A gridRPC-based environment for grid computing

Ho, Quoc Thuan

2006

Ho, Q. T. (2006). A gridRPC-based environment for grid computing. Master's thesis,
Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/2473>

<https://doi.org/10.32657/10356/2473>

Nanyang Technological University

Downloaded on 26 Apr 2023 22:13:25 SGT

NANYANG TECHNOLOGICAL UNIVERSITY

A GRIDRPC-BASED ENVIRONMENT FOR GRID COMPUTING

Submitted to
School of Computer Engineering
in Fulfillment of the Requirements
for the Degree of Master of Engineering

by

Ho Quoc Thuan

Supervisor: Assoc. Prof. Cai Wentong

Division of Computer Science
School of Computer Engineering
Nanyang Technological University
Singapore 639798

June 2006

Acknowledgement

First of all, I would like to express my deepest gratitude and respect to my supervisor, Dr. Cai Wentong, who has been giving me much guidance, thoughtful criticism and invaluable suggestions. Without his support, the thesis would not be possible.

I highly appreciate Dr. Lee Bu Sung and Dr. Ong Yew Soon for their support and encouragement. Especially, I would like to thank other researchers of Grid Computing Group in Parallel and Distributed Computing Center, Nanyang Technological University, who gave me motivation and valuable ideas.

I would like to thank Mr. Ng Hee Khiang and Mrs. Ng-Goh Siew Lai, who helped me to build Grid environment for the experiments. I thank Mr. Dudy Lim for his cooperation in developing the Graphical User Interface for the Grid Application Deployment Kit.

I would also like to thank the Nanyang Technological University for giving me opportunities and environments to do research.

Finally, I would like to thank my wonderful wife, Kathy, for her love and support over the years.

Contents

1	Introduction	1
1.1	Overview of Grid Computing	1
1.2	Motivation	2
1.3	Goals and Approaches	3
1.4	Organization of the Report	4
2	Related Work	6
2.1	Overview	6
2.2	Tools and Environments for Grid Computing	9
2.2.1	Globus	9
2.2.2	Nimrod-G	12
2.2.3	AppLeS	13
2.2.4	GrADS	14
2.2.5	iScheduler	15
2.2.6	Condor-G	16
2.2.7	Summary	17

Master of Engineering's Thesis**CONTENTS**

2.3	Remote Procedure Call	19
2.3.1	Overview of RPC	20
2.3.2	Performance Improvement for RPC	21
2.3.3	Summary	23
2.4	Remote Procedure Call on Grid Environment	23
2.4.1	NetSolve	25
2.4.2	Ninf-G	26
2.4.3	Summary	27
3	Architecture of GridRPC Execution Environment	29
3.1	Overview	29
3.2	Grid Client	31
3.2.1	Overview	31
3.2.2	Classification of Grid Resources	32
3.2.3	States of GridRPC Requests	33
3.2.4	Resource Monitor	33
3.2.5	Metascheduler	34
3.2.6	Request Dispatcher	35
3.2.7	Service Monitor	36
3.3	Grid Resources	36
3.3.1	Overview	36
3.3.2	Mechanism of GridRPC Services	37

Master of Engineering's Thesis

CONTENTS

3.4	Information Service	38
3.4.1	Overview	38
3.4.2	Sample of Information Service	39
3.4.3	Retrieving Information	40
3.5	Implementation	41
3.6	Summary	42
4	Composite Service	43
4.1	Introduction	43
4.2	Mechanism of Composite Service	44
4.3	Performance of Composite Services	48
4.4	Performance Improvement	50
4.5	Summary and Discussions	55
5	Metascheduling and Bundling Mechanisms for GridRPC Applications	56
5.1	Introduction	56
5.2	Metascheduling Mechanism	57
5.2.1	Resource Characterization	58
5.2.2	Resource Selection Heuristics	60
5.3	Request Bundling Mechanisms	61
5.3.1	Unbundled Mechanism - Immediate Dispatching	61
5.3.2	Unbundled Mechanism - Background Dispatching	63
5.3.3	Static Bundling	63

Master of Engineering's Thesis**CONTENTS**

5.3.4	Dynamic Bundling Mechanism	64
5.4	Summary and Discussions	67
6	Performance Analysis	68
6.1	Introduction	68
6.2	Performance of GridRPC Mechanisms on A Grid Resource	69
6.2.1	Immediate Dispatching versus Background Dispatching	72
6.2.2	Unbundled versus Dynamic Bundling	74
6.2.3	Dynamic Bundling versus Static Bundling	75
6.2.4	Effects of Maximum Waiting Time in Dynamic Bundling	79
6.2.5	Summary	81
6.3	Performance of GridRPC Mechanisms on Multiple Grid Resources	82
6.3.1	Single Application	84
6.3.2	Multiple Applications	86
6.3.3	Comparison between Mincost, Mincost-O, and Random Heuristics	87
6.3.4	Summary	88
6.4	Conclusions	89
7	GridRPC Programming Environment	91
7.1	Introduction	91
7.2	GAD Kit Architecture and Design	94
7.2.1	GAD Kit's GUI	94
7.2.2	Service Provider	95

Master of Engineering's Thesis

CONTENTS

7.2.3	Service Consumer	97
7.3	Realistic Airfoil Analysis and Design on the Grid using GAD Kit	99
7.4	Conclusions	102
8	Conclusions and Future Work	103
8.1	Work Done	103
8.2	Contributions	104
8.3	Future Work	106
	Appendix	115
A	Publications	115

List of Figures

2.1	RPC model	20
3.1	GridRPC Architecture	30
3.2	Architecture of a GridRPC Client	31
3.3	GridRPC Request States	34
3.4	Architecture of a Grid Resource	36
3.5	Mechanism of GridRPC Service	37
3.6	Architecture of Information Service	39
3.7	GRIS's Information	39
4.1	Mechanism of Composite Service	44
4.2	Scheduling Steps of a Composite Service	47
4.3	Execution under Different Degrees of Concurrency	49
4.4	Response Time of Composite Service under Different Degrees of Concurrency	51
4.5	New Scheduling Algorithm of NetSolve Agent	52
4.6	Performance of New Local Scheduling Algorithm	54
5.1	Unbundled Mechanism - Immediate Dispatching	62

5.2	Metascheduling and Dynamic Bundling Mechanism	64
5.3	Metascheduling/Dynamic Bundling Algorithm	66
6.1	Interarrival Times of GridRPC Requests	71
6.2	Makespan of GridRPC Applications offered by <i>UBD</i> and <i>UID</i>	72
6.3	Traces of Primitive GridRPC Requests in <i>UBD</i> and <i>UID</i> Mechanisms .	73
6.4	Makespan offered by <i>DB</i> and <i>UBD</i> Mechanisms	74
6.5	Traces of GridRPC Requests in <i>DB</i> and <i>UBD</i> Mechanisms	74
6.6	Number of Dispatchings in <i>DB</i> and <i>UBD</i> Mechanisms	75
6.7	Interarrival Times of GridRPC Requests	76
6.8	Makespan Offered by Various Bundling Mechanisms	77
6.9	Traces of GridRPC Requests in Dynamic and Static Bundling Mechanisms	78
6.10	Time Improvement of <i>DB5</i> and <i>DB10</i> versus <i>DB1</i>	80
6.11	Arrival of GridRPC Requests	82
6.12	Makespan of Single Application Cases	84
6.13	Traces of GridRPC Requests in <i>P1 – 10</i>	84
6.14	Concurrent GridRPC Applications	85
6.15	Makespans of Multiple Applications Case	86
6.16	Traces of GridRPC Requests in Testcase 1	86
6.17	Mincost and Mincost-O in Various Scenarios	88
6.18	Makespan Comparison of Various Resource Selection Heuristics	88
7.1	GAD Kit's Architecture	93

Master of Engineering's Thesis

LIST OF FIGURES

7.2	Front-end GUI of GAD Kit	93
7.3	GFD Specification	94
7.4	Service Provider Interface	95
7.5	GFD of a Grid service	96
7.6	Service Consumer Interface	97
7.7	A Sample Application Program generated by GAD Kit	98
7.8	“Gridifying” of the GA for Aerodynamic Design	100
7.9	Makespans of Analysis of Airfoil Designs	102

List of Tables

2.1	Summary of Grid Computing Tools and Environments	18
2.2	Optimizations for Remote Procedure Call Paradigm	23
2.3	GridRPC API	24
2.4	NetSolve	25
2.5	Ninf-G	26
3.1	Implementation Description	41
4.1	Configuration of Two Clusters	50
6.1	Configuration of A Grid Resource	72
6.2	Configuration of Benchmarks	83
6.3	Configuration of Testbed	83
7.1	Configuration of Testbed for Airfoil Analysis and Design	101
8.1	List of Work Done	103

Chapter 1

Introduction

1.1 Overview of Grid Computing

A couple of decades ago, the idea of using multiple distributed resources to cooperatively work on a single application was proposed. It has been termed as “network operating systems” at late seventies, then as “distributed operating systems”, “heterogeneous computing”, “metacomputing”, and “Grid computing” as nowadays [SN02]. The term Grid has been defined as a set of resources distributed across Wide-Area Networks that can support large-scale distributed applications [FK98b]. Its meaning was borrowed from electric power grid, where electric power can be accessed easily, pervasively, and following standard as plugs and outlets for electric appliances. The Grid problem is defined as *“flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources”* [FKT01]. Here, the resource concept has a broadened meaning that includes data, computers, scientific devices, software, etc. A *virtual organization* is formed by a set of participants that share resources with various relationships in a dynamic scenario. The goal of a Computational Grid is to enable easy, fast, and inexpensive access to resources across an organization, state, country, or even the world.

1.2 Motivation

- Demands of Automatic Resource Discovery and Metascheduling

Grid computing offers many opportunities for complex and highly parallel applications by providing access to a huge amount of shared resources. However, accessing and utilizing Grid resources are still hindered because of the following issues.

- Grid resources are heterogeneous, i.e., they have different capabilities, platforms, administrative policies, etc.
- Grid resources are dynamic and shared by many users simultaneously. Therefore, aggregating resources while achieving high performance is challenging.
- Since Grid resources are geographically distributed over Wide Area Network (WAN), network latency and delays are considerable barriers for the communication and synchronization among components on the Grid. Hence, problems in distributed computing such as resource contention, load balancing, and scheduling, etc. become more complicated.

Therefore, it is necessary to have mechanisms to hide the complexity of the Grid environment and at the same time achieve high performance for applications. Resource discovery and scheduling are two crucial factors contributing to the performance of applications on the Grid.

- Demands of High-level Interface and Environment for “Gridifying” Applications

Presently, there are many tools and mechanisms supporting users to access Grid resources. The users can program their applications in terms of traditional parallel and distributed programming paradigms using tools and libraries such as MPICH-G2 [KTF03]. They can also develop and execute their applications using new libraries, frameworks, or specific Problem Solving Environments (PSEs) that are

specifically developed for Grid computing such as Globus Toolkit [FK98a], Cactus-G [GAL⁺03], Grid Portal Toolkits [TDM⁺02], and GrADS [BCC⁺01], etc. However, the demands of a high-level interface to address the lack of standardized, portable, and simple programming interface still remain.

In summary, the research reported in this thesis is motivated by the following demands.

- The need for mechanisms to improve the performance of Grid-enabled applications.
- The need for a high-level programming environment that hides the complication of the underlying Grid environment.

1.3 Goals and Approaches

In this work, we focused on solving two issues in Grid Computing as follows.

- Make the Grid transparent to users.
- Improve performance of applications on Grid environment.

Firstly, in order to make the Grid transparent to users, we developed an environment in which jobs can be submitted to the Grid resources in such a manner that the complexity of the underlying system is hidden. Motivated by a simple and effective interface of GridRPC [SNM⁺02], a standard API for RPC applications on the Grid, we developed mechanisms and techniques so that users can have a seamless and transparent access to Grid resources using the GridRPC API.

Secondly, in order to improve the performance of the GridRPC applications, we examined the performance of the local resource manager and the metascheduler. In this

work, we focused on multi-cluster Grid which contains multiple clusters of computers. We examine their performance under different degrees of concurrency, i.e., the number of jobs allowed to run on one processor at the same time. In order to exploit the parallelism provided by a cluster of computers and to reduce communication overhead, composite services that distribute jobs onto different computers were introduced. A load balancing mechanism for composite services within a Grid resource was provided. In order to make use of composite services, the metascheduler at the GridRPC client was equipped with the bundling mechanisms.

1.4 Organization of the Report

The remainder of this report is organized as follows.

- Chapter 2 presents related work in Grid computing. A brief survey of mechanisms to access Grid resources and scheduling mechanisms is also introduced in this chapter.
- Chapter 3 shows the architecture of the GridRPC environment. In this chapter, a concrete design of GridRPC execution environment is detailed.
- Chapter 4 describes the mechanism of the composite services. We also examine the effect of degree of concurrency and introduce a simple load balancing mechanism in a cluster of computers.
- Chapter 5 presents metascheduling mechanisms. We present various mechanisms of request bundling and heuristics of resource selection.
- Chapter 6 presents a performance analysis of bundling and metascheduling mechanisms. Unbundling, static bundling, and dynamic bundling mechanisms are examined in various scenarios.

- Chapter 7 describes the design and implementation of the GridRPC programming environment. “Gridifying” an airfoil analysis application is also presented here.
- Finally, Chapter 8 concludes the work done and outlines the future work.

Chapter 2

Related Work

2.1 Overview

Recently, there are many tools and environments supporting Grid computing. They can be classified into the following categories:

- Infrastructures

An infrastructure provides basic services that enable access to Grid resources. Recently, Globus is a *de-facto* standard toolkit for Grid computing. It provides a wide range of services such as resource management, information service, security management, and data communication. However, in order to use mechanisms provided by the infrastructure, many efforts are still needed since Grid is complex.

- Local Resource Managers

A resource manager, also called local scheduler, manages local resources of an organization. When the organization shares its resources to the Grid users, the local resource manager controls the resource usage and schedules jobs on its resources in such a manner that some system-centric performance metrics such as workload balance index and throughput are optimized. Condor [LLM88], PBS [Alt02],

SGE [Sun01], and Maui [JSC01] are some well-known resource managers for Grid computing.

- Metascheduler

A metascheduler aggregates resources of different systems and at the same time delivers high performance to Grid-enabled applications. It carries out resource selection among different Grid resources for several applications running concurrently. Since the metascheduler operates between resource managers and application-level schedulers, it takes into account performance criteria of both resource side and application side. The metaschedulers included in Condor-G [FTL⁺02], GrADs [BCC⁺01], iScheduler [WAE03], and Silver [Jac04] are some well-known examples.

- Application-level Scheduler

The application-level scheduling approach focuses on achieving performance-efficient schedules for applications. An application-level scheduler performs many tasks such as resource selection, performance prediction, and interacting with resources in order to implement the application. It carries out these tasks from the point of view of the application to obtain high performance for the application. This makes application-level scheduler different from LRM and metascheduler. AppLeS [BWC⁺03] is a well-known example of an application-level scheduler for Grid computing.

- Grid Computing Environments

A Grid computing environment provides mechanisms to deploy and execute Grid-enabled applications. Typically, each Grid computing environment has a development environment and an execution environment. GrADS and Nimrod-G [AGK00] are well-known Grid computing environments. A metascheduler or an application-level scheduler is often included in the environment to assist applications to achieve

high performance.

The performance of applications on the Grid depends on how the applications are scheduled, how the Grid resources are selected, and how these Grid resources execute arriving jobs. These issues occur on different components involved in the execution of the application. Therefore, performance improvement can be carried out on different components such as application-level schedulers, metaschedulers, and local resource managers as follows.

- Improve performance via application-level scheduler:

An application-level scheduler schedules application jobs in such a manner that the application can achieve high performance. It discovers, selects resources, and schedules application jobs to the selected resources on behalf of the applications so that their requirements such as time or cost of computation are satisfied. In order to achieve the goals, an application-level scheduler must have the ability to capture the requirements of applications. Therefore, they are usually tightly coupled with applications. AppLeS [BWC⁺03] (using application templates) and the scheduler of Nimrod-G (using declarative scripting language) are some well-known application-level schedulers which will be discussed in this chapter.

- Improve performance via metascheduler:

A metascheduler receives jobs from many applications. It offers high performance for different applications and may consider performance of Grid resources. In order to improve the performance, different mechanisms have been developed. Contract negotiation, resource harvest, and advance resource reservation are different mechanisms offered by the metascheduler. GrADS [VD02] with contract negotiation policy, Silver [Jac04] with advance resource reservation, iScheduler [WAE03], and Condor [LLM88] with resource harvest are well-known metascheduler and metaschedul-

ing policies. The details of GrADS and iScheduler will be presented in a later section.

- Improve performance via resource manager:

Each resource manager has responsibility for allocating its local resources to Grid applications. The key goal of resource managers is to improve performance of the local system according to some performance metrics such as throughput, average response time, and workload balance. By offering high performance of local resources, individual resource managers can contribute their resources to Grid applications efficiently. Portable Batch System [Alt02], Sun Grid Engine [Sun01], Maui Scheduler [JSC01], and Condor-G's matchmaker [FTL⁺02] are popular resource manager systems in the literature.

In the next section, we examine in detail the state-of-the-art of the tools and environments supporting Grid computing with regard to the above performance issues.

2.2 Tools and Environments for Grid Computing

In this research, we focused more on the infrastructures and execution environments for Grid computing in which mechanisms that provide access to Grid resources and attain high performance for Grid applications are equipped. Firstly, we present an overview of Globus [FK98a], a *de-facto* standard toolkit of Grid. Secondly, Grid computing environments, application-level schedulers, and metaschedulers such as Nimrod-G [AGK00], GrADS [BCC⁺01], AppLeS [BWC⁺03], and iScheduler [WAE03] are discussed.

2.2.1 Globus

Globus is one of the early pioneers providing infrastructures for Grid computing. The Globus Toolkit can be considered as the *de-facto* standard toolkit for the Grid. The heart

of a Globus system is the Globus Toolkit, which consists of a set of components that implement basic services, such as security, resource location, resource management, and communications. These services are glued together in order to enable Grid applications to handle distributed heterogeneous computing resources as a single virtual machine.

Globus's architecture consists of five layers, namely, (1) Application, (2) Collective, (3) Resource, (4) Connectivity, and (5) Fabric, in which high-level services are built upon essential low-level services. The Globus Toolkit version 2 (GT2) provides the following basic services:

- Globus Resource Allocation Manager (GRAM) has responsibility for allocation of computational resources and control of computation on those resources.
- Monitoring and Discovery Service (MDS) provides fundamental mechanisms for discovering and monitoring distributed resources so that applications can discover and select resources for their needs.
- Grid Security Infrastructure (GSI) is employed for enabling secure authentication and communication over an open network.
- Global Access to Secondary Storage (GASS) provides mechanisms to access remotely data via sequential and parallel interfaces. GASS includes an interface to GridFTP.

At the fabric layer, resource managers such as Condor [LLM88], PBS [Alt02], and SGE [Sun01] are employed as local services that schedule, execute, and manage computational tasks of Grid applications. Basically, users can use the Globus API directly to access resources. However, using the Globus API directly to develop applications is inefficient because it is a low-level interface. In the upper layers, various kinds of high-level

services operating on top of Globus services provide a seamless and transparent interface to the Grid users. By using high-level services and tools such as Cactus [ADF⁺01], MPICH-G2 [KTF03], Nimrod-G [AGK00], Condor-G [FTL⁺02], etc., users can access Grid resources more easily and effectively than directly using low-level services provided by Globus Toolkit.

At the time when this report was written, the latest version of Globus, GT3 is developed based on Open Grid Services Infrastructure (OGSI) [TCF⁺03]. It has a different architecture from GT2 since it targets Open Grid Service Architecture (OGSA) in which computational resources, storage resources, networks, programs, databases, and the like are all represented as services [FKNT02]. In GT3, Grid technologies and Web services are integrated to create an infrastructure for applications. Hence, mechanisms to access Grid services are changed considerably from GT2. In GT3, accessing Grid resources is performed through the following standard interfaces defined by OGSA:

- Discovery: enables clients to discover available services and determine characteristics of those services.
- Dynamic service creation: enables clients to use a standard service called “factory” to create a new service.
- Lifetime management: manages transient and stateful services so that reclaiming services and their state is enabled.
- Notification: enables services to notify others asynchronously when their states change.
- Manageability: manages and monitors a large number of Grid service instances.
- Simple hosting environment: is a set of resources providing native facilities for service management in a single administrative domain.

Briefly speaking, OGSA has introduced new concepts and interfaces to access Grid resources. It can also be considered as a “shifting” of technologies where traditional interfaces are replaced by Web services. However, the underlying systems still encounter performance issues such as resource selection, job scheduling, network latency, etc.

2.2.2 Nimrod-G

Nimrod-G [AGK00] was built to address the problem of performing a large number of parameterized simulations on a set of distributed computers, where each simulation has a different parameter set.

Nimrod-G provides a GUI and a declarative template assisting users to perform parametric modeling. Through the GUI, Nimrod-G captures specifications of the experiments, transforms them into a declarative scripting documents and then processes demanded tasks.

The underlying system of Nimrod-G was built on top of the Globus so that it can utilize Grid resources. In order to attain high performance, Nimrod-G schedules applications based on the following parameters:

- Resource Cost (that is set by its owner)
- Price (that the user is willing to pay)
- Deadline (the period by which an application execution need to complete)

This approach offers flexible choices between time and cost for users. The scheduling policy was termed “Computational Economy” in which the Grid is considered as a marketplace where consumers and producers can negotiate for Grid resources [BSGA01].

2.2.3 AppLeS

Application Level Scheduler (AppLeS) [BWC⁺03] is a user-centric scheduling system that provides an efficient access to computation Grid. It focuses on adaptive scheduling for Grid applications so that effective performance can be extracted for the end-users. AppLeS employs existing services provided by the infrastructure such as Globus services, Network Weather Service (NWS) [Wol98] in order to schedule tasks of applications on Grid resources.

In AppLeS, each application has its own AppLeS agent for scheduling. Each agent has responsibilities for resource discovery, resource selection, schedule generation, schedule selection, application execution, and schedule adaption. Given an ordered list of feasible resource sets, the AppLeS agent applies a performance model to determine a set of candidate schedules for the application on potential target resources. It subsequently chooses the “best” overall schedule that matches the users’ performance criteria, e.g., execution time, turnaround time, convergence, etc.

In order to assist users to achieve high performance for different disciplinary applications, AppLeS provides some “templates” that extract common characteristics from various similar (but not identical) AppLeS-enabled applications. Each template targets a class of applications. These templates are effective to construct a performance model for scheduling. Recently, two AppLeS templates, APST [COBW00] targeting parameter sweep applications, and AMWAT [Sha01] targeting master/worker applications, have been developed and demonstrated. A summary of applications using AppLeS can be found in [BWC⁺03].

In particular, the Supercomputer AppLeS (SA) [CB03] has been developed to solve the problem of resource competition, called “the Bushel of Apples” [BW97], in which multiple AppLeSs co-exist and share the same supercomputer. By introducing “moldable jobs”,

an SA groups a number of jobs from different applications and puts them to the suitable “holes” in a space-shared parallel supercomputer. Thus, performance of all concurrent applications can be improved. However, when not all applications use AppLeSs and these applications have different overlapping sets of resources, the performance is questionable due to the resource competition problem.

2.2.4 GrADS

Grid Application Development Software (GrADS) [BCC⁺01] is a program execution environment developed to simplify distributed heterogeneous computing in the same way that the World Wide Web simplifies information sharing over the Internet. Further, GrADS focuses on tuning performance of real applications.

In GrADS, developing and executing an application consist of several phases that follow two key principles. First, an application must be encapsulated as a “configurable object program”, which can be optimized rapidly for execution on a specific collection of Grid resources. Second, the system relies upon “performance contracts” that specify the expected performance of modules as a function of available resources.

At the beginning of the development process, GrADS Program Preparation System (PPS) constructs configurable object programs from basic components or modules provided by the users. It frees the users from many of the low-level concerns so that they can focus on high-level design and performance tuning matters. Subsequently, the configurable object programs are delivered to the GrADS Program Execution System (PSE) that performs resource discovery, resource selection, processing, contract guarantee, and optimization via a set of assisting components such as sensors, actuators, monitors, etc.

In order to achieve high performance for Grid-enabled applications, GrADS has three types of scheduling strategies suitable for different classes of applications.

1. Launch-time scheduling

The launch-time scheduler in GrADS is responsible for mapping an application onto available Grid resources at the launch-time of the application. GrADS is equipped with resource requirements specification and heuristic scheduling algorithm (simulated annealing) with regard to different scheduling objectives such as equal allocation, time balancing, and data locality.

2. Rescheduling

In order to adapt execution of long running applications on a dynamic environment, GrADS provides a rescheduling mechanism to support migration and dynamic load balancing. In order to support this, a set of GrADS's functional services such as contract monitor, migration library, and process swapping mechanism for MPI applications have been developed. The rescheduling mechanism takes into account both performance benefits and overhead of migration.

3. Metascheduling

To tackle the issue that multiple applications run at the same time on shared resources that may lead to resource competition, metascheduling mechanisms are investigated in GrADS. The metascheduler takes into account both the needs of the applications and the overall performance of the system. In addition, the metascheduler has control over all applications running on the specified resources and also the ability to schedule any of those applications at any time.

2.2.5 iScheduler

While resource managers and application-level schedulers co-exist, their goals may be opposed to each other and thus, none of them can achieve their objectives. The iScheduler [WAE03], developed by Jon B. Weisman et al., aims to bridge the gap between

resource managers and application-level schedulers. The key feature of iScheduler is the ability to add or remove resources from a running application in order to leverage execution of other applications with regard to performance of both applications and system.

The decision is made when an application arrives, completes, an active application requests resources, and when an application queue time or application age exceeds a threshold. Upon receipt of an event, iScheduler allocates, harvests resources (i.e., de-allocates resources from some applications and allocates them to other applications), or queues applications accordingly.

Many resource allocation policies are supported by iScheduler. The allocation can favor queued applications or running applications. Resources from running applications can be harvested using harvest policies. In order to improve both application and system performance, iScheduler takes into account an application model and user-centric workload model. Applications are provided with an API so that they can interact with iScheduler and thus enable iScheduler to implement their resource management policies. Many simulations have been conducted to compare iScheduler with other mechanisms such as fixed resource allocation, moldable job scheduling, and backfilling. In most cases, iScheduler outperformed other mechanisms in both waiting time and total time (finishing time) [WAE03].

However, in order to deploy iScheduler in a real system, it must have the ability to dynamically obtain application's information, e.g., execution cost. In addition, iScheduler requires an API for adding and removing resources from a running application.

2.2.6 Condor-G

Condor-G [FTL⁺02] is a combination of Condor [LLM88] and Globus. In Condor-G, resources within a single administrative domain are managed as in Condor. Similar to

LSF, SGE, and PBS, Condor provides job management mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Different types of jobs such as serial jobs, parallel jobs (MPI or PVM programs) can be executed in Condor. Especially, jobs with dependency can be scheduled and monitored by DAGMan scheduler. Moreover, with checkpointing and migration capabilities, Condor enables preemptive-resume scheduling as well as the execution of heavy jobs.

Basically, a Condor pool consists of an agent, a match maker and a set of resources. Jobs are submitted to the agent. The agent and resources publish their information to the matchmaker. The matchmaker will inform two compatible parties and let the agent contact the resources to execute jobs. If there are multiple Condor pools working together, they can employ gateway flocking or directly flocking mechanisms to distribute jobs across multiple Condor pools.

Condor-G employs Globus technologies to achieve interoperability among different resources. This enables submitting jobs to a remote machine through the Globus interface. In addition, external users can submit jobs to a Condor pool, i.e., a set of Condor machines, through the Globus interface. However, with the Globus interface, Condor pools cannot work together as in the flocking mechanisms because communication must pass through GRAM. In order to solve this problem, Condor-G provides another mechanism called “*gliding in*” in which the matchmaker started by the user virtually creates an *ad hoc* personal Condor pool from the existing Condor pools that span across multi-domains. Here, Condor-G’s matchmaker acts a metascheduler.

2.2.7 Summary

Table 2.1 presents a brief summary of existing Grid Computing Tools and Environments that provide access to Grid resources and improve performance for applications and systems.

Systems	Category	Comments
Globus	Infrastructure	Provides basic functionalities to access Grid resources
AppLeS	Application-level scheduler	Uses templates to capture characteristics of Grid applications and schedules applications accordingly
GrADS	Grid computing environment	Supports metascheduling, incorporates preemptive scheduling into program development environment and supports task migration
Nimrod-G	Grid computing environment	Selects resources for applications based on time and cost of resources
iScheduler	Metascheduler	Provides preemptive scheduling with many policies w.r.t performance of both applications and system
Condor/Condor-G	Local resource manager, metascheduler	Provides preemptive scheduling and executes jobs across multi-domain

Table 2.1: Summary of Grid Computing Tools and Environments

In order to improve the performance of applications, most systems map the application performance model to the system model in such a manner that some performance metrics are satisfied. According to which goals are considered, each system has its own strategy to support execution of Grid-enabled applications. The objective performance parameters to be optimized in these systems can be listed as follows.

- Application-related metrics:
 - Total execution time
 - Timing constraint requirements
 - Cost satisfaction
- Systems-related metrics:
 - Throughput

- Response time
- Fairness

Based on which objectives are considered, each system has its own strategy to support execution of Grid-enabled applications. Although current Grid environments can assist in applications development, few of them take into account the fact that resources can be shared by different applications that may compete for resources with each other. Following are unsolved issues of the above mentioned Grid environments.

- Lack of scheduling solution for decentralized environments.

The problem is exposed when there are different clients using shared resources concurrently. However, these clients do not negotiate for sharing resources, and neither do they submit their jobs through a centralized scheduler. It is impossible to force all clients submitting their jobs through a single entry or manage all resources under a central management.

- Out-of-date resource information.

Due to network latency, the information obtained from resources may not be up-to-date enough for making an accurate scheduling decision. Especially, resource competition caused by a large number of simultaneous applications is difficult to solve on a decentralized environment.

2.3 Remote Procedure Call

In this section, we explore Remote Procedure Call (RPC) paradigms. RPC interfaces are simple, easy to use, and they effectively hide the complexity of distributed systems. In addition, they can support different types of high performance applications using various programming models, such as data parallelism, task parallelism, and master/slave model.

2.3.1 Overview of RPC

Remote Procedure Call is one of the programming paradigms in distributed computing. In an RPC program, a remote procedure call is similar to a local procedure call except that the remote procedure call consumes computational power on a remote machine. When a remote procedure is invoked, the input parameters are passed across the network to the remote machine where the execution of the procedure is actually carried out. After the procedure finishes and produces results, the results are sent back to the caller in the similar manner of a local machine call. The implementation of remote procedure call was first introduced by Andrew D. Birrell et al. [BN84].

Figure 2.1: RPC model

Figure 2.1 presents the RPC model. An RPC system has two types of components: clients and servers. They communicate over the network using runtime systems that send or receive messages. It requires some local procedures, called stubs, to support remote procedure call. These stubs have responsibility for packing or unpacking messages before they are sent over the network or forwarded to the application layer, respectively. In addition, the client stub can block execution of the client application when necessary and the server stub spawns the server application processes.

2.3.2 Performance Improvement for RPC

Multithreading Technique

In order to improve the performance of an RPC program, RPC calls should be served in parallel on different servers if these requests are independent. In fact, most of the RPC systems provide an asynchronous call mechanism that enables an RPC program to send multiple requests simultaneously. An asynchronous call returns immediately after it passes arguments data to the client stub. In order to collect the results of the asynchronous call when necessary, the RPC program can call another function to wait for the results. Combination of asynchronous calls and waiting functions enables RPC programs to achieve flexibility.

In order to implement asynchronous calls, a multithreading technique is employed. When an asynchronous call is invoked, another thread of execution, beside the one that executes the RPC program, will handle the call. This thread runs in the background, passes data across the network and waits for the return of results. If the waiting function is made by the RPC program, the background thread will handle this request and pass the result to the RPC program once the result is ready. This mechanism is supported in many RPC systems such as Alhorn RPC [AGH⁺91], DCE RPC [RD96], and Sun RPC [Sun94] systems.

Sequencing Technique

Another interesting approach to reduce communication cost for RPC programs has been proposed in the DFN RPC [Rab95] system and NetSolve [ABD00]. Communication is reduced by keeping the results of some RPCs at the server as input for further dependent RPCs. This technique has been termed “buffered call” in DFN and “request sequencing” in NetSolve. It is very effective when applications require a large amount of communica-

tion between RPC client and RPC server.

Brokering Technique

Since the RPC systems have a large number of heterogeneous servers, discovering and selecting suitable servers for the clients' needs are difficult. A new component is added to the RPC run-time system to tackle this issue. This component acts as an agent or a broker that maintains a list of servers and their status. Whenever a client has a request, it asks the agent for a list of suitable servers that are capable of processing the request. This approach has been investigated and examined in NetSolve [ACD02] and Ninf [NSS99].

In NetSolve, an "agent" functions as a resource broker. When a client has a request, it sends information of the request to the agent. Based on the complexity and input data size provided to the RPC, the agent evaluates all servers that are able to serve the request and replies with a ranked list of servers back to the client. Subsequently, the client submits the request to the first server in that ranked list.

In contrast to NetSolve, Ninf has a meta-server acting as an "execution proxy". Instead of suggesting a ranked list of servers for the clients, the meta-server forwards the requests to the most suitable server. When the server accomplishes the RPC, results are sent back to the meta-server which will forward these results back to the client.

Actually, NetSolve's execution model is similar to the *handle-driven* model where the agent provides the address of the selected server for a client. In contrast, Ninf's can be considered as the *forwarding* model where the agent is responsible for forwarding the requests to the selected server and sending the results back to the client. These models of coordinations have been proposed and examined in [Adl95] and [AFM02], respectively.

2.3.3 Summary

Table 2.3.3 summarizes some improvements of existing RPC implementations.

Techniques	Implementations	Purposes
Multithreading	Alphorn [AGH ⁺ 91] DCE [RD96] Sun [Sun94]	Avoids delays of synchronous calls
Sequencing	DFN [Rab95] NetSolve [ABD00]	Reduces communication cost between client and server
Brokering	NetSolve [ACD02] Ninf [NSS99]	Performs resource discovery and selection

Table 2.2: Optimizations for Remote Procedure Call Paradigm

Hiding communication overhead and latency by using a multithreading technique is employed to improve the performance of RPC client. On an environment with several servers, resource discovery and selection become crucial to the performance of RPC applications. When the communications have some dependencies, the sequencing technique can reduce the communication cost of RPC applications. These experiences are useful in development of a GridRPC-based environment for a multi-cluster Grid.

2.4 Remote Procedure Call on Grid Environment

Remote Procedure Call for Grid computing (GridRPC) [SNM⁺02] is a standard interface for RPC applications on Grid environments. It is proposed and supported by Grid Remote Procedure Call Working Group under Global Grid Forum [Gri]. It is simple, easy to use, and provides a seamless and transparent access to Grid resources. It is also known as a good means to hide the complexity of the underlying system. For instance, users can use a simple call in a GridRPC program and then harvest results from remote

resources regardless of their locations. Moreover, using asynchronous calls, GridRPC applications can aggregate multiple resources to carry out their tasks simultaneously. Recently, GridRPC are supported by Ninf-G [NTMS03] and NetSolve [VSM02].

Functions	Meaning
<i>grpc_initialize</i>	Initialize GridRPC system
<i>grpc_finalize</i>	Release any resources being used by GridRPC
<i>grpc_function_handle_default</i>	Get a function handle using the default server
<i>grpc_function_handle_init</i>	Allow explicitly specifying the server
<i>grpc_function_handle_destruct</i>	Release the memory allocated for this handle
<i>grpc_get_handle</i>	Get the function handle using a given session ID
<i>grpc_call</i>	Blocking remote procedure call
<i>grpc_call_async</i>	Non-blocking remote procedure call
<i>grpc_call_argstack</i>	Blocking call using argument stack
<i>grpc_call_argstack_async</i>	Non-blocking call using argument stack
<i>grpc_probe</i>	Check whether a call has completed
<i>grpc_cancel</i>	Cancel a previous call
<i>grpc_wait</i>	Wait for a specific call to complete
<i>grpc_wait_and</i>	Wait for all calls in a specified set to complete
<i>grpc_wait_or</i>	Wait for any call in a specified set to complete
<i>grpc_wait_all</i>	Wait for all calls to complete
<i>grpc_wait_any</i>	Wait for any call to complete
<i>grpc_perror</i>	Print the error string of the last call
<i>grpc_error_string</i>	Get the error string given a numeric error code
<i>grpc_get_error</i>	Get the error code for a specified non-blocking call
<i>grpc_get_last_error</i>	Get the error code for the last call
<i>grpc_arg_stack_new</i>	Create a new argument stack
<i>grpc_arg_stack_push_arg</i>	Push the specified argument onto the stack
<i>grpc_arg_stack_pop_arg</i>	Get the top element of the stack
<i>grpc_arg_stack_destruct</i>	Free resources associated with the argument stack

Table 2.3: GridRPC API

In the GridRPC API standard, a “*function handle*” represents a mapping from a function name to an instance of that function on a particular server. Once created, calls using a function handle always go to that server. A “*session ID*” represents a previously

issued non-blocking call. It allows checking status, cancelling, waiting for results, or getting the error code of a non-blocking call. Function handle and session ID are often input parameters of functions in the GridRPC APIs. Table 2.3 shows the GridRPC API in brief.

2.4.1 NetSolve

NetSolve [ACD02] is a Network-Enabled Servers system developed at the University of Tennessee's Innovative Computing Laboratory, USA. A brief description of NetSolve is presented in Table 2.4.

Properties	Description
Interface	NetSolve API is used in C/Fortran/Matlab/Mathematica. GridRPC API can also be used in C programs.
Remote invocation	Server processes are spawned by NetSolve servers upon remote invocation.
Communication	TCP/IP sockets are used for communication.
Architecture	Client/Agent/Server.
Protocol for a request	<ol style="list-style-type: none"> 1. Client stub contacts agent to receive a ranked list of servers. 2. Client contacts "the best" server and sends input parameters. 3. Server runs specific service based on input parameters. 4. Server returns output results or error status to the client.

Table 2.4: NetSolve

An instance of a NetSolve system is a set of computer hosts accessible over the Internet. The system is based on client/server model. The NetSolve agent has functions of an information service and a resource broker. Currently, each NetSolve agent is associated with one instance of a NetSolve system. Hence, the agent is initialized first before all servers are started. Then, all servers must register them to the system through the NetSolve agent. A NetSolve client has to know its NetSolve agent before making RPC requests. As mentioned above, the execution model offered by NetSolve is similar to the *handle-driven ORB* that was introduced in [Adl95].

Deploying applications on a Grid environment using NetSolve has some shortcomings. First, NetSolve uses the agent as an information service as well as a scheduler for a set of clients sharing multiple servers. This introduces a single point of failure. Second, because its GridRPC implementation is based on its own protocol, NetSolve cannot be widely deployed to the Grid where most of resources are managed and operated using Globus, the *de-facto* standard toolkit for Grid computing. Third, most of the Grid resources are clusters of computers, but NetSolve has no mechanism to schedule GridRPC requests across multiple clusters. It cannot interact with the resource managers such as Condor, SGE, and PBS which are commonly used in clusters of computers. Finally, since its scheduler does not have sophisticated job management mechanism, its servers could easily be overloaded under high job arrival rates that often happen in embarrassingly parallel applications. Details of this issue will be further discussed in Chapter 4.

2.4.2 Ninf-G

Ninf-G [NTMS03] is a Grid-compliant version of Ninf [NSS99]. They were developed at Grid Technology Research Center, AIST, Japan. A brief description of Ninf-G is presented in Table 2.5.

Properties	Description
Interface	GridRPC API is used in C.
Remote invocation	Server processes are spawned by GRAM upon the remote invocation.
Communication	Globus I/O library is used for communication.
Architecture	Client/Server.
Protocol for a request	<ol style="list-style-type: none"> 1. Client checks for the required GridRPC service. 2. Client sends the request to server through server's GRAM. 3. GRAM checks security permission and spawns the server application. 4. Server application finishes and sends results to the client through the server stub.

Table 2.5: Ninf-G

In Ninf-G, the design of the GridRPC API relies on Globus Toolkit. The underlying execution of Ninf-G is delivered by Globus services. First, remote procedures in Ninf-G are registered to Globus MDS. Second, the Ninf-G invocation mechanism is based on Globus GRAM that is responsible for spawning server processes to carry out the remote procedures. Finally, data communication between clients and servers in Ninf-G is based on the Globus I/O library. Because of the way it is implemented, Ninf-G can be considered as a GridRPC wrapper of Globus services.

Although Ninf-G can be readily used to “gridify” applications, it still requires mechanisms to select Grid resources for GridRPC requests. Ninf-G does not provide metascheduling and selection mechanisms to users. These tasks are done manually before program execution. It is inefficient on a Grid environment in which there may be a large number of resources. In addition, when resources’ states change dynamically, using Ninf-G may not achieve performance gains for Grid-enabled applications.

2.4.3 Summary

Providing a seamless and transparent access to Grid resources while maintaining high performance presents the following issues:

- Effective programming interface: a programming interface should be easy to understand and use, and it should be able to explicitly express parallelism.
- Interoperability: different components of an application are able to communicate with each other.
- Flexibility: it should be able to handle problems caused by dynamic and decentralized properties of Grid environments.

The first issue is addressed by the GridRPC API. While traditional RPC systems have their own interfaces that are different from each other, the GridRPC API is a standard for Grid environments. In addition, the key feature of GridRPC is that it offers a simple interface so that users can easily “gridify” their applications regardless of the complexity of the underlying execution system.

Although current implementations of the GridRPC API help hide the underlying complexity in accessing Grid resources to a certain extent, they do not fulfill the full requirements for a seamless and transparent GridRPC interface. None of the systems mentioned above support metascheduling across multiple clusters of computers. In the subsequent chapters, we present the design and implementation of a GridRPC execution system that takes into account the above issues.

Chapter 3

Architecture of GridRPC Execution Environment

3.1 Overview

The GridRPC-based environment developed for Grid computing has two parts:

- GridRPC Programming Environment (GPE).
- GridRPC Execution Environment (GEE).

The GPE supports users in developing GridRPC applications. It includes facilities for creating GridRPC services and GridRPC client programs. Details of the GPE will be given in Chapter 7.

The GEE provides mechanisms to execute GridRPC applications on a multi-cluster Grid environment. The overall architecture of the GEE is presented in Figure 3.1 . There are three types of components in the GEE:

- Information Service.
- GridRPC Client.

Figure 3.1: GridRPC Architecture

- Grid Resources.

A *GridRPC client* is an instance of a *GridRPC application*. It runs on a *client computer* and generates *GridRPC requests*. These GridRPC requests are dispatched to the *Grid resources* where the *GridRPC services* will serve these requests. In order to allow the GridRPC client to locate and select suitable Grid resources, the *information service* is employed. The information service retrieves information of Grid resources and provides the information to the GridRPC clients.

The rest of the chapter is organized as follows. Section 2 presents the architecture of a GridRPC client. In section 3, we present properties of a multi-cluster Grid which has multiple clusters of computers. Section 4 presents the architecture and implementation of the information service in GEE. Section 5 presents the implementation of the GridRPC system. Finally, we summarize the design of the GridRPC system in Section 6.

Figure 3.2: Architecture of a GridRPC Client

3.2 Grid Client

3.2.1 Overview

In order to hide the complication of the Grid environment, the GridRPC client has two components working concurrently as follows:

- *GridRPC Client Program* generates GridRPC requests, sends these requests to the lower layer and receives the results from this layer. The GridRPC client program is developed by the users for specific applications.
- *GridRPC Client Run-time System* receives GridRPC requests from the GridRPC client program, schedules the requests to suitable Grid resources, waits for the re-

sults, and passes the results to the GridRPC client program once they are returned.

Since dispatching jobs to Grid resources involves software overhead as well as network overhead and latency, hiding them to improve performance is necessary. In this work, we develop a GridRPC client as a group of functional components using multi-threading techniques to reduce software and network overhead and latency. A GridRPC client has four running threads (see Figure 3.2): *GridRPC client program thread*, *Metascheduler/Dispatcher thread*, *Service Monitor*, and *Resource Monitor* are working concurrently during run-time of the GridRPC application. Details of these threads and different mechanisms of job dispatching will be presented in Section 5.3.

3.2.2 Classification of Grid Resources

From the point of view of a Grid client, the Grid resources are classified into three categories:

- *Unfiltered resources*: are the Grid resources listed in the information service. Whether they can be used or not has not been known yet.
- *Enabled resources*: are the resources that allow the GridRPC client to dispatch its GridRPC requests and these requests will be processed on the resources after their arrival. A resource is *disabled* to a GridRPC client if the GridRPC client does not have permission on that resource or the resource does not have GridRPC services required by the GridRPC client. In Figure 3.2, the Enabled Resource List contains the list of enabled resources.
- *Available resources*: are the enabled resources whose workloads allow them to serve the GridRPC requests at any time. Some enabled resources may be unavailable because of their high workload. Note that the metascheduler only selects available resources for GridRPC requests.

In order to support scheduling, the *Resource Monitor* thread periodically retrieves the information of Grid resources from the information service and updates the *Enabled Resource List* which contains a list of enabled resources. During scheduling, the GridRPC client finds out available resources and any GridRPC request that must be submitted to an available resource.

3.2.3 States of GridRPC Requests

Each GridRPC request generated by the GridRPC Client Program is stored in the GridRPC Request List. Following are possible states of a GridRPC request:

1. *Created*: when the request is created by the GridRPC client program, it is appended to the Request List with this state (see Figure 3.2).
2. *Scheduled*: a request is scheduled if the metascheduler has selected a resource for it. Once it is scheduled, it can be dispatched to the selected Grid resource for service.
3. *Dispatched*: a request is dispatched if it has arrived at the Grid resource and it is being served by an instance of service.
4. *Finished*: the request has been completed at the Grid resource and its results are ready in the GridRPC request list.
5. *Erroneous*: if there is any error during the lifetime of a request, its state changes to erroneous.

The state transition diagram of a GridRPC request is given in Figure 3.3.

3.2.4 Resource Monitor

The resource monitor maintains a list of enabled resources for the metascheduler. It first retrieves a list of unfiltered resources in the information service. Subsequently, it

Figure 3.3: GridRPC Request States

eliminates disabled resources by checking the permission using the Globus GRAM Client API (“*globus_gram_client_ping*”). Then, the resource monitor obtains static and dynamic information of the enabled resources. The dynamic information of the enabled resources is updated periodically.

To improve the performance, the resource monitor is implemented as a different thread that runs concurrently to the GridRPC client program. This allows the GridRPC client program to carry out some required computation of the application at the same time having up-to-date information of the resources.

3.2.5 Metascheduler

The metascheduler carries out resource selection. It uses information of enabled resources and GridRPC requests from the GridRPC client program. When the GridRPC client program executes, it puts the requests to the GridRPC request list. The metascheduler picks the requests from this list one by one and makes selection of resources for the requests. The resource selection uses the system configuration of the Grid resources and their workload. Following are the stages of resource selection carried out by the metascheduler.

- Firstly, the resource monitor obtains a list of *unfiltered resources* from the information service. Then it checks the GRAM located on each unfiltered resource. Those resources which allow job submission from the client side will be put into the *enabled resource list*.

- Secondly, the metascheduler quantifies and qualifies Grid resources before making selection. Each Grid resource has a number of available slots that can fill the requests and cost of execution of the slots. Based on these two types of information, the metascheduler determines *available resources* and decides which resources are to be selected and how many requests are to be dispatched to each resource. The details of quantification and qualification of the Grid resources are presented in Chapter 5.
- Finally, the scheduled requests are passed to the dispatcher. These requests will be dispatched to the Grid resources.

To exploit the parallelism offered by the composite service, the metascheduler can perform request bundling that groups similar requests into bundles to form *composite* requests and submit them to the Grid resources. Two requests are considered to be *similar* if they invoke the same GridRPC service and can be processed on the same Grid resource. Each composite request contains multiple *primitive* requests. The details of metascheduling and request bundling mechanisms will be presented in Chapter 5.

3.2.6 Request Dispatcher

The dispatcher sends requests together with their arguments to the Grid resources using the Globus Toolkit API. If the GridRPC request is synchronous, the request dispatcher keeps the GridRPC client program waiting for the completion of the GridRPC request before doing further operations. Otherwise, if the request is asynchronous, it can return immediately after passing its arguments to the GridRPC client run-time system. The results can be collected later by a separate operation.

Figure 3.4: Architecture of a Grid Resource

3.2.7 Service Monitor

The service monitor is developed to check dispatched requests and collect the results of finished requests. When there is a GridRPC request dispatched, the service monitor will check the status of the request on the GridRPC resource. The checking is carried out periodically. Once the request is completed at the resource, the service monitor will collect the results and make them ready in the GridRPC request list.

3.3 Grid Resources

3.3.1 Overview

Figure 3.4 shows the architecture of a resource on a multi-cluster Grid. Each resource is a cluster of computers that contains a *master node* (or head node) and one or more *compute nodes*. The master node works as a single entry of the cluster for incoming GridRPC requests. It carries out monitoring and management tasks. Therefore, it runs some daemons such as *Grid Index Information Service* (GIIS), *Grid Resource Information Service* (GRIS) [FFK⁺97], and *workload collector*. Each compute node inside a cluster runs a *workload sensor* and a *compute server* for monitoring system state and executing jobs respectively.

Once the master node receives a GridRPC request from a GridRPC client, it creates an instance of a GridRPC service. This instance further submits a job onto a compute node. Upon receipt of a job, the compute server spawns a *job process* to handle the job. The instance of service must wait for the completion of all of its jobs before returning the results to the Grid client.

3.3.2 Mechanism of GridRPC Services

Figure 3.5: Mechanism of GridRPC Service

Figure 3.5 briefly describes the mechanism of a GridRPC service. Upon receipt of a GridRPC request, the GRAM located on the master node initializes an executable as an instance of the service requested (using “*fork*” system call). This instance creates a number of jobs and dispatches them onto available compute nodes. The number of jobs processed by the instance of service depends on whether the service is *primitive* (one job per instance) or *composite* (multiple jobs per instance). We propose a *dynamic bundling mechanism* to enable the GridRPC client to bundle requests dynamically into composite

requests. The composite requests are processed by *composite services* on Grid resources. Each instance of composite services can distribute jobs across different compute nodes of the cluster in order to let them be processed in parallel. The details of composite services will be presented in Chapter 4.

Scheduling of jobs inside the cluster is carried out by the LRM existing in each cluster. When an instance of service has a ready job, it sends a schedule request to the LRM. The LRM replies with a list of servers sorted by increasing cost of execution. Subsequently, selecting the first compute server in this list and dispatching the job to a selected server are carried out by the instance of service. If the first compute server refuses to receive the job, the instance of service sends the job to the second one in the list and so on until the job is admitted. If there is no compute server available, the ready job is kept waiting before the next scheduling is carried out. In this work, we employed and modified the NetSolve agent as the LRM. Details of the scheduling mechanism inside a cluster will also be presented in Chapter 4.

3.4 Information Service

3.4.1 Overview

The information service plays a critical role in the Grid environment since it enables the GridRPC system to carry out resource discovery and selection. Figure 3.6 presents the architecture of our information service in a multi-cluster Grid. To make the information service scalable, we employed Globus MDS with GIIS and GRIS. Further, in order to provide detailed information of clusters of computers, Ganglia [MCC03] and GLUE schema [ARE] were deployed. Workload sensors (i.e., Ganglia daemons) located at different compute nodes obtain system information and update the workload collector (i.e., Ganglia master daemon) located on the master node of the cluster. Subsequently, the

Figure 3.6: Architecture of Information Service

Globus GRIS retrieves Ganglia's data from the master daemon and organizes the data based on GLUE schema.

3.4.2 Sample of Information Service

The information service deployed on the Nanyang Campus Grid is presented in Figure 3.6. The centralized GIIS ("*ntuch*") contains information of its child GIIS (i.e., "*ntuds*") and GRISs (i.e., itself and "*hpc-pdpm*"). "*hpc-pdpm*" is a cluster of computers. Its workload is monitored by Ganglia and this information is obtained by its GRIS.

Browser root				
<div> <div>ec-pdcm</div> <div> <div>Mds-Vo-name=local</div> <div> <div>Mds-Vo-name=ec-pdcm</div> <div> <div>Mds-Host-hn=ec-pdcm</div> <div> <div>cl=datatag-CNAF</div> <div> <div>scl=sub1</div> <div> <div>host=ec-pdcm</div> <div> <div>host=compute-0-7</div> <div> <div>host=compute-0-6</div> <div> <div>host=compute-0-5</div> <div> <div>host=compute-0-4</div> <div> <div>host=compute-0-3</div> <div> <div>host=compute-0-2</div> <div> <div>host=compute-0-1</div> <div> <div>host=compute-0-0</div> </div> </div> </div> </div> </div> </div> </div> </div> </div> </div> </div> </div></div></div></div>				
Name	Value	Type	Size	
objectClass	GlueHost	text attribute	8	
objectClass	GlueHostProcessorLoad	text attribute	21	
objectClass	GlueHostSMPLoad	text attribute	15	
objectClass	GlueHostArchitecture	text attribute	20	
objectClass	GlueHostProcessor	text attribute	17	
objectClass	GlueHostNetworkAdapter	text attribute	22	
objectClass	GlueHostMainMemory	text attribute	18	
GlueHostName	compute-0-0	text attribute	11	
GlueHostUniqueID	HP2-PDCC-SCE-PDCC-compute-0-0	text attribute	30	
GlueHostProcessorLoadLast1Min	0.02	text attribute	4	
GlueHostProcessorLoadLast5Min	0.02	text attribute	4	
GlueHostProcessorLoadLast15Min	0.00	text attribute	4	
GlueHostSMPLoadLast1Min	0.02	text attribute	4	
GlueHostSMPLoadLast5Min	0.02	text attribute	4	

Figure 3.7: GRIS's Information

On “*ntuds*”, the GIIS contains information of its children GRISs (i.e., “*ec-pdccm*” and “*csge0*”) as well as its GRIS with Ganglia. Here, “*ntuds*” has three Grid resources, i.e., “*ntugp*”, “*ec-pdccm*”, and “*csge0*”. “*ntugp*” is a single computer while “*ec-pdccm*” and “*csge0*” are clusters of computers. All of them use Ganglia to monitor workload.

In Figure 3.7, the detailed information of “*ec-pdccm*” resource is displayed. “*ec-pdccm*” is a cluster of 9 computers of which 8 compute nodes are dedicated for computing (*compute-0-0* to *compute-0-7*). Detailed information of each compute node such as number of processors, memory size, and workload are updated by Ganglia. Here, “*ec-pdccm*” computer is the master node of the cluster. It runs the Ganglia master daemon and Globus GRIS to supply information to the GIIS located at “*ntuds*”. The information is further propagated to the centralized GIIS at “*ntuch*”.

3.4.3 Retrieving Information

To retrieve information of Grid resources, the GridRPC client uses Lightweight Directory Access Protocol (LDAP) supported by Globus MDS. There are two methods of retrieving information from the information service.

- Query the centralized GIIS directly.
- Locate Grid resources from the centralized GIIS first. Then query GRIS from the Grid resources.

In order to obtain up-to-date information, a GridRPC client does not query all information directly from the centralized GIIS (the first method). Instead, the GridRPC client locates Grid resources from the GIIS first. Then, it queries GRIS from the Grid resources directly (the second method). This method avoids the bottleneck on GIIS if there are many applications querying the information at the same time. In addition,

Components	Implementation
GridRPC Client	C program using Globus API (GT2.2, GT2.4, GT3.2 pre-OGSA).
GridRPC Resource	NPACI ROCKS [Teac] cluster with NFS, NetSolve [ACD02], Network Weather Service [Wol98], and Globus Toolkit 2 [FK98a].
Information Service	Ganglia and GRIS/GIIS on each cluster.

Table 3.1: Implementation Description

accessing GRIS directly can get more up-to-date information than accessing GIIS whose information may be older because it is propagated through many levels.

3.5 Implementation

Table 3.1 summarizes the implementation of the multi-cluster GridRPC system. We employed and modified NetSolve as the local resource manager. It selects the compute node to spawn job processes. The GridRPC client was developed in C. It uses Globus API to access Grid resources. Globus's Global Access to Secondary Storage (GASS) [Teaa] and GridFTP [Teab] are employed to transfer data between the GridRPC client and Grid resources.

In this work, we focused on Grid resources that are clusters of computers. Each cluster is installed with the NPACI ROCKS software. It helps to reduce administration cost and provides a single system image on the cluster. Each compute node of the cluster is configured as a NetSolve server while the master node of the cluster is configured as a NetSolve agent. All NetSolve servers of a cluster register them to the NetSolve agent. In addition, we used the Network File System (NFS) so that data can be accessed similarly by different NetSolve servers. The instances of GridRPC service running on the master

node of the cluster also access the same NFS.

3.6 Summary

This chapter presents an overview of the GridRPC environment. Especially, we focus on the architecture of the execution environment.

The goals of the design presented above are to enable resource discovery and achieve high performance through metascheduling. To support these goals, the GridRPC client run-time system is equipped with many components that can work concurrently. First, the resource monitor helps make resource discovery and selection by supplying a list of enabled resources to the metascheduler. Second, the metascheduler and the dispatcher can work concurrently to the GridRPC client program in order to hide the overhead of communication between the GridRPC client program and the Grid environment. Developed using a multithreading technique, the GridRPC client run-time system can maximize the advantages of concurrency.

Chapter 4

Composite Service

4.1 Introduction

In many applications, especially parameter sweep applications, a large number of identical jobs with different parameter sets are produced. Submitting these jobs to Grid resources individually may prove to be very inefficient due to high overhead and network latency of the Grid environment. In particular, since Grid resources are geographically distributed on the Internet, performance of applications may decrease drastically due to the high communication latency involved. In order to avoid the above disadvantages, we developed a mechanism called *request bundling* that groups individual requests into composite requests at the GridRPC client side and the corresponding *composite service* on a Grid resource to process these composite requests.

We distinguish between *primitive requests* and *composite requests*. A primitive request is a request that has one job being executed on a compute node of the Grid resource. In contrast, a composite request is a bundle of primitive requests. Hence, a composite request can have several jobs running in parallel on different compute nodes of the Grid resource. Here, a composite service is a GridRPC service that receives a composite request and executes jobs in parallel on the compute nodes of the Grid resource.

In this chapter, we focus on performance issues of the composite services and Grid resources. We investigate two factors, namely, (1) degree of concurrency (denoted by D), i.e., the maximum number of jobs allowed to process at a time on a processor of the Grid resource, and (2) local resource scheduling algorithm on Grid resources. The study helps determine how much concurrency a Grid resource would support to offer the highest performance to the applications. Based on the D value, configuration, and workload of a Grid resource, a GridRPC client can decide how many requests should be dispatched to the Grid resource.

The remaining sections of this chapter are organized as follows. Section 2 presents the mechanism of the composite service. In section 3, we discuss how D affects the performance of a Grid resource. Performance improvement for composite services is presented in section 4. Finally, section 5 presents our summary.

4.2 Mechanism of Composite Service

Figure 4.1: Mechanism of Composite Service

Motivated by packet bundling techniques [Law99] that were used to reduce communication overhead in distributed interactive simulation, we have developed a dynamic

bundling mechanism to reduce communication overhead between GridRPC client and Grid resources.

Packet bundling techniques have been applied to improve communication in a distributed environment. For example, instead of sending one acknowledgement for each TCP packet, the receiver just sends an acknowledgement for a group of packets. This reduces the number of acknowledgement packets and overhead of sending and receiving acknowledgements. In this work, the packet bundling problem was presented as an on-line problem for dynamic TCP acknowledgement [DGS98, FLNU03].

Packet bundling has also been applied in network routers, bridges, or gateways [BCW96, Ber02] to increase the number of packets processed if these devices have a limited bound of packets. In general, bundling techniques are helpful for reducing communication overhead and packet transmission rate in Ethernet but may not be rewarding for ATM networks [BCW96].

An investigation of packet bundling mechanism for reducing the communication cost has been carried out on a hypercube network [BM88]. In this work, messages sent along the same channel are bundled. Once a node receives a bundle of messages, it takes the messages sent to it and forwards the rest to the other nodes. The experimental results show that bundling mechanism can have a significant improvement for a particular communication model but lesser magnitude than that in the full communication model where every node sends a message to every other node.

In Distributed Interactive Simulation (DIS), packet bundling is employed to reduce bandwidth consumption [BCL⁺97, Law99]. In [Law99], packet bundling using multiple priority queues and a single priority queue was studied. The simulation results show that packet bundling makes sense when the arrival rate of packets is not very small and it can reduce the packet transmission rate and bandwidth requirement at the gateway in a DIS environment.

In this work, we examined packet bundling for reducing the overhead of job submission and also investigated how workloads on Grid resources are affected when using bundling techniques. Two mechanisms were introduced: (1) request bundling at the GridRPC client side and (2) composite service at the resource side (see Figure 4.1).

At the GridRPC client side, the request bundling mechanism groups similar jobs into a bundle called a *composite request* and dispatches the request to a Grid resource. This bundling mechanism is applied for asynchronous GridRPC calls that are available in many parameter sweep applications. The details of the bundling mechanisms will be presented in Chapter 6.

On the Grid resources, *composite services* are provided to process composite requests. A composite service is a program that runs on the master node of a cluster and distributes jobs across compute nodes inside the cluster. Upon receipt of a composite request, the GRAM located at the Grid resource creates an instance of the composite service. This instance receives the arguments of the composite request, determines the number of jobs required, and dispatches these jobs across different compute nodes. Once all jobs are finished, the results from different compute nodes are collected and sent back to the GridRPC client.

We employed NetSolve [ACD02] to implement composite services. A composite service is a NetSolve client program running on the master node of the cluster while computation functions are deployed on compute nodes as NetSolve problems [ACD02]. An instance of the composite service uses the NetSolve API to create a number of jobs and distributes these jobs on different compute nodes in the cluster. Scheduling of these jobs on the compute nodes is carried out by the NetSolve agent which runs on the master node of the cluster. Figure 4.2 presents the scheduling steps for a job by the NetSolve agent. When a job is created, its information is submitted to the NetSolve agent. Subsequently, the NetSolve agent creates a list of compute nodes that are able to execute the job. This

Figure 4.2: Scheduling Steps of a Composite Service

list is sorted by increasing execution cost which will be discussed later. The NetSolve agent sends this list back to the instance of the composite service as a hint of scheduling. Subsequently, the instance selects the first compute node in the list and dispatches the jobs to the selected node. If the first compute node in the list refuses to receive the job, the composite service will send the job to the second one in the list. This process is repeated until there is a compute node accepting the job. If all nodes are unavailable, the service must wait for an available node. When all jobs are finished, their results are returned to the instance of the composite service. Finally, the instance returns the results to the GridRPC client.

4.3 Performance of Composite Services

The total execution time of a GridRPC application depends on many factors as follows.

- Communication overhead between GridRPC client and Grid resources
- Waiting time of primitive requests at the NetSolve client side
- Response time of the composite service

In this part, we focus on the response time of a composite service located on a cluster of computers. The response time of a composite service is defined as the interval between the time at which the request arrives on the Grid resource and the time at which the service is accomplished. When a cluster receives a composite request, it executes a number of jobs specified in the request on its compute nodes. Thus, the response time of the composite service located on the cluster depends on the waiting time of its jobs and the execution time of these jobs on the compute nodes of the cluster.

Let us consider two possible scenarios of a composite request. In the first scenario, the number of jobs to be executed is less than the number of available compute nodes. For each job, the agent will return a list of compute nodes, in which there is at least one node available to execute the job. In this case, the response time of the composite service depends mainly on the execution time of its jobs.

In the second scenario, the number of jobs waiting for execution is greater than the number of compute nodes available in the cluster. The composite service keeps the jobs waiting until there are available nodes. As a result, the response time of the service contains not only the execution time of jobs but also the waiting time of jobs.

We investigate the degree of concurrency (denoted by D), i.e., the maximum number of jobs allowed to process at a time on a processor of a compute node. This factor

influences execution of jobs on compute nodes and their waiting time on the Grid resource. Moreover, the degree of concurrency also influences the number of requests to be bundled into a composite request (see Chapter 5).

Figure 4.3: Execution under Different Degrees of Concurrency

Figure 4.3 presents the relationship between the degree of concurrency and the execution of jobs on a compute node. If the degree of concurrency equals 1, jobs are executed sequentially. Hence, later jobs spend more time waiting for an available processor. In contrast, if the degree of concurrency is increased, e.g., 4, more jobs can be executed concurrently. Hence, their waiting time can be reduced. However, execution times of jobs may be longer since several jobs share processor time.

In order to set the degree of concurrency for the cluster, we change the *RESTRICTION* parameter in NetSolve's *server_config* file. This parameter restricts the maximum number of jobs received by a compute node. Three degrees of concurrency will be investigated.

- Low degree: the system allows no more than one job occupying a processor at a time.
- Medium degree: one processor can process up to 4 jobs at a time.
- High degree: one processor can process up to 10 jobs at a time.

Table 4.1 presents the configuration of the two clusters used for the experiments. Their performance under different degrees of concurrency is presented in Figure 4.4.

	Cluster 1	Cluster 2
Number of Computing Nodes	8	8
Number of Processors per Compute Node	1	2
Processing Speed (MFLOPS)	11.872	85.790
Average Background Workload of Compute Nodes (load_one)	0.625	0.596

Table 4.1: Configuration of Two Clusters

Here, we measured response times of the composite service with different number of jobs, namely, 20, 40, 60, 80, and 100. Each job processes 300 Mega Floating Point Operations (MFLOs). In most cases, a small degree of concurrency, i.e., $D=1$, offers the highest response time of the composite service while the medium one, i.e., $D=4$, offers the lowest response time.

As mentioned above, a small degree of concurrency lengthens the waiting time of jobs when the number of resources is limited. Under medium and large degrees of concurrency, a compute node can easily accept jobs to execute and thus, reduces the waiting time of requests. However, if there are too many jobs executed concurrently on a compute node, the compute node is easily overloaded and thus its performance is decreased. This phenomenon is indicated when the degree of concurrency is high, i.e., $D=10$.

4.4 Performance Improvement

In this part, we present our improvement for NetSolve when it is employed as a Local Resource Manager. In NetSolve, the degree of concurrency is specified via the *RESTRICTION* variable in the *server.config* file. Each compute node in the cluster guarantees that the number of running jobs at a time must not be greater than the degree of concurrency. When a compute node has the number of running jobs equal to the degree of concurrency, it refuses any arrival job and sends a negative reply to the NetSolve client. In this case, the NetSolve client can send the job to another compute

Figure 4.4: Response Time of Composite Service under Different Degrees of Concurrency

node or wait until there is a compute node accepting the job.

In order to avoid redundant communication between the NetSolve client and a compute node, we monitor the number of running jobs on each compute node in the NetSolve agent. When the NetSolve agent schedules a job, instead of sending a sorted list of compute nodes to the NetSolve client, the NetSolve agent filters out compute nodes whose total workload is greater than the degree of concurrency. This guarantees that no overloading happens on any compute node since the scheduling considers both background workload and the number of jobs processed by the Netsolve server on each compute node. Further, this also makes sure that the compute node sent to the NetSolve client will not refuse the job.

Given:	D is Degree of Concurrency supported by the cluster; N is the number of compute nodes in the cluster; $C = \{C_1, C_2, \dots, C_N\}$ is list of compute nodes in the cluster, each compute node C_i has: Workload of Compute Node, i.e., $load_one(C_i)$, and Number of currently running jobs: $NumRunningJobs(C_i)$;
Algorithm:	<pre> 1. Initialize $K = 0$, K is the number of available compute nodes in the cluster; 2. Initialize the list of available compute nodes in the cluster, A; 3. Receive a <i>ScheduleRequest</i> from NetSolve client (composite service); 4. FOR $i = 1$ TO N DO 5. IF $CurrentTime - LastWLChangedTime(C_i) < CTIME$ THEN 6. $TotalWorkload(C_i) = load_one(C_i) + NumRunningJobs(C_i)$; 7. ELSE 8. $TotalWorkload(C_i) = load_one(C_i)$; 9. ENDIF 10. IF $TotalWorkload(C_i) > D$ THEN 11. $Availability(C_i) = false$; 12. ELSE 11. $Availability(C_i) = true$; 13. Add C_i to A; 14. $K = K + 1$; 15. ENDIF 16. ENDFOR 17. IF $K \leq 0$ THEN 18. Send <i>NegativeReply</i> to the Master Node; 19. ELSE 20. Sort A by increasing $TotalWorkload$ ($A = \{A_1, A_2, \dots, A_K\}$); 21. Send A_1 to the Client; 22. $NumRunningJobs(A_1) = NumRunningJobs(A_1) + 1$; 23. $LastWLChangedTime(A_1) = CurrentTime$; 24. ENDIF </pre>

Figure 4.5: New Scheduling Algorithm of NetSolve Agent

Figure 4.5 presents the algorithm carried out by the NetSolve agent when it receives a request from a NetSolve client. Note that the $NumRunningJobs$ is decreased by one when there is a job completed but this is not included in the above algorithm. This is carried out when the NetSolve server informs the NetSolve agent with job accomplishment. First, this scheduling algorithm uses the workload captured directly by the NetSolve daemon deployed on each compute node. Second, when a job is completed on a compute

node, upon receipt of a job accomplishment notification, the NetSolve agent updates the number of running jobs on the compute node (*NumRunningJobs*) as well as the time when the workload is changed (*LastWLChangedTime*).

Here, the total workload of a compute node (*TotalWorkload*) can be the workload captured from the compute node, i.e., *load_one*, (see line 8) or *load_one* together with the number of running jobs (see line 6). Note that the *load_one* value indicates the average number of active processes running on the system in the last minute. It is provided by the UNIX operating system.

The first scenario considers the case where the *load_one* does not reflect workload changes. The algorithm uses a parameter *CTIME*. In our experiments, it was set to 120 seconds, so that it is long enough for a job to contribute its workload to the *load_one* value when it arrives or for a job to remove its workload from *load_one* when it is completed.

If the difference between the *CurrentTime* and the *LastWLChangedTime* is less than *CTIME*, the NetSolve agent knows that *load_one* does not include the workload caused by the new arriving jobs. Hence, the total workload of the compute node should include both *load_one* and the workload caused by the new arriving jobs. In this case, we estimate the total workload of a compute node as the sum of *load_one* and the workload caused by running jobs (line 6). This estimation is heuristic and it may not reflect the actual workload of the compute node because it may double count jobs in this case.

If the difference between the *CurrentTime* and the *LastWLChangedTime* is greater than *CTIME*, *load_one* should reflect the workload caused by the number of running jobs and the background workload. Hence, *load_one* is used directly as the total workload of the compute node.

After calculating the total workload of a compute node, the NetSolve agent assigns availability status to the compute node by comparing its total workload and the degree

of concurrency. This makes sure that the sum of the number of running jobs and the background workload of any compute node in the cluster is not over D . Moreover, by filtering unavailable nodes, the NetSolve agent ensures the NetSolve client can successfully send the job to the best compute node in the list and thus it eliminates any redundant communication between the NetSolve client and compute node.

Figure 4.6: Performance of New Local Scheduling Algorithm

Since the new algorithm reduces communication between the NetSolve client, i.e., the composite service, and the NetSolve agent, and at the same time avoids overloading compute nodes, its performance should be improved. In order to compare the performance of the new scheduling algorithm and the original one for the NetSolve agent, we repeated experiments on the clusters with the configuration shown in Table 4.1. The composite

services have 20, 40, 60, 80, or 100 jobs, respectively. Each job processes 300 MFLOs. Figure 4.6 presents the response time of the composite service with the new scheduling algorithm (denoted by $New(D = 4)$) and the original one (denoted by $Orig(D = 4)$) which used Network Weather Service (NWS) [Wol98]. The degree of concurrency we used here is 4. The results show that the new algorithm achieves better performance for the composite service due to its capabilities of workload balancing and overloading avoidance across different compute nodes of the cluster.

4.5 Summary and Discussions

In this chapter, we presented the mechanism of the composite service to facilitate the achievement of parallelism on clusters of computers. The composite service serves GridRPC requests by creating jobs and distributing these jobs across compute nodes inside the cluster.

By increasing the D value and employing the new local scheduling mechanism, the response time of the composite service can be improved. The medium D offers better performance than the low and high D . Under low D , the instance of the composite service keeps jobs waiting for available compute nodes since less concurrency is allowed. Under high D , overloading on compute nodes may happen and thus slows down performance of the cluster. With medium D , the LRM, the NetSolve agent, can balance these above trade-offs and offer the best response time to composite services.

We also presented the new algorithm to schedule jobs in a cluster-based NetSolve system. By considering the workload and at the same time monitoring the number of running jobs on each compute node, we can reduce redundant communication and avoid overloading on compute nodes of the cluster. This helps the composite service to achieve better performance.

Chapter 5

Metascheduling and Bundling Mechanisms for GridRPC Applications

5.1 Introduction

This chapter presents two mechanisms performed by the GridRPC client, namely, (1) metascheduling and (2) request bundling mechanisms.

The metascheduling mechanism is critical since there are many geographically distributed resources that may have different capabilities on a Grid environment. It characterizes Grid resources and selects the most suitable resource for a GridRPC request. Especially, the metascheduling mechanism takes into account the dynamic workload of Grid resources. This ability helps GridRPC applications to achieve high performance on shared resources.

In order to hide communication overhead and at the same time exploit composite services at Grid resources, the client can use request bundling mechanisms. It groups similar requests into a composite request and dispatches this request to a Grid resource where a composite service can process multiple jobs in parallel. In this chapter, we present three implementations: unbundled, static bundling, and dynamic bundling mechanisms.

The rest of this chapter is organized as follows. Section 2 presents the metascheduling mechanism. In this section, how Grid resources are characterized and selected is presented. Section 3 describes request bundling mechanisms. We also discuss how metascheduling and dynamic bundling mechanisms can work together at the Grid client in this section. Finally, section 4 gives the summary and discussions.

5.2 Metascheduling Mechanism

Metascheduling plays a critical role in achieving high performance for Grid applications. It enables a Grid client to submit jobs to the suitable resources so that high performance can be achieved. There are several major tasks involved in a metascheduling process as follows:

1. *Resource Characterization*: In this stage, the metascheduler determines both quantitative and qualitative characteristics of Grid resources. Based on these characteristics, the metascheduler can determine the availability of each resource, capacity of each resource and its cost of execution.
2. *Resource Selection*: Once available resources are determined, the metascheduler selects the most suitable resources for the GridRPC requests based on the cost of execution. In this work, two resource selection heuristics are investigated and compared with a random selection method.
3. *Request Dispatching*: After being scheduled, GridRPC requests are dispatched to the selected Grid resources.
4. *Request Monitoring*: After dispatching GridRPC requests to the Grid resource, the metascheduler keeps track of the execution of these requests. It also takes into

account the real workload of Grid resources to achieve accuracy of scheduling for new requests.

5. *Request Post-processing*: When a GridRPC request is completed and its results are returned to the metascheduler, the metascheduler updates the GridRPC client with these results. The metascheduler also updates itself with the status of Grid resources after the completion of a request.

5.2.1 Resource Characterization

A Grid resource is quantified by *the number of available slots*, i.e., the number of jobs that the resource can execute immediately. If the number of slots available is greater than zero, the Grid resource is *available*. Otherwise, it is *unavailable*.

Given a Grid resource G , the *number of available slots* on G denoted by A is estimated as follows:

$$A = P \times D - L$$

where P is the total number of processors in the Grid resource, D is the degree of concurrency supported, and L is the total workload of G .

The *availability* of Grid resources is determined based on the number of available slots. Let q be the number of jobs required to execute on a resource, the *availability* of G for q jobs, denoted by $AV(q)$, is estimated as follows:

$$AV(q) = \begin{cases} TRUE, & \text{if } A \geq q \\ FALSE, & \text{otherwise} \end{cases}$$

The quality of a Grid resource is characterized by *the cost of execution* which is directly proportional to the processing time of jobs on the resource. Following are some factors that affect the processing time of jobs on a Grid resource: *average processing speed*, *total number of processors*, and *workload of the compute node*.

Processing speed is one of the significant factors to qualify the Grid resource. In this work, the processing speed is obtained from benchmarking the GridRPC service on the compute nodes inside the Grid resource. Although the method is time consuming for the first time, it is more practical than simply using the CPU speed. It is because the performance of compute nodes may vary depending on the platforms although they have the same CPU speed. The benchmarking just needs to be performed once. The results can be used directly in subsequent runs of the metascheduler. The processing speed can be estimated using an average if the compute nodes of the cluster are heterogeneous.

Let S be the processing speed of the Grid resource G , the cost of execution of G , denoted by $ExecCost$, is estimated as follows:

$$ExecCost = \frac{1}{S} \quad (5.2.1)$$

Typically, the workload of compute nodes is used to determine the capacity of a Grid resource and the processing speed is used to estimate the cost of execution using the available slots in the resource. However, the cost of execution is also affected by the overhead for creating GridRPC requests at the GridRPC client, dispatching the requests to Grid resources, and collecting the results from Grid resources. These will be affected by the number of jobs being concurrently executed. Especially, when a Grid resource is going to be overloaded, its cost of execution must be increased. Therefore, taking load information into account may generate a more accurate estimation of the cost of execution.

Based on the above arguments, we define the *occupation ratio* of a Grid resource as the ratio of the number of currently running jobs over the maximum number of jobs allowed to be executed on that Grid resource. Given that L is the total workload of the Grid resource G , P is the total number of processors, and D is the degree of concurrency

supported, the occupation ratio denoted by O is estimated as follows:

$$O = \frac{1 + L}{P \times D}$$

Note that 1 is added to L to avoid making O too small if the workload is very small.

On a resource G with processing speed S , the cost of execution $ExecCost$ can then be defined as:

$$ExecCost = \frac{1}{S} \times O = \frac{1}{S} \times \frac{1 + L}{P \times D} \quad (5.2.2)$$

The average processing speed and total number of processors are static information that can be easily retrieved and reused from the information service. In contrast, the workload information is dynamic and it is necessary to estimate accurately. In our work, we consider the workload of a compute node to include not only the workload obtained from the information service but also the workload caused by recently allocated jobs. The UNIX one-minute load average, i.e., *load_one* is a useful source for characterizing the workload of a compute node. Almost all UNIX systems maintain this source of workload information. In practice, many workload monitoring systems such as Ganglia [MCC03] and Network Weather Service [Wol98] use this information to characterize the workload. However, since it is calculated on the average in a one minute time interval and propagates through the workload monitor (Ganglia) and the information service (GRIS) to the GridRPC client, the information may not be up to date. Thus, the workload caused by recently allocated jobs may not be captured by the information service yet. An approach similar to the one described in section 4.4.2 can be used to improve the accuracy of the workload information.

5.2.2 Resource Selection Heuristics

After the number of available slots of each Grid resource is determined, resource selection can be carried out for individual requests picked from the Request List. Following are

the heuristics used to make resource selection.

- *Mincost*: Given M available Grid resources, the metascheduler sorts the list of resources in increasing order of the cost of execution. The cost of execution is estimated based on the processing speed (i.e., equation 5.2.1). The first Grid resource in the list is selected for the request.
- *Mincost-O*: In this heuristic, resource selection is carried out similarly to the Mincost heuristic. However, the cost of execution on the Grid resource is estimated using processing speed and the occupation ratio (i.e., equation 5.2.2).

5.3 Request Bundling Mechanisms

In this section, we describe four mechanisms that can be employed to bundle and dispatch GridRPC requests to Grid resources, namely, (1) unbundled mechanism with immediate dispatching, (2) unbundled mechanism with background dispatching, (3) static bundling, and (4) dynamic bundling mechanism.

5.3.1 Unbundled Mechanism - Immediate Dispatching

This mechanism is denoted by *UID*. In this mechanism, after a request is created, the metascheduler makes resource selection for the request and dispatches it immediately to the selected Grid resource if a resource is available. If there is no Grid resource available, the request is kept waiting.

The architecture supporting this mechanism is presented in Figure 5.1. It is implemented by three concurrent threads of execution in the GridRPC client process.

- The first thread, *GridRPC client program thread*, executes the GridRPC main program. It creates a GridRPC request by making a GridRPC function call. Once

Figure 5.1: Unbundled Mechanism - Immediate Dispatching

a GridRPC function call is made, the metascheduling algorithm is invoked. The metascheduling algorithm quantifies and qualifies Grid resources, selects the most suitable one for the request, and dispatches the request to the selected Grid resource. In this case, the *GridRPC client program thread* also performs metascheduling for GridRPC requests. Note that the request is stored in the central place, i.e., the Request List, when it is created. States and results of GridRPC requests are also maintained in this Request List.

- The *Service Monitor thread* works concurrently to the GridRPC application thread. It checks the Request List and monitors the services that are currently running to serve the requests registered in the Request List. When an instance of the service completes and its result is ready, the service monitor obtains and puts the result into the Request List.
- The *Resource Monitor thread* retrieves information of Grid resources from the Information Service. It updates the Resource List with the workload obtained peri-

odically. The workload of Grid resources will be supplied to the metascheduler for Grid resource quantification and qualification.

5.3.2 Unbundled Mechanism - Background Dispatching

In order to hide the latency of dispatching requests, we decoupled the request generating and metascheduling into two separate tasks that can be carried out concurrently. In the *UID* mechanism, these tasks are carried out by the *Grid client program thread*. In this *UBD* mechanism, the *Grid client program thread* generates GridRPC requests while a new thread called *Metascheduler/Dispatcher thread* carries out metascheduling and dispatching. The architecture supporting this mechanism is similar to what is presented in Figure 5.1 except that the Grid client program and the Metascheduler/Dispatcher are carried out by different threads.

When a request is created by the *Grid client program thread*, it is stored in the *Request List* with *Created* state. The *Metascheduler/Dispatcher thread* checks the Request List frequently every one second. If there are any created requests that have not been scheduled, the *Metascheduler/Dispatcher thread* schedules these requests using the metascheduling mechanism and changes the requests' state to *Scheduled*. Later, it will dispatch the requests to the selected resources and the *Service Monitor* will check requests' results at the Grid resources and return the results once they are ready. When the results have been collected at the GridRPC client, the request's state becomes *Finished*.

5.3.3 Static Bundling

The static bundling mechanism (*SB*) allows users to group several primitive requests into a composite request statically by adding extra arguments to a GridRPC function call. Hence, the mechanism supporting static bundling is similar to the *UBD* mechanism except that all GridRPC requests are composite ones.

Figure 5.2: Metascheduling and Dynamic Bundling Mechanism

The static bundling mechanism is not transparent to users and considerable effort is required for choosing an appropriate bundle size and generating the composite requests. Details of these matters will be discussed in Chapter 6.

5.3.4 Dynamic Bundling Mechanism

Figure 5.2 presents the architecture of the dynamic bundling mechanism (denoted by *DB*). The *Metascheduler/Bundler thread* carries out metascheduling and bundling for requests in the Request List. First, it performs resource selection for new GridRPC requests which have just been created by the client program. Then, it groups the requests that use the same GridRPC service on the same Grid resource into a bundle. Undispatched bundles of requests are stored in the Bundle List. The *Dispatcher thread* dispatches scheduled and bundled requests stored in the *Bundle List* to the Grid resources as *composite requests*. The *Service Monitor thread* monitors and collects results from instances of GridRPC services. Once results of an instance of a composite service return, the Service Monitor thread separates the results according to different primitive requests

that had been bundled.

Upon receipt of a composite request, the Grid Resource Allocation and Management (GRAM) [FK98a] located on the master node of the cluster creates an instance of the composite service. This instance receives the arguments of the request, determines the number of jobs required, and sends these jobs to the compute nodes. The instance of the composite service gets a hint from the LRM, i.e., the NetSolve agent, and dispatches its jobs on compute nodes within the cluster based on this hint. Once all jobs are finished, the results from different compute nodes are collected and sent back to the GridRPC client.

In order to ensure that waiting time of requests in the Request List does not exceed a threshold called *Maximum Waiting Time* (denoted by MWT), the Metascheduler/Bundler thread performs metascheduling and bundling periodically every MWT interval. In addition, the Dispatcher thread checks the Bundle List frequently to ensure that the request will not spend much time in the Bundle List after being scheduled and bundled.

Figure 5.3 presents the algorithm to schedule and bundle GridRPC requests. In this algorithm, the metascheduler maps created requests (R) onto Grid resources (G). Resource characterization is performed to determine resource availability and the cost of execution before selection is carried out. If there is no available resource, the metascheduler/bundler will schedule the request the next time it is executed. Once a request is bundled, the metascheduler re-calculates the cost of execution and the availability of Grid resources. It loops until there is no created (unscheduled) requests existing in the Request List or there is no resource available. The algorithm has complexity of $O(n(p + m \log m))$, where n is the number of requests in the Request List, m is the number of enabled Grid resources, and p is the maximum number of undispached bundles. Since the maximum number of bundles cannot be greater than the number of resources, the complexity of the

Given:	$R = \{R_1, R_2, \dots, R_n\}$ is a list of n created requests; $G = \{G_1, G_2, \dots, G_m\}$ is a list of m enabled Grid resources; $B = \{B_1, B_2, \dots, B_p\}$ is a list of p <i>undispatched</i> bundles of GridRPC requests;
Algorithm:	<pre> 1. FOR $i = 1$ TO n DO 2. $TotalAvailSlots = 0$; 3. FOR $j = 1$ TO m DO 4. Calculate $NumAvailSlots(G_j)$; 5. $TotalAvailSlots = TotalAvailSlots + NumAvailSlots(G_j)$; 6. Calculate $ExecCost(G_j)$; 7. ENDFOR 8. IF $TotalAvailSlots > 0$ THEN 9. Sort G by Increasing $ExecCost$; 10. Select the first resource G_d in G such that 11. G_d has <i>Service</i> to handle R_i and $NumAvailSlots(G_d) > 0$ 12. Change state R_i from <i>Created</i> to <i>Scheduled</i>; 13. Set <i>Destination</i> of R_i to G_d; 14. Find <i>undispatched</i> Bundle B_k in B such that 15. requests in B_k have similar <i>Service</i> and <i>Destination</i> to that of R_i; 16. IF <i>found</i> THEN 17. Put R_i into B_k; 18. ELSE 19. Create a new bundle B_{p+1}; 20. Put R_i into B_{p+1}; 21. Append B_{p+1} to B; 22. ENDIF 23. ELSE 24. Return (and wait for resource available); 25. ENDIF 26. ENDFOR </pre>

Figure 5.3: Metascheduling/Dynamic Bundling Algorithm

algorithm is simplified to $O(nm \log m)$.

The algorithm is executed by the metascheduler whenever the waiting time of an unscheduled request in the Request List is greater than MWT . The MWT can be selected intuitively by the GridRPC client based on the arriving rate of requests. If the MWT is small, the benefits of dynamic bundling may be small because there is a small number of requests in a bundle. On the other hand, a large MWT may result in a long

response time since requests will have to spend more time in the Request List.

5.4 Summary and Discussions

This chapter presents the metascheduling mechanism and various mechanisms of request bundling. In order to carry out metascheduling for GridRPC requests, the Grid resources are characterized by their quantity and quality, i.e., the *number of available slots* and the *cost of execution*, respectively.

There are many mechanisms to dispatch GridRPC requests to the selected GridRPC resources. The request can be dispatched immediately when it is created and there is an available resource (*UID* mechanism) or dispatched in the background by another thread (*UBD* mechanism). When there are a number of GridRPC requests invoking the same GridRPC service at the same Grid resource, they can be grouped and dispatched to the Grid resource as a composite request. The bundling can be carried out statically (*SB* mechanism) or dynamically (*DB* mechanism). If bundling is carried out statically, it requires programming efforts from users. Otherwise, in the *DB* mechanism, the bundling is carried out transparently to the users.

In order to make the above mechanisms work effectively, multithreading techniques are employed. The GridRPC client run-time system contains many threads of execution. They work concurrently to the main thread, i.e., the GridRPC client program thread. This hides the latency and complication of the Grid environment to the Grid-enabled applications.

Chapter 6

Performance Analysis

6.1 Introduction

The performance of GridRPC applications in different scenarios is examined in this Chapter. Following are the factors considered to compare performance offered by different mechanisms:

- *Response Time of a GridRPC service* is defined as the interval between the time at which the request arrived on the Grid resource and the time the service is accomplished. As discussed in Chapter 3, the response time of a GridRPC service depends on the degree of concurrency and the local resource scheduling mechanism.
- *Turn-around Time of a GridRPC request* is defined as the interval between the time at which a GridRPC request is created at the client and the time at which the result is returned. Turn-around time of a request includes waiting time of requests at the client, overhead of software, communication between the client and the Grid resource, and the response time of the GridRPC service.
- *Makespan of a GridRPC application* is defined as the interval between the starting time and the ending time of an application. In most of the applications, makespan is more significant than the response time of individual GridRPC requests since the

makespan includes computation time of the service on the Grid resource, communication time, and overhead of the whole application.

In this work, we logged makespan of the application, creation time (denoted as $T_{created}$), dispatching time (denoted as $T_{dispatched}$), and finishing time (denoted as $T_{finished}$) of GridRPC requests. These types of information allow us to analyze the performance of scheduling and dispatching mechanisms.

The rest of this chapter is organized as follows. Section 2 presents the performance results of various bundling mechanisms on a single Grid resource. We used a single cluster to investigate the overhead and latency of the bundling mechanisms rather than the impact of the metascheduling mechanism. In section 3, we further examine the performance of GridRPC applications on multiple Grid resources. In this case, the performance is affected by not only the bundling mechanisms but also the resource selection strategy. Section 4 presents some performance results of the metascheduler when it supports concurrent clients. The experiments were also carried out on multiple Grid resources. Finally, section 5 outlines our conclusions.

6.2 Performance of GridRPC Mechanisms on A Grid Resource

The performance of a GridRPC application depends on the metascheduling and bundling mechanisms. In this section, we focus on the effect of the bundling mechanisms on the performance of the GridRPC applications. Hence, we use a single cluster of computers to eliminate the effect of the metascheduling mechanism. We compare the efficiency of different mechanisms such as Unbundled Mechanism with Immediate Dispatching (UID), Unbundled Mechanism with Background Dispatching (UBD), Static Bundling (SB), and Dynamic Bundling (DB).

In order to compare different mechanisms on a single Grid resource, we generated a synthesized workload by using a GridRPC application benchmark described as follows:

- **Number of Requests:** The GridRPC application requires 50 independent jobs of the same type concurrently. The GridRPC client makes an asynchronous GridRPC request for each job and dispatches 50 requests to a Grid resource whose configuration is presented in Table 6.1. Since these requests are asynchronous and require the same type of jobs on the Grid resource, they can be bundled into composite requests in the dynamic bundling mechanism.
- **Job Size:** Job size is defined as the number of operations that the job requires to complete. In the following experiments, job size is set about 300 Mega Floating Point Operations. The workload of each job is synthesized by using a program that can execute a number of floating point operations as demanded.
- **Interarrival Time of GridRPC Requests:** Interarrival times of requests were generated using an exponential distribution. There are three arrival rates generated randomly for the experiments as follows:
 - **High Rate:** The average interarrival time of requests is 1 second.
 - **Medium Rate:** The average interarrival time of requests is 5 seconds.
 - **Low Rate:** The average interarrival time of requests is 20 seconds.

The interarrival times of GridRPC requests generated for various arrival rates are presented in Figure 6.1. In order to generate requests with the above arrival rates, we use a loop of iterations. Each iteration generates one asynchronous GridRPC request and a period of local computation. The time interval of the local computation is generated using the pre-sampled interval shown in Figure 6.1.

Figure 6.1: Interarrival Times of GridRPC Requests

The Grid resource used to run these experiments is a cluster of 8 compute nodes. Its configuration is presented in Table 6.1.

Number of Computing Nodes	8
Number of Processors per Compute Node	1
Processing Speed (MFLOPS)	11.872
Background Workload of the Grid Resource (load_one)	0.625
Average Latency between GridRPC Client and Grid Resource (s)	7.51

Table 6.1: Configuration of A Grid Resource

Figure 6.2: Makespan of GridRPC Applications offered by *UBD* and *UID*

6.2.1 Immediate Dispatching versus Background Dispatching

This part presents the efficiency of multithreaded techniques in hiding communication latency. As described in Chapter 5, in the immediate dispatching mechanism (*UID*), the request is dispatched immediately once it is scheduled. In contrast, the background dispatching approach (*UBD*) has another concurrent thread to handle request dispatching. Hence, in the *UBD* mechanism, the latency of dispatching can be hidden. The local computation can be carried out by the *GridRPC client program thread*, and at the same time metascheduling and dispatching of GridRPC requests are performed by the *Request Dispatcher thread*. Therefore, the *UBD* mechanism offers a smaller makespan to GridRPC applications than *UID* in all the cases.

Figure 6.2 and 6.3 present the makespan and traces of primitive requests in *UID* and

Figure 6.3: Traces of Primitive GridRPC Requests in *UBD* and *UID* Mechanisms

UBD mechanisms respectively. Due to the overhead of request dispatching, *UID* causes delay for the GridRPC application to create the next GridRPC requests. Hence, the finished time of these requests is increased. As a result, the makespan of the GridRPC application is also increased. However, there is not much difference in response time of GridRPC requests in both cases. Based on the results, we will consider *UBD* as the representative of unbundled mechanisms when comparing to static and dynamic bundling mechanisms.

6.2.2 Unbundled versus Dynamic Bundling

We compare the unbundled mechanism with the dynamic bundling mechanism. The dynamic bundling mechanism was examined with maximum waiting time of requests (MWT) set to be 1 second ($DB1$) and 10 seconds ($DB10$), respectively.

Figure 6.4: Makespan offered by DB and UBD Mechanisms

Figure 6.5: Traces of GridRPC Requests in DB and UBD Mechanisms

In all cases, the DB mechanism outperformed the UBD mechanism (see Figure 6.4). Figure 6.5 presents the traces of primitive requests of UBD and DB . In fact, although UBD can hide some latency of dispatching requests from the *GridRPC client program thread*, it still dispatches GridRPC requests one-by-one. In the DB mechanism, the

Figure 6.6: Number of Dispatchings in *DB* and *UBD* Mechanisms

requests are dispatched in bundles. Hence, the number of dispatchings is reduced (see Figure 6.6). As a result, the latency is reduced and thus the makespan of the entire application is also reduced.

6.2.3 Dynamic Bundling versus Static Bundling

In order to examine the effects of bundling overhead on the performance of GridRPC applications in the *DB* mechanism, we compared it to the *SB* mechanism in which the overhead of bundling is zero since the requests are bundled manually by users. The static bundling mechanism bundles 2 or 5 GridRPC requests into a composite request (denoted by *SB2* and *SB5*, respectively). In order to make the arrival rate of GridRPC requests in the *SB* mechanism similar to that of the *DB* mechanism, we used the interarrival times of GridRPC requests sampled in the *DB* mechanism. Then, the interarrival times of the primitive requests belonging to the same bundle were summed accordingly to form the interarrival time between composite requests for the *SB* mechanism. Note that in practice, static bundling requires programming effort and this may be impossible in many applications.

Figure 6.7: Interarrival Times of GridRPC Requests

Figure 6.8: Makespan Offered by Various Bundling Mechanisms

The interarrival times of GridRPC requests in the *SB* mechanism and the *DB* mechanism are presented in Figure 6.7. The makespan of the GridRPC application running with different bundling mechanisms is presented in Figure 6.8. As discussed above, the *SB* mechanism bundles GridRPC requests into composite requests with zero bundling overhead. The results show that the dynamic mechanism can work effectively with small overhead of bundling. Makespans of GridRPC application benchmarks in *DB1* and *DB10* are slightly higher than those of *SB2* and *SB5*.

Figure 6.9 presents traces of GridRPC requests in various bundling mechanisms. In the upper chart, *DB1* and *SB5* are compared. In *DB1*, the number of requests per bundle varies during execution time of the GridRPC application due to the arrival of GridRPC requests and the resource state.

In traces of *SB5*, there is a large gap of dispatched times between the composite request number 5 and 6 (i.e., primitive request 24 and 25). It is because the Grid resources were unavailable after receiving the first 5 composite requests. Hence, later requests had to wait for the availability of resources before being dispatched. This gap does not appear in *SB2* since the bundle size is small. The available slots in the resource are not filled up as quickly as in the case in *SB5*.

Figure 6.9: Traces of GridRPC Requests in Dynamic and Static Bundling Mechanisms

Although the *SB* mechanism offers an improvement to GridRPC applications, this mechanism is not transparent to the users. In addition, there are some issues of selecting bundle size for GridRPC requests. If the bundle size is small, the benefit of bundling may not improve the performance of GridRPC applications much. On the other hand, if the selected bundle size is large, the GridRPC client may spend a very long waiting time for the availability of Grid resources that have a large enough number of available slots. This scenario is possible on the Grid where the states of resources change dynamically.

6.2.4 Effects of Maximum Waiting Time in Dynamic Bundling

In this part, we examine the effects of Maximum Waiting Time (*MWT*) on the performance of GridRPC applications in the dynamic bundling mechanism. *MWT* is the maximum waiting time for bundling of GridRPC requests in the Request List at a GridRPC client. When the waiting time of the request is over *MWT*, the *Metascheduler/Bundler thread* is activated.

There are trade-offs when selecting *MWT* for the bundling mechanism. With the same average interarrival time of primitive requests, bundling with a small *MWT* may not take much advantage of bundling since the number of primitive requests in a composite request, i.e., bundle size, may be as small as one. This does not reduce dispatching time of requests compared to the unbundled mechanism. On the other hand, if the *MWT* is too large, more primitive requests can be bundled. However, bundling with large *MWT* adds more waiting time to the primitive requests. Selecting a suitable *MWT* by taking into account the average interarrival time of primitive requests will optimize the performance of the applications.

In order to examine the impacts of *MWT* on the makespan of applications, we carried out a set of experiments with different average interarrival times of primitive requests. There are 50 primitive requests in the application. Each request requires 300MFLOs to

complete. The average interarrival times of the experiments are 0.2, 0.6, 1, 2, 8, 14, 18, and 20 seconds. The Grid resource used is a cluster of computers with the configuration described in Table 6.1.

Figure 6.10: Time Improvement of $DB5$ and $DB10$ versus $DB1$

Figure 6.10 presents the improvement of $DB5$ and $DB10$ versus $DB1$. The improvement time is calculated as follows:

$$ImprovementTime = makespan(DB_1) - makespan(DB_i)$$

where i is 5 or 10.

If the average interarrival times of primitive requests are small, i.e., from 0.2 to 1.0, $DB5$ and $DB10$ are not as efficient as $DB1$ in general. Within one second of MWT , there are more than one request generated. In this scenario, $DB1$ can take advantage of the bundling mechanism. In $DB5$ and $DB10$, although they may have more primitive requests in a composite service, their longer MWT s add more delay time to the primitive requests. Hence, the makespans of the application in $DB5$ and $DB10$ are longer than the makespans offered by $DB1$.

In the case where the average interarrival times of primitive requests are large, *DB5* and *DB10* work more effectively than *DB1*. Within one second of *MWT*, *DB1* rarely has several primitive requests to bundle into composite requests. Hence, it cannot take advantage of the bundling mechanism. In contrast, *DB5* and *DB10* can capture more than one primitive request during its *MWT* to bundle and hence they can reduce makespans of the application. This happens when the average interarrival times of primitive requests are greater than one.

However, the improvement may not be achieved when the average interarrival time is very large. Once it reaches a certain point equalling to the *MWT*, the advantage of bundling cannot be achieved since the primitive requests are rarely bundled. In these experiments, this happens when the average interarrival times are above 18s. In this case, the waiting time of primitive requests is also high, which also increases the makespan of the application.

6.2.5 Summary

In this section, we present performance results offered by the unbundled and bundling mechanisms on a single cluster of computers. Following are some observations of different mechanisms:

- The dynamic bundling mechanism outperforms the unbundled mechanism in most of the cases. It is because the dynamic bundling mechanism hides the overhead and latency of dispatching when requests are dispatched in bundles.
- The static bundling mechanism can offer a shorter makespan to GridRPC applications than dynamic bundling when the bundle size of composite requests is well selected. However, it is neither transparent to users nor flexible on a Grid environment in which resources may change their state dynamically. Experiments

Figure 6.11: Arrival of GridRPC Requests

show that the dynamic bundling mechanism involves an acceptable small amount of overhead compared to the static bundling mechanism which has zero overhead of bundling.

- There are trade-offs when selecting MWT in the dynamic bundling mechanism. If MWT is small, the number of primitive requests in a composite request is small. This limits the advantages of bundling since the number of dispatchings in the bundling mechanism is not reduced much. On the other hand, if the MWT is large, the waiting time of the primitive requests being bundled is increased. Hence, selecting MWT by taking into account the average interarrival times of primitive requests is necessary to improve performance in the bundling mechanism.

6.3 Performance of GridRPC Mechanisms on Multiple Grid Resources

In order to examine the performance offered by the metascheduler, we created a GridRPC application benchmark that generates GridRPC requests randomly. Each GridRPC client

creates its GridRPC requests during 20 stages. The number of requests created in each stage is randomly generated. We have two random patterns called $P1$ and $P2$ for GridRPC requests. They are presented in Figure 6.11. The total number of requests are 67 and 91, respectively. These random patterns emulate Genetic Algorithm applications in which there are loops of evaluations of a number of populations. After each iteration, crossover and mutation are carried out and a number of new populations are reproduced to evaluate until some criteria are reached.

To generate different arrival rates of GridRPC requests, we adjust the interval time between stages (i.e., *stage interval*). For example, if the experiment is carried out with a high rate of request arrival, the interval between stages is configured to be small. Otherwise, a low rate of request arrival can be achieved if the interval between stages is large. Four cases of the application benchmarks are presented in Table 6.2.

Name	Program Used	Stage Interval
$P1 - 10$	$P1$	10s
$P1 - 40$	$P1$	40s
$P2 - 10$	$P2$	10s
$P2 - 40$	$P2$	40s

Table 6.2: Configuration of Benchmarks

In these experiments, there are three clusters of computers available on the multi-cluster Grid. Their configurations are listed in Table 6.3.

Property	Cluster 1	Cluster 2	Cluster 3
Name	<i>csge0</i>	<i>ecpdccm</i>	<i>ntuds</i>
Number of Compute Nodes	3	8	1
Number of Processors per Compute Node	1	1	2
Processing Speed (MFLOPS)	29.44	11.87	11.87
Initial Average <i>load_one</i>	0.667	0.625	0.5

Table 6.3: Configuration of Testbed

Figure 6.12: Makespan of Single Application Cases

Figure 6.13: Traces of GridRPC Requests in $P1 - 10$

6.3.1 Single Application

In this part, we study the performance of the mechanisms when the Grid resources are used by a single application. The job size of each primitive request used in this experiment is 300 MFLOs. The makespans of GridRPC applications using unbundled and dynamic bundling mechanisms are presented in Figure 6.12, where $DB1$ represents the dynamic bundling case with MWT equal to 1 second and $DB10$ the dynamic bundling case with MWT equal to 10 seconds. We used $Mincost - O$ heuristic to select Grid resources. The results show that the dynamic bundling mechanism can offer a better makespan to the GridRPC applications in most of the cases.

Figure 6.13 presents the traces of GridRPC requests in the $P1 - 10$ application. The resources where the requests were executed are also shown in the figure. In the unbundled

Figure 6.14: Concurrent GridRPC Applications

case (see the left chart), the requests did not wait for bundling. Hence, they are dispatched nearly immediately after being created. In contrast, the dynamic bundling mechanism keeps requests waiting in the Request List up to MWT . However, in the unbundled mechanism, since the primitive requests are dispatched individually, compared to the bundling case, there is a larger dispatching overhead.

The dynamic bundling mechanism effectively increases the granularity of the jobs scheduled to the resources. Instead of dispatching individual jobs corresponding to primitive requests, now the metascheduler dispatches bundles of jobs, corresponding to composite requests. As a result, the “best” resource would be loaded up faster when the dynamic bundling mechanism is used since many requests are scheduled to the “best” resource until it is full. Hence, this gives more opportunities to other resources when the “best” is full. This helps to spread the load amongst clusters. Therefore, the dynamic bundling mechanism can balance the workload better than the unbundled mechanism, as shown in Figure 6.13.

Figure 6.15: Makespans of Multiple Applications Case

Figure 6.16: Traces of GridRPC Requests in Testcase 1

6.3.2 Multiple Applications

In order to further check the performance of dynamic bundling versus the unbundled mechanism, we created multiple applications that shared the same GridRPC client runtime system. Each application was implemented as a thread of execution. Therefore, two applications ran concurrently and dispatched requests to a single metascheduler at the same time. In this case, the metascheduler has to handle multiple sources of GridRPC requests (see Figure 6.14).

Two testcases, each of which has a different combination of the two applications, were created:

- *Test case 1* consists of two applications: *P1* – 10 with 300 MFLOs jobs and *P2* – 10

with 600 MFLOs jobs.

- *Test case 2* also consists of the two applications, but with different job size configuration: $P1 - 10$ with 600 MFLOs jobs and $P2 - 10$ with 300 MFLOs jobs.

The makespan values offered by the various mechanisms are presented in Figure 6.15. These results show that the dynamic bundling mechanism still offers a better makespan to the GridRPC applications than the unbundled mechanism. The sample trace of the GridRPC requests in the first test case is also presented in Figure 6.16. The results obtained are similar to those of the single application case. That is, the dynamic bundling mechanism is able to balance the workload better. The benefits of bundling can compensate for the overhead of bundling and the dynamic bundling mechanism gives better performance than the unbundled mechanism.

6.3.3 Comparison between Mincost, Mincost-O, and Random Heuristics

The Mincost-O (*MO*) heuristic takes into account the overhead of managing jobs on a Grid resource. Similar to Mincost (*MC*), Mincost-O also selects the resource that has the lowest cost of execution. But, the cost of execution includes the occupation ratio besides processing speed. This heuristic makes the metascheduler schedule jobs onto loaded resources more “carefully” because it takes into account the state of resource. When a resource is going to be fully loaded, its number of available slots becomes smaller and thus its occupation ratio is larger. So, the cost of execution on a heavily loaded resource is high, although it may have a higher speed of execution. This gives the metascheduler chances to dispatch jobs onto other low speed resources with lower occupation ratio.

The full comparison of makespans offered by Mincost and Mincost-O with different bundling mechanisms for medium (300 MFLOs) and large (600 MFLOs) job sizes is

Figure 6.17: Mincost and Mincost-O in Various Scenarios

presented in Figure 6.17. It shows that Mincost-O works more efficiently than the Mincost heuristic when the arrival rate is high and job size is large.

Figure 6.18: Makespan Comparison of Various Resource Selection Heuristics

We also compare Mincost and Mincost-O heuristics with Random selection strategies. Figure 6.18 presents makespans offered by various resource selection heuristics under different test cases. In most cases, Mincost and Mincost-O outperform the Random approach.

6.3.4 Summary

The dynamic bundling mechanism offers shorter makespan to applications than the un-bundled mechanism. In this section, a single application and concurrent applications

using both mechanisms were examined. Since the dynamic bundling mechanism dispatches requests in bundles, it can reduce the overhead and latency of dispatching. At the same time, scheduling requests in bundles can help balance workload across Grid resources. Therefore, dynamic bundling outperforms the unbundled mechanism in most cases.

In this section, we also investigated resource selection heuristics employed by the metascheduler. Normally, the Mincost heuristic offers good performance to GridRPC applications. However, under high workload (e.g., applications with a small interval between stages and large job size), it may not be as effective as the Mincost-O heuristic. In the Mincost-O heuristic, the metascheduler schedules jobs onto loaded resource more “carefully” when a resource is going to be fully loaded. This avoids overloading Grid resources and also gives more opportunity to other Grid resources, which enables load balancing across different resources.

6.4 Conclusions

In this chapter, we examined the performance of GridRPC applications in different scenarios.

Firstly, the performance improvement by using a multithreading technique in the unbundled mechanism was examined on a single Grid resource. In this case, the unbundled mechanism with immediately dispatching (*UID*) mechanism and the background dispatching (*UBD*) mechanism were investigated. Since the multithreading technique helps hide the latency of request dispatching, the *UBD* mechanism can offer a shortened makespan to GridRPC applications than the *UID* mechanism.

Secondly, in many experiments on a single Grid resource, the dynamic bundling mechanism offers a better makespan to GridRPC applications than the unbundled mechanism.

The reason behind this is that the total dispatching latency of all requests in a GridRPC application is reduced in the dynamic bundling mechanism. We also compared dynamic bundling to static bundling. The results show that static bundling can offer an improvement of makespan to GridRPC applications. However, it is not transparent to the users and may be inflexible on the Grid. Further, a comparison of different Maximum Waiting Time (*MWT*) values configured in the dynamic bundling mechanism was carried out. Using a very large or a very small *MWT* may not generate good results. There should be a suitable *MWT* for a specific arrival rate of GridRPC requests.

Thirdly, a comparison of different resource selection heuristics was carried out. In most cases, Mincost and Mincost-O outperform the Random selection strategy. Especially, Mincost-O is suitable for GridRPC applications where GridRPC requests arrive at high rate and their job size is large. In this scenario, the Mincost-O heuristic can balance workload on different resources better than the Mincost heuristic and offer shorter makespan to the GridRPC application.

Finally, the dynamic bundling mechanism using Mincost or Mincost-O still offers better performance to GridRPC applications when multiple applications share the same GridRPC run-time system.

In conclusion, the reduction of latency of request dispatching and workload balancing are two major advantages of the dynamic bundling mechanism. Note that the dynamic bundling mechanism does not improve the response time of individual requests.

Chapter 7

GridRPC Programming Environment

In this chapter, we present a Grid Application Development toolkit called GAD Kit. It is the GridRPC Programming Environment developed to simplify the “gridifying” process by providing mechanisms to enable existing applications as Grid services and to consume these services. The GAD Kit includes features for automatic wrapping of applications as GridRPC services, service deployment and service consuming. We also present the use of the GAD Kit for “gridifying” a realistic aerodynamic design application.

7.1 Introduction

Recently, there has been a lot of work done in the areas of core middleware support for building Grid computing environments (e.g., Globus [FK98a]), application specific problem solving environments (e.g., Cactus-G [GAL⁺03] and GrADS [BCC⁺01]) and portals (e.g., Grid Portal Toolkits [TDM⁺02]). However, at present, the tools and environments that allow users to program the Grid are still quite limited. In particular, how to simplify application development on the Grid environment remains an open issue.

In general, “gridifying” is the process of transforming an existing application into

one that can run on the Grid environment. The “gridifying” process generally involves issues such as parallelizing applications (if the application is sequential), deploying Grid services onto the Grid environment, discovering and consuming the services available as well as scheduling them on the Grid. Here, it is assumed that the applications are already parallelized or at least they are constructed from independent services. From the user’s point of view, the “gridifying” process can be divided into two categories, namely, service provisioning and service consumption. To enable existing applications to be suitable for the Grid environment, it is often necessary to wrap the applications as Grid services in such a manner that it can be accessed with ease.

For general users, skills to perform the distribution and deployment of Grid services across the heterogeneous systems may be limited. Further, performing these manual tasks can be very time consuming and tedious for most users. In addition, the typical manner of deploying applications may not be feasible since the Grid resources are heterogeneous and managed by different administrative domains.

Constructing an application to utilize different Grid services spanning across various geographical locations can be a complex task. The need for the users to keep track of the set of resources available on the Grid, i.e., where and what resources are, would be almost impossible. Moreover, since the Grid resources are dynamic and different in capabilities, it is inefficient to carry out resource discovery and selection manually, especially when a large number of requests are often submitted onto the Grid. Therefore, the tasks of enabling existing application to be suitable for the Grid environment and consuming them with ease are not trivial.

The rest of this chapter is organized as follows. In the second section, we present the architecture and design of GAD Kit. How to “gridify” a realistic application using the GAD Kit will be demonstrated in the third section. Finally, the fourth section draws our conclusions.

Figure 7.1: GAD Kit's Architecture

Figure 7.2: Front-end GUI of GAD Kit

<i>< Function ></i>	→	<i>< FunctionDef ></i> <i>< ArgumentList ></i>
<i>< FunctionDef ></i>	→	FUNCTION : <i>< FunctionName ></i> <i>< FunctionLib ></i>
<i>< FunctionName ></i>	→	<i>< string ></i>
<i>< FunctionLib ></i>	→	<i>< Lib ></i> <i>< Include ></i> <i>< Executable ></i>
<i>< Lib ></i>	→	LIBRARY : <i>< string ></i>
<i>< Include ></i>	→	INCLUDE : <i>< string ></i>
<i>< Executable ></i>	→	EXECUTABLE: <i>< string ></i>
<i>< ArgumentList ></i>	→	<i>< ReturnArg ></i> <i>< NumArguments ></i> <i>< Arglist ></i>
<i>< ReturnArg ></i>	→	RETURN : <i>< DataSpec ></i>
<i>< NumArguments ></i>	→	NUMARGUMENTS : <i>< integer ></i>
<i>< Arglist ></i>	→	<i>< Arg ></i> <i>< Arglist ></i> <i>< Arg ></i>
<i>< Arg ></i>	→	ARGUMENT : <i>< ArgType ></i> <i>< DataSpec ></i>
<i>< DataSpec ></i>	→	<i>< StructureSpec ></i> <i>< TypeSpec ></i>
<i>< ArgType ></i>	→	IN OUT INOUT
<i>< StructureSpec ></i>	→	SIMPLE VECTOR MATRIX
<i>< TypeSpec ></i>	→	VOID CHAR INT FLOAT DOUBLE

Figure 7.3: GFD Specification

7.2 GAD Kit Architecture and Design

GAD Kit comprises of a Graphical User Interface (GUI) module, GridRPC service provider module, GridRPC service consumer module, and resource monitor module. Its architecture is presented in Figure 7.1. The GridRPC service provider module allows users to deploy GridRPC services on enabled Grid resources while the consumer module allows users to consume the Grid resources through GridRPC clients. The resource monitor obtains and displays static and dynamic information of Grid resources such as existing services, Grid resources' configuration and workload that assist service deployment and consumption. Following are the details of GUI, service consumer and service provider modules.

7.2.1 GAD Kit's GUI

At the front-end is a GUI provided for ease of access to the various modules available in the GAD Kit. It comprises of a set of interfaces to the mechanisms that provide the

monitoring of Grid resources, deploying Grid services as well as discovering and consuming Grid services. The GUI also assists users in customizing their applications for the Grid environment. This helps to eliminate most of the need for complex low-level coding by users. Figures 7.2, 7.4, and 7.6 illustrate some snapshots of the GAD Kit GUI.

7.2.2 Service Provider

The role of the service provider module in the GAD Kit is to provide automatic wrapping and deployment of existing applications as Grid services. The process of enabling an application on the Grid is summarized as follows:

Generation of Grid Service Interface

	Type	Structure	In/Out
Return Value	INT	SIMPLE	OUT
Argument 1	FLOAT	SIMPLE	IN
Argument 2	FLOAT	SIMPLE	IN
Argument 3	FLOAT	VECTOR	IN
Argument 4	FLOAT	SIMPLE	OUT
Argument 5	FLOAT	VECTOR	OUT
Argument 6	FLOAT	SIMPLE	OUT
Argument 7	FLOAT	SIMPLE	OUT
Argument 8	FLOAT	SIMPLE	OUT
Argument 9	INT	MATRIX	IN
Argument 10	INT	MATRIX	IN

Figure 7.4: Service Provider Interface

Since the Grid is heterogeneous in nature, it is necessary to provide a standard format

```

FUNCTION : airfoillanalysis
LIBRARY : /home/hqthuan/airfoil/libairfoil.a
INCLUDE : /home/hqthuan/airfoil/airfoil.h
EXECUTABLE : /home/hqthuan/airfoil/airfoil
RETURN : SIMPLE VOID
NUMARGUMENTS : 8
ARGUMENT : IN SIMPLE FLOAT
ARGUMENT : IN SIMPLE FLOAT
ARGUMENT : IN VECTOR FLOAT
ARGUMENT : OUT SIMPLE FLOAT
ARGUMENT : OUT VECTOR FLOAT
ARGUMENT : OUT SIMPLE FLOAT
ARGUMENT : OUT SIMPLE FLOAT
ARGUMENT : IN SIMPLE INT

```

Figure 7.5: GFD of a Grid service

for specifying the Grid service interfaces. Hence, a Grid Function Description (GFD) language has been proposed. The GFD is similar in spirit to CORBA IDL [Obj04], Ninf-G's IDL, and NetSolve's Problem Description File. It is simple and both human as well as machine readable. The GFD grammar is presented in Figure 7.3. Figure 7.4 displays the GAD Kit service provider GUI for easy configuration and creation of a Grid service. The corresponding GFD generated is shown in Figure 7.5.

Wrapping Existing Application

To provide a platform-independent Grid Application Development Environment, various domain-specific interfaces may be generated from the standard GFD interface. Currently, the GAD Kit supports the problem description interface file format of NetSolve [ACD02].

Deployment of the Grid Service

Once software applications have been wrapped as Grid services, these Grid services will be deployed automatically onto Grid nodes as specified by the user. Deploying services

onto the Grid comprises of distributing and installing the Grid services. Since the Grid is heterogeneous in nature, it is critical to employ standard Grid technologies for these tasks. Hence, standard GridFTP and Globus remote commands are used.

7.2.3 Service Consumer

The purpose of the service consumer module in the GAD Kit is to assist users to develop and execute GridRPC client programs using the existing Grid services. The GAD Kit service consumer module provides three main mechanisms:

Generation of Code Templates

	Service Used	No. of Requests
Task 1	airfoilshep	n1
Task 2	airfoilana	n2
Task 3	airfoilana	1
Task 4	airfoilana	1
Task 5	airfoilana	1

Figure 7.6: Service Consumer Interface

```

#include "grpc.h"
int main(int argc, char **argv) {
    int i;
    int n1, n2;
    grpc_function_handle_t * handle;
    grpc_initialize(...);
    handle = (grpc_function_handle_t )
             calloc(n1, sizeof(grpc_function_handle_t));
    for(i=0; i<n1; i++) {
        grpc_function_handle_default(handle[i], airfoilana);
        grpc_call_async(handle[i], ...);
    }
    grpc_waitall();

    //Phase 2
    handle = (grpc_function_handle_t )
             calloc(n2, sizeof(grpc_function_handle_t));
    for(i=0; i<n2; i++) {
        grpc_function_handle_default(handle[i], airfoilana);
        grpc_call_async(handle[i], ...);
    }
    grpc_waitall();

    grpc_finalize();
    return 0;
}

```

Figure 7.7: A Sample Application Program generated by GAD Kit

Programming applications for consuming Grid resources can be notoriously tedious and complex. Hence, templates for consuming Grid services are automatically generated in the service consumer module. This helps to reduce programming time and eliminate programming errors. Figure 7.6 illustrates the service consumer GUI provided for selecting the Grid services of interest.

The GUI provides a simple configuration of task workflow. Each task in the workflow may be specified to execute in parallel. The use of multiple asynchronous requests in the module allows parallelism to be exploited. A sample code template for consuming Grid

services, generated by the GAD Kit interface shown in Figure 7.6, is given in Figure 7.7. It generates GridRPC client programs for consuming existing Grid services.

Execution of GridRPC Application

After a GridRPC client program is created from GAD Kit, it can be executed to consume the Grid resources. The GridRPC application can take advantage of the resource discovery and metascheduling mechanisms implemented in the GridRPC API. These mechanisms facilitate efficient service of requests across the dynamically changing resources based on the system configuration, speed, and workload information obtained from the information service.

7.3 Realistic Airfoil Analysis and Design on the Grid using GAD Kit

In this section, we use the GAD Kit for “gridifying” an aerodynamic design optimization application. Aerodynamic design, one of the most frequently tackled problems in aeronautics, is a computationally intensive application. Due to the large design spaces, usually stochastic optimization algorithms such as Genetic Algorithms (GAs) are employed in the aerodynamic search in order to arrive at a near optimum design [YI95, QPPW98]. GAs typically require thousands of function evaluations and hence require excessive CPU time to locate a near optimal solution, where a single function evaluation involving high-fidelity analysis codes could take up to many minutes to hours of computation time. Hence, it is not uncommon that the design of an aircraft wing using evolutionary algorithms may take up several months of CPU time even on a supercomputer. This often poses a serious impediment to the practical application of evolutionary optimization to complex engineering design problems as the design process may prove to be computationally intractable if an optimal design is desired.

Figure 7.8: “Gridifying” of the GA for Aerodynamic Design

In this case study, the application to be “gridified” is a Genetic Algorithm for optimization of airfoil designs. The optimization is carried through a number of generations. In each generation, the Grid-enabled GA produces a population of designs for analysis. Then the results were used as input for the next stage. The “gridifying” stages of the GA are detailed in Figure 7.8(a). Existing airfoil analysis codes were wrapped in terms of GridRPC services. The GFD of one of these services is presented in Figure 7.8(b). The GridRPC client program to carry out the airfoil analysis was created from the template displayed in Figure 7.8(c). The “*airfoilshp*” calls create airfoil shapes from the design parameters. It takes a few seconds to complete the “*airfoilshp*” calls while the “*airfoilana*” calls take a very long time. This is because “*airfoilana*” calls require expensive computation of optimization code.

Property	Cluster 1	Cluster 2	Cluster 3
Name	<i>csge0</i>	<i>ecpdccm</i>	<i>ntuds</i>
Number of Compute Nodes	3	8	1
Number of Processors per Compute Node	1	1	2
Processing Speed (MFLOPS)	29.44	11.87	11.87
Initial Average <i>load_one</i>	0.667	0.625	0.5

Table 7.1: Configuration of Testbed for Airfoil Analysis and Design

The airfoil analysis application was deployed on three enabled Grid resources within the Nanyang Technological University Campus Grid [Nan]. Table 7.1 describes the configuration of these resources.

Figure 7.9 presents the makespan of the airfoil analysis with 40, 50, 60, and 70 designs (corresponding to $n1$ and $n2$ in Figure 7.6, 7.7 and 7.8(c); here, $n1 = n2$). All primitive GridRPC requests are generated almost at the same time. Due to dispatching overhead, *UBD* offers a longer makespan than the *DB* mechanisms in all the cases.

Figure 7.9: Makespans of Analysis of Airfoil Designs

7.4 Conclusions

The availability of a “gridifying” tool for applications plays a crucial role in the success of Grid computing. Over recent years, there have been increasing interest and research efforts toward providing such mechanisms. In this chapter, we presented GAD Kit, a Grid application development environment for simplifying the “gridifying” process and offering a seamless and transparent access to Grid resources. The GAD Kit comprises of the GUI module and a set of mechanisms for providing and consuming Grid services. It covers most of the mechanisms necessary for Grid application development. In addition, the service consumer mechanism frees users from tedious low-level programming and the complexity of the underlying system. Moreover, with abilities to perform resource discovery and scheduling, GAD Kit helps applications developers to submit their requests across multiple clusters of computers. A case study on using the GAD Kit for facilitating the exploitation of parallelism in the aerodynamic design optimization application is also demonstrated.

Chapter 8

Conclusions and Future Work

8.1 Work Done

GridRPC Execution Environment	<ul style="list-style-type: none"> • Implemented information service for GridRPC applications. • Equipped Grid resources with NetSolve • Improved Local Resource Manager. • Implemented GridRPC API for GridRPC client with various bundling and metascheduling mechanisms. • Carried out performance analysis and comparison for bundling and metascheduling mechanisms.
GridRPC Programming Environment (GAD Kit)	<ul style="list-style-type: none"> • Implemented GFD generator for GridRPC services. • Implemented code templates for GridRPC client program and code generator for GridRPC client program using templates. • Developed GAD Kit's GUI for service consumer, provider, and resource monitor.

Table 8.1: List of Work Done

Enabling applications on the Grid environment is difficult and challenging. It needs

to hide the complication of the Grid environment and at the same time achieve high performance for the applications. The work presented in this thesis targets the above goals. It offers a GridRPC-based environment that includes a programming environment and an execution environment. In order to hide the complication of the Grid environment, we provided not only the implementation of a high-level programming interface, i.e., GridRPC API implementation, but also mechanisms that enable Grid applications to automatically make resource discovery, resource selection and metascheduling. A list of work done for the GridRPC-based environment is presented in Figure 8.1.

8.2 Contributions

In this work, approaches to improve performance of embarrassingly parallel applications were examined. In order to deliver high performance to Grid applications, the environment should possess effective local schedulers, metascheduler, and mechanisms for hiding latency and maximizing parallelism. Following are the summary of some contributions in this work.

- Improvement of Local Resource Manager:

The local resource schedulers should be able to handle services that are supplied to Grid applications effectively. Since the Grid resources may be clusters of computers, load balancing amongst compute nodes within a cluster as well as across clusters is significant.

In this work, we examined the effect of the degree of concurrency on the performance of GridRPC services and improved NetSolve's scheduling algorithm on a cluster of computers. We found that the medium degree can offer the shortest response time of service rather than low or high degree of concurrency.

- Metascheduler:

Since the Grid environment comprises of heterogeneous resources, selecting enabled Grid resources in such a manner that they can offer best performance to the application is an issue. In this thesis, we presented a method to determine the quantity of an enabled Grid resource, i.e., the number of requests a resource can accept. Subsequently, the metascheduler can carry out its resource selection among available resources. This avoids overloading on some Grid resources.

In addition, we introduced Mincost and Mincost-O heuristics for resource selection. They work more efficiently than the random approach. Especially, the Mincost-O heuristic enables the metascheduler to balance workload more effectively when the Grid applications require heavy computation.

- Mechanisms for Hiding Latency and Facilitating Achievement of Parallelism:

On the Grid environment, hiding communication latency and overhead is crucial for performance of applications. This work investigated mechanisms of request bundling and composite service. These mechanisms enable scheduling across multiple clusters of computers efficiently.

First, the bundling mechanism reduces communication between GridRPC clients and Grid resources. By grouping the relevant GridRPC requests and dispatching them in bundles, the dispatched time of GridRPC requests can be decreased. It avoids accumulating latency of dispatching when there are many GridRPC requests arriving at high rate.

Second, the bundling mechanism with the aid of a composite service is a good means to facilitate the achievement of parallelism on a cluster of computers. Once a bundle of GridRPC requests arrive at a Grid resource, parallel jobs can be created and distributed across compute nodes within the resource. Parallelism thus can be achieved.

8.3 Future Work

In this work, we investigated the metascheduler embedded into the GridRPC client. Although a case of multi-streams of GridRPC requests was examined, decentralized scenarios in which there are many metaschedulers working at the same time have not been considered yet. There might be problems when multiple metaschedulers dispatch GridRPC requests at the same time. In this situation, the Mincost or Mincost-O heuristics used in the current implementation may be inefficient because of resource competition. This is due to the fact that the metaschedulers that are working concurrently cannot know GridRPC requests of each others. Hence some resources appearing to be the “best” may be overloaded subsequently. Our future work will investigate both centralized and decentralized approaches for scheduling. In the former case, all clients will submit jobs through a centralized scheduler. When all of the clients are decentralized, optimistic scheduling, i.e., predicting new requests and scheduling them before they are actually created, can be used. By considering the workload caused by both real requests and optimistic requests, the metaschedulers of different clients are able to determine possible bottlenecks on some Grid resources and thus their performance can be improved.

Reusing existing modules and libraries is desirable when applications become larger and complicated. A nested GridRPC may prove to be a good form to support service providers constructing complex functional services. A composite service presented in this thesis can be seen as a form of nested GridRPC with two levels. The first level makes calls between a GridRPC client and a Grid resource while the second level makes calls between the instance of composite service and jobs running on the compute nodes within the Grid resource. Thus, the current implementation needs to be further extended to handle multi-level nested GridRPC.

Bibliography

- [ABD00] Dorian Arnold, Dieter Bachmann, and Jack Dongarra. Request Sequencing: Optimizing Communication for the Grid. *Lecture Notes in Computer Science*, 1900:1213–1222, 2000.
- [ACD02] Dorian Arnold, Henri Casanova, and Jack Dongarra. Innovations of the NetSolve Grid Computing System. *Concurrency: Practical and Experience*, 14(13-15):1457–1479, 2002.
- [ADF⁺01] Gabrielle Allen, Thomas Damlitsch, Ian Foster, Nicholas T. Karonis, Matei Ripeanu, Edward Seidel, and Brian Toonen. Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus. *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 52-52, 2001.
- [Adl95] Richard M. Adler. Distributed Coordination Models for Client/Server Computing. *IEEE Computer*, 28(4):14–22, April 1995.
- [AFM02] Istabrak Abdul-Fatah and Shikharesh Majumdar. Performance of CORBA-Based Client-Server Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 13(2):111–127, February 2002.
- [AGH⁺91] Hans-Ruedi Aschmann, Niklaus Giger, Elisabeth Hoepli, Peter Janak, and Hubert Kirrmann. Alphorn: A Remote Procedure Call Environment for

- Fault-Tolerant, Heterogeneous, Distributed Systems. *IEEE Micro*, 11(5):16–19, 60–67, September/October 1991.
- [AGK00] David Abramson, Jon Giddy, and Lew Kotler. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? *Proceedings of 14th International Parallel and Distributed Processing Symposium (IPDPS 2000)*, IEEE CS Press, page 520, May 2000.
- [Alt02] Altair Engineering, Inc. PBS Documentation Set. <http://www.pbspro.com>, 2002.
- [ARE] DATATAG WP4 AREA. *Glue Schema 1.1*. "<http://www.cnaf.infn.it/~sergio/datatag/glue/index.htm>".
- [BCC⁺01] Francine Berman, Andrew Chien, Keith Cooper, Jack Dongarra, Ian Foster, Dennis Gannon, Lennart Johnsson, Ken Kennedy, Carl Kesselman, John Mellor-Crummey, Dan Reed, Linda Torczon, and Rich Wolski. The GrADS Project: Software Support for High-level Grid Application Development. *The International Journal of High Performance Computing Applications*, 15(4):327-344, 2001.
- [BCL⁺97] Mostafa A. Bassiouni, Ming-Hsing Chiu, Margaret Loper, Michael Garnsey, and Jim Williams. Performance and Reliability Analysis of Relevance Filtering for Scalable Distributed Interactive Simulation. *ACM Transactions on Modeling and Computer Simulation*, 7(3):293–331, 1997.
- [BCW96] Mostafa Bassiouni, Ming Chiu, and Jim Wiiiams. Evaluation of Packet Delays and Packet Bundling in ATM Backbones. *Proceedings of IEEE's Military Communications Conference (MILCOMM)*, 1996.

- [Ber02] Michael Berger. Multipath Packet Switch using Packet Bundling. *Proceedings of High Performance Switching and Routing*, pages 244–248, 2002.
- [BM88] Alan M. Baum and Donald J. McMillan. Message Passing in Parallel Real Time Continuous System Simulations. *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications: Architecture, Software, Computer Systems, and General Issues*, 1:540–549, 1988.
- [BN84] Andrew Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [BSGA01] Rajkumar Buyya, Heinz Stockinger, Jon Giddy, and David Abramson. Economic Models for Management of Resources in Peer-to-Peer and Grid Computing. *Proceedings of the SPIE International Conference on Commercial Applications for High-Performance Computing*, pages 13–25, August 2001.
- [BW97] Francine Berman and Rich Wolski. The AppLeS Project: A Status Report. *Proceedings of 8th NEC Research Symposium*, 1997.
- [BWC⁺03] Francine Berman, Richard Wolski, Henri Casanova, Walfredo Cirne, Holly Dail, Marcio Faerman, Silvia M. Figueira, Jim Hayes, Graziano Obertelli, Jennifer M. Schopf, Gary Shao, Shava Smallen, Neil T. Spring, Alan Su, and Dmitrii Zagorodnov. Adaptive Computing on the Grid Using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, April 2003.
- [CB03] Walfredo Cirne and Francine Berman. When the Herd is Smart: Aggregate Behavior in the Selection of Job Request. *IEEE Transactions on Parallel and Distributed Systems*, 14(2):181–192, February 2003.

- [COBW00] Henri Casanova, Graziano Obertelli, Francine Berman, and Rich Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. *Proceedings Supercomputing 2000*, page 60, November 2000.
- [DGS98] Daniel R. Dooly, Sally A. Goldman, and Stephen D. Scott. TCP Dynamic Acknowledgment Delay: Theory and Practice (Extended Abstract). *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing*, pages 389–398, 1998.
- [FFK⁺97] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. *Proceedings of 6th IEEE Symposium on High-Performance Distributed Computing*, pages 365–376, 1997.
- [FK98a] Ian Foster and Carl Kesselman. The Globus Project: A Status Report. *Proceedings of IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.
- [FK98b] Ian Foster and Carl Kesselman. *The Grid: Blueprint for A New Computing Infrastructure*. Morgan Kaufmann, 1998.
- [FKNT02] Ian Foster, Carl Kesselman, J. Nick, and Steven Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Globus Project, <http://www.globus.org/research/papers/ogsa.pdf>, 2002.
- [FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *The International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

-
- [FLNU03] Jens S. Frederiksen, Kim S. Larsen, John Noga, and Patchrawat Uthaisombut. Dynamic TCP Acknowledgment in The LogP Model. *Journal of Algorithms*, 48:407–428, 2003.
 - [FTL⁺02] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3):237–246, July 2002.
 - [GAL⁺03] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan Masso, Thomas Radke, Edward Seidel, and John Shalf. The Cactus Framework and Toolkit: Design and Applications. *Lecture Notes in Computer Science*, 2565:197–227, 2003.
 - [Gri] Grid Remote Procedure Call Working Group, Global Grid Forum. <https://forge.gridforum.org/projects/gridrpc-wg/>.
 - [Jac04] David Jackson. Grid Scheduling with Maui/Silver. *Grid Resource Management: State of the Art and Future Trends*, pages 161–170, 2004.
 - [JSC01] David Jackson, Quinn Snell, and Mark Clement. Core Algorithms of the Maui Scheduler. *Lecture Notes in Computer Science*, 2221:87–102, 2001.
 - [KTF03] Nicholas Karonis, Brian Toonen, and Ian Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, May 2003.
 - [Law99] Lawrence A.H. Liang and Wentong Cai and Bu-Sung Lee and Stephen J. Turner. Performance Analysis of Packet Bundling Techniques in DIS. *Proceedings of the 3rd International Workshop on Distributed Interactive Simulation and Real-Time Applications*, pages 75–82, 1999.
-

-
- [LLM88] Michael Litzkow, Miron Linvy, and Matt Mutka. Condor - A Hunter of Idle Workstations. *IEEE, Proceedings of the 8th International Conference on Distributed Systems*, pages 104–111, June 1988.
- [MCC03] M.L. Massie, B.N. Chun, and D.E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. Technical report, University of California, Berkeley, http://ganglia.sourceforge.net/talks/parallel_computing/ganglia-twocol.pdf, February 2003.
- [Nan] Nanyang Technological University. *Nanyang Campus Grid*. <http://www.ntu-cg.edu.sg>.
- [NSS99] Hidemoto Nakada, Mitsuhsa Sato, and Satoshi Sekiguchi. Design and Implementations of Ninf: Towards a Global Computing Infrastructure. *Future Generation Computing Systems, Metacomputing Issue*, 15(5-6):649–658, 1999.
- [NTMS03] Hidemoto Nakada, Yoshio Tanaka, Satoshi Matsuoka, and Satoshi Sekiguchi. Ninf-G: a GridRPC System on the Globus Toolkit. *Grid Computing: Making the Global Infrastructure a Reality*, pages 625–638, March 2003.
- [Obj04] Object Management Group. Common Object Request Broker Architecture Specifications. <http://www.omg.org>, 2004.
- [QPPW98] Domenico Quagliarella, Jacques Periaux, Carlo Poloni, and Gabriel Winter, editors. *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*. John Wiley & Sons, 1998.
- [Rab95] Rolf Rabenseifner. The DFN Remote Procedure Call Tool for Parallel and Distributed Applications. *Kommunikation in Verteilten Systemen*, pages 415–429, 1995.

- [RD96] David E. Ruddock and Balakrishnan Dasarathy. Multithreading Programs: Guidelines for DCE Applications. *IEEE Software*, 13(1):80–90, January 1996.
- [Sha01] Gary Shao. *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources*. PhD thesis, University of California at San Diego, May 2001.
- [SN02] Jennifer Schopf and Bill Nitzberg. Grids: The Top Ten Questions. *Scientific Programming, special issue on Grid Computing*, 10(2):101–102, August 2002.
- [SNM⁺02] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. GridRPC: A Remote Procedure Call API for Grid Computing. Advanced Programming Models Research Group, The Global Grid Forum, http://www.eece.unm.edu/~apm/docs/APM_GridRPC_0702.pdf, July 2002.
- [Sun94] Sun Microsystems, Inc. Network Interfaces Programmer's Guide, Part 2 - Remote Procedure Call (RPC), 1994.
- [Sun01] Sun Microsystems, Inc. Grid Engine Documentations. <http://gridengine.sunsource.net/project/gridengine/documentation.html>, 2001.
- [TCF⁺03] Steven Tuecke, Karl Czajkowski, Ian Foster, Jeffrey Frey, Steve Graham, Carl Kesselman, Tom Maguire, Thomas Sandholm, Peter Vanderbilt, and David Snelling. Open Grid Services Infrastructure (OGSI) Version 1.0, 2003.
- [TDM⁺02] Mary Thomas, Maytal Dahan, Kurt Mueller, Steve Mock, Cathie Mills, and Ray Regno. Application Portals: Practice and Experience. *Concurrency and Computation: Practice and Experience*, 14(13-15):1427–1443, 2002.

- [Teaa] Globus Team. *Global Access to Secondary Storage (GASS)*. The University of Chicago and The University of Southern California, <http://www.globus.org/gass>.
- [Teab] Globus Team. *GridFTP*. <http://www.globus.org/datagrid/gridftp.html>.
- [Teac] NPACI Team. *NPACI Rocks Cluster Distribution: Users Guide*. UC Regents, <http://www.rocksclusters.org>.
- [VD02] Sathish Vadhiyar and Jack Dongarra. A Metascheduler for the Grid. *Proceedings of 11th IEEE International Symposium on High Performance Distributed Computing*, pages 343–351, July 2002.
- [VSM02] Sathish Vadhiyar, Keith Seymour, and Michelle Miller. Implementation of GridRPC over NetSolve. GGF5 APM Meeting, http://www.eece.unm.edu/~apm/GGF5/GridRPC_NetSolve.ppt, July 2002.
- [WAE03] Jon Bon Weissman, Lakshman Rao Abburi, and Darin England. Integrated Scheduling: The Best of Both Worlds. *Journal of Parallel and Distributed Computing*, 63(6):649–668, June 2003.
- [Wol98] Richard Wolski. Dynamically Forecasting Network Performance Using the Network Weather Service. *Cluster Computing*, 1(1):119–132, January 1998.
- [YI95] KenjiInoue Yamamoto and Osamu Inoue. Applications of Genetic Algorithm to Aerodynamic Shape Optimization. *Proceedings of the 12th AIAA Computational Fluid Dynamics Conference*, pages 43–51, 1995.

Appendix A

Publications

[HOC03] Quoc-Thuan Ho, Yew-Soon Ong, and Wentong Cai. “Gridifying” Aerodynamic Design Problem using GridRPC. *Proceedings of the 2nd International Workshop on Grid and Cooperative Computing, (GCC2003)*, Shanghai, China, December 2003.

[HOC⁺04] Quoc-Thuan Ho, Yew Soon Ong, Wentong Cai, Hee-Khiang Ng, and Bu-Sung Lee. GAD Kit - A Toolkit for Gridifying Applications. *Proceedings of the 5th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT2004)*, Singapore, December 2004.

[HCO05] Quoc-Thuan Ho, Wentong Cai, and Yew-Soon Ong. Design and Implementation of An Efficient Multi-cluster GridRPC System. *Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid2005)*, Cardiff, UK, May 2005.