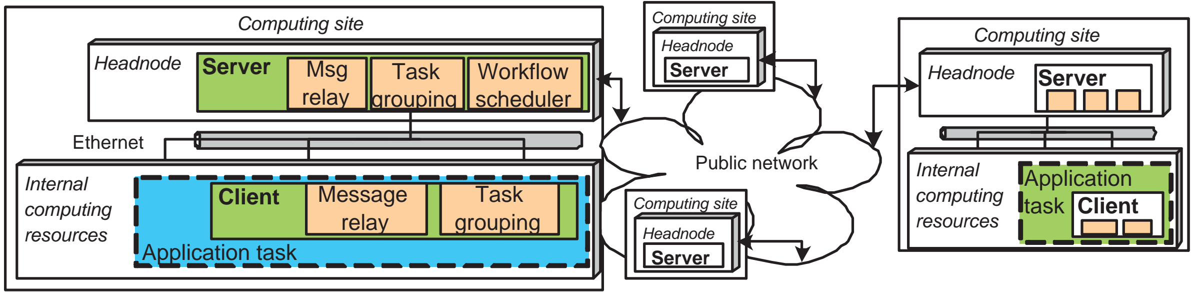

Figure 3.12: The deployment of the framework with workflow scheduling, task grouping, and message relay on a multi-site computing grid.

condition and task executions. After an application task is scheduled and starts to run on a computing site, client-server communication takes place between the application task and its local server process which is running on the headnode at the same computing site.

Figure 3.13 gives an overview of the layered structure of the integrated framework in the execution environment. The SGR framework is located between the application layer and the system and computing grid infrastructure layer. The top layer is the application user layer, which specifies the input information, including candidate computing sites for execution and a workflow application with all tasks' specifications.

Below the SGR framework layer, the system and grid computing infrastructure provides required communication and resource management modules. The resource management module interacts with the scheduling service, providing resource management functions for the framework to manage application tasks' submissions and query the host job queue status. Three communication modules are required:

1. The intra-site application communication module: allows processes of an appli-

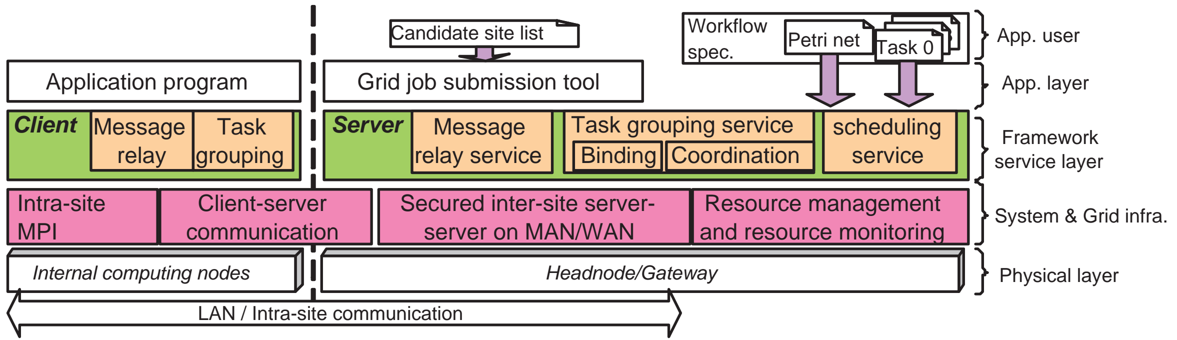

Figure 3.13: The structure of the integrated framework and its execution environment

cation task running in parallel to send and receive messages from one another within a computing site.

2. The client-server communication module: allows an application task to communicate with its server process running on the headnode within a computing site.

3. The inter-site server-server communication module: provides secured communication for server processes running on the headnodes of computing sites.

The intra-site and client-server communication modules rely on vendor-provided communication libraries for a high performance, which are optimized for the host systems. The inter-site server-server communication requires data encryption, since its communication data is exposed on a public network.

The SGR framework, with integrated scheduling, grouping and message relay services, is capable of launching applications on a heterogeneous computing grid. All SGR functions are based on the common system functions which are available on most conventional computing platforms. Without requiring system modifications or any advanced feature to support cross-site resource allocations or cross-site commu-

nication, the SGR framework is easily portabile to different computing platforms. Our SGR framework supports applications complying with MPI, the de-facto parallelization standard, requiring no source code modification to use the system. The framework can launch applications involving multiple executions sequentially or concurrently. Furthermore, application users can use the duplicate resource allocation model to reduce the resource allocation time on a computing grid. Therefore, our framework is adaptable to legacy MPI applications and complex workflow applications, while utilizing resources of computing grids more efficiently.

3.5 SGR framework implementation

In this section, we present our system development platform and how we implement the SGR framework, including communication, interface, and adapters.

We use the Globus Toolkit (GT) as the computing infrastructure for our implementation of the SGR framework. The Globus Toolkit is supported by various computing platforms and has become a popular computing grid infrastructure due to its collection of software and tools required by many applications targeting computing grid environments. Since many computing platforms have installed the GT, our framework implementation can be installed on these systems without additional dependence on system modules. Furthermore, successful deployments of our SGR framework could make it possible for the SGR framework to be included in the GT's future releases.

Figure 3.14 illustrates the structure of the SGR framework services, interface, and dependent system modules based on our development platform. In Figure 3.11 of the

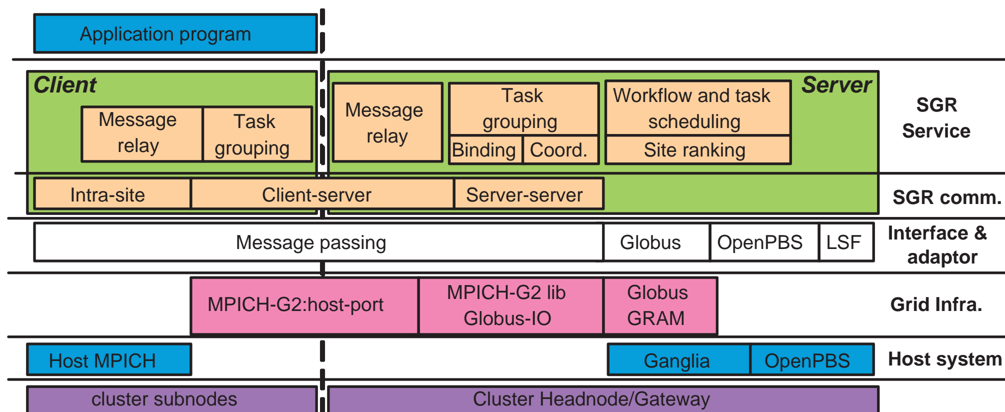

Figure 3.14: The structure of the SGR framework implementation

previous section, we showed that there are three types of communication required by the SGR framework: intra-site communication, client-server communication, and secured inter-site server-server communication. Figure 3.14 illustrates our implementation of these three types of communication, corresponding to Figure 3.11 at the system and grid infrastructure layer.

Intra-site communication for application tasks is based on the standard message passing interface. The client-side communication library of our SGR framework is compiled with the MPICH library installed on our host platforms. When an application task program calls a standard MPI routine for intra-site communication, our SGR framework will invoke the corresponding routine in the host MPICH library. This process is transparent for application developers, and it allows the SGR system to be ported to other vendor-provided MPI implementations.

To establish client-server communication, we use the standard TCP/IP socket connection procedure. A server process requests a host-port pair from its host system and passes this information to the task execution via an environment variable which

is specified in the task invocation script of the resource management system. After the task starts to run, it can read this environment variable to get the connection host-port information. The task then sends the connection request using the host-port socket within our customized MPI initialization routine. After the server process accepts the client connection request, the server and the client task can communicate with one another.

Several software systems implement the socket host-port connection to enable communication between different programs, such as the operating system's TCP/IP library and the Java VM's communication library. We use the MPICH-G2 implementation [33] for our SGR system. MPICH-G2 provides the host-port connection function as part of the process management defined in the MPI 2.0 standard. After the connection is built, MPICH-G2 allows client and server processes to communicate using the standard MPI routines, making it easier to implement the client-server communication. Using the standard message passing interface to implement client-server communication, we can avoid lower-level issues, such as data packet assemble and disassemble, data integrity checks, and memory management.

Because the server program is designed as an MPI application, server-server communication is implemented using MPI communication routines. However, our server program is different from conventional MPI applications which could have synchronized inter-process communication behaviors. The server is a dedicated program, designed to provide fast services for all requests. However, service requests and their communication cannot be scheduled ahead of executions. Unlike conventional paired send-receive programming, the server adopts the probe-receive approach to receive incoming messages from client processes, as well as from other server processes. Ev-

ery server process periodically probes for incoming messages. When a server process detects an incoming message, it starts the receive routine. However, the SGR system's performance is closely related to the frequency of probing. We will evaluate this performance issue in the SGR cross-site communication next chapter.

A server process must be capable of processing multiple message relay requests at the same time so that it allows multiple client processes to send or receive inter-site messages. We implemented communication on the server side, for both client-server and inter-site server-server communication, using message passing in the non-blocking mode . Therefore, data sending or receiving between server or client processes is handled in the background by the operation system, while the server process can monitor all communication operations in a loop.

As shown in Figure 3.14, all three types of communication comply with the MPI standard. Using the MPI standard simplifies development, because the standard hides most lower-level system details. Furthermore, our implementation can achieve better portability, since MPI is installed on most computing platforms. On the other hand, a disadvantage is some performance loss because we are using a high-level communication protocol. The performance loss is a result of additional nested procedure calls within the MPI implementation, and the fact that we are unable tune the client-server and server-server communication for higher performance at the low-level communication library. We will address this issue in the next chapter by analyzing test results of the message relay service in a two-cluster grid environment.

Because a wide range of resource management systems can be deployed on a computing platform, we create adapters to allow the SGR framework to interact with some RM systems. As shown in Figure 3.14, we implemented adapters for

Globus, OpenPBS, and LSF systems within the current release of the SGR system. However, other RM systems can be supported by adding their adapters within the SGR framework.

Although the Globus Toolkit provides an unified job management interface which is built on local resource management systems, we still implemented adapters to different resource management systems to avoid overhead introduced by Globus. The Globus Toolkit's RM interface and its supporting functions are designed for remote job management across different administrative domains. As a result, significant overhead is introduced for authentication, authorization, and data encryption, when performing job management over a public network. Our parallel server design eliminates the need for remote job management, and connecting the native adapter directly to the local RM system for job submissions and queries can yield much faster responses.

In this chapter, we describe how the SGR framework is implemented. The SGR framework, integrated with the three services, allows us to hide different administrative domains, heterogeneous networks, and the multi-task workflow execution for application developers and users. To address the challenges described in the introduction, we designed and implemented the SGR system which is scalable with the size of application specified computing resources, portable to heterogeneous computing platforms, and applicable to common MPI applications. In Chapter 5, we will evaluate our implementation of the SGR framework on a two-cluster grid.

Chapter 4

Motivating Application: Tomosynthesis Mammography

This chapter presents the parallelization work of Tomosynthesis Mammography application, one of our motivating applications. Tomosynthesis Mammography is being used in clinical trials at Massachusetts General Hospital (MGH). In this chapter, we first describe the Tomosynthesis Mammography application, and the parallelization of its core image reconstruction algorithm. We then evaluate the performance of three parallelization methods. The results show that we can achieve a 10 times speedup on a 16-node cluster from the parallel execution as compared to the sequential execution. Finally, we illustrate how the application is executed in a workflow, and conclude that further performance improvement requires more computing resources on a computing grid.

4.1 Introduction to Tomosynthesis Mammography

Mammography is currently one of the most popular techniques used in the detection of breast cancer. A digital mammogram is a digital image produced by a distribution of X-ray attenuation through breast tissues. However, structural information in a three-dimensional (3D) volume may be lost when the object is projected onto a two-dimensional (2D) plane using the conventional mammogram technique. Two-dimensional projections can be misleading due to overlapped and superimposed structures inside the object. Therefore, overlapped tissues inside a breast can obscure a cancer, which causes over 30% of the cancers to be missed when using traditional mammographic techniques [35]. Superimposed normal tissues can sometimes look like an ill-defined tumor in a two-dimensional mammographic image. This causes a large number of patients to be called back to the hospital for additional exams.

Tomosynthesis creates layered 2D images of a 3D object, which provides structural information associated with the object [25]. Based on a set of discrete X-ray projection images which are obtained at different angles while the X-ray detector is held stationary, Tomosynthesis reconstructs the internal structure of the object. Tomosynthesis mammography creates breast images in three dimensions based on multiple mammograms. The goal of this technique is to address the breast tissue superimposition problem [42, 48, 47, 49].

Tomosynthesis mammography has been under investigation at (MGH). A total of 11 to 15 X-ray digital mammograms are acquired by rotating the X-ray source around a patient's breast through a range of 50°. After the X-ray projection images are acquired, Tomosynthesis uses an image reconstruction algorithm to recreate the 3D

volume of the breast and to enhance the visibility of features within the volume, which aid doctors and physicians in the detection and diagnosis of tumorous tissue. The reconstructed 3D images have proven to be effective in distinguishing the overlapped tissues of a breast in a clinical study at MGH [49]. Tumorous tissues that were missed using traditional mammography due to overlapped tissues can be clearly identified in the layered Tomosynthesis images.

In the field of medical imaging, high quality and high-resolution image reconstruction are always demanded. A high-definition X-ray detector panel used by the prototype Tomosynthesis system at MGH is 1900×2304 pixels. A single 3D image reconstruction can consume over 500MB of disk space, over 2GB of memory space, and more than several hours of execution time on a 2.5GHz Pentium 4 workstation. We expect the execution time, memory allocation and disk space will increase significantly, when new panels with higher resolutions are used and the reconstruction process is reconfigured to create smaller voxels (volume pixels). Furthermore, a large number of clinical cases that require Tomosynthesis reconstructions would easily overwhelm high-performance workstations. The long reconstruction time imposes a significant impediment and slows down the research, clinical trails and the practice of the Tomosynthesis. We therefore investigated ways to parallelize the application to reduce its runtime.

The core image reconstruction procedure of Tomosynthesis uses the iterative maximum likelihood (ML) algorithm [49], which is highly computation-intensive and time-consuming. The ML algorithm is illustrated in Figure 4.1. Given an initial guess of the 3D volume, the iterative reconstruction procedure corrects the previous estimation until the result closely resembles the original 3D object. Each iteration proceeds

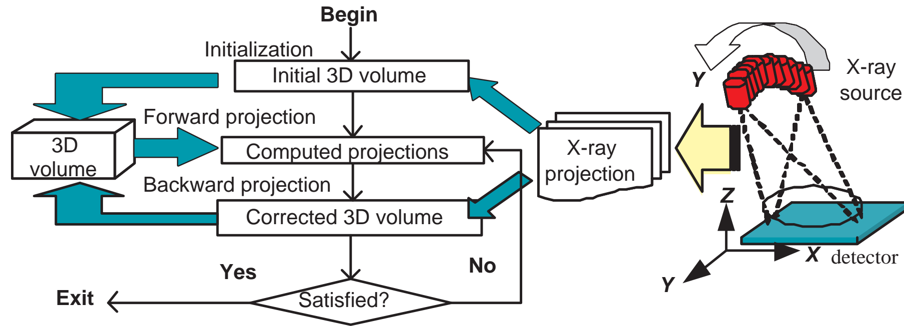

Figure 4.1: Tomosynthesis image reconstruction algorithm.

in two phases. The first phase is the forward projection, followed by the backward projection. Given a 3D volume, the forward projection simulates how x-ray beams are absorbed when they travel through the object, and creates the estimated projection images. During the backward projection phase, each volxel in the 3D volume is corrected based on differences between the estimated projections and the actual x-ray projections. Structural features within the 3D volume are strengthened after multiple iterations. This process typically takes 8 to 10 iterations. The complexity of ML estimation increases linearly with the size of the detector panel, the number of reconstruction iterations, and the depth of the 3D volume.

4.2 Parallelization of Tomosynthesis

The core reconstruction procedure is time-consuming. To improve the performance of the ML reconstruction, we can partition the reconstruction work by dividing the 3D volume into multiple segments, and distribute the reconstruction of segments to different computers. Therefore, we can reduce the reconstruction time by parallel computing.

The core reconstruction procedure requires large amounts of memory spaces. Our profiling study of the sequential Tomosynthesis reconstruction reveals that $40 - 60\%$ of the execution time is the consequence of cache misses and page faults. Data partitioning can contribute to less cache misses and page faults, leading to performance improvement. Thus, we partition the 3D volume into multiple segments, which are processed in parallel on multiple computing resources. At the end, all reconstructed segments are collected to reassemble the complete 3D volume of the final reconstruction result. However, we need to investigate a proper partitioning method, which can expose maximum parallelism and limit the amount of data that must be exchanged between processing nodes.

During the ML reconstruction, the forward projection simulates how X-rays are absorbed, and the backward projection corrects estimated 3D volume by comparing simulated projection results and the actual projection images. Then, the newly corrected volume will be used in the next iteration of the ML reconstruction. In Figure 4.2, as an X-ray beam travels through volxels of the 3D volume and projects to a pixel on the detector panel, data associated with these volxels and the pixel are referenced with one another during the reconstruction. If a partition plane cuts X-ray beams in two neighboring segments, data dependencies exist between the segments. Therefore, we must align partition planes with the X-ray beam's direction, so as to reduce the number of data references between neighboring segments.

Tomosynthesis acquires multiple projection images while the X-ray source shifts over 50° on the pivot point in the Y-axial direction (see Figure 4.3). When the X-ray source rotates, there is more than one X-ray beam traveling through a volxel at different angles. If a partition plane is also aligned with the Y-axial direction, then we

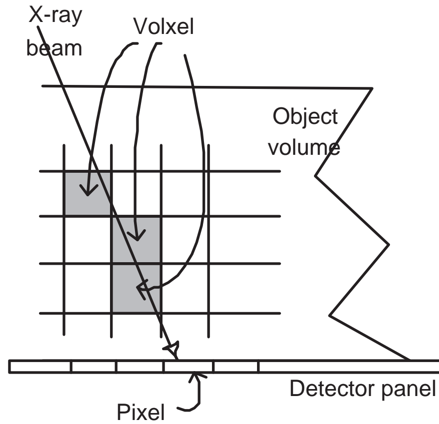

Figure 4.2: Data dependencies among volxels of the 3D volume and the corresponding pixel on the detector panel.

can expect that the smallest number of X-ray beams travel across the partition plane, which results in the fewest cross-segment data references. We choose partition planes aligned with the X-ray beams when the X-ray source is positioned at the highest point and the Y-axial direction, as shown in Figure 4.4, which is viewed in the Y-axial direction. In Figure 4.4, the 3D volume is equally partitioned among multiple processes, as each layer of the 3D volume is partitioned evenly along the Y-axis.

However, our partition method cannot eliminate all data dependences between segments, as illustrated in Figure 4.5. Because the X-ray source shifts, some X-ray beams will travel through a partition plane. To solve this problem, after partitioning a 3D volume in multiple segments an extended region is attached on each partitioning side of a segment. Data in the two extensions of a segment, which has the same value as the corresponding volxel in the neighboring segment, is necessary for the reconstruction process of the segment. The size of an extension can be pre-calculated before reconstruction because it only depends on the geometry of the segment.

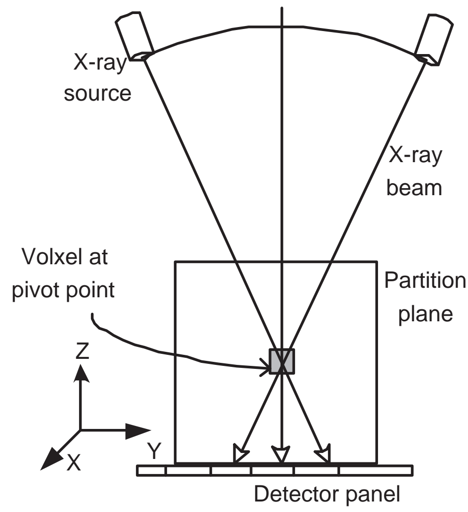

Figure 4.3: Image acquisition while X-ray rotates on the pivot point

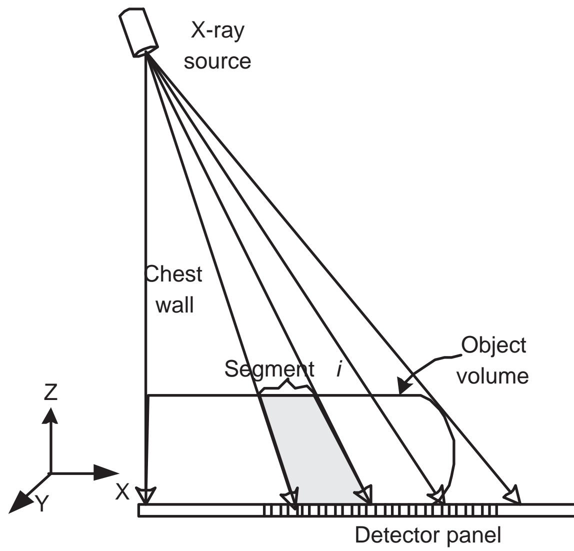

Figure 4.4: 3D volume segmentation method viewed at the Y-axial direction.

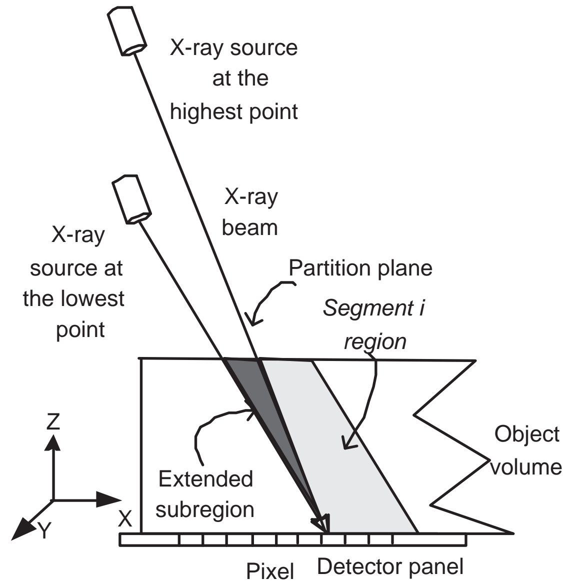

Figure 4.5: The extended region of a segment.

After one iteration of reconstruction, data in the extended regions of all segments must be updated. To update extension regions of a segment, we introduce three methods:

- Non-intercommunication method, which computes both extension regions locally.
- Overlap with intercommunication method, which computes part of the extension locally and transfers the rest from neighboring segments.
- Non-overlap method, which transfers data in both extension regions from neighboring segments.

The three methods differ from one another in how extended regions are updated: by computation, by inter-process communication, or both.

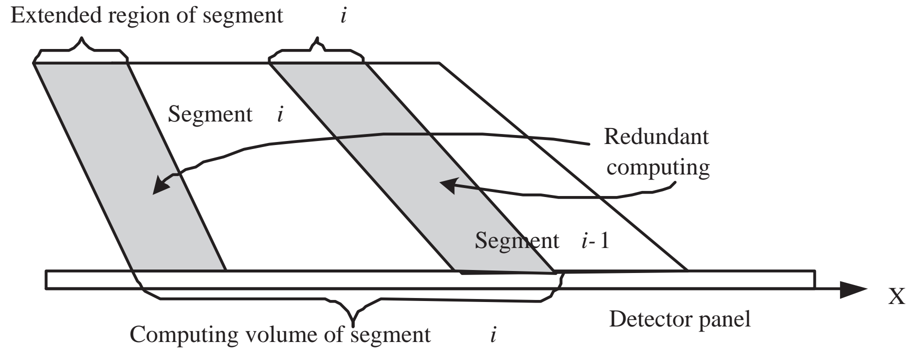

Figure 4.6: The non-intercommunication parallelization method

4.2.1 Non-intercommunication

The non-intercommunication method computes two extended regions for each 3D segment, as shown in Figure 4.6, instead of acquiring them from neighboring segments. This method treats the segment and its extended regions as one consolidated object, except that its geometry is based on the partition location within the original object. After all reconstruction iterations are complete, all reconstructed segments are reassembled and data in extended regions is ignored. This method eliminates all inter-process communication during the reconstruction and allows each partitioned segment to be reconstructed independently. This is a coarse-grained approach which can be implemented using task-level parallelization and deployed on most distributed systems.

During each reconstruction iteration, the reconstruction of each 3D segment and its extended regions are independent from computations in other segments. The results reconstructed by the non-intercommunication method differ from the sequential algorithm. However, research has shown that there is no significant image quality difference between the reconstructed 3D images of the non-intercommunication and the

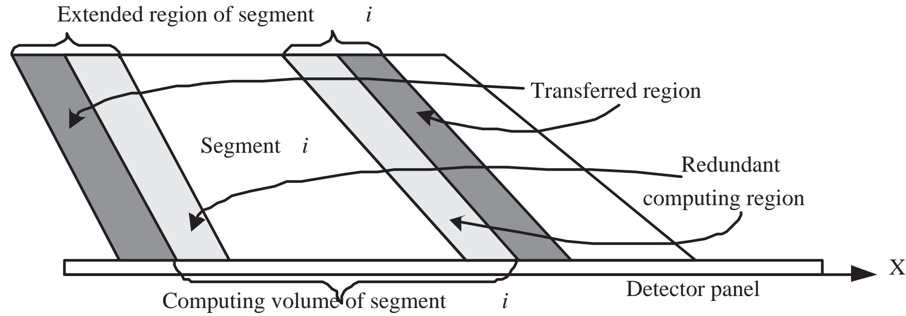

Figure 4.7: The overlap with intercommunication parallelization method.

non-parallel reconstruction [50]. All major tissue features and structural information can be clearly observed. Because two extended regions must be computed with each 3D segment, this method increases the computational workload per process. The extended regions are overlapped with neighboring segments, and redundant computation is introduced. Such redundant computation requires extra computing time and may lower the overall performance when the platform has a slower processing speed.

4.2.2 Overlap with intercommunication

The second method reduces the amount of redundant computation by copying the data from corresponding neighboring segments, as shown in Figure 4.7. Each segment obtains part of its extended region from the corresponding neighboring segment after each iteration of the reconstruction is completed. This method allows the rest of the extended region to be computed locally.

Overlap with intercommunication does not require to compute the complete extended region. However, it introduces communication and synchronization overhead. After forward and backward projection computations are completed, segments ex-

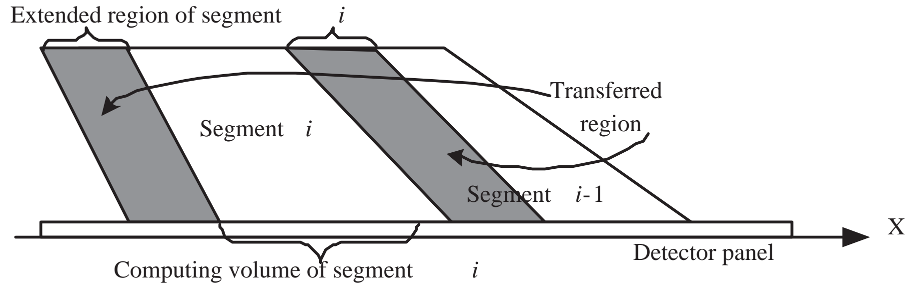

Figure 4.8: The non-overlap parallelization method

change the updated data with their neighbors. The advantage of this approach is that the size of data exchanged between neighboring segments can be fine-tuned to adapt to the network speed. This method allows users to optimize performance on different systems by changing the amount of data transferred between segments.

4.2.3 Non-overlap

The third method eliminates all redundant computation when computing the extended regions. In the non-overlap method, each segment obtains the entire extended regions from its neighboring segments. After each iteration completes, data in the extended regions is transferred from corresponding segments, as shown in Figure 4.8.

The non-overlap method does not have redundant computation, but it has the highest communication overhead. The amount of data exchanged between neighboring segments depends on the thickness of the object and the geometry of the partitioning. Generally, over 10MB of data must be transferred between two neighboring segments. Unlike the first two approaches, the non-overlap method produces the exact same image as the serial execution.

In summary, non-intercommunication is a coarse-grained parallelization approach. It requires less implementation effort, since there is no inter-processor communication performed during reconstruction. Furthermore, its performance does not rely on a high-performance network. However, redundant computation is wasteful, and the performance of this implementation is determined by the processor speed. The non-overlap method, on the other hand, eliminates all redundant computation while requiring communication and synchronization between all neighboring segments. Its performance depends on both the processor speed and on the network bandwidth/latency. In contrast to the first method, non-overlapped uses fine-grained parallelization and is much more difficult to implement. The overlap with intercommunication approach provides a solution that could be more adaptable to platforms than the other two method.

4.3 Parallelization performance comparison

We implemented the three parallelization methods using MPI [51]. Our parallelized code is based on the serial implementation of Tomosynthesis Mammography in C++ developed at MGH [49]. Our parallel implementation can be compiled for two different operating systems, Linux and UNIX (IBM AIX), and can be executed on the five parallel systems shown in Table 4.1. The platforms tested in our experiments range from a low-end Pentium 4 cluster to high-end shared memory supercomputer.

We use a phantom data set for our performance tests, which is also used in the Tomosynthesis image quality control. The size of our image is 1600×2304 pixels. The size of our 3D volume is $1600 \times 2304 \times 45$ pixels, where the thickness of the

	Processor	Interconnection
Intel P4 Cluster	2.5 GHz Pentium 4	100 Mb/s Ethernet
UIUC NCSA IBM p690	1.3 GHz POWER 4	Gbit Ethernet shared memory system
UIUC NCSA Intel Titan cluster	800 MHz Itanium-1 dual-processor	Gbit Myrinet shared L3 cache
SGI Altix 3300 shared memory system	1.3 GHz Itanium-2 dual-processor	NUMA-link interconnect shared memory system
U. of Michigan CAC Hypnos cluster	1.7 GHz Athlon 2000MP dual-processor	Gbit Myrinet

Table 4.1: Processor and interconnection network specifications of testing platforms.

phantom is $45mm$. In experiments from clinical trials, the parallelized ML algorithm is configured for 8 iterations. The length of the extension in the X-axial direction is pre-calculated, and varies from 15 to 20 pixels.

Figure 4.9 shows performance results for the parallel ML implementations running on 4, 8, 16, 32 and 64 processors of the Titan cluster at the National Center for Supercomputing Applications (NCSA) at the University of Illinois. We observed very similar speedup trends for all three methods. In most tests, the non-overlap outperforms the other two methods by 200 to 300 seconds. When running on 64 processors, the non-overlap approach uses only half the execution time of the non-intercommunication method. These tests demonstrated that the Tomosynthesis reconstruction task is dominated by computation, and that redundant computation increases the amount of computation.

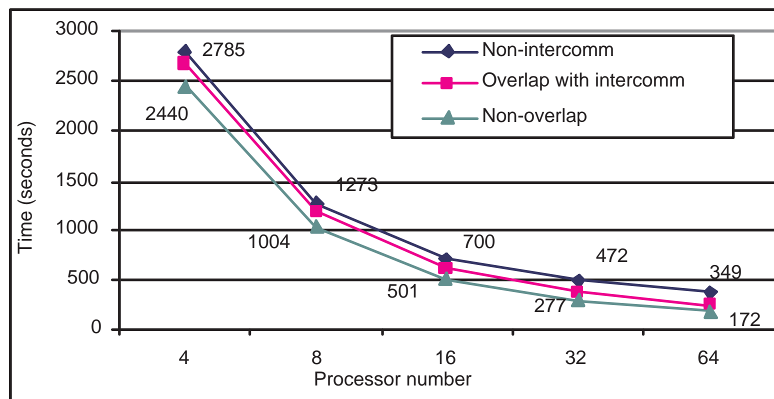

Figure 4.9: Performance of three implementations on number of processors, using UIUC NCSA's Titan cluster.

The results also demonstrate that the performance of the ML algorithm is sensitive to the size of the allocated memory, even though its computational complexity grows linearly with the size of the 3D volume. As shown in Figure 4.9, when the number of processes increases from 4 to 8 the size of each segment deceases by half, yet all three approaches achieve a speedup of more than 2. Such super-linear speedup is a result of the significant reduction in the number of page faults and caches misses.

To understand the behavioral differences in the three approaches, we have developed a fully instrumented parallel implementation that captures timing information of specified events during the reconstruction process, including forward projection, backward projection, synchronization, communication, segments reassemble and file IO. Figure 4.10 shows the cumulative time consumed by these events for an 8-iteration reconstruction running on a 32-node NCSA Titan cluster at UIUC. The profile data is recorded by the root process.

It is clear that the forward and backward projection procedures dominate overall

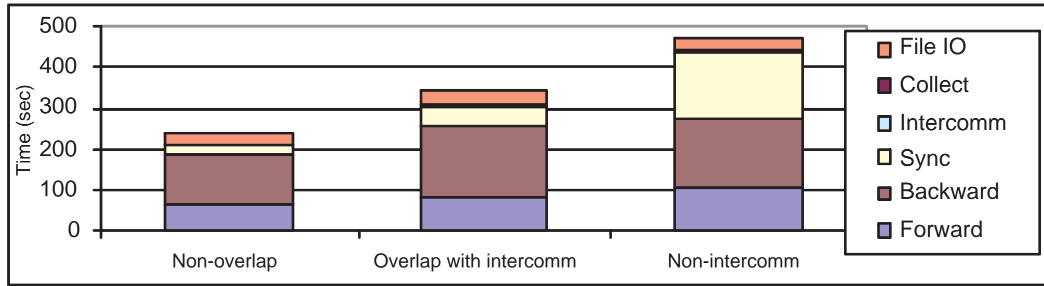

Figure 4.10: Profiling result comparison of three parallelization methods tested on 32 process on the NCSA Titan cluster at UIUC.

performance in all three methods. Tests of the non-overlap method show that evenly partitioned segments lead to well-balanced workloads. However, when extended regions are added to the segment reconstruction, the overlap with intercommunication and non-intercommunication methods show increased synchronization times. The amount of increased synchronization occurring at the final segment reassemble process indicates the imbalance of the workloads between processes.

Figure 4.11 shows the performance of the non-overlap parallelization executed on five different platforms using 32 processors. The SGI shared memory system based on the 64-bit architecture of the Intel Itanium 2 processor yields the least communication overhead and achieves the highest computation speed. The Pentium 4 cluster has a 100 Mb/s ethernet switch, experiencing the largest communication overhead. The times used by the forward and backward projection procedures illustrate the computing power of five different CPU architectures. Our tests show that the Itanium 2 processor can run two times faster than the Pentium 4 processor. The non-overlap parallelization achieves the best performance among the three parallelizations on all five platforms.

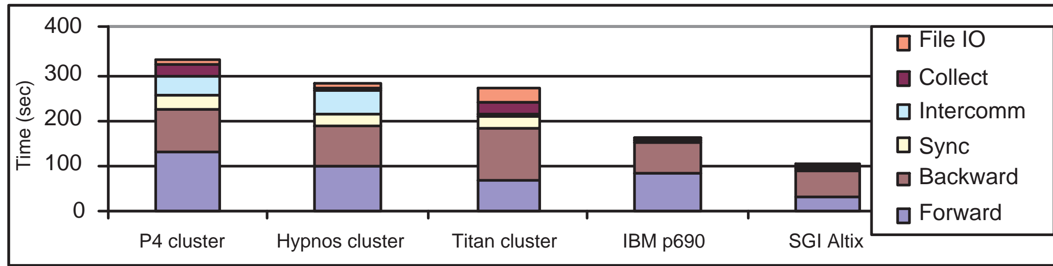

Figure 4.11: System comparison using the non-overlap parallelization running on 32 processors.

As shown in Figure 4.9, fine-grained parallelization is two times faster than coarse-grained parallelization when running on 64 processors. However, the performance difference between the three parallelization methods is barely noticeable on the SGI Altix platform, as shown in Figure 4.12. For the SGI Altix system, the computation of an extension segment requires approximately the same amount of time as transferring the data from a neighboring segment. The SGI Altix system outperforms all other systems in our experiments. However, its performance-cost ratio may not be the most attractive.

4.4 Workflow execution of Tomosynthesis

In previous sections, we have demonstrated the parallelization of the core reconstruction. To process a patient case, Tomosynthesis requires two additional tasks: pre-processing and post-processing, as shown in Figure 4.13. As shown in the figure, the three tasks execute in a workflow. The output of the pre-processing task is the input to the core image reconstruction task, and the output of the core image

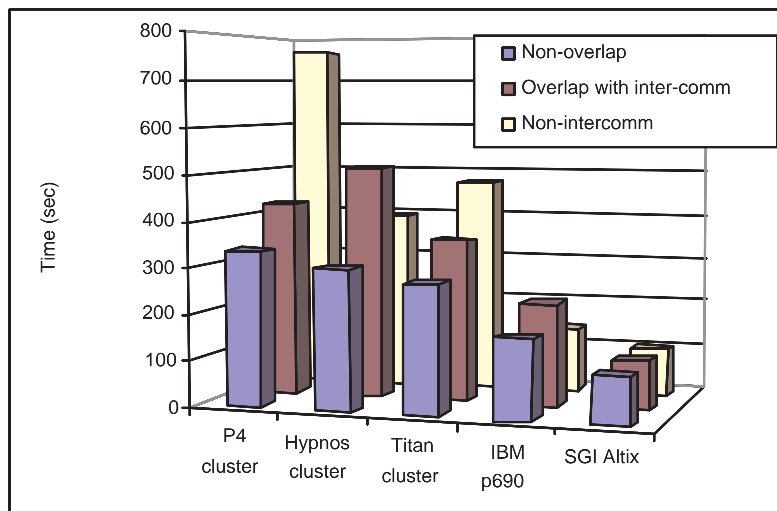

Figure 4.12: Performance test of three parallelization methods using 32 nodes on five platforms.

reconstruction task is the input to the post-processing task.

The pre-processing procedure, which is performed before the core reconstruction, removes artifacts that are introduced by defects in the imaging hardware. The current implementation of the pre-processing task is a sequential program, which takes approximately two minutes on a Pentium 4 workstation.

After a 3D image is reconstructed, the post-processing task performs peripheral equalization(PE) which reduces a broad of differences in volxel values into a smaller range which identifies those suspicious tissues across different layers of images. The results allow the image viewing software to display multiple grayscale images, showing those structures within a range which can be observed by human eyes directly. Specifically, the PE procedure applies the fast Fourier transformation(FFT) to each layer of the 3D volume.

It is a time-consuming process when applying FFT on all layers, and we reduce the

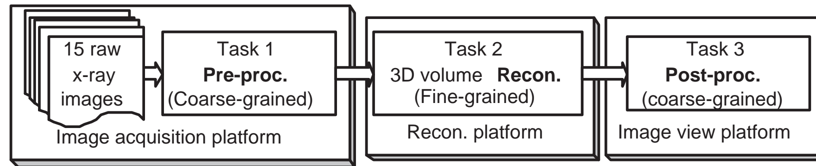

Figure 4.13: Illustration of the three Tomosynthesis tasks executed in a workflow

run time of the post-process task by using a coarse-grained parallelization approach. The computation of each layer in the 3D volume is distributed to a computing node, and each node processes one layer without the need to communicate with other nodes. This high-level parallelization exhibits various degrees of parallelism, determined by the number of layers in the 3D volume. In our tests, the time required to process a layer varies from 3 to 8 minutes on one Pentium-4 node, depending on the size of the layer. Generally, there are 40 to 70 layers in a 3D volume, and therefore, the total processing time could exceed 10 minutes on a 32-node cluster.

The parallelized core image reconstruction algorithm consumes less than 2 minutes of processing time on a high-performance Itanium-2 cluster using 64 nodes. However, such a powerful but expensive platform is not available for Tomosynthesis research, nor for its clinical practice. To achieve high performance computing for Tomosynthesis, one possible solution is to collect computing resources from multiple computing clusters or a computing grid. With this approach, the core image reconstruction can improve its performance via utilizing computing resources from a computing grid based on our SGR system.

The three tasks, exhibiting different levels of parallelism, are implemented in separate programs, and each task is executed independently given individual input data and parameters. There are two distinct benefits from using this approach. First of

all, it provides flexibility that may be needed. For example, if post-processing is not required or if reconstruction is based on previous pre-processed results, Tomosynthesis can be reconfigured easily so that the extra tasks will not be executed necessarily. In addition to flexibility, we can benefit from using separate tasks for better resource utilization. Since the pre-processing, reconstruction and post-processing tasks exhibit different levels of parallelism, a single MPI execution of all three tasks must allocate the greatest number of computing resources as required by all three tasks. Consequently, some computing resources would be wasted during the executions of those tasks using less resources. Using separate task executions, we can specify the adequate number of resources for each task to use, and reduce the number of total resources used by all the three tasks.

In Chapter 3, we introduced the SGR system, which allows an MPI workflow application to run on a computing grid, while hiding physical details of computing environments and enabling cross-site communication. The Tomosynthesis application can be launched on a computing grid using the SGR system. Therefore, we can investigate the performance of the parallelized image reconstruction of Tomosynthesis in a computing grid environment. Based on the SGR system, we will present our findings of running the three parallelization methods on a two-cluster grid in the next chapter.

Chapter 5

Experimental Evaluation of the SGR system

In this chapter, we describe experimental results of using the SGR system on a two-cluster grid. All experiments are parallel executions of MPI programs on specified numbers of computing nodes of the two clusters. Through analyzing queuing times and communication times, we demonstrate the advantages of our resource allocation models, the SGR framework and its implementation.

In the experiments, we found that message relay is scalable with respect to the size of data blocks and the number of MPI processes. We tested the SGR scheduler with synthetic workloads on the two-cluster grid to evaluate our allocation model. The findings based on the real-world experiments validate our earlier simulation results. Based on the SGR system, our motivating application, Tomosynthesis, can reduce the reconstruction time by using resource across the two clusters.

	Joulian	Keys
Num of nodes	16	8
Headnode	Dual PII 350MHz	PIII 1GHz
Subnode	PII 350MHz	PIII 1GHz
Intra-site Ntwk	100Mb/s	1Gb Ethernet
Inter-site Ntwk	Campus network: 10Mb/s	

Table 5.1: Cluster and network configuration of the testing environment

5.1 Communication experiments

We implemented the message relay service based on the intra-site, client-server, and server-server communication modules for the SGR system. This section evaluates the performance of 3-hop message relay in both point-to-point and collective inter-site communication.

The SGR message relay service was tested on a two-cluster grid, which consists of the Joulian and the Keys cluster at Northeastern University. The Joulian-Keys (JK) grid hardware configuration is listed in Table 5.1. The headnode of each cluster is connected directly to a campus network and assigned a public-IP address. Each cluster has an OpenPBS batch job system installed as its local resource management system. Globus Toolkit 4.0 and MPICH-G2 1.2.6 are also installed on both clusters, serving as the grid infrastructure.

We performed nine different communication tests in our testing environment, including two point-to-point performance tests, six collective communication tests and the ring communication application. The two point-to-point tests evaluate the latency

and throughput of cross-site communication between subnodes of the two clusters. The six collective tests are broadcast, scatter, gather, allgether, alltoall and barrier communication tests.

In our communication experiments (aside from the barrier test), we tested messages of different sizes, ranging from 0, 1, 2, 4, and up to 1MB. In all tests, the message buffer used in the communication tests is continuous in memory space. In the collective communication tests across the two clusters, different numbers of processes are used, with each cluster -llocated half of the total number of processes. -

The latency of a cross-site communication operation is the length of the time required to complete the whole operation. In a point-to-point communication test, the latency is measured by using half of the round-trip time. In a collective communication test, the latency is measured by the root process. The throughput of a cross-site communication operation is the size of a message block sent from one process divided by the average value of the communication latency in a period of time.

During all communication tests, no other users or workloads are scheduled on any computing resource across the two clusters, allowing us to obtain the maximum bandwidth and the minimum latency for intra-site communication. However, because the two clusters are connected on a campus network, traffic on the campus network may affect our test results. We tried to reduce this effect by performing our tests late at night when less traffic is expected.

5.1.1 Computing the communication lower bound

We evaluated the performance of the SGR message relay service in a cross-site communication operation by comparing it to a theoretical lower bound on its latency or a theoretical upper bound on its throughput. The lower and upper bounds give the theoretical peak performance of the cross-site communication operation. We compute these two performance bounds by only considering the limitation of the physical network.

A lower bound on the latency of sending a message between two computers, is:

$$Latency_{lower_bound} = T_{init_overhead} + N/P_{bandwidth} \tag{5.1}$$

where $T_{init_overhead}$ is the initial overhead for message passing, N is the size of the message in bytes, and $P_{bandwidth}$ is the physical bandwidth of the network connecting the two computers.

The initial overhead $T_{init_overhead}$ is the time used to transfer a 0-byte message over the network, and therefore represents the time cost needed to communicate a message of any size. $N/P_{bandwidth}$ is the theoretical minimum communication time needed to transfer the N-byte data over the physical network. An upper bound on throughput can be derived from the lower bound on latency as follows:

$$Thoughput_{upper_bound} = N/Latency_{lower_bound} \tag{5.2}$$

The difference between a test result of the point-to-point communication latency and its theoretical lower bound, the performance gap, is a result of additional communication costs. Such costs are introduced by the communication system during the process of transmitting message data over the network connecting the two computers,

such as communication data segmentation, cache misses, and data encryption. We define these additional costs as the communication overhead. The communication overhead is determined by how the communication library and the OS communication kernel are implemented. Since our SGR system relies on the MPICH-G2 library for server-server communication, a major part of the communication overhead of the SGR relay service is inherited from this underlying communication module.

During cross-site point-to-point communication, our SGR system requires three consecutive point-to-point message passing hops to relay a message from the source to the destination process. A lower bound on the total latency is the sum of lower bounds for the three message passing hops. The lower bound of each message relay hop is computed based on Formula 5.1. A cross-site collective communication operation can be divided into two parts: inter-site communication and corresponding intra-site collective communication. Our SGR system relies on the host-provided MPI system to perform the corresponding collective communication within a cluster. We therefore use our test results for the host-provided MPI library as the lower bound for the intra-site collective communication.

In cross-site point-to-point message passing based on the 3-hop message relay service, the performance gap between the observed communication latency and its lower bound includes the three communication overheads occurred during the three message relay hops and the service time used by the message relay service. To relay an inter-site message, it takes time for a server process to detect the incoming relay message and to match its message header to its corresponding receive request. As a result, this SGR-introduced service overhead also contributes to the overall performance gap. We will examine the performance gap and the SGR-introduced service

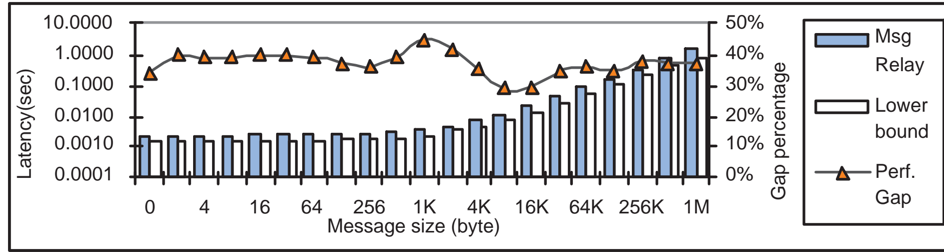

Figure 5.1: Evaluation of the communication latency of the cross-site point-to-point communication.

overhead in the cross-site point-to-point communication next.

5.1.2 Point-to-point communication experiments

Figure 5.1 shows the observed latency of cross-site point-to-point communication between two subnodes, when sending messages of different sizes. Three series of data are plotted: the observed latency for 3-hop message relay, the theoretical lower bound on latency, and the performance gap between the two.

When the message size increases from 0 to 512 bytes, we observe that there is only a small increase in communication latency for the SGR 3-hop message relay. This indicates that the overhead of sending data across the two clusters dominates the overall performance. This overhead includes the initial overhead, the communication overheads for the three message relay hops and the SGR service overhead. When the message exceeds 512 bytes, the observed latency increases linearly with the size of the message, which means that the communication time used by the physical network becomes the dominant factor in determining the overall performance of inter-site communication.

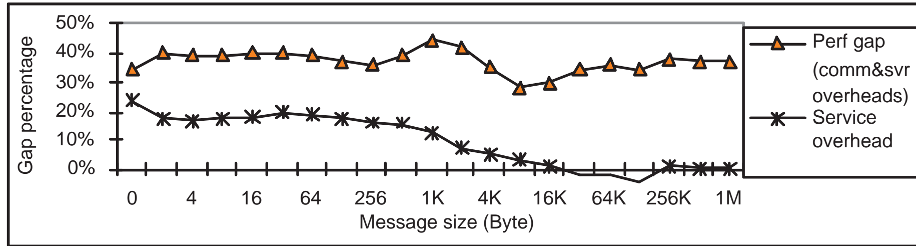

Figure 5.2: Estimation of the SGR-introduced service overhead

For all tested message sizes, Figure 5.1 shows a performance gap of 29% to 45% between the latency of message relay and the lower bound. When the message is 8KB, the performance gap is limited to 29% of 0.012s as the cross-site communication latency. This performance gap includes the communication overheads for the three message relay hops and the SGR-introduced service overhead.

The cross-site point-to-point communication process requires three consecutive message hops and the message relay service. However, neither the SGR service overhead nor the communication latency for each message hop can be measured directly during the relay process, because the communication on the SGR server side is performed in the probe-receive and the non-blocking mode. Alternatively, we can use separate communication tests to estimate the communication latency of each communication hop. The SGR service overhead can be estimated by subtracting the sum of the estimated latencies of the three message hops from the observed cross-site communication latency, as shown in Figure 5.2.

In Figure 5.2, when messages are small, approximately 20% of time is consumed by the SGR relay service. As the size of messages exceeds 4KB, the SGR-introduced service time decreases to around 1%, indicating that the communication dominates

the overall performance. Our analysis demonstrates that the efficiency of the SGR relay service when we use it to send large inter-site messages.

The performance gap shown in Figure 5.2 is the difference between the observed cross-site communication latency based on message relay and its lower bound. The performance gap includes the communication overheads for the three message hops and the SGR service overhead. Therefore, the difference between the SGR service overhead and the performance gap represents the communication overheads for the three message hops, as shown in Figure 5.2. Our tests show that the communication overheads for the message hops increase with the size of the inter-site message during the relay process. Such increases in the communication overheads may result in some performance loss, when sending large inter-site messages, suggesting that further optimization work is needed for the SGR underlying communication module, that is MPICH-G2.

Figure 5.3 shows results of the throughput test across the two clusters, as the size of messages varies. We compute the upper bound on the throughput of message relay based on Equation 5.2. The test results show that 3-hop message relay can achieve up to 70% of the upper bound on the throughput (i.e. the performance gap is at most 30%), when sending messages of size 8KB or 16KB .

When sending messages between two clusters, applications may benefit from larger sizes of messages, as our test results suggest. In our testing environment, messages of size 16KB can best utilize the network and minimize the SGR-introduced service overhead. However, the optimal message size may vary on different networks and computing platforms.

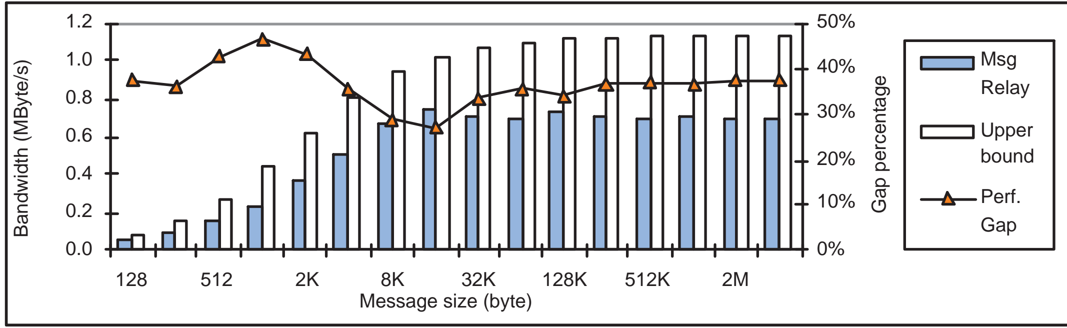

Figure 5.3: Throughput of the cross-site point-to-point communication operation

5.1.3 Collective communication experiments

5.1.3.1 Cross-site MPI barrier

The cross-site barrier operation performs a synchronization operation among root processes of all concurrent tasks, and then a local barrier operation within each task. To synchronize all root processes, each root process sends a barrier signal to all other root processes, and at the same time receives barrier signals from all other root processes. In our 2-cluster grid environment, synchronization of the two root processes consists of two cross-site point-to-point communication operations which can be performed simultaneously using the 3-hop message relay service. Therefore, the lower bound on the latency of the cross-site barrier operation is the bound on a cross-site point-to-point communication operation plus the bound on a barrier operation within a cluster. It is computed as follows:

$$Barrier_{inter-site} = P2P_{inter-site} + Barrier_{intra-site} \tag{5.3}$$

Figure 5.4 shows test results for the cross-site barrier operation based on the SGR message relay service, the theoretical lower bound on the latency, and the performance

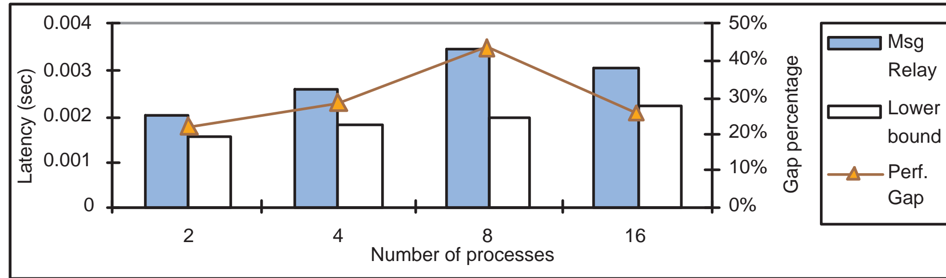

Figure 5.4: Latency of the cross-site MPI barrier operation

gap between the two. The performance gap varies between 30% and 39%, while the number of processes increases from 2 to 16. The latency of the relay-based barrier operation increases more slowly than the size of the execution topology. Therefore, we conclude that the SGR barrier operation is scalable with respect to the execution topology.

In Figure 5.4, the latency of the relay-based cross-site barrier on 8 nodes is higher than the latency on 16 nodes. We believe that this result is caused by increased traffic on our campus network connecting the two clusters during the period when we conducted the test on 8 nodes. Since each test is repeated several times, the standard deviation of the test on 8 nodes is significantly higher than the remaining tests running on 2, 4 and 16 nodes. This suggests that the inter-site communication, which shares the campus network with other users, is affected by other communication during the test.

5.1.3.2 Cross-site MPI broadcast

Figures 5.5 and 5.6 show test results for the cross-site MPI broadcast operation. This operation consists of a cross-site point-to-point communication operation which sends

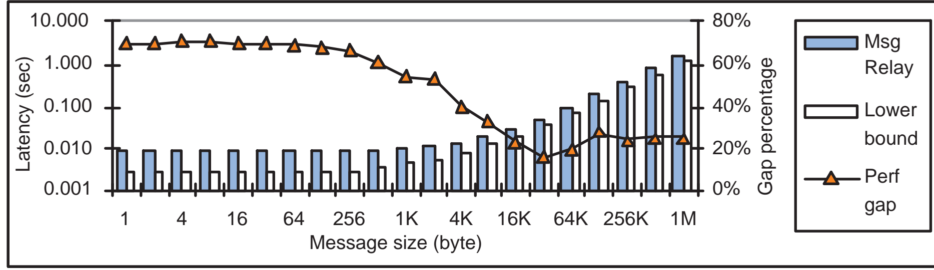

Figure 5.5: Latency of cross-site broadcasting among 16 processes over the two-cluster grid

the broadcast data from one cluster to the other, and two local broadcast operations within each cluster which can be performed concurrently. Therefore, the lower bound on the latency of the cross-site broadcast is the bound on one inter-site point-to-point message passing operation plus the bound on one local intra-site broadcast within a cluster. The lower bound of the cross-site broadcast operation is computed as follows:

$$Bcast_{inter-site} = P2P_{inter-site} + Bcast_{intra-site} \tag{5.4}$$

Figure 5.5 shows a performance gap of 70% to 20% between the SGR message relay-based broadcast operation and the lower bound on latency, when broadcasting messages of different sizes across the two clusters. Smaller messages produce a larger performance gap. When the broadcast message is 16KB, we observe that the performance gap yields the minimum value of 20%, and remains around 25% for message sizes larger than 16KB, demonstrating the scalability of our SGR system in the cross-site broadcast operation.

Figure 5.6 shows the latency of broadcasting messages of size $256KB$ as the number of processes varies across the 2-cluster grid. Test results show a performance gap of 37% to 24%, with the performance gap decreasing for larger topology sizes. This

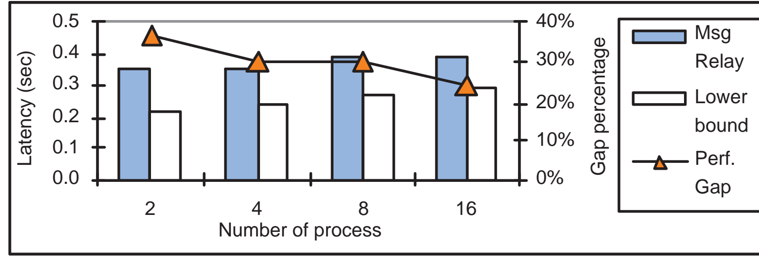

Figure 5.6: Latency of cross-site broadcasting 256 KB message

experiment demonstrates that the SGR message relay-based broadcast operation is scalable with respect to the size of the execution topology.

5.1.3.3 Cross-site MPI scatter

Figures 5.7 and 5.8 show the latency of the MPI scatter operation across the two clusters. The scatter operation sends one specified message block from the root process to every process running on the two clusters. In our experiment, the cross-site scatter operation includes a cross-site point-to-point communication operation, which contains all message blocks for all processes running on the remote cluster, and two local scatter operations within each cluster. We can perform the two local intra-site scatter operations at the same time, after the cross-site point-to-point communication operation completes. Therefore, the lower bound on the latency of the cross-site scatter operation is the bound on one cross-site point-to-point communication operation plus the bound on a local scatter operation within a cluster. The lower bound of the cross-site scatter operation is computed as follows:

$$Scatter_{inter-site} = P2P_{inter-site} + Scatter_{intra-site} \tag{5.5}$$

The test of cross-site scatter produces results similar to the cross-site broadcast

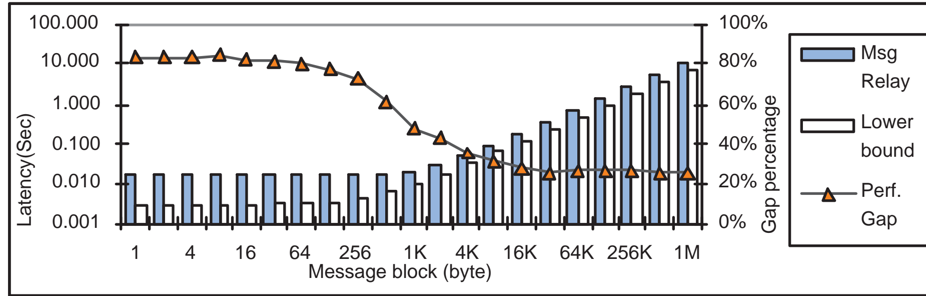

Figure 5.7: Latency of cross-site MPI scatter among 16 processes

test. The results are shown in Figure 5.7. When message blocks are small (below 256 bytes), the SGR relay-based scatter operation yields a performance gap of 70% to 80%. However, the latency increases slowly as we test larger message blocks. When message blocks are larger than 256 bytes, the latency of the scatter operation grows linearly with the size of message blocks. Between 256 bytes and 8KB, we observe a continuous reduction in the performance gap, demonstrating the scalability of the SGR system in the 2-cluster grid environment.

Figure 5.8 shows the latency of cross-site scatter on execution topologies with different sizes. The performance gap between the SGR relay-based inter-site scatter operation and its lower bound decreases from 38% to 26%, while the size of the topology increases from 2 to 16. This suggests that the SGR message relay service is scalable with respect to the size of execution topology in the cross-site scattering operation.

5.1.3.4 Cross-site MPI gather

Figures 5.9 and 5.10 show the observed latency of the cross-site MPI gather operation based on the SGR message relay in our two-cluster grid environment. In a gather

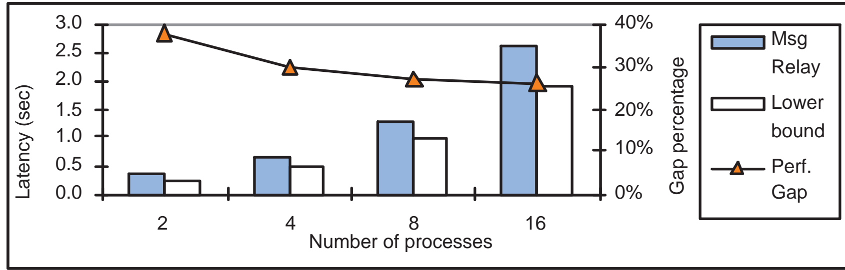

Figure 5.8: Latency of cross-site MPI scatter of 256KB message blocks

operation, the root process collects a message block from each process. The cross-site MPI gather operation requires a local gather operation within each cluster, which can be performed in parallel, and a cross-site point-to-point communication operation to send all gathered message blocks from the remote cluster to the root process. Therefore, the lower bound on the latency of the cross-site MPI gather operation is the bound on a local intra-site MPI gather operation within a cluster plus the bound on a cross-site point-to-point communication operation. The lower bound of the cross-site MPI gather is computed as follows:

$$Gather_{inter-site} = P2P_{inter-site} + Gather_{intra-site} \tag{5.6}$$

The MPI gather operation is the opposite of the MPI scatter operation, and both cross-site operations have similar performance. Figure 5.9 shows that the performance gap between the SGR relay-based gather operation and its lower bound on latency decreases from 78% to 26% as the size of the message block increases from 1 byte to 1MB. Figure 5.10 shows that the performance gap reduces from 36% to 26%, while the size of the execution topology increases from 2 to 16. These test results demonstrate that the performance of the SGR relay service is scalable with respect to the size of message blocks and the execution topology in the cross-site MPI gather operation.

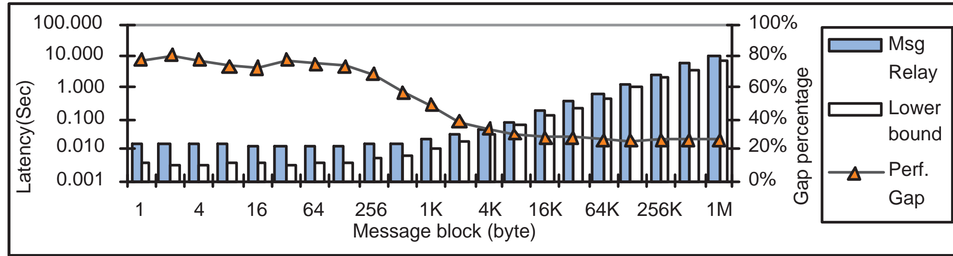

Figure 5.9: Latency of cross-site MPI gather among 16 processes

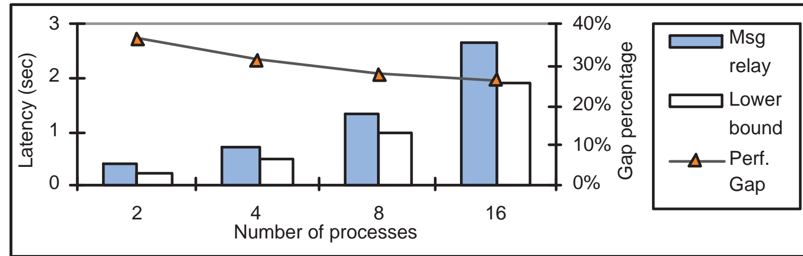

Figure 5.10: Latency of cross-site MPI gather of 256KB message blocks

5.1.3.5 Cross-site MPI allgather

We also tested more complex collective communication: the cross-site MPI allgather operation and the cross-site MPI alltoall operation. The MPI allgather operation collects message blocks from all processes and then broadcasts them to all processes. The cross-site allgather operation consists of three steps.

1. A local MPI gather operation collects message blocks from processes running within each cluster.

2. Each cluster sends its own gathered message blocks to all other clusters.

3. Each cluster performs local broadcasts of its gathered message blocks and message blocks received from remote clusters to all local processes.

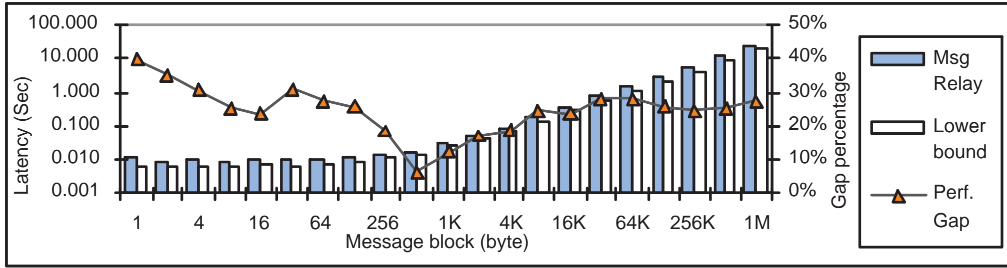

Figure 5.11: Latency of cross-site MPI allgather among 16 processes

In the first and third step, all local MPI gather and broadcast operations within each cluster can be performed in parallel. In the second step, we allow one cluster to perform the cross-site communication at a time, so as to avoid competing with limited public network resources. In our 2-cluster grid environment, two cross-site point-to-point communication operations are required. Therefore, the lower bound on the latency of the MPI allgather operation across the two clusters is the sum of the bounds on the latencies of one local MPI gather operation within a cluster, two cross-site point-to-point communication operations, and one local MPI broadcast operation within a cluster. The bound is computed as follows:

$$Allgather_{inter-site} = Gather_{intra-site} + 2 \times P2P_{inter-site} + Bcast_{intra-site} \tag{5.7}$$

Figures 5.11 and 5.12 show the latency of the cross-site MPI allgather operation based on message relay, as we increase the size of message blocks and the size of the execution topology, respectively. In Figure 5.12, we observe that the performance trend of the MPI allgather operation is similar to that seen with other collective communication operation, as its performance gap decreases from 34% to 25%, while the size of the execution topology increases from 2 to 16.

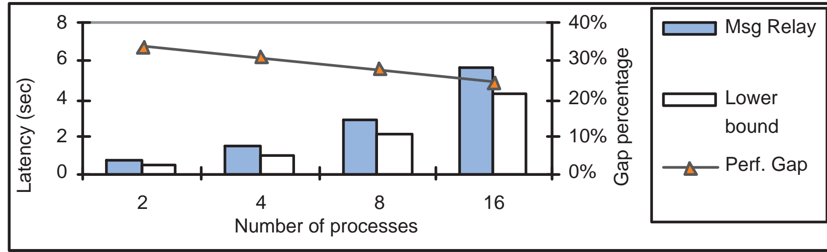

Figure 5.12: Latency of cross-site MPI allgather of 256KB message blocks

In Figure 5.11, the performance gap of the SGR relay-based allgather decreases from 40% to 6%, then it increases to 28% when the size of message blocks increases from 1 byte to 1MB. The smallest performance gap of 6% is obtained when message blocks are 512 bytes, which has $512 \times 2 \times 8 = 8$KB of data being sent across the two clusters. In the cross-site point-to-point communication test, the 8KB message produces the smallest performance gap, as shown in Figure 5.1. This suggests that cross-site communication may dominate the overall performance of the cross-site allgather operation, when the message blocks are large.

The allgather operation exchanges a large amount of data between clusters. To evaluate the fraction of the time used for the cross-site communication in the allgather operation, we measure the communication latency of exchanging messages between the two clusters based on the results of the cross-site point-to-point communication test. Figure 5.13 shows that cross-site communication consumes 90% of the overall latency when the size of message blocks is greater than 512 bytes. Therefore, the performance of the MPI allgather operation across the two clusters is dominated by the performance of cross-site point-to-point communication when the size of message blocks is large.

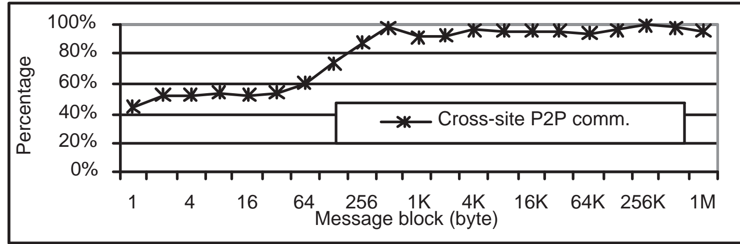

Figure 5.13: Proportion of cross-site communication in the overall cross-site MPI allgather among 16 processes

5.1.3.6 Cross-site MPI alltoall

In the MPI alltoall operation, every process sends specific message blocks to all processes, and receives message blocks from all processes. The operation is implemented in different ways to achieve the best performance in different network environments. In our two-cluster grid environment, the public network connecting the two clusters is the communication bottleneck. We subdivide the cross-site alltoall operation into two types of communication: the intra-site and the cross-site communication. An intra-site MPI gather operation collects message blocks from processes within one cluster, and the collected blocks are sent to the same destination process. For the collected message blocks whose destination process runs on a different cluster from the intra-site gather operation, a cross-site point-to-point communication operation is used to deliver the collected data to the destination process. Therefore, the alltoall operation is carried out by an intra-site gather operation for every process, and cross-site point-to-point communication operations.

To compute the lower bound on the latency of the cross-site alltoall operation, we maximize the time during which the intra-site gather and the inter-site point-to-

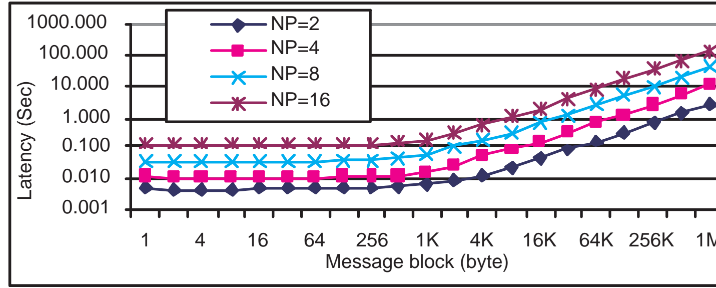

Figure 5.14: Throughput of the cross-site MPI alltoall operation

point communication can be overlapped. The lower bound includes the latency for the first intra-site gather operation. The times for the remaining gather operations are overlapped with the cross-site point-to-point communication operations. Because the total number of cross-site point-to-point operations equals the number of processes, we compute the lower bound as follows:

$$Alltoall_{inter-site} = Gather_{intra-site} + Number_{processes} \times P2P_{inter-site} \tag{5.8}$$

Figure 5.14 shows the throughput of the cross-site MPI alltoall operation as the number of processes and the size of the message block increase. We can observe that the throughput increases linearly with the number of processes. The test results show that the network throughput determines the overall performance of the cross-site alltoall operation which is based on the message relay service.

Figure 5.15 shows the performance gap between the latency of the cross-site alltoall operation based on message relay and its lower bound. The performance gap decreases from 83% when sending 1 byte message blocks to 8% when sending 32KB message blocks. The test results shown in Figure 5.14 and 5.15 demonstrate that the SGR relay system allows the cross-site alltoall operation to be scalable with respect to the

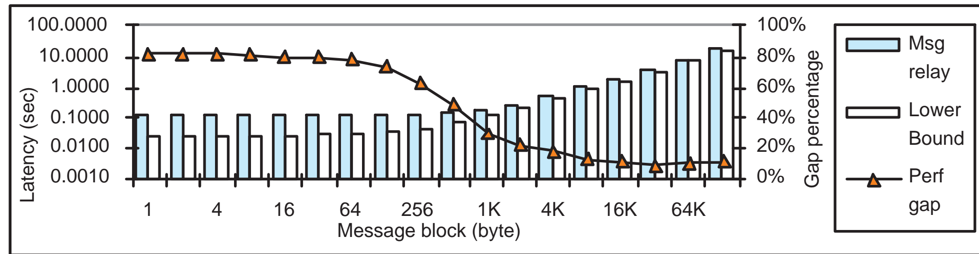

Figure 5.15: Latency of cross-site MPI alltoall on 16 processes

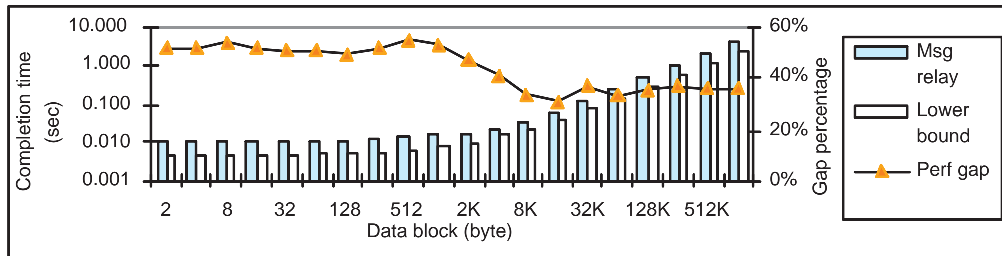

Figure 5.16: Time to complete the cross-site ring communication

size of message blocks and the size of the execution topology.

5.1.3.7 Cross-site ring test

The cross-site ring communication test circulates a message through all processes until the message is sent back to the starting process. The time to complete one circulation among all processes across the two clusters is shown in Figure 5.16. We compute the lower bound on the time needed to complete a circulation by adding the lower bound on the latency of each point-to-point communication operation along its path.

The performance gap between the latency of the cross-site ring communication operation and its lower bound remains around 50% when the message block is below

1KB, and decreases to 32% when the message size increases to 16KB. This performance change is mostly due to the cross-site communication operation, as we can find similar performance changes in the cross-site point-to-point communication test shown in Figure 5.1.

5.1.3.8 Summary of communication tests

In our cross-site communication tests, larger performance gaps are observed when we send smaller inter-site messages using the SGR relay service. However, our SGR relay system has demonstrated its efficiency in handling large messages in cross-site point-to-point and some collective communication operations, such as broadcast, scatter, and gather. As larger message blocks are used in cross-site communication, the communication bottleneck may occur at the public network which connects the two clusters. As we found in the allgather and alltoall experiments, the bandwidth of the public network limited the performance of these collective communication operations.

Based on our experiments and evaluations, we conclude that the SGR message relay service is scalable with respect to the sizes of message blocks and the execution topology. The overhead of the SGR relay service is relatively high when the inter-site message and the execution topology are small. As the sizes of message blocks and the execution topology increase, the SGR service is capable of lowering the fraction of the total communication latency taken by the service overhead. This suggests that our SGR system can be used to solve large-scale problems that involve large numbers of computing resources and a large amount of communication data in a computing grid environment.

5.2 SGR synthetic workload experiments

In Section 2.3, we relied on a simulator to predict the performance of the resource allocation model and the task scheduler using a range of ranking criteria. In this section, we evaluate the performance of the SGR scheduling and grouping services tested in our two-cluster grid environment.

Unlike the communication experiment described in the previous section, we tested the SGR scheduling and grouping services on the grid which processes local resource requests and our test global workload at the same time. Each of the two clusters processes a collection of local job requests throughout the experiment, emulating the conventional usage of a shared-resource environment. Each local job request is queued by the resource management system until its requested resources are available and allocated for it to use.

Each synthetic job is defined by three parameters: the number of computing resources required, the job arrival time, and the job execution time. These parameters are generated randomly based on the same functions and distributions described in the Section 2.3.1. We created two local synthetic workloads using the workload level of 0.7, as each is assigned to one cluster. This means that the resources within each cluster have a utilization of approximately 70% when processing its local workload. A global synthetic workload consists of workflow tasks, each scheduled by our SGR scheduler, targeting the two-cluster grid. The global tasks have a workload level of 0.5 for the Keys cluster, utilizing 50% of the resources on the Keys cluster (or utilizing 25% of the resources on the Joulian cluster since it has twice as many as computing resources). All local jobs and global tasks have an average execution time

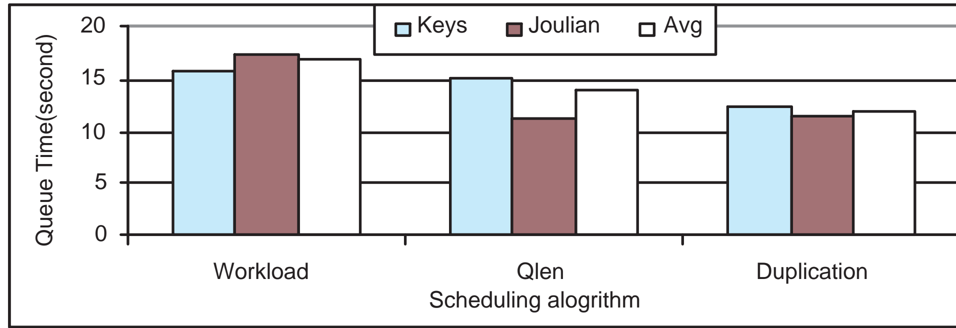

Figure 5.17: Average queuing time for global jobs submitted to the 2-cluster grid

of 60 seconds.

The Joulian cluster with 16 computing nodes allows a job to acquire no more than 8 nodes. The Keys cluster with 8 computing nodes allows a job to acquire no more than 4 nodes. The limitation is used to prevent one large job from blocking all other job requests in a job queue, a policy often adopted by resource management systems. The global synthetic workload only contains tasks that request no more than 4 computing nodes, so that all global tasks can be scheduled and executed by both clusters.

We tested two ranking criteria for the SGR scheduler: *Qlen* (the queuing length) and *Workload* (the system workload). Both of these schedulers have been described in Section 2.3.3. The average queuing times for the global tasks of using the *Qlen* scheduler and the *Workload* scheduler are shown in Figure 5.17. Also presented in the figure are the results using task duplication. Since our testing grid has only two computing sites, each global task is allowed to have one duplicated task. Both tasks are submitted to the two cluster at the same time.

Figure 5.17 shows that task duplication outperforms the *Workload* and the *Qlen*

schedulers by 29% and 15%, respectively, in the average queuing time of global tasks. In our experiment, it takes approximately 10 seconds to submit and invalidate a redundant job request which can delay executions of other tasks and local jobs waiting in the job queue. However, it is shown that by locating the earliest available resources for the global task can increase the overall resource utilization on both clusters, which ultimately outweighs the costs of submitting and invalidating redundant tasks.

In our experiment, the *Qlen* scheduler yields better performance than the *Workload* scheduler, achieving shorter queuing times for global tasks. Because the Joulian cluster is assigned a workload with larger job requests (jobs can specify up to 8 computing nodes), and a larger job request is more likely to be queued for a longer time, the Joulian cluster experiences a longer job queue length and lower resource utilization than the Keys cluster. Therefore, the *Workload* scheduler using the workload as the ranking criterion assigns 73% more global tasks to the Joulian cluster as compared to those assigned to the Keys cluster. On the other hand, the *Qlen* scheduler using the queue length as the ranking criterion assigns 65% more global tasks to the Keys cluster as compared to those assigned to the Joulian cluster. As a result, the *Qlen* scheduler reduces the average queuing time by 35% for the global tasks submitted to the Joulian cluster when it is compared with the *Workload* scheduler. Even though the Keys cluster is assigned more tasks by the *Qlen* scheduler, there are no significant increases in queuing times for global tasks. Therefore, the *Qlen* scheduler using the queuing length as the ranking criterion outperforms the *Workload* scheduler in our experiments.

While the Joulian cluster is processing larger job requests, there could be more idling resources left unused. Some smaller global tasks can use such resources to

reduce or eliminate their waiting times if they are scheduled to the Joulian cluster. However, both the *Qlen* and *Workload* schedulers may not be able to detect such an availability of unused resources based on the information of the job queue length or the workload level. Task duplication, on the other hand, allows global tasks to utilize these resources once they become available. When the SGR system invalidates redundant resource requests, any unnecessary resource used by those duplicated tasks can be eliminated. As a result, the overall system throughput increases, and we can observe less queuing time for global tasks in the duplication test.

In Figure 5.17, we can see that task duplication yields less queuing time on the Keys cluster as compared to the result obtained from using the *Qlen* scheduler. This can be explained by the fact that some small global tasks are executed on the Joulian cluster first, instead of on the Keys cluster, to which they are assigned in the *Qlen* scheduler test. Consequently, the Keys cluster ends up processing less global tasks, leading to shorter queuing time by using task duplication method.

5.2.1 Comparing experiment and simulation

In this section, we have shown that task duplication can obtain 15% to 29% performance improvement over the that two global job schedulers, *Qlen* and *Workload*. This performance improvement obtained on the Joulian-Keys grid is in the same range as shown in our earlier simulation. In Section 2.4.6, Figure 2.9 shows that both *Qlen* and *Workload* schedulers can achieve about 17% performance improvement when using 1 job duplication on a 2-site heterogeneous grid in our simulation environment. However, this marginal difference validates our simulation work.

There are several factors which can cause the difference between the simulation and experiment results. The first factor is the hardware, more specific, the number of computing resources. In the simulation environment, two computing sites are configured with 64 computing resources and 32 computing resources respectively. Joulian and Keys clusters of our testing grid have 16 and 8 nodes, as shown in Table 5.1. Given more resources, a computing site may experience less loaded conditions, and less queuing time for global tasks. As a result, *Workload* scheduler tends to perform worse in the experiment on the Joulian-keys grid, and this allows job duplication to obtain more performance improvement.

The second cause can be found in system overheads In our simulations, we assume the time required by job submission and duplication invalidation is 1 simulation second. However, in our experiments, this cost is varied, as it increases when cluster is heavily loaded. Furthermore, since Joulian has slower processing speed, job submission and duplication invalidation on Joulian tend to take more time than on Keys. These system overheads can affect the global scheduler's performance and the performance improvement when using task duplication.

Finally, the numbers of global tasks tested in our simulation and used in the experiment on the Joulian-keys grid are different. The simulation is much faster, so that we can conduct tests on more than 1000 global tasks. The experiment on the Joulian-Keys grid executes every global tasks and local jobs, which takes hours to complete. Therefore, we are only allowed to use hundreds global tasks and local jobs in the experiment. Given more global tasks in the experiments, we can predict that the differences between the simulation and experiment results will become even less.

In this section, we first present the experiment results using the task scheduling

service in a loaded environment on the Joulian-Keys grid. We show noticeable performance improvement by using task duplication. We also examine the simulation results of a heterogeneous grid, comparing to the experiment results on the Joulian-Keys grid. The simulation findings match our experiment results.

5.3 Experiments using Tomosynthesis

One of our initial objectives is to study the performance improvement of Tomosynthesis when it is launched on two computing sites of a computing grid. We first discuss how we estimate the speedup for a cross-site execution, given the run time of a single-site execution. Then, we compare the estimated speedup of the cross-site execution of Tomosynthesis image reconstruction on the Joulian-Keys grid to the actual experiment results.

5.3.1 Speedup estimation

Amdahl's law [8] is used to predict the achievable speedup of the parallel execution based on the information of its serial execution. Amdahl's law can be described as follows

$$Speedup = \frac{S + P}{S + P/n} \tag{5.9}$$

where n is the number of processes used in the parallel execution, S is the time used by the sequential portion of the execution, and P is the time used by the parallel portion of the execution.

Different from Amdahl law, we compute the performance speedup for a cross-site parallel execution in comparison to the run time used by a single-site parallel

execution. To simplify our study, we only consider the following scenarios. The cross-site execution is referred to as a parallel execution using two equal numbers of processes on two computing sites, while the single-site execution only uses half of the total processes required by the cross-site execution. To compute the speedup, we first show how we estimate the execution time of the cross-site execution based the information of the single-site execution.

We assume that a single-site n-process parallel execution is mostly composed of the inter-process communication, the sequential portion of the execution, and parallel portion of the execution. Let C denote the time used by the inter-process communication during the single-site parallel execution. The sequential portion of the execution can not be parallelized, and we use S to represent its execution time. The parallel portion of the execution is divided amount n processes, and we use P_n to represent the time needed by one of the n processes. The value of P_n should be determined by the process which requires the longest execution time to process its assigned computation work. Then, the total run time of the single-site parallel execution is

$$T_{single-site} = C + S + P_n. \tag{5.10}$$

In a cross-site parallel execution, the intra-site communication network is much faster than the inter-site network. We use $k \times C$ as an estimation of the communication time used by the cross-site execution, where k is the ratio of inter-site to intra-site communication latency. Since the cross-site execution uses doubled computing resources as the single-site execution, we can expect that the parallel portion of the cross-site execution requires half time on one of $2n$ processes, or $P_n/2$. If we assume that the sequential computation does not change in the cross-site execution, and

therefore, we can compute the run time of the cross-site execution as follows

$$T_{cross-site} = k \times C + S + P_n/2. \tag{5.11}$$

Based on the Formula 5.10 and 5.11, the speedup for the cross-site execution is

$$Speedup_{cross-site} = \frac{T_{single-site}}{T_{cross-site}} = \frac{C + S + P_n}{k \times C + S + P_n/2}.$$

In Section 4.3, the Tomosynthesis experiments show that the execution time is dominated by the parallel portion of the execution. S is very small compared to the P_n. Therefore, to simplify the above formula, we can compute an approximate value of the speedup by removing S from the above formula, and we have

$$Speedup_{cross-site} \approx \frac{C + P_n}{k \times C + P_n/2}.$$

When C is not 0, we have

$$Speedup_{cross-site} \approx \frac{1 + P_n/C}{k + 1/2 \times P_n/C}.$$

If we let r replace P_n/C, and we have

$$Speedup_{cross-site} \approx \frac{1 + r}{k + r/2}. \tag{5.12}$$

When there is communication cost in the single-site execution (i.e. $C \neq 0$), r means the ratio of the computing time required by the parallel portion to the communication time used by the parallel execution..

In most of cross-site executions, we can found that the higher intra-site communication cost leads to the smaller speedup, while applications with higher parallel computational workload can achieve better speedup in cross-site executions. Formula 5.12 complies with these conventional observations, because, in the formula, the

speedup decreases with k, the ratio of intra-site to inter-site communication latency, and increases with r, the ratio of parallel computation to communication. In addition, we can quantify value of the possible speedup when given the application's single site execution. Therefore, we are able to estimate the application's performance before it is deployed on a computing grid.

In an actual parallel execution, the performance is determined by the slowest computers. On the Joulian-Keys grid, Joulian's nodes is slower in computing speed and its intra-site network has less bandwidth. Therefore, we use the run time information collected on Joulian to compute r and C. Based on the experiment results presented in Section 5.1.2, the bandwidth for the intra-site P2P communication on Joulian is 10.1 MB/s, and the inter-site bandwidth is 0.7 MB/s. The ratio of inter-site to intra-site communication latency is

$$k = \frac{10.1}{0.7} = 14.3$$

We tested the single-site execution of the Tomosynthesis image reconstruction on both Joulian and Keys clusters. Figure 5.18 presents the execution time of the image reconstruction using the non-overlap parallelization method. The single-site execution is composed of four parts: the forward and backward projection, communication, and the sequential execution. The sequential part includes the MPI initialization and file I/O operation. The forward and backward projection consists of computation work which is parallelized. The communication time includes times used in exchanging data between iterations and collecting all reconstructed segments from all processes.

The ratio of parallel computation to communication of the Tomosynthesis image

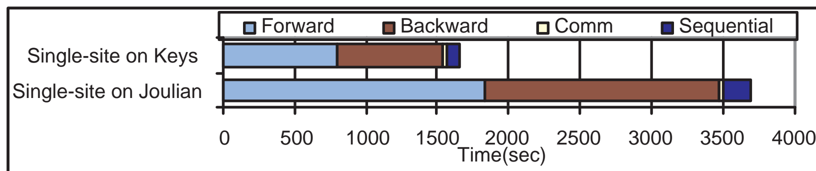

Figure 5.18: Results of single-site executions of the parallel Tomosynthesis image reconstruction using 8 processes on Joulian and Keys

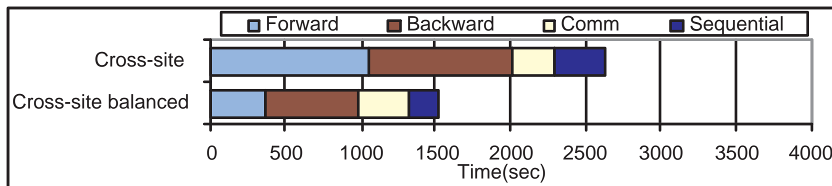

Figure 5.19: Results of cross-site executions of the parallel Tomosynthesis image reconstruction using 16 processes on the Joulian-Keys grid

reconstruction is 90, based on data shown in Figure 5.18. Using Formula 5.12, the cross-site execution can achieve 1.53 speedup compared to the single-site execution.

5.3.2 Experiments on the Joulian-Keys grid

We tested the cross-site execution of the Tomosynthesis image reconstruction on the Joulian-keys grid, while using the SGR system's message relay service to enable inter-site communication. Figure 5.19 presents the execution results of the non-overlap image reconstruction. The observed speedup is 1.40, as shown in the figure.

Previously, we estimated the speedup for the cross-site execution over the single-site execution is 1.53. Our experiments show that the cross-site execution achieves 1.40 times speedup. The difference between the two is due to the fact that the se-

quential portion is not considered in our estimation. In addition, the synchronization time caused by unbalanced workloads between processes could contribute to the difference too. However, this difference is considered small. Using the Tomosynthesis application as an example, we demonstrated that Formula 5.12 can be used in performance prediction for cross-site executions of other similar computation-intensive applications.

Since Joulian has PII processors and Keys has PIII processors, (as shown in Table 5.1), the two clusters differ in computing speed. Figure 5.18 shows that the forward and backward projection takes more than twice as much time on Joulian than Keys, when the both clusters use 8 processes to reconstruct the same 3D volume. In the cross-site execution, we first tested as the 3D volume is evenly partitioned between the two clusters. This test takes 2633 seconds, which is even slower than the single-site execution on Keys.

We fine-tuned the workload assigned to Joulian and keys based on their computing speeds for the cross-site execution. When we reduce the partitioned 3D volume assigned to Joulian to 20% of the total volume, we can achieve the best performance for the cross-site execution. Figure 5.19 shows that the cross-site execution with balanced workloads takes 1533 seconds, over 42% performance improvement than the unbalanced case.

As shown in Figure 5.19, although we can reduce the computational time by using more computing resources from both clusters, there is a large portion of the performance gain offset by the increased inter-cluster communication. The non-overlap method we tested requires a large volume of data transferred among processes. As a result, there are significant increases in communication time for the cross-site ex-

ecution for the Tomosynthesis application. However, the result of using balanced workloads on the two clusters show that we can still achieve at least 8% performance improvement in the cross-site execution than the single-site execution on the Keys cluster.

This section presents our experiment results of running the Tomosynthesis application on the Joulian-Keys grid. We tested the MPI application on a computing grid while using our SGR system to enable inter-site communication. Test results show we can improve performance when the application is launched on the two clusters, given the high inter-site communication cost and varied computing speeds between the clusters. In this section, we also demonstrated how to estimate the performance speedup for a cross-site execution, which can be used to predict possible performance gains before launching applications on grids.

Chapter 6

Summary

Grid computing has emerged as a promising technology to support computation-intensive parallel applications to execute on loosely-coupled and distributed computing resources. Our main focus is on how to enable MPI applications to run in such an environment, hiding heterogeneous networks, geographically separated resources, and multiple administrative domains from application developers and users. To achieve this objective, we addressed workflow scheduling, resource allocation and co-allocation, and cross-site communication at the user level. We specified the workflow scheduling, task grouping and message relay services, and developed an integrated framework for our portable and adaptable SGR system.

In this dissertation, we defined workflow structures to specify the execution process of an MPI application, which involves multiple tasks executed on a computing grid. We introduced a new resource allocation model, which allows application users to perform resource allocation and co-allocation efficiently without relying on sophisticated scheduling algorithms, accurate job execution time prediction, or privileged

system support, such as job preemption. Based on our model and its specification, we implemented the SGR system for multi-task workflow MPI applications to run on Globus-enabled computing grids. Our SGR system provides a location-, topology- and administration-transparent computing environment for application developers and users. We used a simulation environment to evaluate our task scheduler, and obtained clear performance improvement by using duplicated resource allocation. We also validated the simulation results by evaluating resource duplication in our two-cluster grid environment. Performance of Tomosynthesis 3D reconstruction across the two clusters based on the SGR system are also evaluated.

The initial objective of this research was to find a viable method to allow us to execute the Tomosynthesis application and other similar applications on a multi-site computing grid. While some researchers have relied on system level services to enable cross-site resource sharing, and network reconfiguration to enable cross-site communication, our user-level approach demonstrates its unique ability to achieve even better performance without requiring additional services or system-level support. Because we consider common functions which are supported by most computing platforms in the framework design, and use the standard API in the system implementation, our solution is highly portable to heterogeneous computing platforms and adaptable to a wide range of MPI applications. Furthermore, running at the user level, the SGR system can be easily fine-tuned for various individual applications in their specific computing environments.

Using our resource allocation model, applications can allocate resources at the earliest available time by employing duplicated resource requests on multiple computing sites. The SGR system manages all duplications and invalidates redundant

allocations once the application resource requirement is satisfied, eliminating any unnecessary waste of computing resources. This method allows users to handle resource allocations and multi-site resource coordination across heterogeneous administrative domains easily and efficiently, without resorting to privileged resource management or any mechanism that needs to be used to accurately predict job run times.

We studied the new resource allocation model in a simulation environment, given different sizes of computing grids, workload conditions, and different schedulers. Job duplication is shown to be an effective way to achieve better performance in resource allocation and co-allocation. Using just one duplicated job request can reduce the average queuing time for the global jobs by more than 40%, compared to the results of a simple scheduler without duplication. This improved performance is equivalent to the performance achieved by sophisticated schedulers which rely on accurate runtime prediction. We conducted similar tests on the Joulian-Keys grid using live workloads, and found that duplication improves performance by 15% over the *Qlen* scheduler, which ranks computing sites by the queuing length, validating our simulation results.

We described how to use the workflow structures to specify the execution process of a multi-task MPI application. A computation-intensive application, like our motivating application, Tomosynthesis Mammography, can be decomposed into multiple tasks and executed in a workflow. This method allows application developers to divide a large and complex problem into small and simple ones, so that they can focus on individual sub-problems without considering how to allocate and utilize computing resources efficiently throughout the whole execution process. Moreover, the workflow structures enable application users to tailor the workflow application for different purposes, thereby making the application flexible to meet various demands of individual

users.

Communication over heterogeneous networks has always been one of great challenges when deploying parallel applications on computing grids due to several limitations imposed on cross-site communication. Using the parallel forking structure of workflow, we can transform an MPI execution on multiple computing sites into multiple concurrent tasks. These concurrent tasks can use the standard MPI to communicate with one another via the message relay service. Since physical networks are transparent to the application developers, legacy MPI programs can run on a computing grid without source code modification. To implement the message relay service, we introduced the parallel client-server structure to achieve a high level of system performance and scalability. In our cross-site communication tests on the two-cluster grid, the results demonstrate the system's scalability with respect to the number of processes and the size of inter-site messages.

In summary, this dissertation introduced a new resource allocation model, specification of the SGR framework and implementation of the SGR system, while addressing challenges for deploying MPI applications on computing grids. We developed a practical solution that allows us to run Tomosynthesis Mammography, our motivating application, and other similar computation-intensive applications on a computing grid. Our system is optimized for performance, portability, adaptability, scalability and ease of use. Consequently, this system has a great potential to be employed by many other scientific applications on various computing platforms.

Bibliography

[1] Maui scheduler open cluster software. http://mauischeduler.sourceforge.net/.

[2] MPI forum. http://www.mpi-forum.org.

[3] MPI: A message-passing interface standard. http://www.mpi-forum.org/, 2003.

[4] Policy driven heterogeneous resource co-allocation with gangmatching. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, 2003.

[5] The TeraGrid: A primer. http://www.teragrid.org/, 2003.

[6] CSIM19 development toolkit for simulation and modeling. http://www.mesquite.com, 2005.

[7] K. Aida. Effect of job size characteristics on job scheduling performance. In *IPDPS '00/JSSPP '00: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–17, London, UK, 2000. Springer-Verlag.

[8] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. *AFIPS Conference Proceddings*, 30, 1967.

[9] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the grid using apples. 14-4:369 – 382, 2003.

[10] D. Bernholdt, S. Bharathi, D. Brown, K. Chancio, M. Chen, A. Chervenak, L. Cinquini, B. Drach, I. Foster, P. Fox, J. Garcia, C. Kesselman, R. Markel, D. Middleton, V. Nefedova, L. Pouchard, A. Shoshani, A. Sim, G. Strand, and D. Williams. The earth system grid:

Supportloadlevelering the next generation of climate modeling research. In *Proceedings of the IEEE*, volume 93:3, March, 2005.

[11] P. Bradley, K. Misura, and D. Baker. Toward High-Resolution de Novo Structure Prediction for Small Proteins. *Science*, 309(5742):1868–1871, 2005.

[12] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[13] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An architecture of a resource management and scheduling system in a global computational grid. In *Proceedings of the Fourth International Conference Exhibition on HPC in the Asia-Pacific Region*, volume 1, pages 283–289, 2000.

[14] S. Choi, K. Park, S. Han, S. Park, O. Kwon, Y. Kim, and H. Park. An NAT-based communication relay scheme for private-IP-enabled mpi over grid environments. In *International Conference on Computational Science*, pages 499–502, 2004.

[15] K. Czajkowski, I. Foster, and C. Kesselman. Resource co-allocation in computational grids. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 219 – 228, 1999.

[16] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, 2005.

[17] D. Feitelson. Packing scheme for gang scheduling. In *Lecture Notes in Computer Science*, volume 1162, pages 89 – 110. Springer-Verlag, 1996.

[18] D. Feitelson and L. Rudolph. The ANL/IBM SP scheduling system. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 295–303, London, UK, 1995. Springer-Verlag.

[19] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21, 1972.

[20] I. Foster and C. Kesselman. A metacomputing infrastructure toolkit. In *Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing*. SIAM, 1996.

[21] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2), 1997.

[22] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 15(3), 2001.

[23] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. V. Reich. The open grid services architecture, version 1.0, 2005.

[24] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5:237–246, 2002.

[25] D. G. Grant. Tomosynthesis: A three-dimensional radiographic imaging technique. *IEEE Trans. Biomed. Eng.*, 19, 1972.

[26] A. S. Grimshaw, W. A. Wulf, and C. T. L. Team. The legion vision of a worldwide virtual computer. *Commun. ACM*, 40(1):39–45, 1997.

[27] W. D. Gropp and E. Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[28] R. Henderson and D. Tweten. Portable batch system: External reference specification. Technical report, NASA, Ames Research Center, 1996.

[29] S. Herbert. Official administrator's guide to generic NQS. September 1994.

[30] K. Hwang and Z. Xu. *Scalable Parallel Computing*. MCB McGraw-Hill, 1998.

[31] IBM Corporation. *IBM LoadLeveler: User's Guide*, 1993.

[32] B. S. Jr., T. Finholt, I. Foster, C. Kesselman, C. Beldica, J. Futrelle, S. Gullapalli, P. Hubbard, L. Liming, D. Marcusiu, L. Pearlman, C. Severance, and G. Yang. NEESgrid: A distributed collaboratory for advanced earthquake engineering experiment and simulation. In *13th World Conference on Earthquake Engineering*, August 2004.

[33] N. T. Karonis, B. R. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *J. Parallel Distrib. Comput.*, 63(5):551–563, 2003.

[34] K. Keahey, M. E. Papka, Q. Peng, D. Schissel, G. Abla, T. Araki, J. Burruss, E. Feibush, P. Lane, S. Klasky, T. Leggett, D. McCune, and L. Randerson. Grids for experimental science: The virtual control room. In *In proceedings of the Challenges of Large Applications in Distributed Environments (CLADE)*, June 07, 2004.

[35] D. Kopans. *Breast Imaging 2nd ed.* Lippincoot Williams and Wilkins, 1997.

[36] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *In Proceedings of the 8th International Conference of Distributed Computer Systems (ICDCS)*, 1988.

[37] M. Maheswaran, S. Ali, H. J. Siegel, D. A. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop*, pages 30–, 1999.

[38] G. E. Moore. Cramming more components onto intergrated circuits. *Electronics Magazine*, 1965.

[39] A. Mu'alem and D. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543, 2001.

[40] T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, April 1989.

[41] V. Nefedova, R. Jacob, I. Foster, Z. Liu, Y. Liu, E. Deelman, G. Mehta, M. Su, and K. Vahi. Automating climate science: Large ensemble simulations on the TeraGrid with the GriPhyN virtual data system. *eScience Conference*, December, 2006.

[42] L. T. Niklason, B. T. Christian, L. E. Niklason, D. B. Kopans, D. E. Castleberry, B. H. Opsahl-Ong, C. E. Landberg, P. J. Slanetz, A. A. Giardino, R. M. Moore, D. Albagi, M. C. DeJule, P. A. Fitzgerald, D. F. Fobare, B. W. Giambattista, R. F. Kwasnick, J. Liu, S. J. Lubowski, G. E. Possin, J. F. Richotte, C.-Y. Wei, and R. F. Wirth. Digital tomosynthesis in breast imaging. *Radiology*, 205, 1997.

[43] D. C. Rapaport. *The Art of Molecular Dynamics Simulation*. Cambridge University Press, 2004.

[44] W. Smith, I. Foster, and V. Taylor. Scheduling with advanced reservations. In *Proceedings of the IPDPS Conference*, pages 127 – 132, 2000.

[45] M. Socolich, S. Lockless, W. Russ, H. Lee, K. H. K. H. Gardner, and R. Ranganathan. Evolutionary information for specifying a protein fold. *Nature*, 437, 2005, sept.

[46] P. Srisuresh and K. Egevang. Traditional IP network address translator (traditional NAT), 2001. RFC 3022.

[47] S. Suryanarayanan, A. Karellas, S. Vedantham, S. P. Baker, S. J. Glick, C. J. D'Orsi, and R. L. Webber. Evaluation of linear and nonlinear tomosynthetic reconstruction methods in digital mammography. *Acad. Radiol.*, 8, 2001.

[48] S. Suryanarayanan, A. Karellas, S. Vedantham, S. J. Glick, C. J. D'Orsi, S. P. Baker, and R. L. Webber. Comparison of tomosynthesis methods used with digital mammography. *Acad. Radiol.*, 7, 2000.

[49] T. Wu, A. Stewart, M. Stanton, T. McCauley, W. Phillips, D. B. Kopans, R. H. Moore, J. W. Eberhard, B. Opsahl-Ong, L. Niklason, and M. B. Williams. Tomographic mammography using a limited number of low-dose cone-beam projection images. *Med. Phys.*, 30(3), 2003.

[50] T. Wu, J. Zhang, R. Moore, E. Rafferty, D. Kopans, W. Meleis, and D. Kaeli. Digital tomosynthesis mammography using a parallel maximum-likelihood reconstruction method. *Medical Imaging: Physics of Medical Imaging*, 5368:1–11, 2004.

[51] J. Zhang, W. Meleis, D. Kaeli, and T. Wu. Acceleration of maximum likelihood estimation for tomosynthesis mammography. In *Proceedings of the ICPADS*, 2006.

[52] S. Zhou. LSF: Load sharing in large-scale heterogeneous distributed systems. In *Proc. Workshop on Cluster Computing*, 1992.