Coordinated Virtual Partition: A Resource Provisioning Framework for Grid Applications

by

Tarek Ibne Mizan

A dissertation submitted in partial satisfaction of the
requirements for the degree of

Master of Science

Department of Computer Science

Faculty of Graduate Studies
University of Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

COORDINATED VIRTUAL PARTITION: A RESOURCE PROVISIONING FRAMEWORK FOR GRID APPLICATIONS

BY

Tarek Ibne Mizan

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

Of

MASTER OF SCIENCE

Abstract

Grid computing is an emerging computing paradigm to execute applications across geographically distributed shared resources. Grid applications usually have strict QoS requirements which must be met with guaranteed resource allocation. Furthermore, a Grid application may have multiple component tasks that run simultaneously on different resources. Performance of such Grid applications not only depends on the proper resource co-allocation but also on performance isolation and coordinated usage of allocated resources. Therefore it is necessary for a Grid resource management architecture to incorporate functionalities for ensuring coordinated resource usage during the execution of a multi-component Grid application.

This thesis presents *coordinated virtual partition* (CVP), a mechanism for performing coordinated resource provisioning for multi-component Grid applications. Here, resource provisioning means regulating the supply of the resources that is already allocated to the components of an application. With coordinated resource provisioning, we regulate the resources supplied to different components in unison according to an agreed relative proportion. This study shows that coordinated resource provisioning has several benefits including: (a) reducing the wait times experienced by a component task of a Grid application and (b) improving the overall application performance by reducing the wait times. The CVP achieves these benefits by releasing resources from "fast" running application components that can be reallocated by the Grid to other applications. A prototype of CVP was implemented on Linux and was used to carry out various experiments. We demonstrated the utility of CVP by mapping a representative multi-component application written in *Message Passing Interface* (MPI) onto CVP. The results show that CVP can improve the performance of a MPI program by reducing the barrier wait times.

Acknowledgements

In the very beginning, I express my gratitude to **Allah**, the Almighty, Who shows me the right path of life and has helped me to finish this thesis.

I am very pleased to work under the supervision of **Dr. Muthucumaru Maheswaran**. His deep insights and innovative ideas have given me the driving force to jump into my research on grid computing. Whenever I got stuck in my research, he was always there to help me out. He is the kind of supervisor that any graduate student will always cherish for. I am also thankful to the committee members of my thesis, **Mr. Rasit Eskicioglu** and **Dr. Bob McLeod** for reading and evaluating this thesis.

I thank **TR*Labs*** for not only supporting me financially but also providing me an excellent research environment and superb working place. TRLabs has made my research efforts very smooth. It has been a great honor to be a part of this institution.

My thanks go to Mr. Vasee Vaseeharan, TRLabs' system administrator, for his technical advice. He has made my life much easier by helping me in various system setups. Also, many thanks to Mr. Ramandeep Bhinder for helping with valuable code.

I am grateful to Mr. Farag Azzedin who I first met when I came to Canada. His guidance, honest suggestions and inspirations was an invaluable asset during my graduate study. I respect him honestly from the bottom of my heart. I also extend my thanks to all my friends in Winnipeg for their co-operation and suggestions on various problems.

My utmost thanks go to my family, my father, my mother, my elder brother and sister-in-law. My parents sacrificed their whole lives to take me up to this position. I will never be able to sufficiently express my gratitude towards them. My brother has always been my great help for me whenever I have a problem. His guidance in my early academic years has put me on the right track of career. Also, my deep love goes to my little nephew, Anannya, who is a source of great joy and inspiration to me.

Contents

List of Figures

Chapter 1

Introduction

The advent of very high-performance computing and communication resources coupled with widespread growth of the Internet has created enormous potential to deliver reliable, powerful and wide range of IT services to consumers. The growing popularity of e-business and e-science also demands for a service-oriented computing architecture. Research in distributed computing architecture has gained momentum in recent years with the aim to stage services over a wide-area network by integrating geographically dispersed resources from different administrative domains. Examples of such distributed computing architectures include Grid Computing [FoK01], Peer-to-Peer Computing [Ora01] and Public Computing Platform [RoS01]. However, these emerging technologies are facing a number of challenging issues in order to provide seamless, transparent and secure access to IT resources for the consumers.

Resource management is one of the major concerns in a wide-area computing architecture. Wide-area applications can run in an environment that involves heterogeneous resources from different institutions interlinked with complex resource-sharing relationships. In addition, the demand for services and availability of resources can vary dynamically. Therefore, the task of guaranteeing quality of service (QoS) to wide-area

applications requires proper allocation and coordination of the resources.

In this thesis, I propose a mechanism to perform the task of coordinated usage in a Grid resource management architecture.

1.1 Motivation

Grid Computing [FoK01] is an emerging computing infrastructure for deploying wide-area applications over geographically distributed resources. It has enabled the development of interesting applications that require the integration of high-performance resources and data sources. Examples of such applications include distributed supercomputing, remote visualization, distributed scientific simulation, and online gaming. Applications running on a Grid environment encounter some unique resource characteristics compared to traditional computing environments (e.g., single node or cluster-based). Following are some of those characteristics:

- *Heterogeneity of resources*: Resources present in a Grid can vary widely in terms of hardware architecture, operating system, capacity and policy.

- *Uncertainty in resource availability*: The collection of resources in a Grid is dynamic. A resource can join and leave the pool of resources at any time.

- *Shared hosting environment*: As supply of resources can be limited, for ensuring higher resource utilization and lower cost, multiple applications may need to consume resources by sharing same physical machines.

- *De-centralized control domains* : There is no centralized control for the resources in a Grid. Resources from different institutions are under the control of their respective domains. Each domain imposes its own domain-specific policies.

Because most of the applications targeted at Grids arise from cluster or parallel computing systems, coping with unique Grid resource characteristics is a major consideration in migrating applications to Grids. The resource management mechanism has to include the functionalities to create proper execution environment for applications on Grid. We identify the following issues that are necessary to provide support for Grid applications:

- When multiple applications with strict QoS requirements are mapped onto a shared hosting platform, it is imperative to guarantee appropriate resource allocation for each of the applications. Performance isolation among these applications can ensure guaranteed resource. Operating system researchers have made significant progress in achieving performance isolation of applications on shared resources [BaD99, ArD00]. These approaches work by modifying the kernel of the native operating system. However, in a Grid environment, sites are autonomous and modifying the kernel may not be an option. Therefore, a middleware approach would be more appropriate for achieving performance isolation on the diverse types of Grid resources. An advantage that makes a middleware more appealing is that it could be easily integrated into the Grid substrate as a low-level component of a Grid toolkit [FoK97].

- A Grid application can have multiple component tasks that execute in parallel on different machines. The MPICH-G2 [KaT03] from the Globus toolkit is a Grid-enabled system for running Message Passing Interface (MPI) programs. This system uses the Globus toolkit [FoK97] services to address issues like authentication and authorization across domains, staging executables, resource co-allocation, handling different communication protocols and status monitoring. However, MPICH-G2 does not isolate the different application components that are mapped onto the same physical machine or coordinate the execution of different components that

belong to a single application. A component of an application mapped onto a machine is allowed to share the resources with components of other Grid applications and non-Grid applications that may be simultaneously using the machine. It is essential that co-allocated resources for the component tasks of a Grid application be coordinated to maintain similar progress rates of all the components.

1.2 Contribution

This thesis presents *Coordinated Virtual Partition* (CVP), a framework for partitioning and coordinating resources for Grid applications on shared hosting platforms. Following are the two key contributions:

First, CVP implements a middleware-based partitioning of a shared resource to ensure performance isolation among the tasks hosted on that resource. The system is designed to run on top of a general-purpose operating system without modifying the operating system. Therefore, the implementation strategy is more suitable for a Grid environment. Another key benefit of such an implementation is the flexibility of establishing different policies on different partitions without affecting the local policies.

Second, CVP implements a coordination mechanism for the co-allocated partitions of a Grid application. The goal is to ensure that the co-allocated partitions always maintain their capacity in the same proportion such that tasks running on these partitions maintain similar progress rates. The coordination mechanism is integrated with the partitioning middleware. Experiments show that coordination can improve the performance of Grid-enabled MPI programs over wide-area networks.

1.3 Organization

The rest of this thesis organized as follows. In Chapter 2, we introduce a general introduction of an application deployment model in a Grid environment. Chapter 3 presents in detail the design of the CVP mechanism. Chapter 4 discusses the implementation issues and experimental results using the prototype implementation of CVP. Chapter 5 provides an overview of related work. Chapter 6 presents the concluding remarks and future directions of this research.

Chapter 2

Grid Application Deployment Model

Grids provide an infrastructure to bring together diverse geographically distributed resources from different administrative domains to form virtual organizations [FoK01]. These virtual organizations essentially act as hosting platforms where multiple complex Grid applications can be executed simultaneously. Ensuring desired performance for applications requires the provision of high quality of service. The Grid resource management model bears the primary responsibility to deliver the required QoS to applications. Therefore, resource management has to deal with issues that are related to appropriate resource allocation and deployment of applications on heterogeneous resources in a uniform fashion.

In a Grid environment we have three major parties:

- **Resource providers:** Owners of resources.
- **Service providers:** Originators of wide-area applications. They are also clients for resources.
- **End users:** Clients for wide-area services.

Service providers require resources such as processors, memory blocks, disk storage

and network bandwidth from the resource providers. However, because end-users can be geographically dispersed and resource availability can vary, a service may require resources from multiple resource providers. Each resource provider administers the resources according to its own independent policies. Also there is no centralized controller for all the resource providers. As a result, appropriate resource allocation to launch a service is done in a distributed and hierarchical manner. Figure 2.1 shows a simple architecture of a Grid computing environment involving the three major parties. Examples of some of the Grid resource management architectures can be found in [KrB02].

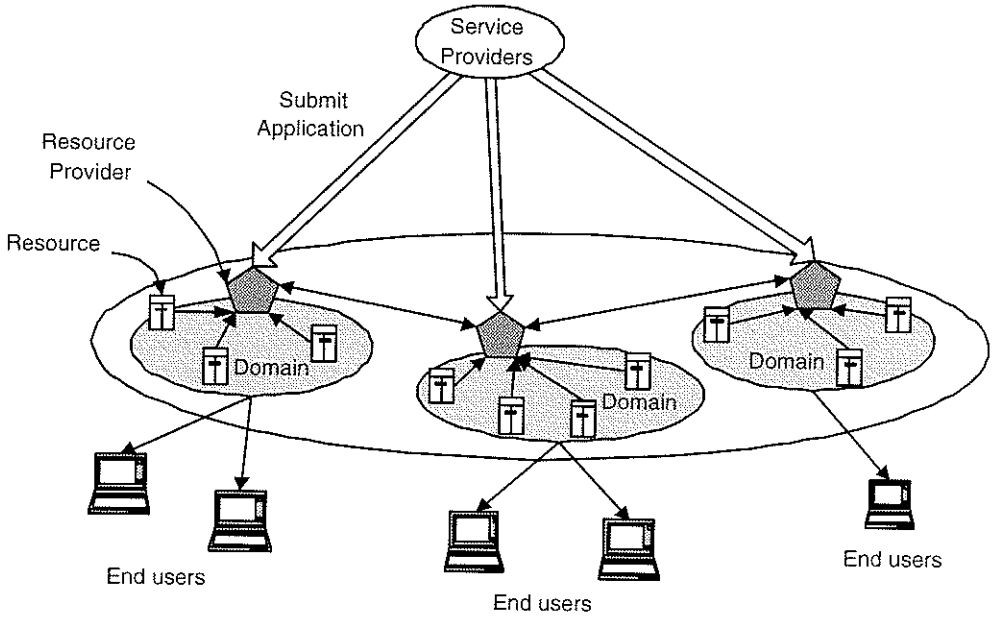

Figure 2.1: A generic architecture of a Grid computing environment.

In general, deployment of Grid applications is achieved through a two-level resource management scheme. In the top level, there is a set of nodes known as *Grid Resource Managers* (GRMs). A Grid resource manager administers a pool of resources that belong to a particular resource provider. A GRM also interacts with other GRMs, establishes resource sharing relationship across resource providers and performs resource discovery. At the bottom level of the Grid resource management model, there are *Local Resource*

Managers (LRM). A LRM operates in association with a GRM. The LRM manages an individual resource and enforces specific policies of the resource provider.

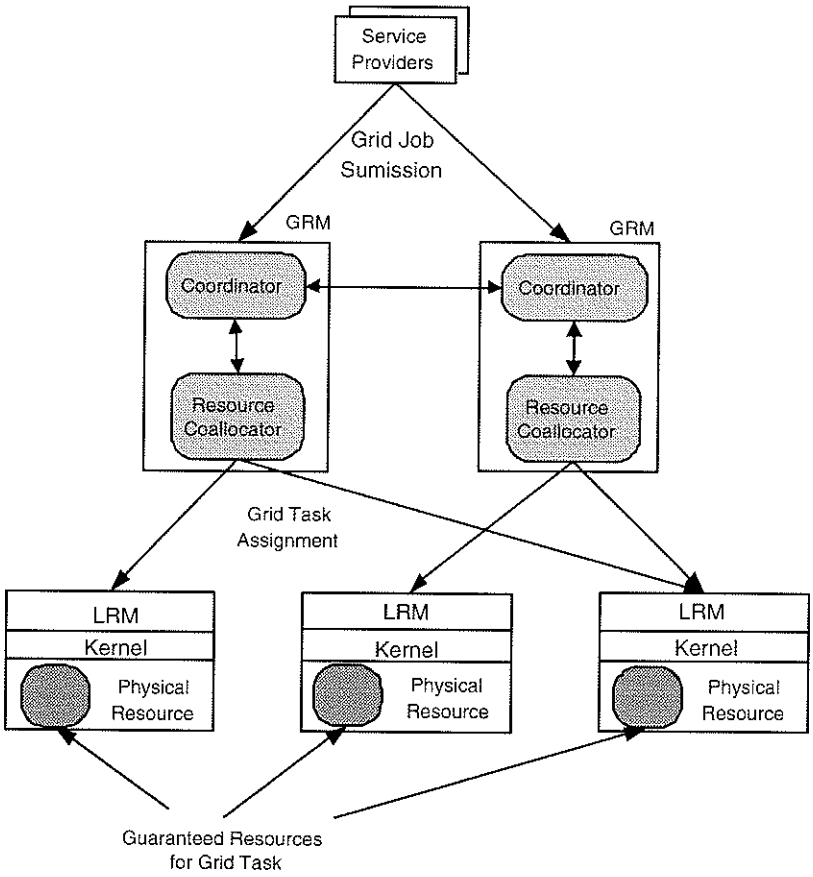

Figure 2.2: A generic application deployment model in a Grid.

Figure 2.2 depicts a generic application deployment model in a Grid. To launch a service, a service provider submits a job request with necessary QoS parameters to the queue of jobs of a GRM. From the QoS parameters, the GRM evaluates appropriate resource requirements (such as CPU, memory, disk storage) for that job. Depending on resource availability, the GRM can select resources from its own resource pool or can coordinate with other GRMs to select resources from multiple resource providers. At the end of selection mechanism comes the actual resource allocation or co-allocation phase. In this phase, a job is mapped to a machine or a set of machines. For a multi-

component Grid job, the component tasks might need simultaneous allocation on different machines. Sometimes advance reservation of resources might precede this allocation or co-allocation phase. Once a Grid job is mapped to the machines, LRMs on those machines will initiate and execute the job after performing necessary authentication and security measures. The LRM is responsible to guarantee the desired QoS for a Grid job. For the purpose of scalability and maximum utilization, local resources usually host multiple tasks. These tasks will compete for resources and will get eventually scheduled by the local OS scheduler. LRM interacts with the local OS scheduler and ensures that Grid tasks get their guaranteed share of resources.

Chapter 3

Coordinated Virtual Partitions

3.1 Overview and Assumptions

To deliver QoS to different competing Grid applications on a shared platform, performance isolation among those applications is a prime necessity. One possible approach to achieve performance isolation is to divide a resource into virtual partitions and assign each partition to a single task. As a result, the resource consumption of a task is bounded by the size of its partition. The size (or capacity) of the partition should be determined in a way such that it conforms to the task's QoS requirement.

In this thesis, we present *Coordinated Virtual Partition* (CVP), a framework for partitioning and coordinating resources for Grid applications on shared hosting platforms. Before we discuss our CVP model in detail, we note the following key assumptions that we have made about the system:

- The shared resource that is partitioned for Grid applications is CPU capacity. Therefore the term "resource partition" means a portion of CPU capacity. In rest of this thesis, the terms "node" and "host" are used interchangeably to refer to the shared local resource.

- A Grid application will have multiple component tasks. Each of these tasks will be assigned to a specific partition on a shared resource.

- The process of determining the necessary size of partitions for Grid tasks and assigning the tasks to different partitions is handled by a Grid resource management system as discussed in Chapter 2. The proposed CVP mechanism is effective only after a task is mapped and initiated on a resource.

- The tasks that are allocated resource partitions are executable processes.

- All nodes are considered to be single-processor machines. Nodes can be connected over a wide-area network.

A Grid application can have multiple components tasks that are mapped onto partitions on different shared hosts. Figure 3.1 illustrates a scenario of such partitions in an example Grid. There are three Grid applications $W1$, $W2$, and $W3$. Components of $W1$ and $W3$ are mapped onto two hosts, while components of $W2$ are mapped onto all three hosts. On each host, a guaranteed partition is created for each of the application components.

Grids are dynamic systems, where new applications can arrive and current applications can depart or become inactive at any given time. This dynamic nature creates variable load on the resources. Whenever new or previously idle applications are mapped onto a shared resource, it consumes resources from the unused resource partition. When Grid applications terminate or become inactive, they free up the partitions they previously consumed. Therefore, the size of the unused partition of a host varies with time. In this thesis, the portion of a resource that is not allocated to any task is referred to as "unused partition" or "idle partition".

When a host is under utilized, as indicated by a non-zero unused partition size,

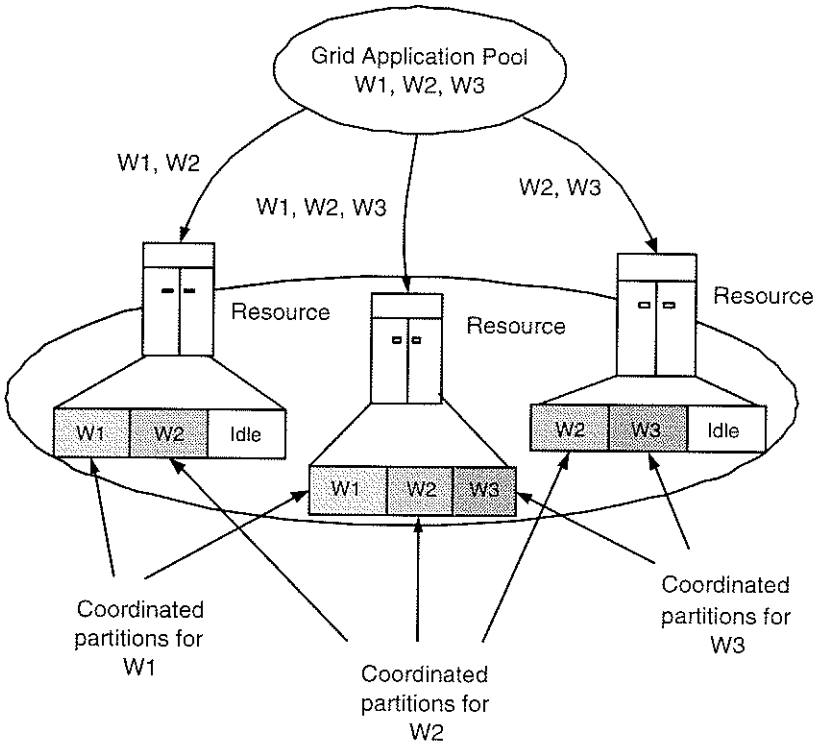

Figure 3.1: Resource partitioning for wide-area applications.

the tasks mapped onto other partitions of that host can exceed their partition sizes and consume resources from the unused partition. The objective of permitting the tasks to use extra capacity beyond their guaranteed partition is to increase application performance and overall resource utilization. However, in case of a multi-component Grid application, a component task has limited freedom in consuming resources from unused partitions.

In this thesis, the executable instance representing a component task of a multi-component Grid application is defined as a *component process*. A component process cannot consume resources beyond its guaranteed partition even if it has the opportunity to use the unused partition, unless all other component processes of the same Grid application have the same opportunity. Otherwise there will be asynchrony in the progress rates of the component processes. In other words, component processes will exhibit dissimilar progress rate in their computation. In Figure 3.1, the leftmost and rightmost

hosts have idle partitions, but the component tasks $W1$, $W2$ and $W3$ mapped on those hosts cannot take up any chunk of the idle partition because the component processes mapped onto the host in the middle do not have idle partition to grow. If all the component processes have room to grow, new temporary partition sizes are computed for each of them such that similar progress rates are maintained. In this thesis the proposed computation method is to scale all the partition sizes by the same scaling factor.

Increase in unused partition sizes means more opportunity for component processes to extend their partitions. However, any increment in a component process' partition size should happen only when sufficient idle partitions are available on those shared hosts, where other components are mapped. When the size of the idle partition decreases at a host, a component process has to relinquish any section of an idle partition it is holding for fairness within that host. As a result, component processes hosted on other nodes have to reduce their usages of idle partitions as well. In rest of this thesis, the task of performing such adjustments in idle partitions to maintain similar progress rate is defined as *coordination.*

Now, I present CVP to achieve two objectives—performance isolation and coordination. Therefore, CVP incorporates two mechanisms:

1. Partitioning - for performance isolation.

2. Coordination - for ensuring similar progress rates for component processes of same Grid application.

The next two sections discuss the functionalities of these two mechanisms.

3.2 Partitioning Model

Once a component process is mapped onto a host, a CPU partitioning mechanism will control the CPU consumption of that process. The partitioning works by controlling the allocation of CPU quantum to every process mapped onto a host. The mechanism employs the *Stride scheduling algorithm* [WaW95], a proportional-share scheduling algorithm, to allocate CPU quanta to the processes. In the stride scheduling algorithm, a process' right for the CPU capacity is represented by the number of *tickets*. A fixed number of tickets represent the total capacity of CPU. Each process holds a certain number of tickets from the total number of tickets in accordance with its guaranteed share of CPU capacity. The scheduling algorithm allocates CPU in a way such that in a given time interval, the percentage of CPU quanta given to a process is equal to the ratio of tickets held by the process and total tickets representing CPU capacity. There are several features that make stride scheduling an attractive choice for CPU partitioning. The most important are the following:

- **Simplicity:** Proportional-share allocation of a resource can be guaranteed by using the ticket distribution in an intuitive way. Also resource share can be varied smoothly with ticket allocation.

- **Deterministic behavior:** Unlike some randomized resource allocation mechanisms such as Lottery Scheduling [WaW94], processes get CPU quantum in a deterministic way. As a result, stride scheduling achieves improved accuracy in throughput rate.

- **Flexibility in resource sharing:** Because tickets represent the right for a resource, complex resource sharing relationship among resource providers can be established in a convenient way. For a Grid environment, flexibility in sharing

resources is a desirable attribute.

Section 5.1.3 presents stride scheduling algorithm with examples. Original implementation of stride scheduling was done by changing the Linux kernel code. However, such an implementation strategy would be in contrary to middleware concept of the CVP mechanism. Therefore, we have implemented stride scheduling at the user-level on a Linux System. Following are the major benefits for the middleware implementation:

Support for Heterogeneous Platforms: Because no modification of the native operating system scheduler is needed, the mechanism can be easily set up on heterogeneous machines.

Modular Resource Management: Different resource management policies can be implemented on different partitions without affecting local policies at the kernel.

In Chapter 4, we discuss the implementation issues of stride scheduler at the middleware level.

3.3 Coordination Model

The function of coordination model is to adjust the partition sizes of the component processes when load on a resource changes. Before describing how the coordination model works, we first show one important behavior of stride scheduling and the way coordination can tackle the effect of load variation.

Consider three processes A, B and C holding 30, 20 and 50 tickets, respectively out of a total of 100 tickets. Therefore, according to stride scheduling A, B and C will get 30%, 20% and 50% of CPU capacity as guaranteed allocation. Let process C leaves or becomes idle. The 50% of CPU capacity that process C was holding now becomes available or idle. The nature of stride scheduling is to divide the idle capacity among the active processes in proportion of their tickets. The ratio of tickets of A and B is

3:2. Therefore, the total CPU allocations of A and B become 60% and 40% respectively. When process C again becomes active, A and B will have to relinquish their share of idle capacity. Such behavior of stride scheduling suggests that in a dynamic environment where processes leave and join time to time, the CPU partitions of the active processes can be of any arbitrary size exceeding the guaranteed amount.

The aim of coordination is to control the partition sizes of the component processes of a Grid application such that similar progress rate is maintained despite the load variation. To achieve this objective, coordination attempts to ensure the resource consumption (partition sizes) of component processes change in same proportion. Let a Grid application A has two component processes $A1$ and $A2$ mapped onto two different machines with guaranteed allocation of $G1\%$ and $G2\%$ of the CPU capacity, respectively. If the amount of idle CPU capacity consumed by these two component processes are $I1\%$ and $I2\%$, then the goal of coordination is to satisfy the following:

$$\frac{I1}{G1} = \frac{I2}{G2} \tag{3.1}$$

3.3.1 Definitions and Architecture

From the set of nodes that host the component processes of a Grid application, one node is selected as the *coordinator*. Due to efficiency considerations, the coordinator is selected as the node having the least communication delay with all nodes hosting other component processes. However, factors such as capacity and load of the nodes may be used instead or in addition to select the coordinator. The introduction of a coordinator obviates the need for explicit communication among the set of nodes. Because a coordinator is local to the set and the set is assumed to be of moderate size, scalability is not a major concern. Some of the functions performed by the coordinator include gathering information about

partition sizes and controlling the partition growths and shrinkages.

Following are some important terms we define to describe the *coordinated virtual partitions* (CVPs).

- *Asynchrony:* is the difference between the amount of CPU quanta a component process actually gets from the middleware scheduling system and the amount of CPU quanta the process should ideally get to maintain similar progress rate with other component processes.

- *Coordination Cycle:* is the time interval after which partitions of component processes of a Grid application get coordinated. Therefore, a component process can accumulate asynchrony within this period. We denote the coordination cycle length as L.

- *Asynchrony Bound:* is the upper bound of asynchrony (A) that a component process can accumulate within a coordination cycle. It is the guarantee provided by our model to a Grid application.

On each node, the middleware scheduling system repeatedly goes through a cycle that can have three phases:

- Monitoring phase

- Communication phase

- Correction. phase

A cycle can have multiple monitoring and communication phases but should always end with a correction phase.

- *Monitoring Phase:* At the start of this phase, the CVP scheduler at a node determines the CPU allocation for the component processes on that node. The CVP scheduler calculates the appropriate CPU allocation based on the global information it receives from the coordinator. A component process is scheduled according to this allocation. The allocation includes the guaranteed partition for the component process and possibly a portion of idle capacity. During monitoring phase, the CVP scheduler monitors for any change in idle CPU capacity. The monitoring phase ends whenever a change is detected. The CVP scheduler then performs some appropriate actions and moves into the communication phase. Following are the actions that the CVP scheduler takes at the end of the monitoring phase:

  - If there is an increase in idle CPU partition, the CVP scheduler restricts a component process' allocation to the initial amount set at the start of the monitoring phase. The CVP scheduler does not allow a component process to take up more CPU capacity from the increased idle partition.
  - If there is a decrease in idle CPU capacity, a component process' CPU allocation can get decreased as it might be using portion of CPU idle capacity. In this case, the CVP scheduler allocate as much idle capacity as possible to the component process so that the initial allocation set at the start of the monitoring phase is maintained. However, if there is not enough idle capacity, asynchrony starts to accumulate. The CVP scheduler keeps track of the accumulated asynchrony. In the event the accumulated asynchrony exceeds the guaranteed asynchrony bound A, the CVP scheduler sets an asynchronized state for the node

After performing the above actions the CVP scheduler moves to the communication phase.

- *Communication Phase:* In this phase, the CVP scheduler at a node sends notification to the coordinator about the changes in the available CPU capacity that occurred during the monitoring phase. The coordinator gathers such notifications from all the nodes that host component processes and periodically sends its decision for coordinated allocation to all those nodes. The actual information that is communicated through these notifications and decisions are discussed in section 3.3.2. The duration of the communication phase is not just the time required for a two-way communication between a node and the coordinator, rather it depends on how often the coordinator sends its decisions to the CVP schedulers at the nodes. While the CVP scheduler at a node is waiting for the decision from the coordinator, it continues to schedule the component process based on it actions taken at the end of monitoring phase. The communication phase ends when a decision is received from the coordinator. Based on that decision, the CVP scheduler readjusts the partition sizes for the component processes. Once the readjustment is complete, the CVP scheduler goes back into the monitoring phase. However, if at least one of the nodes is at asynchronized state then the coordinator asks all the nodes to initiate correction phase. In that case, CVP schedulers on all the nodes move into the correction phase instead of moving to monitoring phase.

- *Correction Phase:* The purpose of this phase is to remove the asynchrony that a component process accumulates during monitoring and communication phases. For the duration of this phase, a component process is assigned a newly calculated resource partition that is required to correct the asynchrony. Component processes with asynchrony get portions of the idle CPU partition while component processes without asynchrony get resources less than their guaranteed allocation. When this phase is over, the CVP schedulers at all the nodes go back to the monitoring phase

with each component process making same portion of overall progress.

3.3.2 Coordination

To make similar progress rates, each component process should know about the progress of all other component processes of the same Grid application. Based on this knowledge, a CVP scheduler at a node can make appropriate allocation for a component process. For a given period, the coordinator gathers notifications from the set of nodes hosting the component processes. At the end of this period, it sends to the nodes the decision it makes based on those notifications. If the coordinator does not receive a notification from a node, it assumes that the last received notification still represents the current CPU allocation of the component process hosted on that node. The notification from a node includes the current CPU allocation (including both the guaranteed partition and the share of the idle partition) and the maximum possible allocation (considering all available idle partition is allocated to a component process) for a component process. The coordinator calculates the average of all current CPU allocations and sets this value as the new coordinated allocation for all the component processes. However, if the average is greater than the maximum possible allocation of some component processes, then the coordinator chooses the minimum of all the maximum allocations as the new coordinated allocation. After computing the value for coordinated allocation, the coordinator sends this value as its decision to all nodes.

To further describe the above coordination technique, we introduce a notion called the *scale up factor*. The scale up factor is defined as the current allocation divided by the guaranteed allocation. We define two terms: $currSF$, which is the current scale up factor from the guaranteed CPU allocation and $maxSF$, which is the maximum scale up factor from the guaranteed CPU allocation. Suppose the guaranteed partition of a component

process is $x\%$ and the idle partition is $y\%$. With a proportional share scheduler, the component process gets $\frac{x}{100-y}$ share of the idle partition. Therefore, the idle partition utilized by the component process is given by:

$$z = \frac{x * y}{100 - y}\%. \tag{3.2}$$

The total CPU allocation for the component process will be

$$(x + \frac{x * y}{100 - y})\%.$$

Then, the $currSF$ is computed as

$$currSF = 1 + \frac{y}{100 - y}.$$

Considering a component process is allowed to use whole of the idle partition, the maximum possible CPU allocation for the process will be $(x + y)\%$. Thus, the $maxSF$ is computed as,

$$maxSF = 1 + \frac{y}{x}.$$

Let there be n number of component processes of a Grid application and notification for component process i be $(currSF_i, maxSF_i)$, where $currSF_i$ is the current scale up factor from guaranteed CPU allocation for component process i and $maxSF_i$ is the maximum possible scale up factor from guaranteed CPU allocation for component process i. Then the coordinated scale up factor, $syncSF$ for all the component processes, is computed as:

$$syncSF = \text{avg}(currSF_i), \tag{3.3}$$

where $1 \leq i \leq n \mid \forall i \ \text{avg}(currSF_i) \leq maxSF_i$ or

$$syncSF = \min(maxSF_i), \tag{3.4}$$

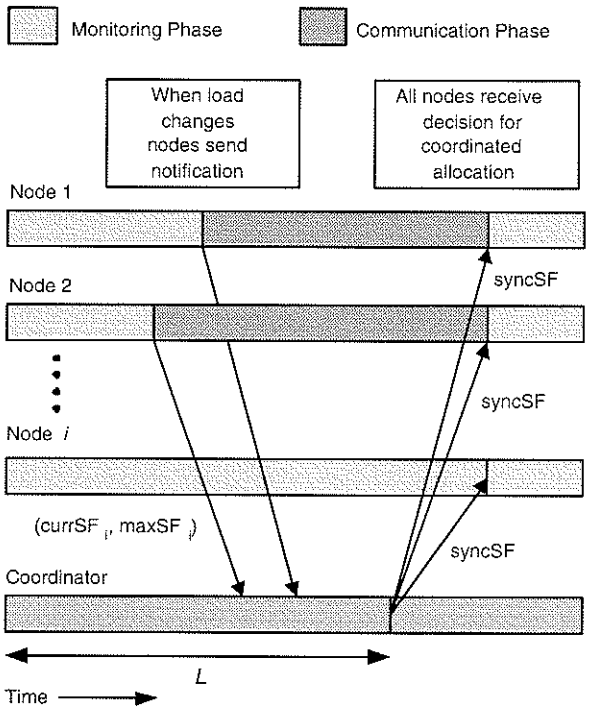

Figure 3.2: Coordination mechanism

where $1 \leq i \leq n \mid \exists i$ avg$(currSF_i) > maxSF_i$.

The coordinator performs the above calculation for $syncSF$ and sends this value to all nodes hosting the component processes. Thus, $syncSF$ defines the coordinated partition for all component processes. In fact, this is the decision sent to all the nodes from the coordinator. Figure 3.2 shows how nodes communicate their $(currSF_i, maxSF_i)$ and the coordinator responds with $syncSF$. Those nodes that did not send notifications also gets $syncSF$ from the coordinator to readjust their partition size to remain coordinated with other nodes.

3.3.3 Computing Asynchrony

During the monitoring phase, a component process will have certain $syncSF$ and $maxSF$ values. With a decreasing idle partition size, the value of $maxSF$ reduces. When value

of $maxSF$ becomes less than $syncSF$, the component process is unable to maintain proportionate progress with other component processes. As a result, asynchrony starts to accumulate and continues to grow until a response from the coordinator is received. Let t be the time in seconds between onset of the asynchrony condition and the corresponding reception of a response from the coordinator on redefining the partition. Further, let $x\%$ be the guaranteed CPU allocation of that component process and q be the length of a CPU quantum. Then, asynchrony a is calculated as

$$a = (syncSF - maxSF) * \frac{x}{100} * \frac{t}{q} \tag{3.5}$$

The CVP scheduler at a node calculates a for a component process and checks whether it exceeds A.

From Equation (3.5), it can be observed that asynchrony grows with t, the time to get coordinator's response. If the period L, by which the coordinator turns around the decisions is reduced, t as well as a are also reduced. Consequently, reducing L will control growth of asynchrony. However, reducing L will result in more load on the coordinator and more communication overhead.

3.3.4 Correcting Asynchrony

When the coordinator receives notification about an asynchronized state of a component process from a node, it instructs all the nodes to initiate the correction phase. The coordinator also calculates and sends back the length of the correction phase T_{corr}. Figure 3.3 shows this process. During the correction phase, component processes without asynchrony reduce their partition sizes by a *slow down factor* (SDF) to allow the asynchronized process to remove its asynchrony. Usually, SDF is chosen in [0,1]. Suppose the process in asynchronized state has a current scale up factor of $currSF_a$, guaranteed CPU allocation of $x\%$, and overall coordinated scale up factor of $syncSF$, then the length of

the correction phase is given by:

$$T_{corr} = \frac{A * q * 100}{(currSF_a - SDF * syncSF) * x} \tag{3.6}$$

where A is the asynchrony bound defined in Section 3.3.1. T_{corr} in equation 3.6 is the time that an asynchronized process needs to make up for A by using up more CPU capacity at a factor $(currSF_a - SDF * syncSF)$ than the other component processes.

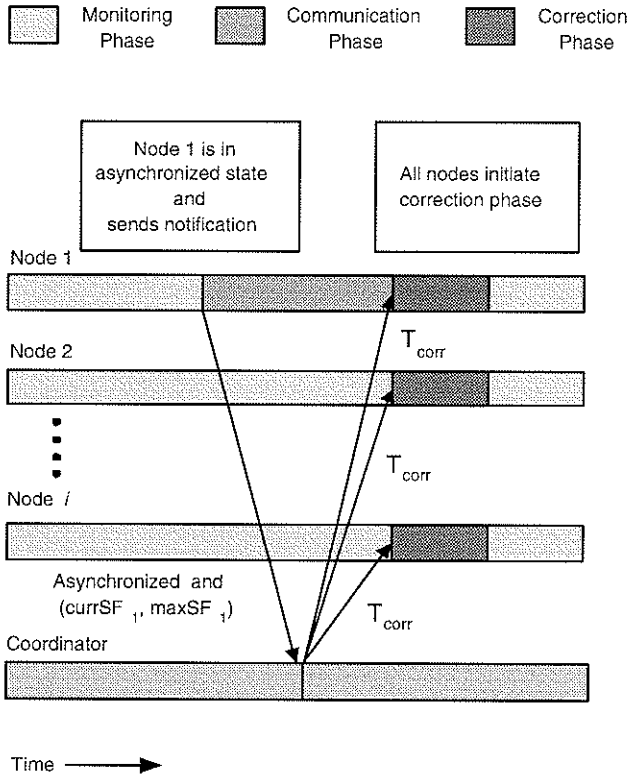

Figure 3.3: Initiation of Correction phase

3.3.5 Estimating Granularity

To understand the notion of granularity and to determine its value, we look at the scenario of asynchrony occurrence and its subsequent correction. When accumulated asynchrony of a component process exceeds the asynchrony bound, the node hosting that process

informs the coordinator who immediately sends back the required corrective decision. However, during this communication, the component process might accumulate more asynchrony. This increment in asynchrony is the *granularity* of our coordination model. We argue that our model can provide an asynchrony bound that can vary due to the value of granularity. That is, if the proposed asynchrony bound is A and granularity is g, then the experienced asynchrony bound that is applied on the component processes is given by, e, where $e = A + g$.

If component processes need to be highly coordinated, then we want the value A as low as possible. Taking the minimum value of A to be zero, we can say that the experienced asynchrony, e equals g. Therefore, g is the minimum asynchrony level that component processes in our model can be provided. This provides an alternate definition of granularity (i.e., *granularity* can be considered as the lower bound of the asynchrony).

For calculating g, let m be the CPU allocation (in percentage of whole CPU capacity) of a process, p be the CPU idle partition (in percentage of whole CPU capacity), and T be the difference between the time asynchrony exceeded A and the time corrective action was initiated by the coordinator.

A primary source for the accumulation of asynchrony during T is the receding of the idle partitions that was provided to the component process previously. To determine this impact consider a scenario where we have idle partitions going from their nominal size of p to 0 (i.e., no idle partitions at all). Due to the disappearance of the idle partition, the CPU allocation of a process decreases by its share of idle partition. Using Equation 3.2 we can measure the size of the share. The reduction of this share of CPU capacity determines the value of g. Therefore,

$$g = \frac{T}{q} * \frac{p * m}{(100 - p) * 100} \tag{3.7}$$

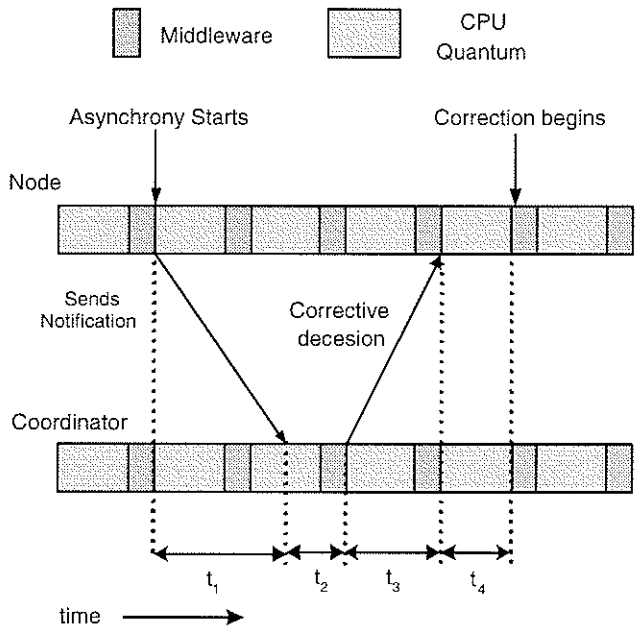

Figure 3.4: The time region that affects granularity.

From Figure 3.4, we can estimate T. It is the summation of communication delay $t_1 + t_3$, time for the coordinator process to get its next CPU quantum t_2, and time for the CVP scheduler at a node to get its next CPU quantum t_4.

$$T = t_1 + t_2 + t_3 + t_4 \tag{3.8}$$

The maximum value of t_2 and t_4 can be q. Because we are calculating granularity we can set t_2 and t_4 at their maximum value of q and take $t_1 + t_3$ as C, the communication delay.

$$T = C + 2q \tag{3.9}$$

Therefore, from Equations (3.7) and (3.9)

$$g = \frac{(C + 2q)}{q} * \frac{p * m}{(100 - p) * 100} \tag{3.10}$$

3.3.6 Handling Stale Information

Stale information about the idle CPU capacity of a node can pose a major problem in the coordination scheme proposed in Section 3.3.2. When monitoring phase at node i detects a change in available CPU capacity, CVP scheduler sends $(currSF_i, maxSF_i)$ values to the coordinator. The coordinator calculates the value of $syncSF$ by using these values. However, by the time the node i receives $syncSF$ from the coordinator, the available CPU capacity of node i may change. So it has different values of $(currSF_i, maxSF_i)$ from the values sent to the coordinator. In the worst case, the available CPU capacity decreases and the new value of $maxSF_i$ can be less than $syncSF$. In that case, the node i would not have enough available CPU capacity to make coordinated allocation for the component process hosted on it.

A prediction mechanism can be useful to address the problem of stale information. Each time the available CPU capacity changes at node i, it has new values of $(currSF_i, maxSF_i)$. The prediction mechanism keeps track of the past values. Whenever a change in available CPU capacity occurs, based on those past values the CVP scheduler predicts the value of $currSF_i$ and $maxSF_i$ for the next change. Let the predicted values be $predCurrSF_i$ and $predMaxSF_i$. If $predCurrSF_i \leq currSF_i$ then CVP scheduler sends $(predCurrSF_i, predMaxSF_i)$ as its notification, otherwise it sends $(currSF_i, maxSF_i)$. The idea is to avoid a situation where current CPU availability at a node leads the coordinator to decide a large CPU allocation although the availability of CPU decreases in immediate future. Therefore, if prediction is worse than the present availability, predicted values are used to calculate the coordinated allocation.

Chapter 4

Prototype Implementation and Performance Evaluation

The novel feature of CVP is that it is implemented as middleware and runs entirely in the user-space without any modifications of the kernel. This chapter discusses the implementation of the middleware and presents experimental results showing the behavior and the performance of the model.

4.1 CVP Implementation

4.1.1 User-level Process Scheduling

We developed the prototype middleware using C++ on top of RedHat 7.3 Linux kernel. The essence of our user-level CPU scheduling mechanism lies in the manipulation of priorities of processes. The underlying operating system (in our case, the Linux kernel) provides the following two supports: (1) interval timer, (2) fixed priorities of processes. The manipulation of priority is done in a way to ensure that kernel scheduler selects

particular process for the next CPU quantum.

The CVP scheduler is a user-level Linux process that runs with root privilege to enable it to manipulate the priority of the processes. Also it has the highest priority in the system so that it can acquire the CPU whenever it is runnable. The middleware has its own run-queue that contains a set of runnable process. These processes might be component processes of Grid applications or any local process requiring QoS. Initially these processes to be scheduled by the CVP scheduler are loaded with a "loader" module. After loading a process, the module sends CVP scheduler the pid and CPU requirement of that process. We implemented the communication between the loader and CVP scheduler through standard Linux message passing scheme using system calls `msgsnd()` and `msgrcv()`. The CVP scheduler gets the pid of the process, inserts it in its own run-queue and controls its CPU allocation.

To schedule a CPU quantum, the CVP scheduler selects a process from a set of runnable processes according to stride scheduling. The CVP scheduler then makes the priority of that selected process the second-highest priority. On Linux, we used the system call `sched_setscheduler()` to change the priority. The middleware then starts an interval timer having duration same as the length of a CPU quantum. We set the length of the CPU quantum to be 10 milliseconds. We used the system call `setitimer()` to setup the timer duration. After starting the timer CVP scheduler sleeps. At that point, the selected process which has the second highest priority automatically gets dispatched by the kernel scheduler to get the CPU. When the timer expires the CVP scheduler wakes up. Because it has the highest priority it takes the CPU away from the running process. The CVP scheduler then lowers the priority of that process to the lowest in the system. This scheduling cycle repeats for all CPU quanta.

A *Self-blocking Condition* can happen when the process that is dispatched to get CPU quantum blocks on some input data or semaphore. For example, such self-blocking

occurs when a process has to block on a read call from a network socket for data not arriving on time. In our implementation, CVP scheduler does not have knowledge of an application's self-blocking condition. Therefore if a process gets self-blocked in the middle of a CPU quantum, the kernel scheduler will select may any other runnable process on the system. If the self-blocked process becomes runnable before the end of the CPU quantum, it will preempt the running process and runs for the remainder of CPU quantum. However, CVP scheduler will record that the process has used the whole CPU quantum. The CVP scheduler assumes that it is the responsibility of the application developers to ensure that processes be runnable throughout the whole CPU quantum. At present, our implementation of CVP requires that applications know the length of CPU quantum prior to their deployment for avoiding self-blocking conditions.

4.1.2 Implementation of Stride Scheduling

Creating appropriate size partitions by using stride scheduling is quite straightforward. The total capacity of CPU is represented by 100 tickets. If the required guaranteed partition of a process is $x\%$ of the CPU capacity then it is assigned basic x tickets out of those total 100 tickets. Besides the guaranteed partition, a process will have partition from the unused capacity if there exists any. Basic stride scheduling enables a process to get share of unused capacity in proportion of its tickets. However, to satisfy the coordination requirements, a component process may have to use an arbitrary portion of the idle partition. We accommodate such arbitrary partitioning of idle capacity in stride scheduling by dynamically modifying the tickets held by the processes. Whenever there is a need of readjusting the size of the unused partition that a process uses, the CVP scheduler just changes the ticket of that process accordingly. The effect of changing the ticket will automatically be reflected on allocation of CPU quanta.

To understand how we modify the tickets, let, from basic stride scheduling, a process will get $y\%$ of CPU capacity from unused partition in addition to its guaranteed partition of $x\%$. We set the ticket of that process to $(x + y)$. If a readjustment requires that usage of idle partition to be changed from $y\%$ to $z\%$, then the number of tickets is also modified to $(x + z)$. If $y > z$, then $(y - z)$ tickets are equally distributed among remaining processes. If $y < z$, then $(z - y)$ tickets are gathered from rest of the processes. However, at a maximum, a component process is permitted to take away only the amount of tickets that defines other processes' idle partition. The value $maxSF$, as discussed in Section 3.3.2, determines this amount of tickets. In our current implementation, we considered that there is single component process per machine and rest of the processes are local processes. Because only the idle partitions used by the local processes are taken away, and not their guaranteed partitions, the QoS requirements of these processes are not compromised.

4.1.3 Implementation of Prediction

We have added an improvement to the coordination algorithm by using prediction at the monitoring phase. The CVP scheduler at a node keeps a history of the changes in the idle partition size. This history information is used to predict future changes in idle partition sizes and these predictions are sent to the coordinator as well. With the predicted values, the coordinator can be informed of future changes with high probability and this improves the overall performance of the coordinator. As a result, growth of asynchrony is controlled. For the on-line prediction mechanism, we have chosen to use autoregressive AR(1) [BoJ76] model, which is simple and has acceptable error limit [ChG02]. In AR(1) model a sample value is predicted as,

$$\alpha_{i+1} = \bar{\alpha} + R(1) * (\alpha_i - \bar{\alpha}) + e \tag{4.1}$$

where $\bar{\alpha}$ and $R(1)$ are mean and autocorrelation of the history values $\alpha_1, \alpha_2, ..., \alpha_N$. e is the white noise component. We assume e to be 0. The autocorrelation $R(l)$ is calculated as,

$$R(l) = \frac{\sum (\alpha_i - \bar{\alpha}) * (\alpha_{i+l} - \bar{\alpha})}{(N-1) * \sigma^2} \tag{4.2}$$

where $1 \leq i \leq N-1$ and σ is the standard deviation of the history values.

4.2 Experiments

In all the experiments we used machines having identical hardware and software configurations: Pentium-III 550MHz, 256MB machines each running Linux RedHat 7.3. Although these machines were connected with high-speed links, we emulated a wide-area network by introducing delays in the IP packets using the `iptables` tool. In this section, we present results that validate the desired performance of our model. We tested our model under different system parameters to examine its behavior. Also, a parallel program written in MPI was executed on top of our prototype to demonstrate its support for parallel programs.

4.2.1 Partitioning under static conditions

To validate the ability of our model to partition the CPU capacity, we ran the prototype CVP scheduler on two machines. Then, we loaded one component process on each of the two machines. The component processes were identical programs each performing floating-point arithmetic. The experiment was under two different setups. In the first

setup, both component processes were given a guaranteed 40% of CPU. We measured the progress of the component processes by counting the number of floating-point operations performed within certain lengths of time. Figure 4.1 shows the progress of the two component processes under a static load condition (where the background load remains constant). We found that both processes performed almost the same number of MFLOPs within the same length of time. In the second setup, one process was allocated 20% of CPU on one machine and the other process was allocated 40% CPU on the other machine. In this case, as Figure 4.2 shows, we found that the process with more CPU always did twice as much operation as the other within the same length of time. These results demonstrate the partitioning ability of our model.

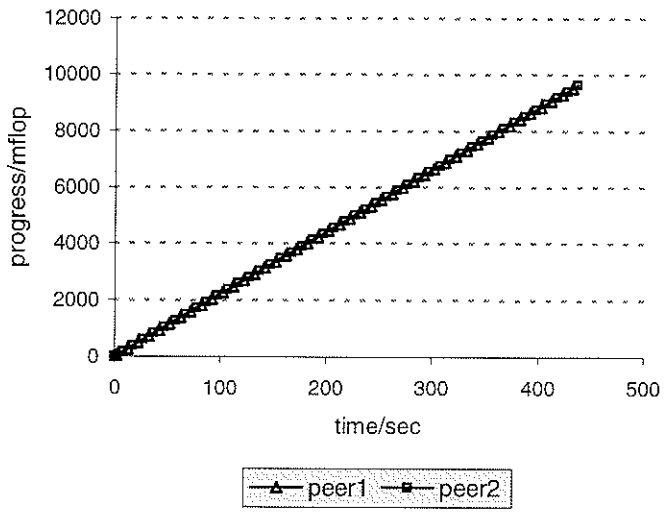

Figure 4.1: Guaranteed partition for two component processes at allocation ratio 1:1.

4.2.2 Maintaining coordinated allocation with asynchrony bound

This experiment showed the ability of our model to coordinate the co-allocated partitions within a guaranteed asynchrony bound. In this experiment, we used four machines. One component process was loaded onto all of the machines. Each component process had

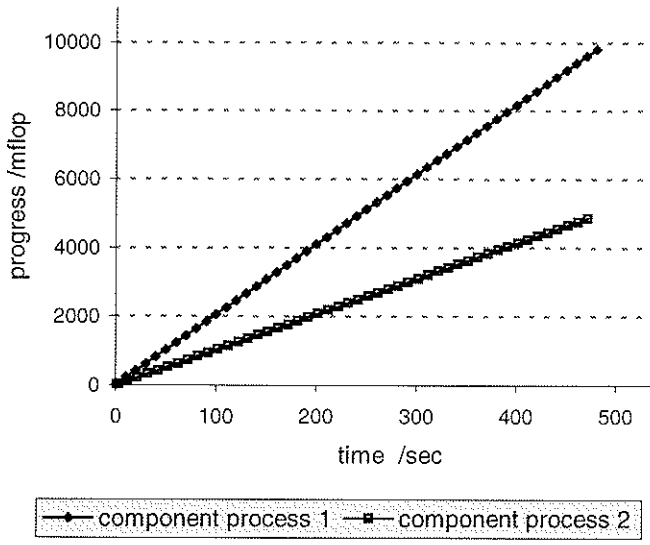

Figure 4.2: Guaranteed partition for two component processes at allocation ratio 2:1.

a 40% guaranteed allocation. We kept the load on machine 1 fixed and changed the load on all other machines within the range of 0–40% CPU. In Figure 4.3, we show the variation of asynchrony on machines 2, 3, and 4 with time with correction turned on. The asynchrony bound A was set at 10 CPU quantum for this experimentation. From all the three curves we found that asynchrony grew in a stepwise fashion. Because after each monitoring and communication phases, our model set a new coordinated allocation that stabilized the growth of asynchrony of a component process. As load was changed further, asynchrony rose again with yet another phase of resulting synchronization. This pattern continued until asynchrony on any of the node reached the upper bound, A. In that case, corrective action took place on all the nodes and asynchrony was sharply reduced. Figure 4.3 reflects this behavior.

4.2.3 Effect of dynamic variation of load on asynchrony

In this experiment, we were interested in determining how asynchrony changes with load variation at the hosting nodes. We used two machines same as Experiment 1. The idle

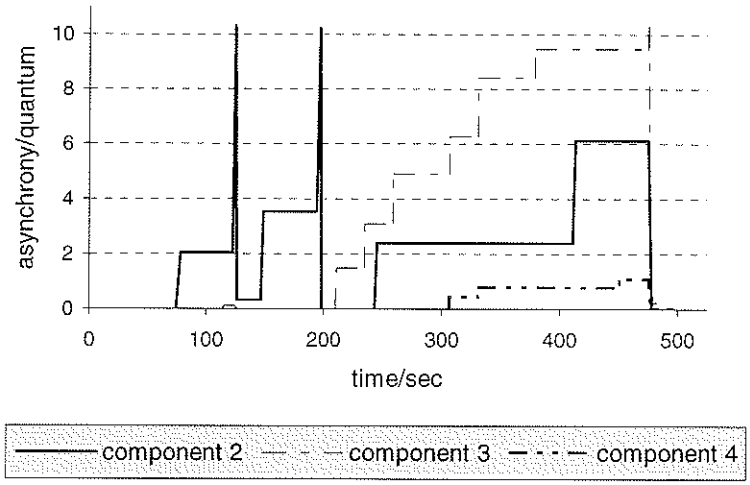

Figure 4.3: Maintaining coordinated allocation.

partition of machine 1 was maintained fixed at 60% of CPU while the idle partition of machine 2 was varied in a uniform range where the average value of the range was fixed at 40%. The uniform range had different spreads ranging from 20%, 25%, and 30% on both sides of the average. For example, with a 20% spread, we can have values from the range [20%, 60%] for the idle partition size. Figure 4.4 suggests that asynchrony rises more quickly as the load variation occurs in a larger range. The asynchrony curve is much flatter when load varies in a smaller range. This behavior can be explained by noting that larger load variations create opportunities for a component process to grab larger amount of idle capacity. However, when idle capacity decreases, the process has to relinquish that large share. Therefore, the process is likely to incur large asynchrony.

4.2.4 Effect of coordination cycle length

In this experiment, we changed the length for the coordination cycle, L, and studied its impact on the growth of asynchrony. Again, we used two machines same as Experiment 1. For the experiment setup, we maintained fixed load on machine 1 while varying

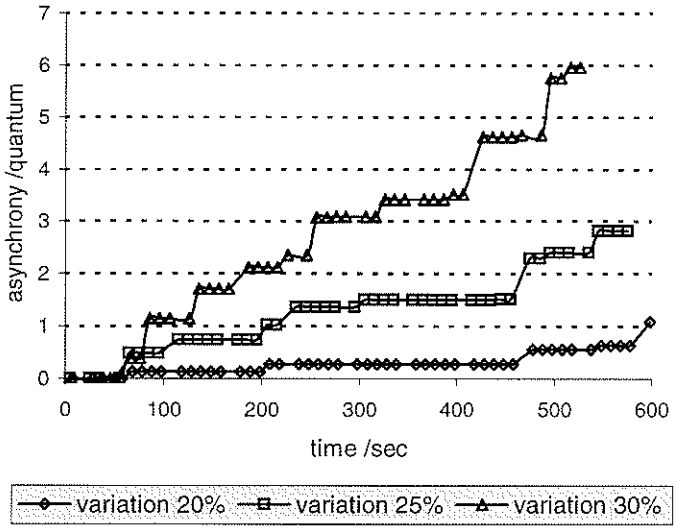

Figure 4.4: Effect of load variation on asynchrony.

idle capacity on machine 2 within a uniform range with an average 40%. Figure 4.5 shows the variation of asynchrony with time for three values of L. We found that for smaller values of L, the coordinator passed its decisions to the component processes more frequently and as a result, growth of asynchrony could be checked more quickly. This result essentially validates the Equation (3.5), where we measured asynchrony by multiplying the asynchronized allocation with time t, which is the time required to get a decision from the coordinator. With increasing L, it took more time to get a decision back from the coordinator. Hence, larger L lead to steep asynchrony growths.

4.2.5 Effect of prediction

This experiment examined the effect of using prediction to estimate the idle partition size at a node. Although the coordinator could use the current value of the idle partition as an estimate, using the predicted value can improve the performance. For example, current value of the idle partition sizes might indicate that a scale up of the allocated partition sizes is possible. However, due to future arrivals, the idle partition sizes can shrink

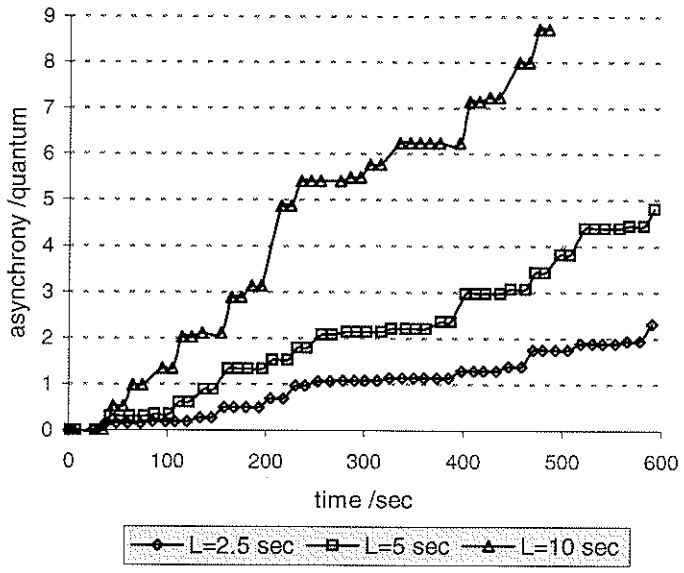

Figure 4.5: Effect of coordinator cycle length on asynchrony.

and scaling up the allocations may not be possible. Using prediction can help in these scenarios. We used the same setup as the previous experiment and ran the CVP scheduler with and without the predictor module. Figure 4.6 shows the variation of asynchrony with time with and without prediction. The prediction itself was performed using an AR(1) predictor. This experiment showed that prediction can reduce the asynchrony growth rate.

4.2.6 Effect of idle partition on granularity

In our model, granularity is a parameter that varies with idle partition size. This experiment was performed to verify the relationship between granularity and the idle partition size. We used the same setup as in Experiment 4. Figure 4.7 shows the results of the experiment. A point along the X-axis defines the maximum value of the range within which idle partition can vary. In accordance with the derivations shown in Equation (3.10), the results in Figure 4.7 show that as the idle partition size increases the granu-

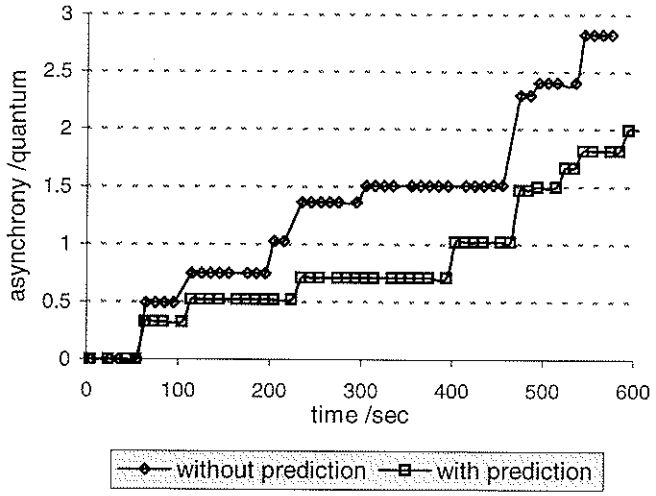

Figure 4.6: Effect of prediction on asynchrony.

larity increases. Granularity can take fractional values as shown in Figure 4.7 because, although a process might loose an integer number of quanta during a single asynchronized state, the average number of quanta that a process looses at different asynchronized state can be a fraction.

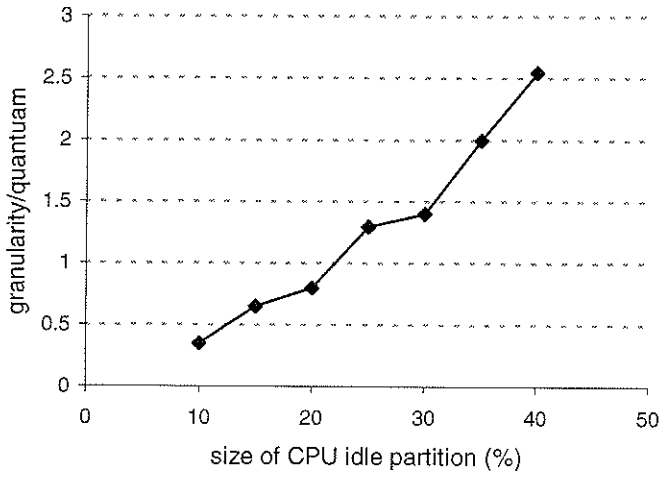

Figure 4.7: Effect of idle partition size on granularity.

4.2.7 Effect of CPU allocation of a process on granularity

In this experiment, we examined the variation of granularity with guaranteed CPU allocation of a component process. As the guaranteed CPU allocation increases, a process' weight increases. Consequently, it acquires a larger fraction of the idle partition. Therefore, with increasing guaranteed CPU allocations, the granularity increases. On the other hand, with smaller guaranteed CPU allocations, asynchrony can be controlled more precisely because the granularity decreases as well. This can be observed from the experimental results shown in Figure 4.8.

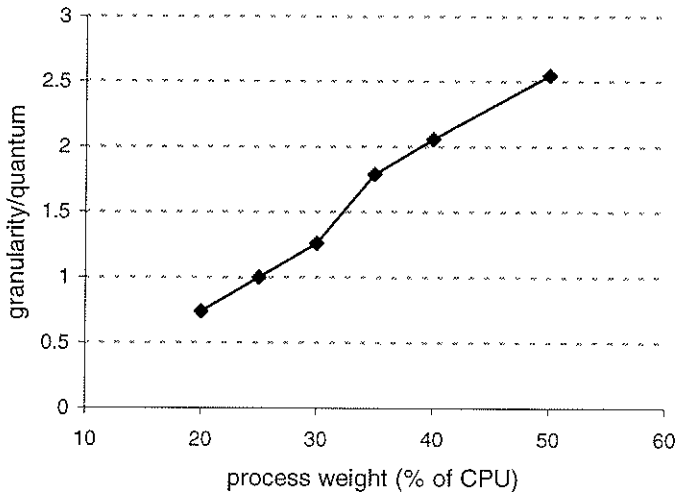

Figure 4.8: Effect of CPU allocation on granularity.

4.2.8 Effect of communication delay on granularity

In this experiment, we examined the variation of the granularity with the communication delay. Because our experimental setup involved two machines in a local area network, we used a delay simulator to introduce the latencies packets would experience in wide-area networks. To vary the IP packet delay between machines, we used the `iptables` tool

and the `libipq` library. We ran two component processes on the two machines each with 40% guaranteed CPU allocation. The load on one machine was held fixed while the load on the other machine was varied between 30% and 60% of the CPU. We varied the communication delay in the range 20ms to 200ms, which is typical of the delays encountered in wide-area networks. The results indicate that granularity increases as the communication delay increases. This is intuitive because with low delay, the coordinator can send its decisions quickly to the nodes. Therefore, nodes would have to spend less time to accumulate asynchrony. As a result, more fine-grain coordination is possible. We can also find from figure 4.9 that the growth of granularity is slower relative to the increase in communication delay.

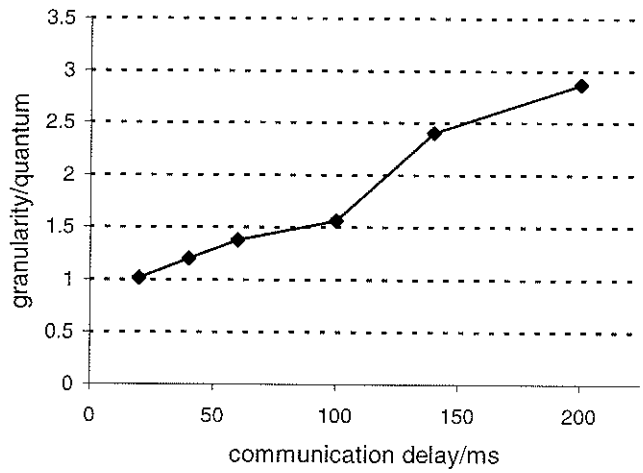

Figure 4.9: Effect of communication delay on granularity.

4.2.9 Overhead of the middleware

In this experiment, we wanted to determine the overhead caused by our CVP scheduler. We measured the overhead in terms of the CPU capacity the middleware consumed. The overhead is caused by two major functions: context-switch time and processing

time. Because the middleware schedules for a fixed CPU quantum, it introduces a fixed context switch overhead. To determine the context-switch overhead, we ran the CVP scheduler without any component processes. On a Pentium-III 550 MHz machine, we measured overhead to be 0.06% of CPU time. Next, we ran the CVP scheduler with a number of component processes. In this case, the CVP scheduler took 0.263 ms on the average to schedule a quantum of length 10 ms. This implies a processing overhead of 2.63% of CPU time.

4.2.10 Support for MPI programs

The experiments up to now have focused on characterizing the CVP mechanism. The goal of this experiment was to show how parallel programs running on shared hosts in a Grid environment can benefit from our CVP mechanism. We used four machines and a wide-area network was emulated by introducing delays at the IP level. We used a MPI program written in *single program multiple data* (SPMD) model that performs image correlation on 4MB image data [ArM98]. This program uses globally synchronizing operations to scatter the image data at the beginning and gather the results at the end of the execution. For our tests, we loaded two of the four machines with background workload such that the MPI component process got 20% of the CPU. We measured two parameters: total execution and barrier wait times. The barrier operations were in the code just before the global scatter and gather operations. The results of execution without and with the CVP scheduler are presented in Table 4.2.10.

	Without CVP	**With CVP**
Avg. Barrier Wait Time	105.66 sec	28 sec
Max. Barrier Wait Time	148 sec	55 sec
Computation Time	88.75 sec	57.75 sec

These results indicate that CVP can improve the performance of MPI programs when they are executed over wide-area environments even if the programs use synchronizing operations sparingly.

Chapter 5

Related Work

This chapter discusses a number of research related to CVP scheduler. We examine research work from CPU scheduling, process synchronization and grid resource management. Section 5.1 explores various CPU scheduling algorithms, Section 5.2 looks at some resource partitioning methods, Section 5.3 presents concepts of different process synchronization approaches and Section 5.4 discusses the toolkits and models for resource management and application development on Grid.

There have been significant amount of research addressing the problem of CPU partitioning in a fair and proportional way. However, there are fewer projects that examine resource coordination across machines over wide area. Therefore, we compare general process synchronization approaches from the literature to give us insights to develop a coordination model over the Grid.

5.1 CPU Partitioning Approaches

5.1.1 Start-time Fair Queuing

Start-time Fair Queuing (SFQ) [GoG96] proposes a hierarchical CPU partitioning algorithm. It partitions the CPU capacity among different application classes in a tree like structure where each application class can further partition its capacity into sub-classes. Each node in the tree represent an application class. The rood node schedules its child nodes and child nodes further schedule their child nodes. At the bottom of tree, the leaf nodes schedule the actual processes or threads. Different schedulers can be employed for different application class (node) as shown in Figure 5.1. Each class gets CPU capacity from its parent class in proportion of its weight. It achieves fairness by scheduling processes based on their weights and start time, the time from which they become runnable. Let in any interval $[t_1, t_2]$, two threads i and j are runnable and their weights are r_i and r_j respectively. If $W_i(t_1, t_2)$ and $W_j(t_1, t_2)$ are the number of instructions executed by i and j respectively, then the fairness criteria $|\frac{W_i(t_1,t_2)}{r_i} - \frac{W_j(t_1,t_2)}{r_j}|$. SFQ provides a near-optimal upper-bound for this fairness criteria. SFQ algorithm is applicable for single processor nodes. Implementation of this algorithm requires kernel modification. A SFQ-based Linux kernel called QLinux has also been developed.

5.1.2 Lottery Scheduling

Lottery Scheduling [WaW94] is a randomized scheduling algorithm for flexible and efficient resource management. The idea behind this algorithm is very simple. Each of the competing processes for CPU holds a certain number of tickets. The tickets assigned to any process are numbered sequentially. The algorithm selects a ticket randomly and owner of that ticket will be winner to get the next CPU quantum. Because each ticket

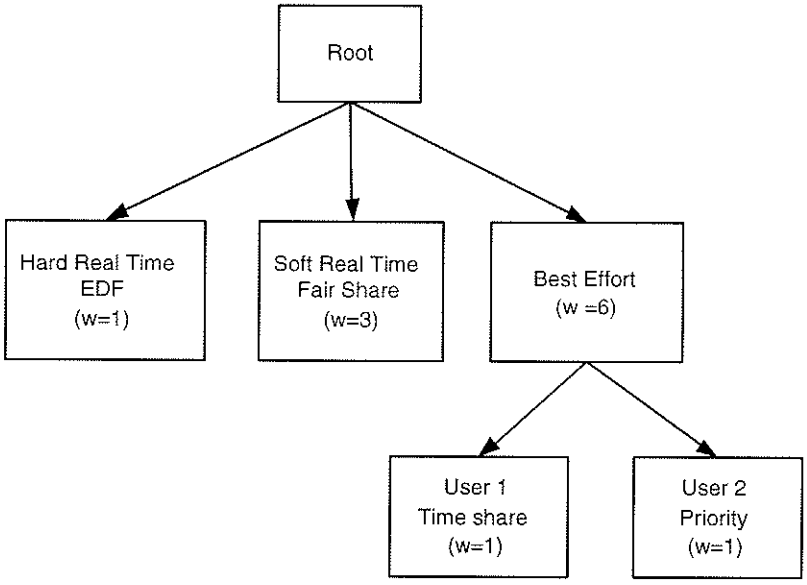

Figure 5.1: Hierarchical partitioning of CPU capacity by start-time fair queuing

has equal probability of getting selected, the probability that a particular process will be winner is directly proportional to the number of tickets that it holds. It introduces a *currency* abstraction where value of tickets can vary with demand for resources. The currency abstraction provides a simple, flexible, and modular way of resource management. However, because of its non-deterministic nature, when the scheduling period is short, the algorithm may exhibit large scheduling inaccuracy. Also the random allocation technique results in processes experiencing variable response times.

5.1.3 Stride Scheduling

Stride Scheduling [WaW95] is a deterministic algorithm that maintains the same simple and flexible approach of Lottery scheduling. In this algorithm, each process has a value called *stride*, which is a time interval for which the process to wait between successive allocations. The value of stride is calculated by taking the inverse of the number of tickets that a process hold. Proportional partitioning is achieved because with more tickets a

process will have less value of stride and, therefore, will more frequently get the CPU. For example, a process with twice as much tickets than another process, will have half the stride value and will get CPU twice frequently. Each process is also associated with another variable called *pass* which is initially set as equal to stride. Each time a process gets a CPU quantum its pass value is increased by stride. The scheduling decision for a CPU quantum works by selecting the process which has minimum pass value. Figure 5.2 shows CPU allocation using stride scheduling among three processes with stride ratio 3:2:1.

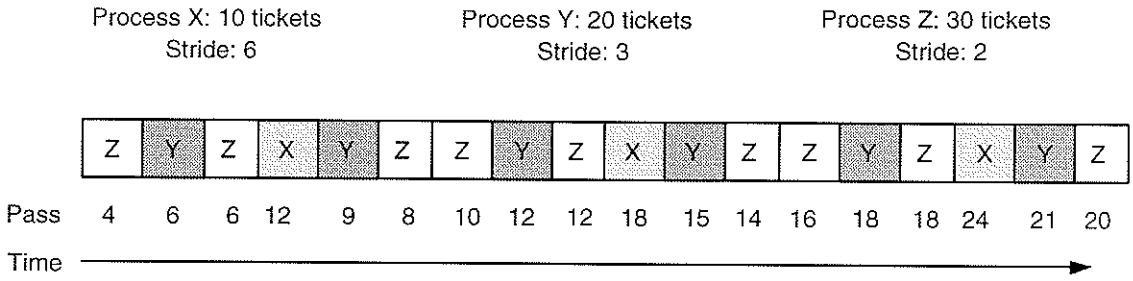

Figure 5.2: Basic Stride Scheduling

An extension to the basic stride scheduling implements support for a dynamic environment where processes can leave and join the system. It introduces two new variables: *global_tickets*, which is the sum of all the tickets, *global_stride*, inverse of global_tickets and *global_pass*, which is incremented by global_stride for every CPU quantum allocation. Whenever a process leaves (becomes idle), the difference between its pass value and global_pass value is recorded. Later when the process joins then it pass value is calculated by adding the difference to current value of global_pass. Figure 5.3 shows an example stride scheduling for three processes X, Y and Z where Z becomes inactive for some time. During the absence of Z, CPU capacity is divided between X and Y in proportion of their tickets. From the point Z becomes active, CPU capacity is again divided according to the ticket ratio of all three processes. The idea is to give a process

a fair allocation only during the time it remains active. In our implementation of CVP scheduler, we used this version of stride scheduling.

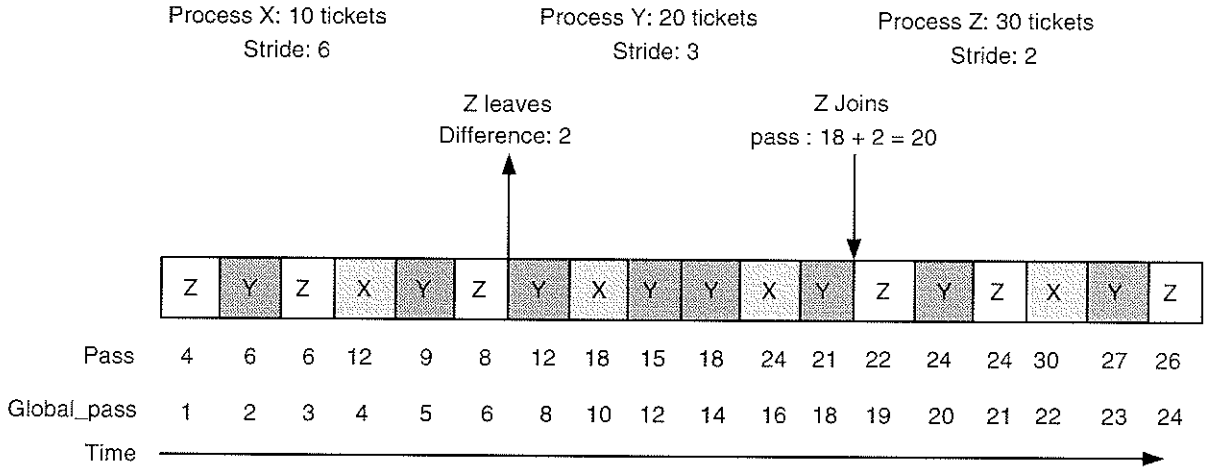

Figure 5.3: Stride Scheduling with support for dynamic client leave and join

Stride scheduling guarantees that the absolute allocation error for a given pair of processes is never more than one quantum [Wal95]. Other examples of proportional-share scheduler include *Surplus Fair Scheduling* [ChA00] which has been developed for symmetric multiprocessors. However, it relies on kernel level modifications.

5.1.4 Dynamic Soft Real Time Scheduler

Dynamic Soft Real Time Scheduler (DSRT) [Chu99] is an interesting work for implementing CPU partitions. It partitions the CPU among different service classes based on processor usage pattern. Applications, specially multimedia applications, may require constant or variable CPU usage time over a fixed or variable length period. DSRT extracts the processor usage patterns of different processes to determine their class and parameters. Based on these values, advance reservation of specific amount of CPU resource is made for a process. DSRT also monitors application's execution and tried to adapt to any mismatch between allocation and actual usage pattern. It divides the CPU

capacity into three partitions: RT partition for reserved allocation, Overrun partition for bursty usage and TS partition for general time-sharing processes.

The attractive feature of DSRT is that it is implemented in the user space. DSRT provides a good collection of APIs for applications to specify reservation and scheduling parameters and to collect process and CPU statistics. Although the notion of processor usage pattern is absent in our CVP scheduler and advance reservation is handled by a Grid resource management model, DSRT still provides important techniques for its middleware-based implementation strategy.

5.2 Resource Partitioning

5.2.1 Resource Container

Resource container [BaD99] introduces a new operating system abstraction for fine-grain partitioning of resources both at the user and the kernel level. This work identifies that a portion of resource consumption for a process takes place at the kernel level on which the process has no control. Therefore, it is possible that in the kernel mode resources are not fairly partitioned based on the priority of the processes. Resource container proposes creation of an operating system entity that will contain all the necessary resources (CPU time, network buffers etc.) required by a specific activity (a process or a thread). The mechanism can be useful for performance isolation among services to individual requests in a web server supporting numerous users. The concept of performance isolation is important to our design of CVP scheduler, but its implementation strategy is to modify a general purpose operating system which is not suitable in a Grid environment.

5.2.2 Cluster Reserves

Cluster Reserves [ArD00] extends the basic notion of performance isolation presented in Resource Container for a network of servers environment. Essentially, it allocates resource containers on different back-end cluster nodes shared by multiple web-services. However, the mechanism also provides support for dynamically adjusting the container capacity across the cluster nodes to improve resource utilization and average throughput of applications.

5.3 Process Synchronization Approaches

5.3.1 Explicit Co-Scheduling

Explicit Co-Scheduling [Ous82], also known as gang scheduling, introduces the idea of *co-scheduling*, where co-operating processes of a parallel application are scheduled on multiple processors at the same time as if the parallel application is running on a single machine. A simultaneous global context-switch across all the processors is required for this approach. Synchronization is performed by having a process spin and wait for another communicating process. This approach is more suitable for synchronization within a tightly coupled parallel system. It is less suitable for a geographically distributed setting as the one targeted by this study.

5.3.2 Implicit Scheduling

Implicit Scheduling [DuA96] is an approach that lets local schedulers independently schedule parallel jobs provided that processes themselves coordinate using a *conditional two-phase waiting* algorithm. In traditional two-phase waiting algorithms, a parallel process spins for a fixed waiting time for an event to occur and then blocks if waiting

is unsuccessful. In conditional two-phase waiting, a parallel process can dynamically increase its waiting time for variable length. However the decision to wait or block is made by each parallel process on the basis of implicit information that is associated with the communication inherent to parallel application. For example, by assessing the round-trip time of getting a reply message and incoming message rate a local process can infer whether remote process is scheduled or not. Then a local process can decide whether to spin or block. Implicit scheduling works for parallel applications on tightly coupled cluster of workstations. Nevertheless, its idea of process synchronization, specially the use of message arrivals, provides us with useful insights to develop our coordination model.

5.3.3 Dynamic Co-Scheduling

Dynamic Co-Scheduling (DCS) [DuC98] is a demand-based co-scheduling technique of parallel jobs. Generally, demand-based coscheduling techniques exploit communication between processes by observing the fact that the communicating processes are the ones that need to be coscheduled. In dynamic coscheduling, all message arrivals are treated as demands for coscheduling and scheduling decisions are made for destination process by explicitly controlling priorities. The concept of DCS is similar to implicit coscheduling but it handles a broader range of parallel applications more effectively.

5.4 Grid Resource Management

5.4.1 Globus Resource Management Architecture

Globus [FoK97, CzF98] is currently the most well-known and well-specified Grid resource management architecture. It is a toolkit that packages a set of services to handle various issues in a meta-computing system such as: resource location and allocation, authentica-

tion, other necessary tasks to prepare resources for application deployment. Some of the major challenges of resource management in a meta-computing environment are site autonomy, resource heterogeneity, implementing generalized extensible policy, co-allocation and adaptation to resource availability. Globus architecture is developed to addresses these challenges.

As shown in Figure 5.4, Globus introduces a number of entities to define its architecture.

- **Local Resource Managers:** These entities act as interfaces to local policies, security and access control. Such an interface helps to deal with site autonomy and heterogeneous configurations of resources. Local resource managers implements Grid Resource Allocation Managers (GRAM) to 1) process allocation request, 2)launch jobs, 3) enable monitoring and 4) update resource status.

- **Resource Specification Language (RSL):** is a specification language to enable negotiations among different entities.

- **Resource Brokers:** Applications specify their high-level requirements through RSL to *resource brokers* who translate those requests into actual resource requirements. The translated requirements are called ground RSL.

- **Resource Co-allocators:** These entities are responsible to coordinating resource allocations across multiple sites simultaneously. They interact with the local resource managers to make resource allocation requests.

- **Information Service:** This entity provides information about the resource availability and characteristics. The information service organizes and represents data in Lightweight Directory Access Protocol (LDAP) for uniformity, extensibility and distributed storage.

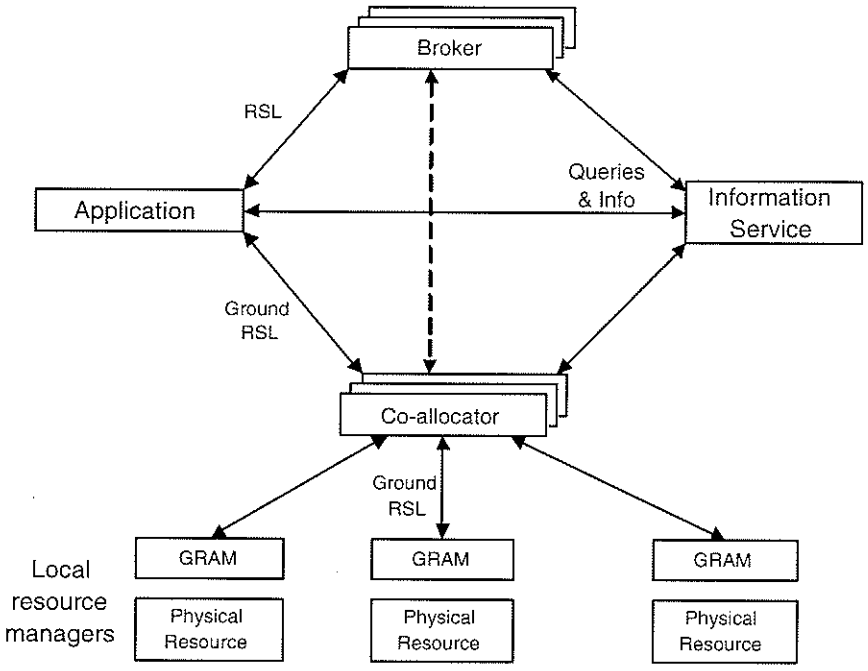

Figure 5.4: Globus Resource Management Architecture

Globus Architecture for Reservation and Allocation (GARA) [Fok99] extends the basic resource management architecture to include functionalities for advance reservation, co-allocation and dynamic resource discovery. GARA supports in a uniform fashion diverse resource types (e.g., networks, CPUs, memory, disk).

5.4.2 MPICH-G: Grid Enabled MPI

MPICH-G [Fok98] is a tool for running MPI programs over resources within a wide area environment. It allows a user to run MPI programs over machines from different geographically distributed domains by using same set of commands used to run parallel programs in clustered environment. Essentially MPICH-G uses the services provided by the Globus toolkit to 1) get information about accessing computers, 2) securely stage and launch executables, 3) utilize different site-specific communication methods and 4) monitor progress and terminate processes. However, MPICH-G library does not have

any functionality to control the resource allocation at runtime. This functionality is necessary in order to adapt to the varying nature of resource availability while MPI programs running. In our CVP design, we wanted to incorporate such a run-time control and dynamic coordination of resources that would facilitate the MPICH-G tool to manage a controlled resource provisioning mechanism in a dynamic environment.

Chapter 6

Conclusion and Future Work

6.1 Summary

This thesis describes the design and implementation of CVP, a mechanism for coordinated resource provisioning for wide-area Grid applications. The major objective of CVP is to orchestrate the local resource management systems such that the virtual partitions that are allocated to the different components of a Grid application are adequately provisioned for the application components within them to make simultaneous progress. The CVP mechanism builds a coordination tree for the virtual partitions that belong to a single application.

This thesis makes the following contributions:

- It provides a mathematical model for the coordination problem between virtual partitions. Based on this model, important limiting parameters such as asynchrony and granularity are defined. The relationships between these parameters and other variables such as CPU quantum are derived.

- It describes a middleware-level implementation of the CVP mechanism and the

experiments done using this implementation. The experiments were used to show the feasibility, performance, and utility of the CVP. The results indicate that CVP is capable of regulating the resource supplies to the virtual partitions such that similar progress rates are achieved.

- It presents early results from mapping an MPI application components onto a CVP. The results demonstrate that CVP can reduce the wait times for globally synchronizing operations. Even for applications that sparingly use *scatter-gather* operations the CVP mechanism proved to be valuable.

6.2 Future Work

We identify the following work that need to be done to enhance the capabilities of our CVP mechanism:

- A generalized CVP mechanism should support execution of multiple Grid application components per machine. In our current design we have assumed a single Grid application component can share a machine with other local processes. Therefore, we need to relax this limitation. We believe such an effort will require us to consider resource sharing relationships among application providers.

- We have considered coordinated resource provisioning for compute-bound applications. Future endeavors will focus to support other types of applications (e.g., interactive or I/O bound).

- Coordinated resource provisioning can be part of a Grid middleware such as Globus. For this integration, we will need to define specifications and interfaces to interact with other Grid services.

- Prediction plays an important role in assessing future resource availability. We used a simple but less accurate AR(1) model for our model as it takes little time to execute. However, more accurate prediction algorithms (ARMA, ARIMA) can improve the performance of our model if they can be executed with similar efficiency.

Appendix

List of Abbreviations

AR	Autoregressive Model
CVP	Coordinated Virtual Partition
DSRT	Dynamic Soft Real Time
GARA	Globus Architecture for Reservation and Allocation
GRAM	Grid Resource Allocation Manager
LDAP	Lightweight Directory Access Protocol
LRM	Local Resource Manager
MPI	Message Passing Interface
QoS	Quality of Service
RSL	Resource Specification Language
SFQ	Start-time Fair Queueing
SPMD	Single Program Multiple Data

Bibliography

[ArD00] M. Aron, P. Druschel, and W. Zwaenepoel, "Cluster reserves: A mechanism for resource management in cluster-based network servers," *Measurement and Modeling of Computer Systems*, June 2000, pp. 90–101.

[ArM98] J. Armstrong, M. Maheswaran, M. Theys, H. Siegel, M. Nichols, and K. Casey, "Parallel image correlation: Case study to examine trade-offs in algorithm-to-machine mappings," *J. Supercomputing, Special Issue on High-Performance Computing and Applications in Computer Graphics, Image Processing and Computer Vision*, Vol. 12, No. 1/2, January 1998, pp. 7–35.

[BaD99] G. Banga, P. Druschel, and J. Mogul, "Resource containers: A new facility for resource management in server systems," *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, February 1999, pp. 45–48.

[BoJ76] G. Box and G. Jenkins, *Time Series Analysis: Forecasting and Control*, Holden-Day, San Francisco, CA., 1976.

[ChA00] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, "Surplus-fair scheduling: A proportional share CPU scheduling algorithm for symmetric multiprocessors," *Fourth ACM Symposium on Operating System Design and Implementation(OSDI)*, October 2000, pp. 45–48.

[ChG02] A. Chandra, W. Gong, and P. Shenoy, "An online optimization-based technique for dynamic resource allocation in GPS servers," Technical Report UM–CS–2002–030, University of Massachussets, July 2002.

[Chu99] H. Chu, *CPU Service Classes: A Soft Real Time Framework for Multimedia Applications*, Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1999.

[CzF98] K. Czajkowski, I. Foster, C. Kesselman, N. Karonis, S. Martin, W. Smith, and S. Tuecke, "A resource management architecture for metacomputing systems," *IPPS/SPDP'98 Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998, pp. 62–82.

[DuA96] A. Dusseau, R. Arpaci, and D. E. Culler, "Effective distributed scheduling of parallel worklods," *1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1996, pp. 25–36.

[DuC98] A. Dusseau, D. E. Culler, and A. M. Mainwaring, "Scheduling with implicit information in distributed systems," *SIGMETRICS'98/PERFORMANCE'98 Joint Conference on Measurement and Modeling of Computer Systems*, June 1998, pp. 233–243.

[FoK01] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the Grid: Enabling scalable virtual organizations," *International Journal on High Performance Computing Applications*, Vol. 15, No. 3, November 2001, pp. 200–222.

[FoK97] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *Int'l J. Supercomputer Applications and High Performance Computing*, Vol. 11, No. 2, Summer 1997, pp. 115–128.

[Fok98] I. Foster and N. T. Karonis, "A grid-enabled MPI: Message passing in heterogeneous distributed computing systems," *In Proceedings of Supercomputing Conference*, November 1998, pp. 1–11.

[Fok99] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy, "A distrbuted resource management architecture that supports advance reservation and co-allocation," *Proceedings of the International Workshop on Quality of Service*, 1999, pp. 27–36.

[GoG96] P. Goyal, X. Guo, and H. Vin, "A hierarchical CPU scheduler for multimedia operating systems," *USENIX Symposium on Operating System Design and Implementation '96*, October 1998, pp. 107–122.

[KaT03] N. Karonis, B. Toonen, and I. Foster, "MPICH-G2: A Grid-enabled implementation of the message passing interface," *Journal of Parallel and Distributed Computing*, 2003, to appear.

[KrB02] K. Krauter, R. Buyya, and M. Maheswaran, "A taxonomy and survey of grid resource management systems," *Software Practice and Experience*, Vol. 32, No. 2, February 2002, pp. 135–164.

[Ora01] A. Oram, *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, O'Reilly and Associates, Sebastopol, CA, 2001.

[Ous82] J. Ousterhout, "Scheduling techniques for concurrent systems," *Third International Conference on Distributed Computing Systems*, May 1982, pp. 22–30.

[RoS01] T. Roscoe and P. Shenoy, "New resource control issues in shared clusters," *8th International Workshop on Interactive Distributed Multimedia Systems (IDMS'01)*, September 2001, pp. 2–9.

[WaW94] C. Waldspurger and W. Weihl, "Lottery scheduling: Flexible proportional-share resource management," *First Symposium on Operating System Design and Implementation*, November 1994, pp. 1–11.

[WaW95] C. Waldspurger and W. Weihl, "Stride scheduling: Deterministic proportional resource management," Technical Report MIT/LCS/TM-528, MIT Laboratory of Computer Science, June 1995.

[Wal95] C. Waldspurger, *Lottery and Stride Scheduling: Flexible Proportional-share Resource Management*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.