Parallel Job Scheduling Policies on Cluster Computing Systems

by

Jemal H. Abawajy

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University,

Ottawa, Ontario, Canada

November, 2003

©2003, Jemal H. Abawajy

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-89881-4

Our file Notre référence

ISBN: 0-612-89881-4

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Cluster computing has come to prominence as a cost-effective parallel processing tool for solving many complex computational problems. The key to making the cluster computing work well is the middleware technologies that can manage the policies, protocols, networks, and job scheduling across the interconnected set of computing resources. The research question addressed in this thesis is the on-line job scheduling problem for multi-cluster systems. We believe that, in order to fully harness the aggregate power of cluster computing systems, an efficient dynamic resource scheduling policy is required. To this end, we propose and experimentally validate several on-line dynamic scheduling policies that manage multiple job streams across both single and multiple cluster computing systems with the objectives of improving the mean response time and system utilization. Based on hierarchical resource management architecture, we present a dynamic parallel processor scheduling policy. We then extend this policy for dual (i.e., CPU and I/O) resource allocation to competing parallel jobs. Finally, we incorporate a fault-tolerant approach into the proposed job scheduling policy such that both the scheduler and the applications survive compute node and network failures. As part of performance analysis, we also extended several existing scheduling policies to make them suitable for cluster computing environments. We present results of the performance analysis under various system and workload parameters that demonstrate the viability of the proposed scheduling policies.

Contents

1	Introduction	1
1.1	Background and Motivation	1
1.2	Job Scheduling Problem	4
1.2.1	Job Scheduling Definition	4
1.2.2	Job Scheduling on Cluster Computing Systems	4
1.3	Thesis Focus and Contributions	7
1.3.1	Thesis Goal	7
1.3.2	Thesis Focus	8
1.3.3	Thesis Contribution	10
1.4	Chapter Summary and Thesis Organization	11
2	Background	12
2.1	Basic Concepts and Terminology	12
2.2	Uniprocessor Job Scheduling Approaches	13
2.3	Multiprocessor Job Scheduling Approaches	16
2.3.1	Parallel Computers	16
2.3.2	Review of Job Scheduling Approaches	17
2.4	Grid Computing	26
2.4.1	Resource Management Systems	28
2.4.2	Job Scheduling on Grid Computing Systems	30

2.5	Cluster Computing Systems	33
2.5.1	Cluster Management Systems	33
2.5.2	Work Stealing Approaches	37
2.5.3	Job Scheduling on Cluster Computing Systems	39
2.6	Coordinated Allocation of Multiple Resources	44
2.6.1	I/O Resource Considerations	45
2.6.2	Memory Considerations	47
2.7	Fault-Tolerance Resource Allocation	48
2.7.1	Types of Failures	49
2.7.2	Fault-Tolerance Mechanisms	49
2.7.3	Fault-Tolerant Scheduling Middleware	51
2.8	Summary	51
3	Dynamic Hybrid Scheduling Policy	53
3.1	Introduction	53
3.1.1	Motivation	53
3.1.2	Problem Statement	55
3.1.3	Contributions	56
3.2	Hierarchical Distributed System Architecture	57
3.3	Dynamic Hierarchical Scheduling Policy	60
3.3.1	Request Sequencing Algorithm	60
3.3.2	Pull Algorithm	64
3.3.3	Push Algorithm	66
3.3.4	Comparative Analysis	68
3.4	Baseline Parallel Job Scheduling Policies	71
3.4.1	Time Sharing Policy	71
3.4.2	Space Sharing Policy	73
3.5	Performance Evaluation	75

3.5.1	Experimental Environment Setup	77
3.5.2	Parallel and Background Workload Models	78
3.6	Simulation Results and Discussions	81
3.6.1	Dynamic versus Static Hybrid Policies	82
3.6.2	Dynamic versus Baseline Policies	83
3.6.3	Sensitivity Analysis	89
3.7	Summary	96
4	Coordinated Resource Scheduling	97
4.1	Motivation and Problem Statement	98
4.1.1	Problem Statement	99
4.1.2	Contributions	101
4.2	System and Workload Models	101
4.2.1	System Architecture	101
4.2.2	Workload Models	103
4.3	Coordinated Resource Scheduling Policies	105
4.3.1	Proposed Scheduling Policies	105
4.3.2	Adaptive Local Cluster Scheduling Policy	107
4.3.3	Baseline Scheduling Policies	111
4.4	Performance Evaluation	113
4.4.1	System Model	114
4.4.2	Workload Model	115
4.5	Simulation Results and Discussions	116
4.5.1	Validation of the Baseline Policies	117
4.5.2	Unified versus Separate Resource Scheduling	118
4.5.3	Relative Performance of Unified Scheduling Policies	118
4.5.4	Sensitivity to I/O Frequency	121
4.5.5	Sensitivity to Service Demand Variations	122

4.5.6	Sensitivity to Arrival Time Variations	124
4.5.7	Sensitivity to I/O Factor	126
4.5.8	Sensitivity to Background Workload Arrival Variations	128
4.5.9	Sensitivity to Number of Disks	129
4.5.10	Sensitivity to Multiprogramming Level	130
4.5.11	Sensitivity to Network Bandwidth	131
4.6	Summary	132
5	Fault-Tolerant Job Scheduling	133
5.1	Background and Problem Statement	134
5.1.1	Motivation	134
5.1.2	Problem Statement and Related Work	135
5.1.3	Contributions	137
5.2	System Architecture and Failure Modes	139
5.2.1	Failure Modes	139
5.2.2	System Configuration	140
5.3	Dynamic Fault-Tolerant Scheduling Policy	146
5.3.1	Application Scheduling and Fail Management	147
5.3.2	Switchover and Switchback Strategies	153
5.4	Baseline Scheduling Policies	156
5.4.1	Fault-Tolerant Scheduling Algorithm	157
5.5	Performance Analysis	160
5.6	Results and Discussions	162
5.6.1	Deterministic Node Failure Environment	162
5.6.2	Non-Deterministic Node Failure Environment	165
5.6.3	Multiple Node Failure in a Cluster	166
5.6.4	Sensitivity to Mean Times Between Failures	167
5.6.5	Sensitivity to Link Failures	168

5.7	Summary	170
6	Conclusions and Future Directions	172
6.1	Processor Scheduling	172
6.2	CPU-I/O Intensive Parallel Job Scheduling	173
6.3	Fault-Tolerant Resource Allocation	174
6.4	Extending Previous Approaches	175
6.5	Future Directions	175
A	Glossary	177
B	Dynamic/Static Hybrid Policies	180
B.1	Dedicated Heterogeneous Environments	180
B.2	Shared Heterogeneous Environments	182
B.3	Shared Homogeneous Environments	183

List of Figures

1.1	Multicluster computing system.	9
2.1	Multiprocessor Systems: (a) Shared-memory systems; and (b) Distributed-memory systems.	17
2.2	Architecture of a typical grid computing system	28
2.3	Architecture of a typical batch queuing system	34
3.1	On-line processor allocation problem	55
3.2	An example of cluster tree (SS: System scheduler; CS: Cluster scheduler; LS: Local scheduler; and P_i : workstation i)	58
3.3	Architecture of the adaptive hierarchical scheduling policy. RFC: request for computation; RFB: request for load-balancing; CS: cluster scheduler; SS: system scheduler.	61
3.4	Pseudo-code of the request sequencing algorithm	63
3.5	An illustration of the request sequencing	64
3.6	Description of the pull algorithm.	65
3.7	An illustration of the pull algorithm. (For simplicity, local schedulers associated with processors are not shown in this figure.)	66
3.8	Dynamic push algorithm	67
3.9	An illustration of the load balancing activity.	69
3.10	Job-based time sharing policy	72

3.11 Adaptive space sharing policy	74
3.12 Relative performance under shared homogeneous environments . . .	86
3.13 Relative performance under the dedicated heterogeneous environment.	87
3.14 Relative performance under shared heterogeneous environments. . .	90
3.15 Sensitivity to service demand variations: Homogeneous shared settings; Heterogeneous dedicated settings; and Heterogeneous shared settings.	92
3.16 Sensitivity to arrival time variations: Homogeneous shared settings; Heterogeneous dedicated settings; and Heterogeneous shared settings.	94
3.17 Sensitivity to cluster size.	95
4.1 Coordinated processors and I/O resources allocation problem	100
4.2 An overview of the system architecture. P : Processors; LS : Local Scheduler; CS : Cluster Scheduler; SS : System Scheduler; R_{IO} : remote I/O request and L_{IO} : local I/O request.	102
4.3 The I/O properties of parallel applications.	103
4.4 The I/O properties of parallel applications: (a) non-overlapped execu- tion, (b) overlapped execution.	103
4.5 Adaptive global scheduling policy	106
4.6 An illustration of the adaptive global scheduling policy	107
4.7 Adaptive local scheduling policy	109
4.8 An illustration of the adaptive local scheduling policy: (a) Job submis- sion. (b) Self-scheduling and job assignment.	110
4.9 An illustration of the adaptive local scheduling policy.	111
4.10 Validation of the baseline and modified policies.	117
4.11 Relative performance of AGS and AHS policies.	119
4.12 Relative performance of the four policies in shared homogeneous envi- ronments.	120

4.13	Relative performance of the four policies in shared heterogeneous environments.	122
4.14	Sensitivity to service demand CV in shared homogeneous environments.	123
4.15	Sensitivity to service demand CV in shared heterogeneous environments.	124
4.16	Sensitivity to arrival time CV in shared homogeneous environments. .	125
4.17	Sensitivity to arrival time CV in shared heterogeneous environments.	126
4.18	Sensitivity to I/O factor in shared homogeneous environments.	127
4.19	Sensitivity to I/O factor in shared heterogeneous environments.	128
4.20	Sensitivity to background workload utilization: shared homogeneous environments (top) and shared heterogeneous environments (bottom).	129
4.21	Sensitivity to the number of disks.	130
4.22	Sensitivity to the multiprogramming level.	131
4.23	Sensitivity to the network bandwidth.	132
5.1	Definition of fault-tolerant parallel job scheduling problem.	135
5.2	Reconfigurable system architecture	140
5.3	Health monitor architecture	141
5.4	Failure detection with health monitor.	142
5.5	Failure detection architecture. The dotted lines indicate the missing <i>alive</i> messages.	144
5.6	Dynamic fault-tolerant scheduling policy	148
5.7	Application replica placement algorithm	149
5.8	Application replica placement algorithm	149
5.9	Monitoring Jobs and Tasks.	151
5.10	Illustration of failure detection and recovery.	152
5.11	Link failures: (a) Between two cluster schedulers; (b) Between a cluster scheduler and the system scheduler	154

5.12 Repair: (a) System scheduler failure scenarios; (b) Repaired system scheduler	155
5.13 Node Failures: (a) Backup scheduler fails; (b) Primary scheduler fails.	156
5.14 Fault-tolerant scheduling algorithm	158
5.15 Performance of the policies in dedicated heterogeneous environments	163
5.16 Performance of the policies in shared homogeneous environments . . .	163
5.17 Performance of the policies in shared heterogeneous environments . .	164
5.18 Performance of the policies in non-deterministic dedicated heterogeneous environments.	165
5.19 Impact of multiple processor failures in dedicated heterogeneous environments.	166
5.20 Performance of the policies as a function of mean time between failures/hour.	168
5.21 Sensitivity to link failures.	169
 B.1 Relative performance under dedicated heterogeneous environments .	180
B.2 Relative performance under dedicated heterogeneous environments .	181
B.3 Relative performance under the shared heterogeneous environment. .	182
B.4 Relative performance under shared homogeneous environments	183

List of Tables

3.1	Default parameters and values used in the experiments.	76
3.2	Processor type and speed used in the simulation.	78
3.3	Non-uniform system configuration.	95
4.1	Default parameters and values used in the experiments.	114
5.1	Default parameters and values used in the experiments.	161

Chapter 1

Introduction

Cluster computing systems have recently generated enormous interest in processing large-scale scientific applications such as high-energy physics simulations and data mining for business applications. One of the most important problems that must be addressed in order to realize the advantages of *cluster computing* systems, and the focus of this thesis, is that of job scheduling problem. This chapter motivates the need for an efficient job scheduling policy for cluster computing systems. An outline of the approach taken to address the scheduling problem is discussed. Then the goals of the thesis within the context of this approach is established. Finally, the contributions of the thesis are described.

1.1 Background and Motivation

In the last few years, the trends in parallel processing system design and deployment have been moving away from a single isolated powerful supercomputers to cooperative networked distributed systems such as *grid computing* and commodity-based *cluster computing* systems. One of the main driving forces behind the move to network-based distributed systems is the unprecedented technological advances in high performance microprocessors and high-speed networks coupled with falling prices of these tech-

nologies. Another important factor is the continuous rise of application demands for processing speed and power. For example, many scientific applications such as high-energy physics simulations and data mining for business applications, often require the use of *supercomputing* systems to achieve computational power needed to solve these problems. Although *supercomputers* are fast, they are also extremely expensive making them out of reach for many small to medium-sized research groups and organizations. As a result, network-based parallel processing computing has generated enormous interest in processing large-scale scientific and commercial applications.

In this thesis, we consider *cluster computing* systems. Cluster computing systems are generally classified as either *High Throughput Computing (HTC)* or *High Performance Computing (HPC)* systems [24]. HPC brings enormous amounts of computing power to bear over relatively short periods of time whereas the goal of HTC is to deliver large amounts of processing capacity over long periods of time. Within these two categories, cluster computing systems can be further classified into three main classes as follows:

1. **Volunteer Cluster Computing (VCC):** Individual workstation in the system is considered to be privately owned. The workstations may be used for processing externally submitted tasks only when the owner is not using it. The VCC cluster type is commonly used as *HTC* [24]
2. **Community Cluster Computing (CCC):** All machines are shareable and may be homogeneous or heterogeneous. The CCC cluster computing are used both as *HTC* and as *HPC*.
3. **Private Cluster Computing (PCC):** PCC is a dedicated cluster of workstations, which is commonly referred to as *Beowulf*. The PCC is intended for use either as a dedicated *HTC* or as a dedicated *HPC* platform.

The primary benefits of using a cluster computing system are attaining access to

resources that would otherwise be unavailable, and allowing the system to hide the complexity of resource management from the user. The key to making the cluster computing work well is the middleware technologies that can manage the policies, protocols, networks, and job scheduling across the interconnected set of computer resources. A tremendous effort has been made and is still underway to provide and fine tune a number of middleware technologies such as scheduling service, resource discovery, data management and so forth allowing cluster computing users to access large resources in order to solve the most challenging computational problems.

Existing tools and systems are useful for putting together a cost-effective off-the-shelf high-performance computing (HPC) or high-throughput computing (HTC) systems as well as developing application programs and executing them remotely. For example, the availability and maturity of parallel and distributed toolkit systems such as PVM [128] and MPI [116] that facilitate communications between the workstations as well as writing parallel and distributed programs for cluster computing environments have played a major role in the popularity of the cluster computing systems. Also, successful *cluster management* infrastructures such as the one described in [23, 106, 113, 62] have simplified cluster access and usage, allowing the cluster to be useful to a wider range of scientists and engineers. Being able to bring enough computing cycles together in a cluster is one thing, but to be able to use these cycles efficiently so that the aggregate power of a cluster can be fully harnessed is another.

A principal consideration of any resource management systems is the efficient *assignment* of system resources to applications. With efficient resource allocation, system owners and users can realize good performance and increased resource usage in an economical manner. Unfortunately, existing tools and resource management software provide little or no assistance for efficiently scheduling applications as well as utilizing resources effectively. As a result, extracting the best possible performance out of a cluster computing system has remained a challenging problem to users. The

research reported in this thesis addresses this problem. In the following section, we formulate the job scheduling problem and then show how the underlying characteristics of the cluster computing environments complicate the design and development of scheduling policies for cluster computing systems.

1.2 Job Scheduling Problem

1.2.1 Job Scheduling Definition

Large-scale applications that consist of a large number of tasks such as those that arise from parameter studies or probabilistic analysis can benefit from a large number of machine resources often spanning multiple clusters across the network. However, this requires that system resources be allocated to the application in a manner that reduces the application completion time. Collectively, we refer to this as *application scheduling* or *job scheduling* problem.

This problem has been extensively studied on parallel computers (e.g., [50, 19, 124, 54]) and to a lesser extent on cluster computing systems (e.g., [23, 106]). Moreover, its difficulty is well documented in the literature [54]. Existing solutions to the job scheduling problem can be generally divided into *static scheduling policy (SSP)* and *dynamic scheduling policy (DSP)*. In a *SSP*, the schedule is computed at compile-time. This requires advance knowledge of the system and application execution times. In the *DSP*, the schedule is computed at runtime.

1.2.2 Job Scheduling on Cluster Computing Systems

Job scheduling is a complex problem, yet it is fundamental to sustaining and improving the performance of parallel processing systems. Although the scheduling problem is conceptually the same across different systems and parallel workloads, there is no portability across the architecture spectrum as noted in [47]. The feasibility and

performance of possible solutions are very sensitive to the underlying architecture, network and workload changes [47]. In particular, cluster computing systems have several subtle but significant characteristics that influence and complicate scheduling decisions, which are not an issue in the design and development of scheduling policies for parallel computers. Some of these factors include:

1. **Scale:** The number of resources available can potentially be much larger than the conventional parallel processing systems. This requires an effective way of aggregating these resources.
2. **Control and Management:** The advantages of massively parallel processors (MPPs) are that they present a uniform computing environment managed by one operating system, one communication method, and one organization. In contrast, there is less autonomy and control over resources (e.g., workstations) in cluster computing.
3. **Heterogeneity:** Multiprocessor systems typically consist of homogeneous processors. Workstations in cluster computing, on the other hand, may be different from each other in architecture, CPU speed, available disk space and so on.
4. **Variation in Availability:** The availability of the system resources vary over time. This variation is both highly dynamic and unpredictable.
5. **Interconnection Network:** Clusters of workstations rely on interconnection technologies that typically exhibit high overhead and low bandwidth. Moreover, the network is characterized by variable load due to both parallel and interactive workloads.
6. **Fault-Tolerance:** Relative to the MPP nodes, workstations in cluster computing are vulnerable to failures. This is mainly due to the fact that users may

reset/switch off their computers, load the system heavily, and several other logistical problems associated with non-dedicated shared systems.

Due to the inherent characteristics of cluster computing systems, scheduling strategies developed for the traditional distributed systems have to be extended significantly to support the dynamics of cluster computing systems [127]. Particularly, the *adaptability* and *fault-tolerance* are important issues in resource allocation policies for cluster computing systems. Fault-tolerance is important as the scheduling policy must be able to provide continuous service to users despite system disruptions and failures. Dynamic scheduling is required since resources (i.e., machines and networks) may suffer dynamic load fluctuations or may be added or removed during the course of application execution.

Cluster computing environments require new approaches for efficiently allocating and re-allocating resources to competing jobs in a dynamic, unpredictable and failure-prone environment. Unfortunately, neither adaptability nor fault-tolerance have been factored into the design of most existing scheduling strategies. Research coverage of fault tolerant scheduling is limited as the primary goal for nearly all scheduling algorithms developed so far has been high performance by exploiting as much parallelism as possible. Only little attention has been paid to approaches which combine scheduling with fault tolerance. Also, most of the conventional cluster-based systems use a static scheduling model (e.g., LSF [106] and Condor [23]). We argue that static policies do not adapt well to network-based distributed systems, particularly those built to support HPC and HTC on cluster computing systems. Obstacles include resource sharing and heterogeneity of resources. Adaptive scheduling policies are based on the assumption, commonly made for traditional distributed systems, that the workstations are dedicated and homogeneous have been proposed for cluster computing systems (e.g., [121]). Moreover, existing solutions for the cluster computing systems, to a large extent, are based on requiring static and dynamic application and system

resource information, and performance prediction models. This kind of information is not always available and is often difficult to obtain specially in the presence of resource failure.

Despite the attention cluster computing has received, job scheduling problem in cluster computing environment has remained an important and open problem [48]. Therefore, there exists an opportunity to study and develop scheduling policies for cluster computing systems with the objectives of achieving good application performance, while ensuring that the computational resources are utilized effectively and jobs are executed to their logical termination in the presence of failures.

1.3 Thesis Focus and Contributions

1.3.1 Thesis Goal

The goal of the thesis is to design and evaluate efficient dynamic *parallel job scheduling policies* for cluster computing systems. Job scheduling is an important aspect of distributed computing systems, particularly in the case of *cluster computing systems*, where the scale of the system, local computation and resource heterogeneity give rise to a large number of scheduling choices. To date, the focus on cluster computing systems is on seamless integration of multiple workstations into a virtual single system. However, aggregation of many resources is not enough to guarantee good performance — careful scheduling must be employed to achieve the best performance possible [26]. Without the support from well-designed cluster job scheduling policy, resources are shared in an ad-hoc manner, limiting performance as well as the utilization of the resources. Therefore, in order to be widely useful, cluster infrastructure technology must provide more functionality beyond the technologies that enable access, selection, and aggregation of resources and services. In particular, the ability to tolerate failures while effectively exploiting the cluster computing resources in a transparent manner

must be an integral part of cluster computing infrastructure. This problem is very important as the potential for faults in cluster computing is a significant complicating factor for users [120].

1.3.2 Thesis Focus

Cluster computing systems can be a single cluster systems as in [16] or multiple cluster systems as in [32]. We define a *cluster* as a collection of independent workstations interconnected by a local area network (LAN) or a wide-area network (WAN) such that the collection appears to an end user or application as one large virtual computing system. For the most part, job scheduling research in cluster computing environments has focused on single-cluster computing systems (e.g., [104, 76, 48]). As noted in [78], a common assumption in the previous work has been that all processors in the system have equal processing capacity (i.e., homogeneous) (e.g., [121]) and dedicated (e.g., [121]). In contrast, we focus on *shared*, *heterogeneous*, and *multi-programmed* cluster computing systems composed of *multiple clusters* spanning campus-wide area see Figure 1.1).

We assume that a cluster is geographically compact and contains a set of workstations of similar architecture, connected by a fast Ethernet (100 Mbits/sec). Similarly, inter-cluster interconnection network is a fast Ethernet (100 Mbits/sec). We also assume that a network segment contains a single cluster only. Computers in different clusters do not share the communication bandwidth. Note that these assumptions can be easily relaxed, but they allow us to ignore the mechanisms involved in getting jobs to run, and concentrate on policy issues for selecting which jobs as well as where and when they should run.

In addition, we restrict our focus to two dimensions of heterogeneity: processor speed and cluster sizes as these two are expected to be the major components of system heterogeneity in networked-systems [78]. We further restrict our focus on

Figure 1.1: Multicluster computing system.

the *Volunteer Cluster Computing (VCC)* and *Community Cluster Computing (CCC)* environments. Note that job scheduling research in cluster computing environments has focused mainly on *PCC* (e.g., [121, 17]) and to a lesser extent on *VCC* (e.g., [104]) while there is very little work on *CCC* environments. Scheduling on *PCC* is essentially the same as that of the traditional multiprocessor systems since the workstations in *PCC* are dedicated and homogeneous.

We also focus on dynamic scheduling policies that combine both space-sharing and time-sharing approaches. In [48], a scheduling policy that combines both *space-sharing* and *time-sharing* approaches was suggested as an attractive alternative for cluster computing systems but never further developed.

Finally, we focus on the moldable parallel workload model as this type of workload constitutes a significant portion of the high-performance computing centers workloads [37] and likely to increase in future [52]. Moldable jobs are parallel jobs that are flexible in the number of processors at the time the job starts, but cannot be reconfigured during execution.

1.3.3 Thesis Contribution

1. In an environment such as cluster computing where the number of resources can be potentially large, an architecture in which the system resources are effectively utilized is paramount. To achieve this goal, we propose a hierarchical architecture that effectively aggregates multiple cooperating clusters of computers so that job scheduling is done not only within the clusters but also among them [8].
2. We propose a dynamic scheduling policy that has the ability to allocate and reallocate system resources along both time and space axes, among a number of competing jobs, giving each the required level of service [7, 9, 10]. Our performance analysis shows that policies with the capability to distribute and redistribute the load perform consistently better than the classical load distribution policies [11].
3. An approach that couples scheduling and job replication mechanisms as a means to achieve fault tolerant job scheduling is proposed and studied experimentally [5, 6]. The results of the experiments show that the proposed scheme incurs reasonable overhead and the performance of the application gracefully degrades in the presence of failures.
4. We address the problem of job scheduling in the presence of multiple resource requirements with emphasis on CPU and I/O resources. We describe a scheduling framework that addresses these problems. Within this framework, we develop a family of dynamic job scheduling policies and use simulation studies to evaluate their performance [13, 12]. We also compare performance of the proposed scheduling policies against two baseline policies under various system and workload parameters. Our results show that the proposed scheduling policies substantially outperform the baseline policies under all load and system

parameters.

5. We extend previous work in adaptive scheduling approaches by identifying several issues that impact the performance of the conventional adaptive space-sharing scheduling policies in shared heterogeneous cluster computing environments. We proposed several solutions and validated the viability of the proposed approaches through trace-based simulation analysis. The performance of these two policies with and without the optimizations were studied. The version with the proposed optimizations produced significantly better response times than the original policies.

1.4 Chapter Summary and Thesis Organization

To properly place cluster-based distributed computing into its historical perspective, a review of current and influential cluster computing projects is presented in Chapter 2. The classical approaches of cluster control via batch queuing systems and wide-area managers are discussed. A literature survey on scheduling research is presented. In Chapter 3, the proposed scheduling policy along with its performance analysis is presented. In Chapter 4, we address the problem of combined CPU and I/O resources scheduling problem. We propose and study the performance of the new scheduling policies. We also compare the performance of the proposed scheduling policies against two similar scheduling policies. In Chapter 5, we address the fault-tolerant job scheduling problem. The conclusion and future directions are described in Chapter 6.

Chapter 2

Background

This chapter provides background in the area of scheduling jobs across a range of parallel processing systems. The literature related to job scheduling is quite extensive as job scheduling is a well-studied topic and obviously no survey study can ever capture all of the development in a field as broad as this one. Therefore, it is beyond the scope of this thesis to survey it all. A summary of resource management and scheduling on workstations clusters is presented. Fault-tolerance approaches for cluster computing environment is reviewed.

2.1 Basic Concepts and Terminology

In this thesis, we define resource as any capability that must be scheduled, assigned, or controlled by the underlying implementation to assure non-conflicting usage by processes. Example of resources include processors and storage devices. Also, a node refers to a processor, workstation, or a computing machine in a cluster environment and the terms are used interchangeably. In the thesis, the terms jobs and applications are used interchangeably to refer to a request made by a user to run a given application or program with a given input. Each job can be viewed as composed of a set of independent units of computational work, which we refer to as *tasks* or *processes*

that may run independently or in parallel.

The performance of schedulers is influenced by the workloads that serve as their input. A useful classification of parallel jobs from the view-point of scheduling is provided in [53]. This paper divides parallel jobs into four classes: (i) Rigid, (ii) Moldable (iii) Evolving, and (iv) Malleable. A rigid job requires a certain number of processors at arrival time and executes on these processors exclusively until completion. Moldable jobs are flexible in the number of processors at the time the job starts, but cannot be reconfigured during execution. Moldable jobs can be further subdivided into fixed and adaptive jobs. The program structure and parameters (e.g., number of tasks) of fixed jobs are determined at compile time while in adaptive jobs they are determined at run time [31]. Since the allocation of processors to jobs is performed at run time, it is possible that a fixed moldable job can be assigned fewer processors than the number of tasks it has. In this case, some processors can be assigned more tasks. Both Evolving and Malleable jobs can change their processor requirements during execution. For evolving jobs, changes are application initiated. If the system cannot satisfy the job's demand, the job cannot proceed. For malleable jobs, the decision to change the number of processors is made by an external job scheduler.

2.2 Uniprocessor Job Scheduling Approaches

Uniprocessor scheduling deals with the problem of deciding which of the processes in the ready queue are to be allocated the CPU. There are many different CPU scheduling algorithms, which can be generalized into *preemptive* and *non-preemptive* scheduling policies. By a *non-preemptive* scheduling policy, we mean that once a task has begun execution, it cannot be interrupted. In contrast, a *preemptive* policy can take away the CPU from a running task and give it to another ready task.

Examples of *non-preemptive* scheduling policies include: (1) *First Come First Served (FCFS)* policy in which the tasks are serviced in the order of their arrival; (2) *Shortest-Job-First (SJF)* policy in which the CPU is assigned to the process that has the smallest CPU demand; and (3) *Longest-Job-First (LJB)* in which the CPU is assigned to the process that has the highest CPU demand. In both *SJF* and *LJB*, if two processes have the same length of the CPU demand, scheduling is done based on the FCFS policy.

Examples of *preemptive* scheduling policies include: (1) *Round Robin (RR)*; (2) *Multilevel Queue (MQ)*; (3) *Priority based (PB)*; and (4) *Stride Scheduling (SS)*. The *RR* policy rotates the CPU among the tasks in the ready task queue, giving only a small quantum of service to a task at a time, then quickly switching to another task. The quantum length used in popular workstations range from 20ms to 200ms. The *MQ* policy partitions the ready queue into several separate queues and processes are permanently assigned to a queue, scheduling is done on these queues. Another variation of *MQ* is the *Multi-level Feedback Queue (MF)* policy in which each queue has a different time quantum. The high-priority queue has a short quantum. The middle-priority queue has a longer one, and the lowest priority queue has none. Since jobs move down the priority queue when their time quantum expires, we get a good time quantum selection for the various jobs – the more CPU-bound ones have a longer (or no) quantum, and the I/O bound jobs not only execute in preference to the CPU-bound jobs, but they have a shorter time-quantum.

In *PB*-based, the highest priority process obtains the CPU first. Then, as a process continues to use the CPU, its priority degrades so that lower priority processes waiting in line may have a turn. Stride scheduling [131] uses the notion of tickets to fairly allocate resources, including CPU time. Each job gets a proportion of the CPU, according to the proportion of tickets assigned to the job. A time line is then produced for each job containing the periods when the job is scheduled to run. That

is, the time quantum is placed at strides along the timeline. The timelines from all the jobs are pushed down onto a single timeline, and idle time squeezed out.

The advantages of non-preemptive policies is that they are very easy to implement. However, they may not be good when there are various types of processes in the system. For example, a large process can monopolize CPU and lead to substantial delays for small processes. The ability to preempt jobs permits the scheduler to more efficiently utilize the system in several ways: (1) long running jobs and those with high resource requirements can be executed without monopolizing resources; (2) interactive and other high priority jobs can be provided with near real-time response, even jobs with high resource requirements during periods of high system utilization; (3) jobs with high processor requirements can be initiated in a timely fashion, without waiting for processors to be made available in a piecemeal fashion as other jobs terminate; (4) low priority jobs can be executed, provided with otherwise unused resources, and preempted when higher priority jobs become available; and (5) high system utilization can be sustained under a wide range of workloads.

However, preemption does incur an overhead commonly referred to as the context switching. The context switch cost is in the range of $200\mu s$. Moreover, the increase in storage requirements is possibly substantial. All running or preempted jobs must have their storage requirements satisfied simultaneously. Preempted jobs must also vacate memory for other jobs, with the memory contents written to disk. Choosing the time-quantum (the interrupt interval) is important. If the quantum is too long, the interactive jobs will run too slowly and lose their ability to be effective interactively. If the quantum is too short, then the operating system spends an inordinate amount of time context switching between jobs and the system becomes inefficient.

2.3 Multiprocessor Job Scheduling Approaches

2.3.1 Parallel Computers

Parallel computers are architecturally divided into *shared-memory multiprocessors* and *distributed-memory multicomputers* as shown in Figure 2.1. In the *shared-memory systems* (see Figure 2.1a), all processors have access to a single uniform virtual address space and communication between processors is implicit as well as transparent. This makes inter-processor communication easy to implement.

Shared-memory systems are classified into *uniform memory access (UMA)* systems and *non-uniform memory access (NUMA)* systems. The UMA systems are suitable for small-scale systems as they have inherent scalability problem since they are based on a shared bus to connect processors to the memory system. The symmetric multiprocessor systems (SMP) is an example of the UMA architecture. In the NUMA architecture, the physical memory of the system is distributed among the individual processors. However, the memory modules are still globally addressable by all processors.

In the *distributed-memory systems* (see Figure 2.1b), each CPU has its own memory. Communication and synchronization between the processors is performed by exchanging messages. The Intel's iPSC/860 and Paragon are examples of distributed-memory system.

In addition to the well known computing paradigms of shared memory and distributed memory, there is a third paradigm – *Distributed Shared Memory (DSM)*. A DSM system logically implements the shared-memory model on a physically distributed-memory system. System designers can implement the specific mechanism for achieving the shared-memory abstraction in hardware or software in a variety of ways. The DSM system hides the remote communication mechanism from the application writer, preserving the programming ease and portability typical of shared-memory systems.

Figure 2.1: Multiprocessor Systems: (a) Shared-memory systems; and (b) Distributed-memory systems.

2.3.2 Review of Job Scheduling Approaches

Scheduling in multiprocessor systems can be classified into *space-sharing* and *time-sharing*. It is also possible to combine these two types of policies into a *hybrid policy*. In the following subsections, these three classes of scheduling policies are reviewed.

2.3.2.1 Space-Sharing Policies

In the space sharing approach, processors are partitioned into disjoint sets and each application executes in isolation on one of these sets. Space-sharing approaches can be classified as static-space sharing [112], adaptive space sharing [111] and dynamic space sharing [65]. These three different space-sharing policies are discussed in the following subsections.

Static Space Sharing Policy

Static space-sharing policies require each job to declare the number of processors needed at arrival time. If the number of processors required by the job is available, it is immediately scheduled. If the job cannot be scheduled upon arrival, it is queued in a central job wait queue until such time that the required number of processors becomes available. The jobs in the job wait queue are scheduled in FCFS bases.

Static scheduling can perform well if the jobs tend to have low variation in parallelism and service times. The major drawback of the static space sharing policy is that it fails to recognize that most parallel applications have sub-linear speedups, thus cannot fully utilize all the processors allocated to them during their execution. Because applications hold the same number of processors until they terminate, new applications cannot be started immediately. Both of these factors could have a negative impact on system performance, especially when the system load is high. As a result, static scheduling can lead to relatively low processor utilization and system throughput.

Some of the problems associated with the static policies can be addressed by taking into account job characteristics such as the service demand of the jobs as in the Smallest-Number-of-Processes-First (SNPF) policy and the Smallest-Cumulative-Demand-First (SCDF) [79]. In SNPF, free processors are allocated to the job with the smallest number of waiting processes. In SCDF the cumulative demand of jobs is considered instead of the parallelism level when choosing the next job to be assigned to the available processors. Majumdar, Eager and Bunt [79] showed that SCDF perform significantly better than FCFS-based static policies but SNPF perform only slightly better than FCFS-based policy, unless there is a positive correlation between the number of processes in an application and the cumulative demand of the application.

Backfilling [55] is proposed to address the processor fragmentation problem associated with the static policies. Backfilling refers to the process of making reservations for jobs to run either immediately or at some later time. A backfilling scheduler searches for free slots in the schedule and utilizes them with suitable jobs without delaying other already scheduled jobs. Backfilling scheduling has been shown to increase the machine utilization by up to 15% compared to FCFS job sequencing approach [55]. The drawbacks of backfilling includes: 1) an unbounded delay for job start, 2) unpredictability of the schedule; and 3) fragmentation if there are not enough small

jobs or if the jobs are of the wrong size and duration.

Adaptive Space Sharing Policies

The main idea of adaptive space-sharing policies is that the number of processors assigned to a job is a compromise between the user's request and what the system can provide. These policies can be classified as work-conserving adaptive policies and non-work-conserving adaptive policies. Work-conserving policies allocate all available processors to the jobs waiting for service. An example of work conserving policy is the PWS [45] and the Adaptive Static Partitioning (ASP) [111]. In ASP, if a job finds idle processors on arrival, it is allocated the lesser of the idle processors and the job's maximum parallelism (P_{max}). Otherwise, the job waits in the job wait queue. When a job completes, the released processors are allocated uniformly among the waiting jobs.

A non-work-conserving policy may intentionally leave some processors idle (i.e., unallocated) even when there are jobs waiting for service. Three families of non-work-conserving policies called Fixed Processors Per Job (FPPJ), Equal Partitioning with a Maximum (EPWM) and Insurance Policy (IP) have been presented in [101]. In FPPJ the processor set is divided into a fixed number of equal sized partitions. In EPWM, the set of currently free processors is equally divided among the jobs in the waiting queue with a Maximum partition per job set as a configuration parameter. The Insurance Policy (IP), always reserves a percentage of the available free processors for later job arrivals depending on the system load.

An alternative way of being non-work-conserving is implemented by the Processor Saving Adaptive (PSA) policy [102]. This policy does not force any *a priori* constraint on the size of the partition. Rather, processor saving decisions are based on the system history in which the scheduler is made to remember past allocations and make current decisions based on the recent past behavior.

The Robust Adaptive scheduling policies (i.e. AP1, AP2, AP3, P4, and AP5) are another set of non-work-conserving families discussed in [101]. These five policies (AP1 through AP5) differ mainly in the way they determine a partition size to the parallel jobs. A variation of AP2 called Modified Adaptive Policy (MAP) is studied by [124] in which the scheduler takes the running jobs into account when calculating the target partition size. They show that AP2 policy with the new partition size algorithm provides appreciable improvements in performance as compared to the original algorithm.

Brecht [29] has carried out an experimental evaluation of scheduling in systems where processors are identified with clusters or pools, and intra-cluster memory access is faster than inter-cluster access. A surprising result is that worst-fit scheduling, where each job is allocated to the pool with the most available processors, beats best-fit scheduling, where jobs are placed where they come closest to filling out a pool. This is a result of using a model of evolving jobs, where it is best to leave these jobs some space to grow.

Dynamic Space-Sharing Policy

The basic idea behind the dynamic space-sharing policies is to make the jobs in the system share the processors equally as much as possible [65, 90]. This is achieved by varying the number of processors allocated to an application during its execution. This means processors may be reclaimed from a running job and distributed to newly arrived jobs, or additional processors may be added to an executing job when processors become available. A heavily studied form of dynamic space sharing is equi-partitioning policy [81], where each job is allocated an equal fraction of the processors, up to their maximum parallelism. Equi-partitioning based algorithms have been shown to perform well over a wide range of workloads and a wide range of distributions in service demand [77].

McCann and Zahorjan [82] introduced a policy called Folding for a parallel system

with mesh-based interconnection network. On job arrivals, the scheduler selects a job with the largest existing partition size. It then splits the partition in half and then job's tasks are folded onto half of their previous allocation, and the other half is given to the new job. Processors are rotated between differently sized partitions to give equal average service rates to all jobs. This folding policy is compared with a version of equipartitioning that attempts to allocate partitions with at least one power-of-two dimension, and performs mass reallocation on every job arrival and departure. Folding tends to perform better when the number of jobs in the system is a power of two as rotations are minimized. Also, folding was shown to outperform equi-partitioning policies [82].

Dynamic space-sharing policies are typically used for scheduling malleable jobs in shared-memory systems. The benefits and limitations of dynamic partitioning are discussed in [118]. Malleable jobs are jobs that can have their processor allocation changed during their execution in response to a request by the scheduler. Malleable jobs are not a generally supported programming model in distributed memory system, since often significant programming effort and execution overhead must be expended to change a job's processor allocation during execution. In a distributed memory environment especially, the costs of data repartitioning can overwhelm the scheduling benefits realized by malleable job support.

2.3.2.2 Time-Sharing Policies

In a time-sharing scheme processors are shared over time by executing different applications on the same processors during different time intervals, which is commonly known as time-slice or quantum. Time-sharing policies may be either independent or coordinated. Two independent or uncoordinated time-sharing policies that are extended from the classical round robin policy are the *Round-Robin-Process (RRprocess)* policy and the *Round-Robin-Jobs (RRjob)* [79]. In RRprocess, there is a fixed

quantum per processes. When a processor completes its quantum of service on a process, it goes to the global queue, puts the process at the end of the queue, then takes the process at the head of the queue to service for the next quantum. In contrast, the quantum length of the jobs in RRjob varies and is computed as $\frac{N}{P} \times q$, where q is the unit size of system wide quantum, N is the number of tasks in the job and P is the total number of the processors in the system.

Performance analysis by Leutenegger and Vernon [77] of the RRJob policy shows that RRJob perform better than policies such as FCFS and classic round robin. This is attributed to the fact that RRJob gives roughly equal share of processing power to all jobs, whereas the other policies tend to favor one class of jobs at the expense of another. Ayachi and Dandamudi [19] introduced a modified version of RRjob policy in which each processor has its own ready task queue with a size of known multiprogramming level (MPL) in addition to a central ready queue. The processors pick up tasks from the central queue (i.e., self-scheduling), equal in number to their multiprogramming level. Each processor applies the RR policy only on the tasks that are in their local queues and preempted tasks are inserted back in the processor's local queue and not to the central queue. The performance of the modified version of RRjob is studied in [19] and shown that it performs substantially better than the original policy.

The problem with the uncoordinated time-sharing policies is that they are not good in scheduling communication-intensive jobs. To address this problem, several variations of a static quantum-based parallel processor allocation policies in which each job is configured for a static number of processors and timesharing is used to share the processors among the jobs. Ousterhout's [89] *co-scheduling* policies are examples of this class of policies. In the co-scheduling policy, the tasks of the same parallel job are assigned to a set of processors at the same time and run concurrently across the processors for only certain period of time which is called scheduling slot. At

the end of a scheduling slot, the processors will context-switch simultaneously to give service to tasks of another parallel job. Therefore, all tasks in the queue take turns to receive service in a co-ordinated manner across the processors. The problem with *co-scheduling* policy is that a straightforward implementation requires both global co-ordination and centralized scheduling, which is very difficult to achieve in systems with independent local schedulers studied in this thesis.

2.3.2.3 Hybrid Scheduling Policies

Purely time-sharing policies are in general inferior to purely space-sharing policies when the service time variance is not high. On the other hand, space-sharing policies perform poorly when variance in service time is high. Several policies that combine the time-sharing and space-sharing policies into a single framework have been proposed in the literature. In this section, a review of some of these scheduling policies is presented.

2.3.2.4 Semi-Static Quantum-based Policies

In semi-static quantum-based (SSQ) parallel processor allocation policies, each job is configured for a set of processors determined at the time of scheduling. In addition, timesharing is used to share the processors among the jobs. A class of SSQ policies called *Adaptive Multiprogrammed Partitioning* (MP) was proposed in [86]. The MP policy is adaptive in that the partition size and the degree of multiprogramming can change in response to changes in the workload. Also, they always try to provide equal sized partitions so as to perform better than all the other adaptive policies. MP is compared to non-preemptive ASP [111] policy both quantitatively and qualitatively. Both simulation and analytical studies show that MP achieves better response time, particularly at heavier loads and larger partition sizes. This is attributed to its ability to overlap communication and waiting time of a process with the computation time

of other processes.

Parsons and Sevcik [91] generalized two adaptive space partitioning policies namely ASP [111] and PWS [45] to feedback preemptive disciplines referred to as *FB-ASP* and *FB-PWS* respectively. In FB-PWS and FB-ASP, jobs are assigned processors upon job arrival and departure events as in the original policies. However, every quantum, the jobs that are running are preempted and in their place the jobs with the least acquired processing time are selected to run. Through simulation, the authors show that the feedback versions of the policies lower mean response time than the run-to-completion (RTC) versions for different workloads and offered loads, with more pronounced tendencies as the coefficient of variation (CV) of the job demand increases. The semi-static quantum-based FB-PWS processor allocation defined in [91] is also studied in [33] under synthetic workloads that have not previously been considered.

A scheduling policy called Process Control (PC) is proposed in Tucker and Gupta [125]. Under PC, parallel jobs must be written as multi-tasked applications keeping their tasks in a task queue. The scheduler divides the number of processors in the system by the number of parallel jobs to compute the partition size. The system dynamically makes known to each parallel application the partition size, and the application maintains, as many tasks as the partition size. The process simply removes the tasks from the application's task queue and runs them until they block, at which point they take another task. If more parallel jobs exist than there are processors, the scheduler timeshares processor sets among the parallel jobs.

2.3.2.5 Gang Scheduling Policy

Gang-scheduling policy schedules tasks of a job together and deschedule them at the same time [50]. It uses a hierarchy of controllers to control a multi-user, multiprogrammed parallel system. The processors are partitioned recursively into groups that

are a power of two, as in a buddy system. Each group of processors has a controller, creating a hierarchy of controllers. A new job that requires N processors is allocated to the least loaded controller of size $2 \times \log N$ which has all the processors controlled by it available in some slot. If none of the existing slots can accommodate the new job, a new slot is created.

A variation of gang scheduling involves providing service cyclically among a set of fixed partition configurations, each having a number of partitions equal to some power of two is analyzed in [132, 133]. They find that long jobs benefit substantially from this approach, but only at the cost of longer response times for short jobs. However, achieving gang scheduling can be difficult unless special hardware exists to perform simultaneous context switching across all of the processors. In a tightly coupled MPP, this is easier to accomplish than in a distributed system.

2.3.2.6 Hierarchical Scheduling Policy

A novel hybrid job scheduling policy called Hierarchical Scheduling Policy (HSP) for multi-user, multiprogrammed shared memory parallel systems is presented in [19]. This policy is extended to distributed memory systems [124] and network of workstations [7]. In this system, there is a global queue and each processor has a local queue. In addition, there exists a hierarchy of queues in between the global queue and the local queues forming a tree with all processors with their local queues attached to the bottom level of the tree (i.e., as leaf nodes)[42]. All incoming jobs are placed in the global queue at the root node from which the available processors retrieve work to execute. Each access to a parent queue in the tree transfers a set of tasks/jobs down one level. The actual number of jobs transferred from a parent queue to the child queue in the tree is a function of the *transfer factor*, *branching factor* and *partition reach*. A tunable parameter called the transfer factor controls the number of tasks transferred from a parent queue to the child queue, branching

factor denotes the number of children of a node and partition reach is the number of processors reachable from a given node. HSP was analyzed both quantitatively and qualitatively for shared memory systems and distributed systems and shown that it eliminates/minimizes the drawbacks associated with both centralized and distributed co-ordination while preserving their advantages.

Performance analysis of HSP against Equipartitioning and RRjob policies shows that HSP performs better than the other two policies specially for medium to high system loads. At high system loads, the performance of Equipartitioning policy is observed to be bad.

2.4 Grid Computing

Grid computing is a type of parallel and distributed system that enables sharing, selection, and aggregation of geographically distributed autonomous resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements [22, 41, 70]. Grid systems can be classified depending on their usage as follows [70]:

1. **Computational Grid:** Denotes a system that has a higher aggregate capacity than any of its constituent machine. Computational grid can be further categorized based on how the overall capacity is used as follows:
 - (a) **Distributed Supercomputing Grid:** Executes the application in parallel on multiple machines to reduce the completion time of a job. For example, Grand challenge problems typically require a distributed supercomputing Grid.
 - (b) **High Throughput Grid:** This type of grid increases completion rate of a stream of jobs arriving in real time. For instance, processor design verifications tests would be run on a high throughput Grid.

2. **Data Grid:** Grid systems that provide an infrastructure for synthesizing new information from data repositories such as digital libraries or data warehouses. Applications for these systems would be special purpose data mining that correlates information from multiple different high volume data sources.
3. **Service Grid (SG):** Systems that provide services that are not provided by any single machine. SG systems can be subdivided based on the type of service they provide as follows:
 - (a) **Collaborative Grid:** Connects users and applications into collaborative work-groups (i.e., enables real time interaction between humans and applications via a virtual workspace).
 - (b) **Multimedia Grid:** Provides an infrastructure for real time multimedia applications – requires the support of quality of service across multiple different machines whereas a multimedia application on a single dedicated machine can be deployed without QoS synchronization between network and end-point QoS.
 - (c) **Demand Grid:** This category dynamically aggregates different resources to provide new services. For example, data visualization workbench allows a scientist to dynamically increase the fidelity of a simulation by allocating more machines to a simulation.

Larger scale grids in the "real world" typically exhibit a number of different properties:

1. Nodes in the grid are typically scattered across a number of different administrative domains.
2. Each domain will typically have their own resource allocation policy, determining for example, if resources must be surrendered to local users.

3. Each domain will have their own job queuing system, their own access cost, and various amounts of available computational power.

The current focus in grid computing is on building infrastructures (e.g., resource selection services for locating Grid resources that match application requirements) intended to provide user-level services for building and running wide-area applications. Globus [64] and Legion [32] are two well known Grid management systems.

2.4.1 Resource Management Systems

Figure 2.2 presents the core Grid architecture, which provides mechanisms for Grid clients having access to diverse Grid computing resources (e.g., supercomputers and clusters of workstations) managed by a wide range of local resource management systems (e.g., LSF [106]). Using the Grid client application program interface (API), Grid clients can request computations on specific Grid computational resources from the Grid broker.

Figure 2.2: Architecture of a typical grid computing system

Globus [64] provides the basic infrastructure with capabilities and interfaces for

communications, resource location, and data access. For instance, it provides a standard network-enabled interface to local resource management systems. Resources and computation management services are implemented in a hierarchical fashion. Note that Globus does not make resource allocation decisions, instead the objective is to build toolkits on top of the existing systems.

Legion [32] is a meta operating system that utilizes host operating system services to create the illusion of a single virtual machine. Legion addresses challenging issues in Computational Grid research such as parallelism, fault-tolerance, security, autonomy, heterogeneity, legacy code management and so forth. It uses the basic resources (Hosts and Vaults), the information database (Collection) populated with information describing the resources, the schedule implementer (Enactor), and the Scheduler that computes the mapping of objects to resources. Resource selection is performed using resource availability and affinity, which is an extension of Condor [23, 113].

Legion also has the concept of a jurisdiction magistrate that contains and enforces the local policies. It is this object, which handles all placement failures, security breeches and querying of placement decisions. Fault tolerance is addressed by encapsulating fault tolerance protocols in base classes, which may be extended by users. Latency-tolerant, relatively large-grained parallel jobs are targeted. Programs are represented by graphs (or DAGs) and parallel execution is based on a macro data flow model, where edges in the graph denote dependence relations between nodes, which represent operations (or actors). Memory within Legion is treated as a single persistent object space. Scheduling in Legion is static in that sufficient resources must be available before the placement takes place.

As Grids mature, they will play a role similar to today's production parallel supercomputers serving a variety of high-end scientific applications including a mixture of traditional regular parallel applications (e.g. single MPP-style applications) and ensemble applications. Efficient scheduling of Grid resources across this mixed work-

load becomes crucial to achieving high performance for both application classes. The following section reviews some of the research in this direction.

2.4.2 Job Scheduling on Grid Computing Systems

Although Grid infrastructures such as Legion and Globus provide the basic resource selection functionality, work allocation functionality, and scheduling mechanisms, applications must interpret system performance information in terms of their own requirements in order to develop performance-efficient schedules. Therefore, efficient usage of Grid resources require a powerful and flexible Grid scheduling. The successful deployment of Grid technology must deliver automatic features in order to determine available Grid resources and to coordinate the allocation of these resources in accordance with the requirements, dependencies and objectives of the user. Unless significant support is provided from a Grid management system the necessity of manually performing this task would significantly slow down the proliferation of computational Grids [58]. The definition of a scheduling architecture is becoming necessary which supports the cooperation of different scheduling instances for arbitrary resources available in the Grid. Especially, the interaction between local resource management (e.g., PBS) and data management shall be considered in such an architecture. While most approaches are focused on compute resources with local batch schedulers in mind, a general Grid scheduling architecture must at least include network, software, data, storage and processing units.

Ideally, a Grid management system includes a separate scheduling layer that collects the Grid resources specified in a job request, checks the compliance with all necessary conditions, such as security requirements, and interacts with the local scheduling systems of the individual Grid resources. Hence, the paradigm of such a Grid scheduling layer will deviate significantly from that of local or centralized schedulers. While local scheduling usually involves a single scheduling instance that

has access to all system information, Grid scheduling requires interaction with remote sites and their local scheduling systems. It is therefore obvious that a Grid scheduler will consist of more than a single scheduling layer. However, the details of such a scheduling architecture have not yet been established.

Parallel job scheduling research in computational grids is slowly but surely increasing [58, 134, 115, 26, 127]. In [134], a static model for scheduling in wide-area systems is described. The model is federated and utilizes a collection of local site schedulers that control the use of their resources. The scheduler allocates and schedules tasks defined by a static task graph onto a set of networked computational resources. Resource allocation is implemented by a scheduling manager, which coordinates scheduling requests and a local manager, which manages the resources at a local site. Each site manager keeps track of the locally available resources by periodically probing them. Jobs are submitted to the site manager that is responsible for locating and allocating resources.

Resource selection is performed by attempting to minimize the execution time of task graph as predicted by a performance model for the application and the prospective resource. However, as the minimization procedure and the cost model are static, there is no support for policy extensibility. Another drawback is that the scheduler depends on the ability to reserve resources. There is very little performance analysis of the policy especially under various system load conditions and other parameters. The published results apply to one cluster and one job situation, which is hardly reflective of the networked distributed systems, for which the policy was supposedly designed in the first place.

The AppLeS project [63] is an application management system which sits on top of grid middleware, such as Globus [64] and Legion [32]. AppLeS uses static and dynamic application and system information to select viable resource configurations and evaluate their potential performance. Static information is gathered through agents,

where each agent is responsible for a particular task. For example, an agent uses a "Heterogeneous Application Template" (HAT) to allow a user to specify the structure, characteristics and the current implementations of the application that needs to be run and its tasks. The "User Specification" agent is also used to collect information on the users performance, timing and other criteria. With respect to dynamic information, AppLeS uses the Network Weather Service (NWS) to monitor the varying performance of resources potentially usable by their applications. AppLeS then interacts with the relevant resource management system to produce a customized schedule for each application waiting to be scheduled. The scheduler called Application-Level Scheduling (AppLeS) [26] attempts to maximize application performance, measured as runtime or speedup, with little regard to overall system performance. Therefore, depending on the characteristics of the application being scheduled, different amount of information is required. The main objective of AppLeS is to optimize the performance of an individual application rather than to cooperate with other applications sharing the resources. Since multiple users share the resources, optimizing the performance of an individual application may dramatically affect the completion time of other applications.

A number of computational economy based scheduling systems have also been discussed in the literature [67, 57, 43]. For example, Nimrod [44] is a grid scheduling system for parametric computing problems across a set of computing resources on the basis of a computation economy where by the user could determine the available budget and the priority of the job. It is suggested in [44] that Nimrod worked successfully for static sets of computational resource but that it exhibited a number of flaws which rendered it unsuitable when implemented in the large-scale dynamic context of a computational grid. Nimrod/G [43] is an extension of Nimrod [44]. It utilizes a "Job Wrapper", which is responsible for the staging of application tasks and data. The "Job Wrapper" acts as a mediator between the parametric engine and the actual

machine on which the task runs. Its primary function is to send the results from the remote client back to the main engine. It is therefore conceivable that it could also monitor the resource utilized by the remote client and report that data to the resource accounting system. Nimrod/G supports an integrated computational economy in its scheduling system. This means that Nimrod/G can schedule jobs on the basis of deadlines and budget. Therefore users can make assertions such as: "Please complete this task by date X and within budget Y", "If you can complete task X by date Y I shall provide reward Z".

The key difference between traditional systems and market-based systems is that in the latter case resources will be allocated not only on the basis of available capacity but also on the basis of affordability. In other words, users will not necessarily purchase resources on the basis of their availability. They might purchase the cheapest resources (thus allowing them to purchase as many resources as possible) or they might purchase expensive resources (perhaps hoping these are of better quality). The problem is that the efficiency depends on the users ability to assess their needs and then make rational decisions that maximize their utility.

2.5 Cluster Computing Systems

2.5.1 Cluster Management Systems

As local area networks spanning large numbers of powerful workstations become commonplace, researchers have come to realize that at most sites, many nodes are idle much of the time [14, 104, 84, 85]. Ideally there should be a way to recapture some of these lost cycles, which grow increasingly formidable in the aggregate as workstations grow more powerful. A cluster manager is the name given to the software that provides such management functions and abstractions that allow clusters to be treated as a single computing resource. Several cluster management systems (CMSs)

are available for queuing and scheduling jobs on clusters of networked computers, including Beowulf clusters. They all provide queuing software to allow users to easily submit batch jobs to run on the cluster, a scheduler to organize for the job to be run using appropriate compute resources in the cluster, and a loader to start the job running on the specified node.

Figure 2.3: Architecture of a typical batch queuing system

An idealized CMS is shown in Figure 2.3. Jobs are submitted to a centralized Queue Master. The user characterizes application resource requirements either explicitly by some type of job control language as in ClassfiedAd [23] or implicitly by selecting the queue to which a request is submitted as in [106]. The submitted job along its resource requirements is then analyzed by the Queue Master. One of the main functions of the CMS is to determine the state of the resources (Idle or available), enforce usage policy and task management (e.g., checkpointing and process migration). Many systems provide process migration component for relocating a running process from one machine to another, and allowing the transplanted process to continue execution on the new host. The classical mechanism for detection of ma-

chine availability is through periodic review. In the periodic review approach, the system is monitored at discrete, equally spaced instants of time. Unfortunately, this method increases the overhead on the scheduling policy, non-scalable and intrusive in a sense that it impacts on the execution of the application, as the nodes must halt the execution in order to respond to the state request. In Chapter 3, we will discuss a different strategy that ameliorates the problems associated with the periodic review strategy.

There are many different type of CMS. Some of these cluster management systems are:

1. **Condor** is a high throughput computing (HTC) system that manages large collections of distributively owned workstations [23]. Condor runs as a user level process monitoring the usage of each workstation in the cluster. Idle workstations are placed into a resource pool and are available for execution of batch jobs. The owner of a workstation has complete control over its usage in the cluster and has the highest priority on it. The system can migrate jobs from one workstation to another; therefore, if its owner reclaims a workstation, a batch job running on that workstation is stopped and migrated to other workstation in the cluster. For checkpointing running jobs, Condor requires the users' code to be linked with its own software libraries. Since Condor uses only idle machines and will migrate a job from a machine if the workstation user begins to use this machine, the system throughput achieved by Condor will be limited.
2. **Load Sharing Facility (LSF)** supports many advanced features such as transparent access to heterogeneous resources across the network, scheduling serial, parallel and interactive workloads, process migration, network wide load monitoring and load balancing [106]. Hosts are grouped into clusters (either physical or virtual) and all machines are considered shareable. Within a cluster, a mas-

ter host is in charge of the system and application information such as *CPU load, memory, disk accesses* and so forth.

LFS provides two different ways to share resources between clusters: *Job Forwarding (JF)* model and *Resource Leasing (RL)* model. The JF model is a sender-initiated model in which overloaded clusters offload jobs onto underloaded clusters, whereas the RL model is a receiver-initiated model in which underloaded clusters offload work from overloaded clusters.

LSF system is a more scalable system that runs on hundreds of workstations. The system also has some application fault-tolerance provisions. As long as a single node is up, LSF is available and no jobs are lost even when the rest of the nodes are down. Also, the system supports distributed filesystems such as AFS and DFS.

There are a number of limitations. For example, a parallel job can be forwarded to another cluster, but the job cannot start unless the execution cluster has enough hosts and resources to run the entire job (i.e., the static space-sharing). Another limitation is that a parallel job cannot span clusters (i.e., no co-allocation) even if there are resources that can satisfy the requirements of the job across multiple clusters.

3. **LoadLevler** is a commercial workload management and job scheduling system developed by IBM [113]. It is based on the Condor system [23] to which many new features were added, making it much more versatile. Current versions support load balancing across the workstations in the cluster, multiple resource queues, serial and parallel workloads, and distributed file systems (AFS, DFS). It also provides a well-documented application-programming interface for customizations and extensions to the system.
4. **Portable Batch System (PBS)** was developed at NASA to support the

programming environments at all of the NASA centers [62]. It was designed to handle the unique demands of the heterogeneous computing clusters at NASA research centers. PBS uses an independent scheduling module, which allows the system manager to define what types of resources and the amount of each resource that can be used by each job. It supports both serial and parallel computing environments, allowing for multiple queues and interactive usage. The system also provides for file staging in which user required files are made available on the remote executing machines either by remote mounting of the file system automatically or ‘ftping’ the files over the network.

In general, network batch queuing systems are not useful for a multi-cluster environment, which is the main focus of this thesis. Also, they do not individually schedule the programs that comprise a job; they merely place the job onto a sufficiently unloaded server. The only form of scheduling they perform is the matching of available computational capacity with the job’s requirements.

2.5.2 Work Stealing Approaches

In this section, we review the work related to work stealing concept as this pertains to the work of this thesis. We wish to note that the goals of these systems are to create a programming environment in which parallel applications are easy to implement and executed in a load balancing fashion. In contrast, we use it for parallel job scheduling as in [19, 124].

Random Stealing: In the *Random Stealing* approach [69], an idle node N_i randomly selects a node N_j in the system and sends a steal request. When N_j receives the steal request, it can respond to it by sending work to N_i or negatively. In the later case, N_i repeats the steal attempts until it succeeds. For single cluster systems, Random Stealing (RS) is known to achieve optimal load balancing. However, RS is inefficient

when applied to hierarchical wide-area systems where multiple clusters are connected via wide-area networks (WANs) with high latency and low bandwidth [69]. Second, the policy is oblivious to the load level of the local clusters as jobs can potentially be brought to the cluster while there may be work available inside the local cluster. Also, since the steal attempts are synchronous, the stealing processor has to wait for response for the results, while there may be work available inside the local cluster. This leads to low system utilizations (due to high idle time).

Random Pushing: In the *Random Pushing (RP)* approach [69], a threshold is associated with each node and any time the current load on the node exceeds the threshold, the excess jobs are sent to a randomly chosen peer in the system. A problem with the *RP* is that under high system loads job pushing causes useless overheads as all nodes already have work. Like the *RS*, the Random Pushing also suffers from excessive communication overheads.

Cluster-Hierarchical Stealing (CHS): The *Cluster-Hierarchical Stealing (CHS)* [20] arranges processors in a cluster in a tree topology, and idle nodes send steal messages along the edges of the tree to its children. If the children are also idle, steal messages will recursively descend the tree. Only when the entire subtree is idle, steal messages will be sent upward in the tree asking parents for work. Compared to the *RP* and *RS* approaches, the *CHS* sends fewer inter-cluster messages. However, it suffers from bad performance inside clusters and stalls the entire cluster when inter-cluster steals are issued. Also, it exhibits much locality as work inside a subtree will always finished before the steal requests are sent to the parent of the subtree. CHS has two main drawbacks. First, the root node of a cluster waits until entire cluster becomes idle before starting the inter-cluster steal attempts. During the round-trip time of the steal request, the entire cluster remains idle. Second, the

preference for stealing further down the tree results in jobs with finer granularity to be stolen first, leading to high LAN communication overhead, due to many job transfers.

Cluster-aware Stealing: In the *Cluster-aware Stealing (CAS)* approach [69], an idle node sends a steal request to two randomly selected nodes in the system such that one of the nodes is a remote node while the other one is within the same cluster. Cluster-aware Random Stealing (CAS) is highly efficient and easy to implement. CAS adapts itself to network conditions and job granularities, and does not require manually-tuned parameters. It has at most 4% overhead in run time compared to RS on a single, large cluster, even with high wide-area latencies and low wide-area bandwidths [69]. The problem with *CAS*, however, is that it sends more steal requests than the other approaches. Also, CAS performs poorly when the bandwidth requirements of the applications are too high. In addition, if the problem size is too small, then presumably there is not enough work to keep all CPUs busy, resulting in excessive inter-cluster control messages.

2.5.3 Job Scheduling on Cluster Computing Systems

2.5.3.1 High Throughput Cluster Computing Systems

The scheduling policy used in HTC systems is commonly known as *opportunistic scheduling* [137, 2, 1, 59]. A non-preemptive self-scheduling policy called *Eager Scheduling* is discussed in [22]. It is shown that Eager Scheduling works very well in situations where CPU speeds on the different machines are not equal, the machines have dynamically varying loads, the machines are prone to failure, and new machines can join the computation. However, eager scheduling produces sub-optimal turnaround times in cases where the number of tasks is slightly higher than the number of processors. Similar approach is used in Piranha system.

The *Piranha system* harnesses workstation idle cycles to run explicitly parallel

programs [49]. It is an adaptive version of master-worker parallelism. The application defines a worker function designated by the distinguished name `piranha()`. Whenever a node becomes idle, the system creates a new worker process on that node, using the `piranha()` function as a template. The `piranha` application must also define a function called `retreat()`, invoked automatically by the system on any node that stops executing `piranha()` and goes back to its owner's commands. Finally, a `feeder()` user-defined routine is included that runs on the submitting node and manages the ongoing computation.

In *Up-Down* policy [84], each job is tagged with a user id and the system keeps track of the total CPU time allocated to each user. When jobs are to be scheduled, jobs belonging to the user with the least total CPU allocation are given priority. Within a user's job scheduling of the jobs is done on the basis of FCFS. When a workstation owner starts using it, the job executing on the workstation is evicted from the workstation. The evicted job is placed at the head of the queue so that it is scheduled as soon as possible. In Ghare and Leutenegger [59], an extension of the Up-Down [84] scheduling policy is discussed. The job rotate (JR) policy is identical to the UP-Down [84] policy except that instead of adding an evicted job at the head of the queue, it is added at the end of the queue. Simulation analysis shows that the JR policy perform substantially better than the default FCFS policy used in Condor [84] with respect to the mean slowdown while there was no gain in the mean response time.

Another scheduling policy called MultiQueue-based Scheduling (MQS) is discussed in [1]. In MQS, jobs that have arrived into the system for execution are classified into new jobs and evicted jobs, where new and evicted mean that the new job has not received any service since arrival while evicted refers to the fact that the job has received some service. To represent these two classes of jobs, the MQS policy keeps two queues: one for holding new jobs and the other for holding the evicted jobs. We

refer to these queues as New and Evicted respectively. The New queue entries are sorted in the order of the arrival while entries in the Evicted queue are sorted in descending order based on the CPU service received by the jobs. Scheduling is done such that the job at the head of the New queue is always scheduled first and only if the New queue is empty then the job at the head of the Evicted queue is assigned to the workstations. Through simulation, it was shown that these scheduling policies outperform both JR and RR policies.

Ryu and Hollingsworth [105] present a scheduling policy called Linger-Longer that exploits the fine-grained availability of workstations to run sequential and parallel jobs. They present a two-level workload characterization study and use it to simulate a cluster of workstations running their policy and compare two variations of their policy to two previous policies: Immediate-Eviction and Pause-and-Migrate. They show that the Linger-Longer policy can improve the throughput of parallel jobs on a cluster by 60 percent with only a 0.5 percent slowdown of local jobs. For parallel computing, they show that the Linger-Longer policy outperforms reconfiguration strategies when the processor utilization by the local process is 20 percent or less in both synthetic bulk synchronous and real data-parallel applications.

In Ghare and Leutenegger [59], a scheduling policy that resembles the round robin policy (RR) is discussed. The round-robin policy (RR), each job executes for a given quantum length and then preempted and is added at the end of the queue. A job at the head of the queue is then scheduled. Simulation analysis of the RR policy shows that it performs substantially better than the default FCFS policy used in Condor [84] with respect to the mean slowdown while there was no gain in the mean response time. The major problem with this policy is that evicting jobs based on quota is really of theoretical interest and may not be feasible to implement.

To address this, a scheduling policy called Proportional Local Round Robin (PLRR) is proposed in [2] is discussed. The PLRR policy associates with each processor a

local ready queue, a quantum length and a multiprogramming level (MPL). Each processor applies the RR policy only on the jobs that are in their local queues and the preempted jobs are inserted back in the processor's local ready queue and not to the central queue. In addition, a variable quantum length is assigned to the jobs in the local ready queue based on the proportion of CPU consumption since the arrival of the job to the system. Simulation results show that PLRR policy performance is several magnitudes order better than the FCFS policy [84] and the RR policy [59].

2.5.3.2 High Performance Cluster Computing

Anastasiadis and Sevcik [121] proposed a scheduling policy called Adaptive Equipartition Policy (AEP) for networks of workstations environment based on the assumption that all processors in the system have equal processing power and dedicated to parallel jobs. In AEP, when a job arrives into the system, it is allocated a number of processors equal to the minimum of its maximum parallelism, the target partition size and the number of available processors. The target partition size is computed as the total number of processors divided by the total number of jobs, both waiting and running, in the system.

In the Maui Scheduler [66], if there is lack of resources to start the next job in line, the scheduler makes a reservation of the requested resources in the future. The notable difference from most other scheduling efforts is the presence of these advance reservations in a future schedule, taking the dimension of time into account. The schedule which is recreated repeatedly by Maui resembles a meeting calendar where the meetings are jobs and the scheduler makes constant efforts to fill every gap but still paying respect to the job priorities. The authors claim that the Maui scheduler provides a surprisingly good scheduling strategy for Beowulf clusters.

In [22], three preemptive algorithms called Optimal Algorithm, Fault-tolerant Round Robin Algorithm and Preemptive Task Bunching Algorithm are presented. All

these three algorithms require the existence of task migration facility. The Optimal Algorithm is targeted for situations where the number of tasks to be executed is slightly larger than the number of machines available. This algorithm pre-computes a schedule that minimizes execution time and number of context switches needed. However, it requires that the task execution time be known in advance and therefore is not always practical. The Distributed, Fault-tolerant Round Robin (DFTRR) algorithm is a variation of the well-known round-robin algorithm. It is designed for a system state in which a set of n tasks scheduled on m machines, where $n > m$. Initially, the first m tasks are assigned to the m machines. Then, after a specified time quantum, all the tasks are preempted and the next m tasks are assigned. This continues in a circular fashion until all tasks are completed.

The Preemptive Task Bunching (PTB) implicitly assumes that either the machines are heterogeneous or the n tasks are not of equal length (i.e. the tasks do not have an equal service demand). In PTB, all n tasks are bunched into m bunches and assigned to the m machines. If one of the machines in the system finishes its assigned bunch, all the tasks on all the other machines are pre-empted and all the remaining tasks are collected, re-bunched (into m sets), and assigned again. These policies suffer from the same problems as RRprocess and RRjob [79] discussed previously.

In [18] two extensions of stride scheduling policy for NOWs called Loan & Borrow and System Credit are discussed. Both approaches are built upon exhaustible tickets, which are simple tickets with expiration time. In Loan & Borrow approach, exhausted tickets are traded among competing clients. When a user temporarily exits the system, other users can borrow these otherwise inactive tickets. The borrowed tickets expire when the user rejoins the system. When the sleeping user wakes up, it stops loaning tickets and is paid back in exhaustible tickets by the borrowing users. This policy can keep the total number of tickets in the system constant over time; thus, users can accurately determine the amount of resources they receive.

With system credits, clients are given exhaustible tickets from the system when they awaken. The idea behind this policy is that after a client sleeps and awakens, the scheduler calculates the number of exhaustible tickets for the clients to receive its proportional share over some longer interval. The system credit policy is easy to implement and does not add significant overhead to the scheduler on sleep and wakeup events.

Two similar co-scheduling policies for scheduling communicating processes in a cluster of workstation environment have been proposed in [117, 48]. The implicit co-scheduling [48] uses spin-block synchronization primitives and the priority boost provided by the SVR4 scheduler. By observing and reacting to implicit information, local schedulers in the system make independent decisions that dynamically coordinate the scheduling of communicating processes. Their results show that this mechanism to be competitive with explicit co-scheduling; a maximum slowdown of 30% is observed.

The dynamic co-scheduling [117] achieves co-scheduling among communicating processes of a parallel job based on local events that occur naturally within the communicating applications by observing message arrival at a processor and speculatively assumes that the sender is active and will probably send more messages in the near future. Each local scheduler makes independent decisions that dynamically coordinate the scheduling actions of cooperating tasks across processors.

2.6 Coordinated Allocation of Multiple Resources

Most of the contemporary research on job scheduling concentrates on processors only. In reality, however, jobs require other resources as well, and the interaction between the demands for different resources can have a significant effect on possible schedules. In this section, we present a review of work related to coordinated resource scheduling.

2.6.1 I/O Resource Considerations

One of the important characteristic of applications that has a significant impact on scheduling is I/O activity. Although we are beginning to see consideration of I/O in scheduling work, quite often it is simply ignored. As applications obviously do perform I/O (and sometimes even a lot of it), this is a severe deficiency in current practice. Recently, I/O bottleneck has been receiving attention and research is being pursued in several fronts that can facilitate the design of scheduling policies. These areas are briefly discussed next.

Application I/O Characterization: Several researchers looked at the access patterns of I/O-bound applications [114, 100, 71, 93]. Also, basic issues underlying the characterization of I/O behavior of applications is discussed in a number of papers including [80, 99, 83, 114, 100, 71, 93]. In addition, models of parallel applications with large computation and I/O requirements are discussed in several papers including [100, 71]. The general consensus is that:

1. I/O activity in parallel jobs tends to be regular, recurrent and predictable (e.g., [93]).
2. Applications tend to access data in chunks ranging from 32 KB to 512 KB and the access sizes are relatively constant [83]. Applications with access sizes different from this range is also discussed in a number of papers including [114, 100, 93].

Management of I/O Resources: The bottleneck and performance factors' of parallel I/O in a cluster environment is addressed in [123]. In [80, 71], management of I/O in multiprogrammed parallel systems is discussed. Through simulation, they demonstrate a strong relationship among I/O characteristics of applications, I/O management strategies, and system performance. The problem of scheduling I/O in

multiprogrammed parallel systems is discussed in [71]. Using I/O characteristics of jobs such as the total outstanding I/O demand is observed to be useful in devising effective scheduling strategies. Applications that can simultaneously do computation and I/O and the strategies that can incorporate centralized I/O management are found to be beneficial for a multiprogrammed parallel environment. In [35], heuristics to choose the number of I/O servers and placement on physical processors in a way to better exploit disk I/O resources and minimize total I/O time in heterogeneous cluster environments is discussed. Several approaches to minimizing I/O-related overheads in the context of Grid computing has also been proposed [63, 46, 97].

Architecture: In [46], a system that links jobs and data by binding execution and storage sites into I/O communities that reflect the physical representation of the resources is discussed. As they present building blocks for such communities but do not address policy issues, this work is complementary to our work on scheduler policy. The same comment applies to work on Execution Domains [46], a framework that defines bindings between computing power and data resources in a grid such that applications are scheduled to run at CPUs that have access to the required data and storage.

Mechanisms to enable programs to access data in a manner independent of data and program location is also one of the fronts research is proceeding. For example, in [56], a file access paradigm for remote parallel file systems called RIO is described. In RIO approach, programs use *remote I/O library* to access files located on remote filesystems in a manner that is independent of physical location.

Integrated Processor and I/O Scheduling: The interactions between gang-scheduling and I/O-intensive jobs are discussed in [74] and showed that the CPU can remain severely underutilized during time-slices dedicated to I/O intensive tasks.

Scheduling policies for I/O intensive applications on a parallel machine with parallel disks is examined in [99]. They show that performance improves if the I/O subsystem is explicitly scheduled as well [99]. Casanova et al. [63] describe an adaptive scheduling policy for parameter sweep applications in Grid environments that takes data storage issues into account. Their approach is to place files strategically for maximum reuse. Casanova et al. [63] make scheduling decisions centrally, assuming full knowledge of current loads, network conditions and topology.

Heuristics like Max-min and Min-Min are used for Level-by-Level scheduling of DAGs in [15]. They consider data location as a parameter but assume that resource performance characteristics are perfectly predictable. What sets our work apart from other scheduling research is that we consider dynamic data replication as a fundamental part of the dynamic scheduling problem.

2.6.2 Memory Considerations

Since memory is also a critical resource for many parallel jobs, the allocation of memory and processors must be coordinated to allow the system to operate most effectively. Feitelson [51] studied the memory requirements of parallel jobs in a CM-5 environment. He found that memory is a significant resource in high-performance computing, although he observed that users typically request more processors than naturally correspond to their memory requirements. Parsons [92] studied bounds on the achievable system throughput considering memory demand of parallel jobs. Peris et al. [94] developed analytical models for studying processor allocation trade-offs while taking into account the impact of memory requirements. The effects of memory performance on parallel job scheduling is discussed in [122].

Setia et. al. [109] studied the impact of job memory requirements on gang scheduling performance. The authors claim that, counterintuitively, having more physical memory may lead to reduced performance, because more large batch jobs may fit in,

causing problems for shorter interactive jobs. The solution is to limit the degree of multiprogramming, and specifically, allow only one batch job at a time.

In [60], the problem of co-scheduling under memory constraints in a NOW environment is discussed. Batat [25] improved gang scheduling by imposing admission control based on the memory requirement of a new job and the available memory of a system. This modified gang scheduler estimates the memory requirement for each job and assigns a job into a time slice only if the memory is large enough for all jobs in the time slice.

McCann and Zahorjan [82] propose and evaluate several scheduling policies that take into account the constraints imposed by the memory requirements of parallel applications. Each job has a minimum processor allocation due to its memory requirement. They find that a discipline based on buddy system allocation consistently does well.

Setia [110] studied the constraints imposed by memory requirements of jobs under adaptive partitioning policies for scheduling parallel jobs on distributed memory parallel computers. The results show that these constraints can have a negative impact on the performance of adaptive partitioning policies. Setia also show that virtual memory offers little benefit to space-sharing policies as any performance benefit resulting from the easing of minimum processor constraints imposed by the memory requirements of jobs will be negated by the overhead due to paging.

2.7 Fault-Tolerance Resource Allocation

Large scale clusters gather thousands of nodes and there is a significant probability of failures in the nodes (perhaps due to hardware or software errors that require rebooting of particular PCs) or network components. This volatility reduces the mean time between failure (MTBF) of the whole system to hours or minutes [28].

Functionally, the probability of failure increases exponentially with the number of nodes. Therefore, the need to incorporate reliability into the design of a cluster becomes imperative.

2.7.1 Types of Failures

A system can experience several kinds of failures, which can be classified according to their locality (processor or link) and their nature as follows:

- **Permanent failures:** In the case of a processor, it stops executing its program forever. In the case of a communication link, this can be interpreted as a definitive rupture of the communication service.
- **Intermittent failures:** These are more difficult to cope with. A processor can have a Byzantine behavior, and a communication link may lose, duplicate, reorder or modify messages in transit.
- **Transient failures:** For a processor, such failures may occur following a crash and repair operation, or temporary shutdown of its power supply. For a communication link, electromagnetic fields may lead to similar problem.

2.7.2 Fault-Tolerance Mechanisms

Numerous research groups have been examining the problem of fault tolerance for parallel applications in distributed networks. There have been three major mechanisms:

1. **Checkpoint-recovery:** In checkpoint-recovery approach, parallel applications typically instrument themselves with the ability to checkpoint their computation states to stable storage. When one or more processors on which the application was running fails, the application may be restarted from the most

recent checkpoint, thereby reducing the amount of re-computation that must be performed.

2. **Job replication:** The job replication approach allows multiple copies of a job (task) to be scheduled on different processors either simultaneously (active replication) or one replica, called the primary replica, is active while other replicas, called backups, are in a stand by mode (passive replication). The backup is only activated if a fault occurs while executing the primary replica.
3. **Process groups:** The process group approach allows for groups of programs (possibly distributed) to cooperate and work together in order to achieve some goal or task to achieve reliable distributed computing [27].

Replication techniques are becoming popular approaches to provide fault-tolerance capabilities in distributed systems. For example, an active replication for fault-tolerance is used in [73]. An active replication requires tighter synchronization among the replicas than do passive replication techniques. Thus, they are ill suited for an environment in which communication latency can be high. On the other hand, passive replication can tolerate non-deterministic servers and uses little processing power when compared to the other replication techniques. However, passive replication suffers from a high reconfiguration cost in case of the failure of the primary.

An example of a system that uses checkpoint and rollback approach is the Fail-safe PVM [75]. It is an enhancement to PVM and masks fail-stop, single-node failures of the application. Checkpointing is a feasible technique for tolerating transient failures and to avoid total loss of work on small-scale systems.

An example of a system that uses *Process groups* is the Nile system [73]. The target of the process groups approach is that applications require a very strong consistency model. It is interesting to see that the performance degradation experienced by the system outweighs the benefits that may be gained by targeting applications that do

not require such a strong consistency model [73]. Moreover, in many cases, the time needed to develop and debug the fault-tolerance function was at least five times as much as was required to develop the core function [73].

The Chamelon [68] implements different levels of fault tolerance, depending on the penalty that the application-writer is willing to pay.

2.7.3 Fault-Tolerant Scheduling Middleware

Although fault-tolerance is an active area of research in cluster computing environments, the issues of providing fault tolerance have generally been addressed separately from the issues of job scheduling. The LSF [106] has some measure of system-level service fault tolerance built into the system. In LSF, if the master host (which makes scheduling decisions) fails, another host in the cluster is automatically voted to be the master. Hosts are elected to be the master in the order they appear in a static file, which must be visible to all machines in the cluster. In the event that the cluster becomes partitioned by network failure, the partition that has access to the LSF [106] log files continues working, while the remaining partitions sit idle. Therefore, there is an effort going on in solving this problem but so far it is not completely resolved.

2.8 Summary

In this chapter, a survey of the current state of art in parallel job scheduling is discussed. The scheduling literature provides solutions for a number of special cases. The objectives and the issues that arise in the design of a parallel job scheduling system for cluster computing were described. Unfortunately, the existing research considers a collection of heterogeneous systems and assume that either: 1) tasks can run on every machine in the processor pool; 2) nodes are dedicated; 3) the nodes do not fail; 4) a centralized scheduler is acting in isolation and is in possession of up-to-

date global state information. In the final analysis, there are no scheduling research efforts that approach the problem in the same manner as the work carried out in this thesis.

There is a certain amount of synergy between the research discussed in this chapter and our proposed work in that the techniques employed by these systems provide guidelines for what is needed in a general-purpose job placement framework design. Also, it is evident from the above discussion that the reliability provision for the scheduling subsystem, with a few exceptions, is largely ignored in the existing cluster-based distributed systems. Moreover, most research in the area has focused solely on the allocation of processors to jobs. However, since I/O is also a critical resource for many parallel jobs, the allocation of memory and processors must be coordinated to allow the system to operate most effectively. Last but not least, the issues of providing fault tolerance have generally been addressed separately from the issues of job scheduling.

Chapter 3

Dynamic Hybrid Scheduling Policy

In this chapter, we address the problem of parallel job scheduling on multi-cluster computing systems. We propose a dynamic scheduling policy and evaluate its usefulness under various system and application parameters¹. We also compare the proposed policy with modified adaptive space-sharing and modified time-sharing policies².

3.1 Introduction

3.1.1 Motivation

The research question addressed in this chapter is the on-line job scheduling problem for multi-cluster systems. There are a number of motivating factors for studying this problem. First, multi-cluster systems are gaining more and more importance in practice [30]. Second, with its growing popularity, a wide variety of applications are being hosted on such systems as well. Apart from the scientific applications that

¹Jemal H. Abawajy and S. P. Dandamudi, "Distributed Hierarchical Workstation Cluster Coordination Scheme," *Proceedings of the IEEE International Conference on Parallel Computing in Electrical Engineering (PARELEC'00)*, page: 111-116, 2000.

²J. H. Abawajy and S.P. Dandamudi, "Parallel Job Scheduling on Multi-Cluster Computing Systems," *Proceedings of IEEE International Conference on Cluster Computing (Cluster 2003)*, Hong Kong, China, December 1- 4, 2003

have traditionally needed supercomputing power, a large number of commercial applications such as database, web service, and e-commerce have started using clusters. These types of applications can benefit from a large number of machine resources often spanning *multiple clusters* across the network. Therefore, when a large number of computers are to be shared by a variety of users, proper scheduling and resource allocation becomes a critical issue. The system may have powerful machines and interconnected by a high-speed network, but poor choice of a scheduling algorithm will lead to sub-optimal system performance and utilization.

Third, although multi-cluster systems are gaining more and more importance in practice, to date less attention has been paid to parallel job scheduling on multi-cluster systems [30]. Instead, much research in the area of scheduling has been and continue to be concentrated on single cluster systems [30]. It has been shown that parallel applications that have been written for homogeneous single cluster systems do not run efficiently on multi-cluster systems [129].

Last but not least, cluster computing systems, compared to the classical parallel computers, posses several technical challenges that introduce additional degree of complexity to the scheduling problem while amplifying the existing ones. This is mainly due to the fact that resources in cluster computing are: (1) distributed; (2) heterogeneous; and (3) highly shared in both time and space. Therefore, continuously arriving jobs and dynamically changing available CPU capacity make traditional scheduling algorithms difficult to apply in networked systems. On-line scheduling algorithms promise better results by adapting schedules to changing situations. The following section gives a formal definition of the scheduling problem and a review of some common approaches used to address it.

3.1.2 Problem Statement

The on-line processor allocation problem can be formally stated as shown in Figure 3.1. Although this is an intractable problem [55], a good approach to attack this problem is fundamental to achieving the cluster computing high-performance objectives.

On-line Processor Allocation Problem

1. **Given:** *A system composed of P shareable (i.e., community-based) processors and a set of J jobs that arrive to the system in a random fashion.*
2. **Processor Allocation:** *The problem is to schedule the J jobs onto the P processors such that the mean response time is improved under the following constraints:*
 - (a) *the scheduler has no information on: (a) the service times of the jobs until the jobs complete execution; (b) remaining processing requirements of the tasks; (c) the job arrivals; and (d) how many processors each job needs until the job actually arrives;*
 - (b) *no knowledge of the set of processors available for scheduling to the scheduler and resource availability may change dynamically, and the patterns of these changes are not known before hand.*

Figure 3.1: On-line processor allocation problem

Several classes of scheduling strategies are possible to address this problem, each differing in the way the system resources are shared among the jobs. These include:

1. The class of *space-sharing* policies that share the processors in space by partitioning them among different parallel jobs. As described in Chapter 2, space-sharing policies can be classified as *static* (e.g., [139, 55]), *adaptive* (e.g., [86, 92]), and *dynamic* (e.g., [119, 65, 81]).
2. The class of *time-sharing* policies that share the processors by rotating them among a set of jobs in time. Time-sharing policies, as described in Chapter

- 2, can be either *coordinated* in which all the tasks in a job are scheduled and de-scheduled simultaneously (e.g., *gang scheduling* [50]) or *uncoordinated* (e.g., the *job-based round robin policy* [79]).
3. The class of scheduling policies that combine both *space-sharing* and *time-sharing* policies, which we refer to as *static hybrid (SH)* scheduling policy (e.g., [50, 19, 124]).
4. The class of scheduling policies that integrate *space-sharing*, *time-sharing* and *load-redistribution* approaches into a single unified framework. We refer to this class of scheduling policies as *dynamic hybrid (DH)* scheduling policies.

In this Chapter, we focus on the *dynamic hybrid (DH)* scheduling policy. We believe that dynamic scheduling approaches that do not require *global coordination* and that combine both *space-sharing* and *time-sharing* policies with a capability to adapt to changes by dynamically re-mapping (i.e., *load-redistribution capabilities*) tasks to processors are attractive for cluster computing systems. We are not aware of any scheduling policy that combines *load redistribution*, *space-sharing* and *time-sharing* into a single framework. The work reported in this chapter is the first attempt in this direction. To this end, we make the following contributions.

3.1.3 Contributions

1. We propose a class of *dynamic hybrid* scheduling policies that employ a push-pull scheme for load distribution and re-distribution over multiple clusters in shared heterogeneous systems. The proposed policy is based on a vertically distributed architecture and on a federation of schedulers, each responsible for a set of nodes in the system.
2. The performance of the proposed policy is compared against a space-sharing policy, and a time-sharing policy and a static hybrid policy. The results of the

experiments show that dynamic hybrid scheduling policy produces significantly better response time than the other three policies.

3. We modified existing *space-sharing* and *time-sharing* policies such that the schedulers determine *where* and *when* applications will be executed (i.e., on which processors and in what order). Moreover, we address the *processor fragmentation* problem [124] associated with the classical *adaptive space-sharing* policies. The performance of these two policies with and without the optimizations were studied and the version with the proposed optimizations produced significantly better response times than the original once.

3.2 Hierarchical Distributed System Architecture

A key issue in developing scheduling infrastructure is the system model upon which the scheduling middleware and the aggregation of the resources are built. This has a direct impact on the services provided, performance, scalability, reliability, load sharing facility, level of support provided to the programmers. There are several ways of organizing workstation clusters into a single virtual system and the choice is usually between distributed control mechanism and centralized control mechanism, each offering good but different and often conflicting features. We use a hierarchical approach as it can combine the merits of centralized and distributed coordination mechanisms into a single framework suitable for shared cluster computing environments. In the hierarchical framework, the W workstations are grouped into a set of clusters, each with P processors (e.g., according to their architectures, processing speeds, and physical locations). In our case, a cluster is geographically compact and contains a set of workstations of similar architecture.

As shown in Figure 3.2, the core system architecture is designed around L -levels of virtual hierarchy, which we refer to as a *cluster tree*. First, the cluster nodes are

Figure 3.2: An example of cluster tree (SS: System scheduler; CS: Cluster scheduler; LS: Local scheduler; and P_i : workstation i)

organized in a tree shape. Multiple clusters are then interconnected at their root nodes via inter-cluster links. This process is repeated L times to create the cluster tree of Figure 3.2. At the top of the cluster tree, there is a *system scheduler* while at the leaf level there is a *local scheduler* (LS) for each node. In between the *system scheduler* and the *local schedulers*, there exists a hierarchy of *cluster schedulers* (CS). At higher levels, the schedulers coordinate the assignment of parallel jobs to clusters. At the cluster level, the cluster schedulers coordinate the scheduling of the components of a parallel program (i.e., tasks) to workstations. At the lowest level, the local schedulers coordinate the execution of parallel tasks and local tasks.

We refer to all processors reachable from a given node in the cluster tree as its

partition-reach. The number of children of a node in the cluster tree is called a *branching factor* (BF). The *processing speed* (ω_l) of a given non-leaf node at level l is computed as follows:

$$\omega_l = \sum_{i=1}^{\text{partition-reach}} \text{speed}(P_i) \quad (3.1)$$

where $\text{speed}(P_i)$ is the processing speed of processor P_i within the partition reach of the node.

The processors at the leaf level are assumed to run a *round-robin* local scheduling policy. We use a parameter called *multiprogramming level* (MPL) to control the number of tasks among which the processor is time-shared at any given time. Since the processors may have different processing speed, the MPL of processor P_i is determined as follows:

$$MPL(P_i) = \left\lceil \frac{\text{speed}(P_i) \times \text{Base MPL}}{\text{speed}(P_{\text{slow}})} \right\rceil \quad (3.2)$$

where $\text{speed}(P_{\text{slow}})$ is the speed of the slowest workstation in the system.

Finally, we associate with each node in the cluster tree a parameter called *base load level*. For all the leaf-level nodes, the base load level is the same as the multiprogramming level of the node. For non-leaf nodes, the base load level is set to zero. Whenever the *load* of a node falls below its *base load level*, the node is said to be in the *receiver* state. In contrast, when the load of a host exceeds the *base load level*, we say that the node is in the *sender* state. A node that is neither in the *sender* nor in the *receiver* state is said to be in the *neutral* state.

It is important to note that the ideas used in the design of the system architecture of Figure 3.2 is not really new. Similar approaches have been used in various contexts and under different system architectures as in [42, 19, 124]. The contribution lies in how these ideas are integrated and modified to address the problem we are trying to solve. For example, our structure is designed with scalability, fault-tolerance, flexibility and simplicity in mind. Unlike [124, 19], we allow the movement of jobs

and tasks vertically (both up and down) and horizontally in the cluster tree. Also, every cluster scheduler in the system is reachable from any other cluster scheduler unless there is a failure in the network or the node housing the cluster scheduler. In addition, the number of tasks in a given processors at any time is proportional to the processing capacity of the processors as will be shown later. Last but not least, by virtue of its push and pull algorithm, the cluster tree allows job submission at any node in the cluster tree.

3.3 Dynamic Hierarchical Scheduling Policy

On the basis of the system structure discussed in the previous section, we designed a parallel job scheduling policy that combines *space-sharing*, *time-sharing* and *load-balancing* into a unified framework. We refer to the proposed policy as *Adaptive Hierarchical Scheduling (AHS)* policy. Figure 3.3 shows the basic components of the adaptive hierarchical scheduling policy.

All incoming jobs are submitted to the system scheduler where they are placed in the *job wait queue* until a placement decision is made. There is no central authority that has the knowledge of and control over when and where different tasks should be executed. Instead, the nodes determine when and how much work they are willing to do. This is achieved through negotiations between the parent scheduler and its children schedulers as discussed in the following subsection.

3.3.1 Request Sequencing Algorithm

The proposed scheduling policy is demand-driven where nodes in the system look for work when their load is below a given threshold. This must be done carefully in such a way that the receiving node is not overloaded and the communication overhead is minimized. To achieve these goals, we use an algorithm that we refer to as a *Request*

Figure 3.3: Architecture of the adaptive hierarchical scheduling policy. RFC: request for computation; RFB: request for load-balancing; CS: cluster scheduler; SS: system scheduler.

Sequencing (RSA) and is shown in Figure 3.4.

The *RSA* is essentially a mechanism of negotiation between a sender and receiver of work, where the receiver sends a request when the need for more work arises. We refer to such requests as *Pull Requests (PLR)*. Specifically, whenever the *current load level* of a non-root node in the cluster tree falls below its *base load level*, the node sends a request for T_{req} units of computation to its parent, where T_{req} is computed as follows:

$$T_{req} = \text{base load level} - \text{current load}. \quad (3.3)$$

After sending the *Request for Computation (RFC)* to its parent, the node updates its *base load level* to ensure that it can have only one outstanding RFC at any given

time. As a consequence, any RFC that arrives at a node with outstanding RFC will be backlogged and processed when work becomes available. Note that this approach minimizes communication overhead as no communication occurs until the *current load level* of a node falls below its *base load level*.

The *RSA* also provides the *Push Requests (PSR)*. This allows the nodes in the cluster tree whereby unfulfilled PLR can be filled from any node in the system. This capability is also useful in heterogeneous environments, as it is common to allocate a disproportionately high percentage of jobs to the more powerful computers in order to improve performance. However, what happens if the system state on which the initial placement decision is made is no longer exists? This is possible in shared system such as cluster computing environments. In this case, a node can send a push request to its parent.

When a parent receives a RFC (i.e., pull request) from a child, the parent processes the request as follows:

1. If the parent does not have unscheduled work, it will store the RFC from the child. It then checks to see if it has an outstanding RFC or not. If there is a pending RFC, then the parent will do nothing. Otherwise, the RFC recursively ascends the cluster tree until the RFC reaches the *system scheduler*, where it is backlogged and processed when work becomes available.

An example illustrating the RFC sequencing is shown in Figure 3.5a. Here it is assumed that there is no unscheduled work in the system. Here, P_1 of *cluster₁* sends a RFC to its parent CS_1 . When the parent receives the RFC from P_1 it will forward it to its parent CS_{10} on the next level of the cluster tree, which in turn forwards it to the system scheduler. Note that when the RFC from CS_3 arrives at CS_{10} the request is simply backlogged since CS_{10} has an outstanding request at the time of the RFC from CS_3 arrival.

Request Sequencing Algorithm

1. Pull Request

- (a) Request for T_{req} units of computation where T_{req} is computed as follows:

$$T_{req} = \text{base load level} - \text{current load.} \quad (3.4)$$

- (b) Send the request to the parent scheduler.
(c) Update the base load level as follows:

$$\text{base load level} = T_{req} + \text{current load.} \quad (3.5)$$

2. Push Request

- (a) Send a push request message to the parent scheduler
(b) Update the pending request queue.

3. Process the request.

- (a) Pull Request.
- i. If there is unassigned work, invoke the *push algorithm*.
 - ii. Else if there is a pending request, queue the RFC.
 - iii. Else generate *RFC* as discussed in step 1a.
- (b) Push Request.
- i. if the node is in *neutral* state, queue the request
 - ii. Otherwise, invoke the *pull algorithm*.

Figure 3.4: Pseudo-code of the request sequencing algorithm

2. If a parent scheduler has unassigned jobs when a RFC from a child arrives, the *Pull* algorithm gets invoked to send a set of jobs down the hierarchy along the path the RFC has traveled. The following subsection presents the details of *Pull* algorithm.

Figure 3.5: An illustration of the request sequencing

3.3.2 Pull Algorithm

The *Pull* algorithm is a form of *space-sharing* policy in that it assigns a partition size to jobs based on the parent-child relationship and negotiations between the nodes in the cluster. Figure 3.6 is the pseudo-code of the *pull* algorithm. First, the algorithm determines an ideal number of jobs that can be moved down one level to the child scheduler as a single assignment. The key is to transfer a suitable number of jobs from a parent to its children. This amount is determined dynamically during parent and child negotiations and the number of unscheduled jobs. The algorithm uses a parameter, T_r , which we refer to as the *transfer factor* and is set to:

$$T_r = \frac{\text{partition-reach of the child node}}{\text{partition-reach of the parent node}} \quad (3.6)$$

Once the target number, T_{target} , of jobs that can possibly be transferred to a child is determined, the algorithm then considers the size of the RFC from the child as

a hint to adjust the number of jobs that will actually be transferred down one level to the child. Finally, if the target size determined above is smaller than the T_{req} from the child, then the parent will send the lesser of T_{req} and the *number of tasks queued*. Otherwise, the set of tasks sent to the child is the lesser of T_{target} and *child partition-reach*. Note that this is a dynamic load distribution algorithm that changes the size of the batch at run time, allocating large size to larger clusters while smaller clusters are allocated small size.

Pull Algorithm

1. Determines an ideal number of tasks that can possibly be sent to a child scheduler as follows:

$$T_{target} = \lceil T_r \times \text{number of tasks queued} \rceil \quad (3.7)$$

where T_r is the *transfer factor* and is computed as using Equation 3.6.

2. Adjust the number of tasks actually transferred as follows:

$$T_{target} = \begin{cases} \min(T_{req}, \text{number of tasks queued}) & \text{if } T_{req} > T_{target} \\ \min(T_{req}, \text{child partition-reach}) & \text{Otherwise.} \end{cases} \quad (3.8)$$

where T_{req} is the is the number of computations requested by the child scheduler.

3. Forward T_{send} tasks to the child scheduler.
-

Figure 3.6: Description of the pull algorithm.

Figure 3.7 illustrates how the Pull algorithm works. In the example, it is assumed that B is 4, MPL is 2, all processors are idle, and there are 32 tasks waiting to be scheduled at the system scheduler. The T_{req} from P_1 is 2 since the base load at the leaf-level is equal to the MPL of the nodes.

When the *system scheduler* receives the RFC from CS_{10} , it removes 11 tasks from the ready queue and sends them to CS_{10} . By this move, these 11 tasks are assigned a partition-size equal to the number of processors reachable from CS_{10} . When CS_{10}

Figure 3.7: An illustration of the pull algorithm. (For simplicity, local schedulers associated with processors are not shown in this figure.)

receives the 11 tasks, it also applies space-sharing to these tasks, which results in 4 tasks being dispatched to CS_1 . This means the partition-size of these 4 tasks is now set to the four processors in cluster 1. At this point, one of the 4 tasks is sent to P_1 while the remaining three tasks are queued in the task wait queue at CS_1 such that any one of the four processors within the CS_1 cluster can retrieve and execute it. Note that this process leaves a varying number of tasks along the way. This means, RFCs from other processors don't have to travel all the way to the root as there are jobs and tasks waiting at various levels of the cluster tree. This substantially decreases the number of requests to be processed by the system scheduler.

3.3.3 Push Algorithm

The push algorithm intrinsically allows a batch of tasks to be relocated from overloaded clusters to underloaded clusters. Factors considered when this amount is de-

terminated include processing speeds of different nodes, the current load state of both parent and children, and the processing demands of tasks eligible for relocation. Figure 3.8 is the pseudo-code of the push algorithm. There are two instances in which the push algorithm is invoked: (1) when a load level of a given cluster exceeds its capacity (e.g., when child scheduler receives work from the parent scheduler in excess of its capacity) and (2) when a RFC cannot be satisfied by the system scheduler.

Push Algorithm

1. Each child node sends the total number of tasks, W_i , within its partition-reach to its parent.
2. The parent sends to each child the average workload per child, w_{avg} , computed as follows:

$$w_{avg} = \frac{\sum_{i=1}^B W_i}{\Phi} \quad (3.9)$$

where Φ is the sum of the CPU speeds within the partition-reach of the parent node and B is the number of children.

3. When a child node receives w_{avg} from its parent, it computes its quota and sends excess tasks to its parent. The number of tasks a child gives up is determined as follows:

$$w_{send} = \text{current load} - \Phi_i \times w_{avg} \quad (3.10)$$

Figure 3.8: Dynamic push algorithm

When the push algorithm is initiated, first it determines the average load in the entire system (w_{avg}). It then identifies those clusters that are above or below w_{avg} and move tasks from the most loaded queues of the overloaded clusters to the least loaded queues of the underloaded clusters. The algorithm tries to keep the tasks of a job together if possible by first trying to move tasks that can execute together in the

destination clusters.

Figure 3.9 illustrates how the pull algorithm works. Note that each node in the cluster tree has three entries. The first entry is the current load (i.e., unassigned jobs) whereas the second and third entries denote the total load and processing speed of the nodes within its partition-reach. When a request from CS_6 arrives at S_0 , there are no unassigned tasks to send to CS_6 . Therefore, the *system scheduler* initiates the load balancing activity. Each node determines the relative load of the sub-tree and sends it to the system scheduler, which computes the average workload per node and sends it to the nodes below it. Each node then determines the total amount of load it has to keep and the number of jobs/tasks to transfer to its parent.

3.3.4 Comparative Analysis

We now compare and contrast the proposed approach with some of well-known existing approaches that have some commonalities with our approach. The motivation for doing this is to highlight the unique features of the proposed algorithm as well as its advantages relative to some of the policies in the literature.

First, the proposed policy dynamically switches between a load distribution phase (i.e., pull algorithm) and a load-balancing phase (i.e., push algorithm). This is important since the load distribution algorithm can assign jobs to already overloaded site, slower site or due to unexpected arrival of local tasks.

Platform computing extended the standard Load Sharing Facility (LSF) to multi-cluster environments called the *LSF Multi-Cluster (LSFMC)* system, where multiple clusters are interconnected in peer-to-peer fashion. However, LSFMC uses *static space-sharing* approach. There are a number of limitations. For example, a parallel job can start executing only when the execution cluster has enough hosts and resources to run the entire job. Another limitation is that a parallel job cannot span clusters (i.e., no co-allocation) even if there are resources that can satisfy the requirements

Figure 3.9: An illustration of the load balancing activity.

of the job across multiple clusters. The proposed policy is dynamic scheduling and combines space-sharing, time-sharing and load-balancing in a coordinated manner. It also allows co-allocation based on the system load and the needs of the jobs.

Several hierarchical schedulers have been discussed in the literature [50, 19, 124, 20], but they all focus on special applications. None of these approaches actually integrate time-sharing, space-sharing and load-balancing as is done by the proposed policy. Also, the work of [50, 19] focus on shared memory systems while the [124]

focus on dedicated homogeneous distributed systems. However, Atlas [20] deals with shared heterogeneous cluster computing systems. Also, once a job is assigned to a cluster scheduler, the parent scheduler (e.g., system scheduler) has no further direct or indirect influence on further scheduling of the job. In addition, a job cannot be moved to another cluster even if the load at the other cluster becomes lower at a later time. The proposed scheduling policy does not have this problem.

The Atlas system [20] arranges processors in a tree topology and uses *hierarchical work stealing* to implement self-scheduling. There are a number of problems with Atlas. First, it can generate a lot of inter-cluster messages. Moreover, Atlas suffers from bad performance inside clusters and stalls the entire cluster when inter-cluster steals are issued. Each successful steal brings down one task, which leads to excessive communication overhead. Last but not least, processors initiate the steal request only when their ready queue is empty. This leads to the processors to idle while the steal is transcending the tree. In contrast, our approach minimizes the inter-cluster communication by allowing only parents to generate inter-cluster job requests. It also has high locality as the algorithm only seeks work provided that there is no job within the cluster. To avoid whole cluster stalling, the parent prefetches a set of jobs where prefetching is controlled by load information in the clusters. Our approach further reduces this overhead by making the nodes to send request messages only when their load changes. Finally, the proposed approach hides the possibly high inter-cluster trip time by overlapping communication and computation, because inter-cluster prefetching messages are asynchronous. Also, each job is allocated a fraction of processors that corresponds to its share of the total anticipated number of processors allocated in the system. Under light loads, a job will be assigned more processors while under heavy loads it will be allocated fewer processors.

Last but not least, AHS has some characteristics of the Backfill [139, 55] policy without its drawback as it tends to restrict large jobs to a subset of clusters, which

allows smaller jobs to have better response times than they could have under normal circumstances.

3.4 Baseline Parallel Job Scheduling Policies

We compared the performance of the AHS policy with a derivative of the job-based time-sharing policy [79] and an adaptive space-sharing policy originally introduced in [101] and subsequently modified in [124] to take into account running jobs when determining the partition size. The next two subsections give details of these two policies.

3.4.1 Time Sharing Policy

In the original policy proposed in [79], a central ready queue is used to hold unscheduled tasks and as each task completes its quantum on a given processor, it is preempted and returned to the central ready queue. A major problem in implementing such type of time sharing policies in cluster computing environments is the impact on disk space or network bandwidth due to the requirements of saving the task state before migration to the central queue. If the image of the process's state is too big then migration can be time consuming and defeats the main purpose of the scheduling policy, which is to minimize the response time. Moreover, a task may have to execute on several processors of different speeds and architectures. Also, it increases the frequency of accesses to the central queue. This last problem becomes more severe as the quantum size decreases. On the other hand, if we increase the quantum size to decrease the contention problem, performance becomes sensitive to service time variance.

In [19], some ways of addressing these problems are discussed for multiprocessor environments. We build on the work of [19] and introduce modifications to the original

policy such that it is suitable for cluster computing environments. As in [19], the modified policy uses a two-level scheme in which at the upper level there is a system wide ready queue and at the processor-level each processor has its own ready task queue. Also, we associate with each processor, a multiprogramming level (MPL) that controls the maximum number of tasks that can be present in a given processor ready queue (i.e., MPL). This number is given as per Equation 3.2. We refer to the new policy as the *Job-based Round-Robin (JTS)* time-sharing policy. This new policy works as shown in Figure 3.10. Note that since preempted tasks are inserted back in the processor's local queue, the new policy eliminates the second problem mentioned above.

Job-based Time Sharing Policy

1. When a job arrives into the system, it is partitioned into its constituent tasks and each task is assigned a quantum length as per Equation (3.11) and then inserted into the system ready queue.

$$quantum = \frac{\text{Total Processors} \times \text{Base Quantum}}{\text{Maximum Parallelism of the job}} \quad (3.11)$$

where *Base Quantum* is the default quantum size in the system and *Maximum Parallelism* is the number of tasks in a job.

2. When the number of tasks in a given processor falls below its multiprogramming level (MPL), then the processor picks up a task from the central ready queue and begins to execute.
 3. Each processor applies the RR policy only on the tasks that are in their local queues and preempted tasks are inserted back in the processor's local queue.
-

Figure 3.10: Job-based time sharing policy

3.4.2 Space Sharing Policy

The space-sharing policy we implemented in this chapter is based on the *Modified Adaptive Partition (MAP)* algorithm originally introduced in [101] and subsequently modified in [124]. Adaptive space-sharing policies for cluster computing systems should have at least two components: *processor allocation* (i.e., how many processors is allocated to a job) and *processor selection* (i.e., which processors should be allocated to a job). However, the classical *adaptive space-sharing* policies [124, 86, 92, 103], consider only the processor allocation aspect while they are oblivious to processor selection problem. Also, they are characterized with *all-or-nothing* approach, which means that jobs will hold on to the processors allocated to them even if they do not need it. Existing adaptive policies do not actively correct such a resource imbalance state. As a result, some clusters may be overloaded while others may sit idle. This leads to poor resource utilization due to *processor fragmentation*.

In order to adapt the MAP policy to cluster computing, we introduced some modifications to the MAP policy that essentially mimics the *dynamic space-sharing* approach such that idling processors are allocated to other jobs that can fruitfully utilize them. To achieve this, we keep a *pool of available processors* from which the scheduler allots a set of processors to unscheduled jobs. When a processor is assigned to a job completes the allocated task, it immediately returns to the pool where it becomes available for reassignment to another job. We also modify the activation of the scheduler such that it is invoked in three instances: (1) when a job arrives; (2) when a job departs; and (3) when there are α processors in the *pool*, where α is a tunable parameter and is $0 \leq \alpha \leq P$, where P is the total number of processors in the system.

The pseudo-code of the new algorithm is shown in Figure 3.11. We use a parameter called a *fitness factor (ff)* for each processors P_i in the system that is defined as

Adaptive Space Sharing Policy

1. Compute the ideal partition size as follows:

$$\text{partition size} = \left\lceil \frac{\text{Total processors}}{(\text{Queued Jobs}+1) + (0.5 \times \text{Executing Jobs})} \right\rceil \quad (3.12)$$

2. Determine target partition size as follows:

$$\text{target} = \min(\text{maximum parallelism of the job, partition size}) \quad (3.13)$$

3. Processor selection phase.

- (a) Select processors within the same cluster if possible
- (b) Select processors taking cluster proximity into account

4. For each processor P_i in the target partition:

- (a) Determine the number of tasks to be assigned as follows:

$$\text{bunch} = ff(P_i) \times \frac{\text{Job Maximum Parallelism}}{ff(\text{System})} \quad (3.14)$$

Reset the system fitness factor

$$ff(\text{System}) = ff(\text{System}) - ff(P_i) \quad (3.15)$$

Reset the job maximum parallelism

$$\text{Job Maximum Parallelism} = \text{Job Maximum Parallelism} - \text{bunch} \quad (3.16)$$

- (b) The tasks assigned to processor P_i are *folded* onto the target partition size allocated to the job.
-

Figure 3.11: Adaptive space sharing policy

follows:

$$ff = \frac{\text{CPU speed} \times \text{Default MPL}}{\text{slowest processor speed in the system}} \quad (3.17)$$

Hence, $ff(system)$ is the *fitness factor* of the sum of all processors in the system.

When the algorithm is invoked, it assigns a set of processors equal to the lesser of the partition size and the job's maximum parallelism. The scheduler then goes into processor selection phase, which selects appropriate processors for the job. Finally, the scheduler determines the exact number of tasks assigned to each processor and the tasks assigned to processor P_i are *folded* [82] onto the target partition size allocated to the job. For example, if P_i is assigned 3 tasks, these tasks are folded into one large task assuming that jobs are malleable.

3.5 Performance Evaluation

We used discrete event simulation written in C programming language to study the performance of the proposed policy and compare it with the baseline policies. Without a complete prototype and access to all the resources to be used, we have a choice between analytic and empirical approaches to evaluate the performance of new scheduling algorithms. Although theoretical analysis is possible, it is inherently complex for cluster computing systems due to various dependencies between the software and hardware components. Also, in practice, isolating the effect of various parameters on a system, while holding the others constant requires exploring a variety of scenarios. Moreover, the analytic approach requires a formal theoretical model to describe processor availability patterns. Given our current understanding of processor availability patterns, creating an accurate formal model promises to be a difficult task. The complexity of parallel applications and parallel processor scheduling policies make exact analysis for large systems very difficult as well. Simulation can provide an attractive mechanism to study the combined influence of hardware and software failures, and the possible interactions between them on the overall failure behaviour of the system. Simulation can also offer a viable mechanism to supply carefully controlled,

homogenous data sets, and to study the overall failure behaviour of a system as well as the influence of various parameters/factors on the failure behaviour. Owing to these facts, we chose to use the simulation-based approach to evaluate and validate the performance of the processed scheduling policy.

Parameters	Values
Multiprogramming level	2
Service demand coefficient of variation	3.5
Parallel job arrival coefficient of variation	1.0
Parallel job mean service demand	14.06
Background job mean service demand	2.0
Number of clusters in the system	8
Number of processors in each cluster	8
Space-sharing threshold (α)	2

Table 3.1: Default parameters and values used in the experiments.

Table 3.1 shows the default system and workload parameters used in the simulation. Several classes of experiments were performed; these include experiments that study the effect of scheduling and contention overheads on the response time, the effect of the coefficient of variation of service demands and arrival times as well as the effects of other factors such as the cluster and the system sizes.

Performance Metrics: For relative performance evaluation, we use the *mean response time (MRT)* and *average utilization*. The MRT is defined as the average job response time of completed jobs. The *average utilization* is defined as the job arrival rate times the mean service demand of the jobs divided by the number of processors in the system.

Validation: In all the experiments, a batch strategy is used to compute confidence intervals. We used 31 batches in simulation run and a batch size of 10,000 jobs were used for the results reported in this chapter. In each experiment, we always ignored the results of the first batch as a system warmup. This strategy produced

95% confidence level that were less than 1% of the mean response times confidence interval when the system utilization is low to moderate and less than 5% for moderate to high system utilization (in most cases, up to 90%). Also, wherever possible, we verified the results in this chapter with previously published results.

3.5.1 Experimental Environment Setup

We organized the 64 nodes in the system into eight clusters and each cluster is managed by a cluster scheduler. We then create a 4-level cluster tree with the root as the system scheduler, four second-level cluster schedulers that act as children of the system scheduler, and 64 local schedulers at the bottom. We studied performance of the scheduling policies in the following environments:

1. **Shared Homogeneous Environment:** In this setting, all processors run at the same speed. However, each workstation may receive background workload at any time.
2. **Dedicated Heterogeneous Environment:** In this environment, there is no background workload. However, the processors in the system may differ with respect to their processing speed.
3. **Shared Heterogeneous Environment:** In this setting, processors differ in speed. Also, each workstation may receive background workload at any time.

Processor Model: In order to model the processors speed, we used the SPECfp2000 results for a family of Intel Pentium 3 and 4 processors with different clock speeds Table 3.2 shows the four types of Pentium processors used in this thesis. The *relative* column in this table gives the node rating relative to the slowest node in the system. Also, we assume that each workstation runs a round-robin time sharing local scheduling policy. We used $200\mu\text{s}$ for the context switch overhead and 100ms for the

time-slice, which is consistent with most of modern operating system for workstations.

System Name	Base	Relative
SE440BX-2 (650 MHz P3)	215	1.0
VC820 (1.0 GHz P3)	284	1.32
OR840 (1 GHz P3)	335	1.56
D850GB(1.4 GHz, P4)	538	2.50

Table 3.2: Processor type and speed used in the simulation.

Communication Model: We used a simple model to capture the communication overhead as follows:

$$T_{comm} = \text{Startup} + \frac{\text{Message size}}{\text{Bandwidth}} \quad (3.18)$$

The same model is used in several studies (e.g., [124]) and the model has been shown to be quite sufficient in [130]. In all the experiments discussed here, the communication network latency to be $50\mu\text{sec}$ with the transfer rate of 100Mbits/sec. These values are typical of modern light-weight messaging layers running on top of gigabit switched LANs [130].

3.5.2 Parallel and Background Workload Models

3.5.2.1 Background Workload Models

Each workstation services a stream of background tasks that arrive at individual workstations independently. As in [104, 76], we model the inter-arrival times and the service demand of the background jobs with an exponential distribution and a 2-stage hyper-exponential distribution respectively.

3.5.2.2 Parallel Workload Models

We use synthetic workload to study the performance of the scheduling policies described in this thesis. An alternative is to use traces of actual workloads. The problem

with trace-based approach is that the traces reflect specific system environments for which the applications were developed and executed. Synthetic workloads, on the other hand, allow us to model applications for general-purpose systems using various parallel workload characteristics observed in supercomputing centers. However, we intend to study the performance of the proposed policy using various workload traces in future.

We assume that a set of n jobs, $\mathbf{J}=\{J_1, ..., J_n\}$ arrives to the system in a stochastic manner. Each job, J_i , can be decomposed into t tasks, $\mathbf{T}=\{T_1,...,T_t\}$. Each task T_i executes sequential code and is fully preemptable. The abstract model for parallel programs consists of a set of input parameters along with a job structure. As in [124], we assume that jobs only request processors and we do not include in the model any other type of resources. Each job is characterized by *arrival time*, *service demand time* in a dedicated environment, *maximum parallelism*, and the *size of the job* in Kbytes.

1. **Job arrival:** The job arrival process is characterized by two parameters: λ and CVA, where λ represents the job arrival rate and CVA represents the coefficient of variation of the inter-arrival times.
2. **Maximum Parallelism (P_{max}):** The maximum parallelism of a parallel program refers to the maximum number of tasks that can be run in parallel, if an unlimited number of processors is provided.
3. **Mean Service Demand (S_{demand}):** The service time needed to execute a job in a dedicated environment. We use the uncorrelated approach in which the overall service demand of a parallel program is independent of the number of tasks. For example, if $S_{demand} = 20$, then, on average, a parallel program requires 20 units of service time from the system.

4. **Coefficient of Variation of Service Demand (CVS):** The CVS is the coefficient of variation of service demand of parallel programs.

As noted earlier, each parallel job is composed of a set of tasks. The cumulative service demand of a job is divided among its constituent tasks such that three different workload types described in [124] are created as follows:

1. **Workload W1:** In this workload, service demand of the job is divided evenly among the tasks. For example, if a job's service demand is 2 hours and the job is decomposed into 4 tasks, the service demand of each task will be 30 minutes. Examples of this type of application include matrix multiplication algorithms.
2. **Workload W2:** In this workload, 50% of the tasks of the job receive 50% of the evenly distributed job service demand while the remaining service demand is shared equally amongst the other tasks. For example, if a job with 4 tasks arrives with a total service demand of 8 minutes, two tasks will have a service demand of 1 minutes (i.e., 50% of 2 minutes) and the other two tasks will have 3 minutes of service demand.
3. **Workload W3:** This workload distributes a job's total service demand such that 50% of the tasks receive 25% of the job service demand while the remaining service demand is shared equally amongst the other tasks. For example, suppose a job with service demand of 16 minutes and 4 tasks arrives at the system. Then two tasks will have a service demand of 1 minute (25%) each while the remaining two tasks (50%) will have a service demand of 7 minutes each.

The maximum parallelism of the jobs is uniformly distributed over the range of 1 to 64 [124], while the service demands of the jobs are generated using hyper-exponential distribution with mean 14.06 [121, 124]. The default arrival CV is fixed at 1 (i.e., we assume Poisson arrivals) and the default service time CV is fixed at 3.5 as empirical

observations at several supercomputer centers indicated this to be a reasonable value [121]. However, these studies have also shown that both arrival time CV and service time CV can be very high as well. Therefore, we study the sensitivity of the scheduling policies to these variances in Section 3.6.3.1 and 3.6.3.2 respectively.

3.6 Simulation Results and Discussions

For all the results presented in this section, we plot the mean response time (in seconds) on y-axis. Also, the parameters in all the experiments are set to their default values as shown in Table 3.1, unless otherwise noted. Note that performance at low load levels is not useful in general and we concentrate in the range between 50% to 90% as in [124]. The rest of the section discusses the results of the simulation.

3.6.1 Dynamic versus Static Hybrid Policies

In this section, we study the performance of the static hybrid and dynamic hybrid policies. Note that we refer to the class of scheduling policies that combine both *space-sharing* and *time-sharing* policies as *static hybrid (SH)* scheduling policy (e.g., [50, 19, 124]). In contrast, we refer to the class of scheduling policies that integrate *space-sharing*, *time-sharing* and *load-redistribution* approaches into a single unified

framework as *dynamic hybrid (DH)* scheduling policies. Figure 3.12 shows the relative performance of the dynamic and static policies for the three workloads under shared homogeneous, dedicated heterogeneous and shared heterogeneous environments respectively. The general trend is that there is no significant difference between the performance of the two policies at low system loads. This is because at this load, there are plenty of processors idle most of the time. However, as the load increases the dynamic hybrid policy performs better (by about 20% to 30%) than the static hybrid policy.

Note that the static policy does not reconsider prior scheduling decisions, which may lead to load imbalance due to workload characteristics or system environments. However, the dynamic hybrid policy is able to redistribute work and maintain almost uniform load among the clusters based on the capacity of the clusters. In light of this observation, there is a compelling reason to include mechanisms for modifying scheduling decisions in response to changes in the system to improve the overall performance by redistributing loads from heavily loaded to lightly loaded computers. This is important since external computations from other users are present, a system may become grossly unbalanced, resulting in degraded performance.

3.6.2 Dynamic versus Baseline Policies

3.6.2.1 Shared Homogeneous Environments

Figure B.4 shows the relative performance of the three scheduling policies under the shared homogeneous environment for the three workloads. Note that both W_2 and W_3 workloads are slightly inferior in speedup to W_1 workload. For this reason, the mean response times for W_2 and W_3 workloads are higher than that for W_1 workload.

The data in the graph shows that at low system loads, there is no significant difference among the three policies as there are plenty of processors idle most of the time at this load level. However, under the load of interest (i.e., medium to high load), the performance of the three policies is quite different. The results show that AHS policy is the best for all the three workloads at medium to high loads. The MAP policy performs better than the JTS policy for W_1 workload and the JTS performs better than the MAP policy for W_3 while there is only a marginal difference between these two policies for W_2 workload. This trend can be explained by the fact that in the MAP policy jobs tend to be allocated smaller partition-size as the system load increases and in the presence of local workload. As the number of processors allocated to a job decreases below its maximum parallelism, the service demand of the tasks also increases. This makes the jobs sensitive to the presence of the background load compared to the other two policies, which can increase the wait time of the jobs. Also, the JTS policy can degenerate to run-to-completion as tasks with larger quantum size can block the execution of tasks with smaller tasks. In contrast, the AHS is able to transfer work along the appropriate branch at the root much more efficiently. In addition, the load balancing component tends to maintain almost uniform load amongst all the clusters based on the capacity of the clusters.

Note that the impact of the background load on the parallel job depends on the service demand of the parallel tasks; the longer a parallel task occupies the processor

the more likely its execution is interrupted by the background workload. In addition, the MAP policy suffers from a form of *processor fragmentation* induced by the presence of the background load and the way the partition-size is assigned to the jobs. Since partition size is computed based on the total number of processors in the system, the actual number of idle processors can be lower than the partition-size computed. This is because of the fact that some of the processors run background load at the time and the MAP policy does not take into account the background loads when computing the partition size. In this situation, the processors will remain idle as the scheduler will not assign them to jobs even if there are small jobs that can fruitfully utilize them.

3.6.2.2 Dedicated Heterogeneous Environments

Figure 3.13 shows the performance of the three policies under the dedicated heterogeneous environment for the three workloads. There is no significant differences between the three policies at low system loads, while the AHS outperforms the other two policies for all three workloads when the system load is medium to high.

From the data in the figures, we also observe that the JTS policy substantially outperforms the MAP policy when there is imbalance among the tasks of a job (i.e., W_2 and W_3 workloads). One possible explanation for the performance difference between the AHS and the other two policies is due the presence of large jobs. In JTS, for instance, since round robin policy is applied to the local queues, large jobs tend to make the system work in a quasi-FCFS mode.

The AHS policy avoids this type of behavior as it first uses space sharing to limit large jobs to a set of processors in a partition. In addition, it applies time sharing to derive the benefits of the time sharing policy. Furthermore, the load balancing component tends to maintain almost uniform load among the clusters based on the capacity of the clusters. The poor performance of the MAP policy is due to excessive

wait times experienced by the jobs compounded by a long job getting assigned to a slow machine. This is because of the fact that jobs tend to be allocated small partition-size as the load increases. As a result, when a large job is confined to slow processor (that becomes a bottleneck), its execution time increases.

Figure 3.12: Relative performance under shared homogeneous environments

Figure 3.13: Relative performance under the dedicated heterogeneous environment.

3.6.2.3 Shared Heterogeneous Environments

Figure 3.14 shows the relative performance of the three policies under the shared heterogeneous environment for the three workloads. Relative to the dedicated heterogeneous environment (see Figure 3.13) the mean response time of all three policies have slightly increased under the shared heterogeneous environment.

Also, for W1 workload, the data in Figure 3.14 shows that the MAP policy is more sensitive to the combined effects of background load and processor heterogeneity than the other policies (compare Figure 3.14 and Figure 3.13). However, the trends for W2 and W3 workloads is similar. Here the JTS policy tends to be more sensitive to the combined heterogeneity and shared environment at low to medium loads than the other two policies. However, at high system loads, the MAP policy tends to be more sensitive than the other two policies. This is due to the fact that, in addition to processor heterogeneity, the jobs experience slowdown due to interference from the background jobs. Note that heterogeneity can lead to a load imbalance situation. In such situations, the completion time of the jobs increases. When the execution of jobs in an already imbalanced environment is interrupted due to background jobs, the finishing time of the jobs is further increased. Since AHS policy employs a load balancing algorithm that monitors the load in the system and corrects in case the load is unbalanced, its performances tends to be better than the other two policies, which do not have this capability.

The relatively poor performance of the MAP policy is due to the space sharing nature of the policy. Note that in the extreme case, the MAP policy ends up allocating a single processor to each job. Therefore, all the optimizations proposed to alleviate the problems associated with MAP policy have no impact at this stage. In addition, the presence of the background load also exacerbates the situation by creating interference for currently running jobs. This interference is observed as an increase in the completion time of the parallel jobs due to resource contention. The

MAP policy reaches a point at which performance dramatically falls off. When the load is sufficiently high with all the processors running at least 1 job, the interference increases rapidly and performance can be very poor. Also under the MAP policy a job may have to be executed in one of the slower processors, considerably degrading the execution time.

Note that in all these figures (i.e., Figures B.4, 3.13 and 3.14) the response time of W_2 and W_3 workloads are increased slightly relative to the W_1 workload. This is because of the fact that W_2 and W_3 workloads exhibit slightly inferior speed up.

3.6.3 Sensitivity Analysis

In this section, we study the sensitivity of the three policies to variances in inter-arrival and service times using W_2 workload. When the arrival CV is varied, the service CV is held at 3.5; arrival CV is fixed at 1 when the performance sensitivity to service time CV is studied. The system utilization for both parallel and background load is fixed at 80%.

Figure 3.14: Relative performance under shared heterogeneous environments.

3.6.3.1 Sensitivity to Service Demand Variations

Figure 3.15 shows the performance of the three scheduling policies as a function of the service time coefficient of variations (CVS). The MAP policy is quite sensitive to the variance in service demand as compared to the other two policies. This is because the fact that the MAP policy does not provide time-sharing, rather uses the FCFS allocation of processors to jobs. It is well known that the FCFS is sensitive to service time variance. FCFS allocation of processors to jobs results in a situation where small jobs could be held up by a large job that had arrived earlier. This problem gets more serious as the variance in service demand increases. The other policies are not as sensitive to the variance in service demand because of the time-sharing nature. An increase in CVS only increases the variance of job service demand and does not affect the processor allocation decision of these policies.

Figure 3.15: Sensitivity to service demand variations: Homogeneous shared settings; Heterogeneous dedicated settings; and Heterogeneous shared settings.

3.6.3.2 Sensitivity to Arrival Time Variations

Figure 3.16 shows the sensitivity of the three scheduling policies as a function of the parallel job arrival time coefficient of variations (CVA). As in [124], we used the two-stage hyper-exponential distribution for $CVA > 1.0$ in the experiments. Note that, higher variance in CVA implies a clustered arrival of parallel jobs into the system.

As can be seen from the graph, the mean response time of MAP increases at a faster pace as compared to AHS and JTS policies. This trend can be explained by the fact that clustered arrival forces the MAP policy to assign smaller partitions to jobs as a batch of jobs arrive into the system. Since jobs arrive in clusters, jobs may have to wait longer before they can be scheduled to run. This increases the waiting time, which in turn increases the response time. The higher the CV, the more clustered the arrival is. Thus, the queue length becomes the dominant factor in determining the partition size, resulting in decreased response time difference among the three policies.

The performance of the JTS and AHS policies is substantially better than the MAP. This is mainly achieved by multiprogramming the tasks on a processor. Between the two, AHS performs better than the JTS policy as there is less likelihood of a large job blocking small jobs behind, which is the case in JTS. This is because of the fact that AHS employs the space sharing policy, which works in quasi-policy.

Figure 3.16: Sensitivity to arrival time variations: Homogeneous shared settings; Heterogeneous dedicated settings; and Heterogeneous shared settings.

3.6.3.3 Sensitivity to Non-Uniform Cluster

In this section, we study the impact of cluster size heterogeneity on the AHS policy. The number of workstations in each cluster is varied as shown in Table 3.3. The first row is the 8 clusters while the second and third rows denote the size of each cluster (i.e., the number of workstations in each cluster) and the processing speed of the workstations, respectively.

C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_7
12	10	8	7	10	11	5	9
1.0	1.0	1.32	1.32	2.50	2.50	1.56	1.56

Table 3.3: Non-uniform system configuration.

Figure 3.17: Sensitivity to cluster size.

Figure 3.17 shows the performance of AHS under uniform and non-uniform cluster sizes. The labels SHD denote *shared homogeneous environment*, HED denote *dedicated heterogeneous environment*, and SHED denote *shared heterogeneous environment*. In the experiments, clusters vary in size for but we use the same set of jobs as discussed before. The results of the experiment show that AHS policy is insensitive to the *cluster size heterogeneity* as the policy task into account, among other factors, the size of the cluster at the time of the load distribution as well as load balancing activities.

3.7 Summary

In this chapter, we investigated parallel job scheduling problem on heterogeneous networks of workstations, where computing power varies among the workstations, and local and parallel jobs may interact with each other in execution. We proposed a scheduling policy based on a virtually rooted tree structure that employs a pull-push scheme for scheduling and load balancing parallel jobs over multiple clusters. The proposed policy allows multiple job streams to share the system concurrently in an environment where the actual load distribution is not completely predictable in advance and scheduling is done dynamically. Also, the policy integrates several approaches (i.e. task scheduling, load balancing, self-scheduling, and time-space sharing) into a simple framework for parallel job scheduling on a system composed of multiple clusters. We studied the performance of the proposed scheduling policy through simulation. The results indicate that the proposed scheduling policy significantly better than the other scheduling policies used in the study.

Chapter 4

Coordinated Resource Scheduling

In Chapter 3, we considered the problem of job scheduling under the assumption that the jobs require only processors. In reality, applications may require multiple resources and to avoid negating some or all of the benefits derived from parallelizing computations, the scheduling decisions must be based on all of the resources required by the applications. Moreover, advanced general-purpose parallel processing systems should be able to support applications with different resource requirements without compromising effectiveness and efficiency. In this chapter, we address the problem of coordinated allocation of resources with emphasis on the processor and I/O resources¹. We propose two scheduling policies and compare their performance with static space sharing policy².

¹J. H. Abawajy and S.P. Dandamudi, "A Unified Resource Scheduling Approach on Cluster Computing Systems," *Proceedings of the 16th International Conference on Parallel and Distributed Computing Systems (PDCS-2003)*, August 13-15, 2003, Nevada, U.S.A.

²J. H. Abawajy and S.P. Dandamudi, "Scheduling Parallel Jobs with CPU and I/O Resource Requirements in Cluster Computing Systems," *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS'03)*, Orlando, Florida, October 12-15, 2003.

4.1 Motivation and Problem Statement

Cluster computing systems provide a large number of different resources, such as compute cycles, memory, communication network, and I/O resources for parallel and distributed job execution. First, managing and utilizing these resources to meet performance and usability goals of high performance applications has become a significant challenge. Second, as cluster computing gains popularity, it is increasingly being used for parallel and sequential applications with significant I/O requirements in addition to compute-intensive parallel and sequential applications. Hence, as applications require multiple resources (e.g., CPU and I/O) and to avoid negating some or all of the benefits derived from parallelizing computations, the scheduling decisions must be based on all of the resources required by the applications. Third, with the tremendous advances in processor and memory technology, I/O has risen to become the bottleneck in high-performance computing for many applications. A key approach to overcoming these problems is through efficient organization of the resources and scheduling policies [99, 46, 138].

Current job scheduling approaches do not normally consider these requirements when assigning resources to jobs. In fact, the focus of job scheduling research has been on independently scheduling processors (e.g., [124]) or I/O resources (e.g., [72]). However, separated processor and I/O resource scheduling can lead to poor quality of service and utilization of resources when one type of resource is not available while others are idle. For instance, under gang-scheduling, the CPU can remain severely underutilized during time-slices dedicated to I/O intensive tasks [74]. Therefore, a unified resource scheduling policy is important specially in the face of ever widening performance gap between different system resources (e.g., between CPU and I/O resources).

Some studies have considered coordinated allocation of processors and I/O resources based on ideal parallel job and system environment assumptions [97, 15,

63, 99]. What is lacking is an approach that combines processor and I/O resources scheduling in a unified manner while addressing the unique requirements of the cluster computing environments that includes:

1. **Heterogeneity:** Processors and the I/O resources are more likely to be heterogeneous. This heterogeneity can cause load imbalance and reduce the performance of parallel applications.
2. **Availability:** Resource availability can change dynamically at run-time.
3. **Remote I/O:** Jobs may execute at sites that are distant from their data. This leads to a remote I/O access across networks, thus the I/O cost is many times more than the local I/O cost.

4.1.1 Problem Statement

The problem of coordinated allocation of processor and I/O resources has three fundamental aspects that are strongly interrelated:

1. The behavior of parallel applications and workloads with respect to computation and I/O as a function of the number of processors and disks on which they are executed;
2. A framework that binds the computing power and the data resources; and
3. Allocation of the resources among the system workloads to satisfy various objective functions (e.g., minimize response time).

The first issue has been studied quite extensively while the last two have not been satisfactorily addressed especially in cluster computing environments. In this chapter, we address these two key issues of the general multiple resource allocation problem with emphasis on cluster computing systems. In this section, we give a formal

definition of the resource allocation problem and briefly describe the contribution of this chapter.

Coordinated CPU and I/O Resources Allocation Problem

Given: *A set of jobs that arrive in a stochastic manner to a system composed of $S = \{s_1, s_2, \dots, s_i\}$ clusters, each cluster with a set of sharable (i.e., community-based), $s_i = \{p_1, p_2, p_j\}$, workstations and $D = \{d_1, d_2, d_k\}$ sharable disks.*

Scheduling Problem: *The problem is to design a scheduling policy that schedules the set of jobs onto the set of clusters with the objectives of improving the average response time and the system utilization, where the choice of scheduling must be made on-line under the following constraints:*

1. The scheduler has no information on the service time of the jobs until the jobs complete execution. Also, there is no knowledge on the remaining processing requirements of the tasks. Also, there is no advance knowledge about the job arrivals and how many processors each job needs until the job actually arrives.
2. A task can execute only after all input data have been received, amount of input data sets needed from a given data storage is known at the time the job is submitted and at most one task can access any data storage at any given time.
3. There is no knowledge of the set of processors available for scheduling to the scheduler and resource availability may change dynamically, and the patterns of these changes are not known in advance.

Figure 4.1: Coordinated processors and I/O resources allocation problem

The solution to the general scheduling problem is known to be NP-complete [126] hence this scheduling policy is an intractable problem as well. The problem becomes much more complicated on clusters of workstations where, in addition to background load, heterogeneity may exist in both processors and I/O subsystems. Large number of users attempting to concurrently execute jobs on the system, parallelism of the applications and their respective computational and storage requirements are all issues that make difficult the resource scheduling problem in these systems.

4.1.2 Contributions

1. We propose a unified framework for resource scheduling in cluster computing systems. The proposed framework is a vertically distributed framework that defines bindings between computing power and data resources.
2. Based on the proposed framework, we propose two scheduling policies. In the first policy data movement operations is tightly bound to job scheduling decisions. In the other policy, scheduling of the data and computation is performed by a decoupled, asynchronous process on the basis of the observed data access patterns and local load conditions.
3. The benefits of using the proposed scheduling policies in terms of reduced mean response time of the jobs is studied through simulation under various system and workload parameters.

4.2 System and Workload Models

4.2.1 System Architecture

The system has P shareable (i.e., community-based) workstations that can be grouped into clusters according to their architectures, processing speed, and physical locations. Managing and utilizing these resources to meet performance and usability goals of high performance applications has become a significant challenge [46, 138]. In order to address this issue, we partition the P processors and the set of N disks into a set of disjoint clusters as shown in Figure 4.2. We refer to the resulting system architecture as a *cluster tree*. Note that cluster tree is defined in the previous chapter.

Any data required to run a job is fetched before the task is run if it is not already present. When the job and the I/O server are located in different clusters we say that the I/O request is a remote request but if they are in the same cluster we say

Figure 4.2: An overview of the system architecture. **P**: Processors; **LS**: Local Scheduler; **CS**: Cluster Scheduler; **SS**: System Scheduler; R_{IO} : remote I/O request and L_{IO} : local I/O request.

that the I/O request is a local request. All I/O requests from a job is sent to the *Data Scheduler* in the I/O subsystem, which coordinates the I/O request on multiple I/O nodes for greater efficiency. Upon arrival, if an I/O request cannot be serviced immediately, it is placed into an I/O wait queue. When an I/O device becomes available, the scheduler schedules a pending request from the wait queue using the FCFS policy.

This system structure is a natural structure for localizing application I/O on the system composed of multiple clusters. By binding CPUs and storage devices together into organizations that reflect the physical entity, we increase the performance of applications and the utilization of system resources. The other benefits is that hierarchy can yield scalability quite naturally.

4.2.2 Workload Models

We assume that parallel applications and a stream of local jobs compete for system resources. The parallel workloads of interest to us are the *compute-intensive* and *CPU-I/O bound*. The CPU-intensive parallel jobs have been described in Chapter 3. Thus we focus on the CPU-I/O bound job characteristics here.

Figure 4.3: The I/O properties of parallel applications.

Figure 4.4: The I/O properties of parallel applications: (a) non-overlapped execution, (b) overlapped execution.

We focus on I/O-bound workload requirements that have been identified in recent

characterization studies on a number of platforms [100, 99, 93, 39, 114, 80, 36, 83]. Based on these studies, the execution behavior of CPU-I/O bound jobs can be naturally partitioned into disjoint intervals as shown in Figure 4.3. Each interval consists of a single burst of I/O with a minimal amount of computation followed by a single burst of computation with a minimal amount of I/O. For example, each job uses the CPU for an average of 5-msec between 20-msec I/O requests. In some cases, the execution of the computation phases and the I/O phases can be interleaved as in Figure 4.4 (b) or sequential (i.e., non-overlapping) as in Figure 4.4 (a).

We studied two CPU-I/O bound workloads referred to as W_1 and W_2 , which are the same workloads used in [72]. These two workloads are characterized by alternating CPU and I/O phases that repeat K times as observed in [93, 114]. In *W1 Workload*, the number of phases is set to 2. In contrast, the number of phases is set to 12 in *W2 workload*.

Each job begins execution by reading its input data from an I/O device and then execution proceeds by alternating cycles of computation and I/O. The last operation of each job is writing its data to an I/O device and when all the tasks of the job have finished their last writes, the job terminates.

We use a model of [99, 72] to determine the proportion of the computation and I/O times for each task as follows:

$$T_n = T_n^{IO} + T_n^{CPU} \quad (4.1)$$

where T_n is the time or length of the n^{th} phase, T_n^{IO} is the time it takes to service the I/O request and T_n^{CPU} is the computation service time of the phase. We can define the fraction of the n^{th} phase that represents the length of the I/O burst in a given phase as follows:

$$T_n^{IO} = T_n \times f_{IO} \quad (4.2)$$

where f_{IO} is the I/O factor, which defines the I/O to CPU ratio for a given phase [72].

4.3 Coordinated Resource Scheduling Policies

4.3.1 Proposed Scheduling Policies

We now describe two adaptive scheduling policies, which we refer to as the *Adaptive Global Scheduling (AGS)* policy and *Adaptive Local Scheduling (ALS)* policy. Both scheduling policies use self-scheduling and favor local execution (i.e., job and data are co-located) in order to reduce response time, network traffic and resource usage cost. Resources not local to the user are only used when there is lack of enough local resources.

4.3.1.1 Adaptive Global Scheduling Policy

The *Adaptive Global Scheduling (AGS)* policy uses the self-scheduling approach described in the previous chapter. For the sake of brevity, we will not describe that policy here; only the scheduling part of the algorithm is described. Jobs are submitted to the system scheduler where they are stored in the job wait queue until a placement decision is made. Figure 4.5 shows the pseudo-code of the AGS policy.

When a node in the cluster tree receives a Request for Computation (RFC) for R_{work} from a child, the algorithm first determines an ideal number of jobs (i.e., N_{ideal}) that can be moved down one level from the parent node to the child node. Once the ideal number of computations to be transferred is determined as per Equation (4.3), the algorithm applies *affinity-scheduling*, which selects J_{send} jobs from the parent job wait queue. The J_{send} jobs are then assigned a *partition-size* (i.e., a set of processors) equal to the partition-reach of the child node on which these jobs can possibly execute. Note that if there is no computation that meet the affinity condition, then the actual

Algorithm 1: AGS policy

1. Determine an ideal number of jobs that can be assigned to a child node as follows:

$$N_{ideal} = \lceil T_r \times N_{work} \rceil \quad (4.3)$$

where N_{work} is the number of jobs waiting to be scheduled when the RFC from a child node arrives at the parent node and T_r is the *transfer factor* defined as follows:

$$T_r = \frac{\text{partition-reach of the child node}}{\text{partition-reach of the parent node}} \quad (4.4)$$

2. Select J_{send} jobs from the parent job wait queue as follows:

$$J_{send} = \begin{cases} \min(N_{home}, N_{ideal}) & \text{if affinity;} \\ R_{work} & \text{Otherwise.} \end{cases} \quad (4.5)$$

where N_{home} are jobs that have their data within the partition-reach of the child node. The R_{work} is the number of jobs requested by the child scheduler.

3. Forward the J_{send} jobs to the requesting node.
-

Figure 4.5: Adaptive global scheduling policy

number transfered down one level from the parent to the child node is set to R_{work} .

Figure 4.6 illustrates how the AGS policy works. Assume that there are nine unscheduled jobs at the *system scheduler (SS)* while first level and second level *cluster schedulers (CS)* are both in a *neutral state*. Also assume that there are 9 clusters in the system each cluster has four processors. Further assume four out of the nine jobs at the *system scheduler* have their data located at *cluster₂*.

Suppose that at some point in time, processor P_1 sends a RFC to its parent which is the CS at *level₁*. Since the CS at both levels are in a *neutral state*, the RFC will eventually reach the *system scheduler*. When the *system scheduler* receives the RFC, it applies *space-sharing* to the nine jobs in the job waiting queue, which results in three jobs being transfered one level down to CS. By this move, these three jobs are assigned a partition-size equal to the number of processors reachable from CS at

Figure 4.6: An illustration of the adaptive global scheduling policy

level₂. When CS receives the three jobs, it also applies *space-sharing* to the three jobs, which results in one of the jobs being dispatched to CS at *level₁*. This means the partition-size of the job is now set to four processors. At this point, the job is split into 4 tasks and one of the tasks is then sent to P_1 while the remaining three tasks are queued in the task wait queue such that any of the four processors within the *cluster₂* can execute it. Notice that this process leaves varying number of jobs along the path. Thus, it is not necessary for some queries from other processors to reach the *system scheduler* to get work.

4.3.2 Adaptive Local Cluster Scheduling Policy

The core of the Adaptive Local Scheduling (ALS) policy is the same as the AGS in that it has components that perform job assignment, affinity-scheduling, and self-scheduling. However, the ALS policy differs from the AGS policy in three aspects: First, local jobs have priority for being executed locally over non-local jobs. In case this is impossible, it employs data pre-staging to the remote site before the job is

actually executed such that the communication and computation are overlapped. Second, jobs are submitted to the local cluster scheduler (i.e., where the data of the jobs is located). Also, each cluster scheduler keeps track of the number of pending jobs for which the data is located in the cluster excluding those being executed. Finally, the schedulers use self-scheduling as in the AGS, but self-scheduling is initiated in either of the following two instances:

1. When there are pending local jobs in the system and the current load level is less than the base load level; or
2. When there is no local pending jobs but the current load is 0.

Figure 4.7 is the pseudo-code of the adaptive local scheduling policy. The scheduler at each node of the cluster tree is invoked in three cases: (1) job arrival events; (2) non-root nodes looking for work; (3) and by the system schedulers. When a job arrives at a node, the node will keep the job provided that the number of jobs at the node is less than its base load level. Otherwise, the scheduler forwards the job to its parent at the next level. This process is recursively followed up the hierarchy until either a scheduler that is able to accept the job is found along the path or the job reaches the system scheduler. In the later case, the job is stored in the *job wait queue* until a placement decision is made. The ALS policy can lead to a situation where a local job is moved to another location for execution even though the local cluster is available to run the job. We plan to address this problem in a future study.

We illustrate this process using Figure 4.8a. In the figure, the arrows show the direction of the job movement. The numbers on the circles indicate the number of jobs waiting to be scheduled. We also set the base load level of the non-leaf nodes to B^{level} , where *level* is the location of the scheduler in the cluster tree. Assume that there are two jobs (i.e., J_1 and J_2) are waiting at cluster 1 when a new job (i.e., J_8) arrives at CS_1 . Since the current load levels at CS_1 and CS_5 are equal to their base

Algorithm 2: ALS policy

1. When a new job arrives at *scheduler*, the following steps are performed:
 - (a) Keep the job if the current load level is less than the base load level.
 - (b) Otherwise, forward the job to parent at the next level.
 2. When a request for computation arrives, the following steps are performed:
 - (a) Determine an ideal number of jobs that can be assigned to a child node as follows:
$$N_{ideal} = \lceil T_r \times N_{work} \rceil \quad (4.6)$$
where N_{work} is the number of jobs waiting to be scheduled and T_r is defined as in Equation 4.4.
 - (b) Select J_{send} jobs from the parent job wait queue as follows:
$$J_{send} = \begin{cases} \min(N_{local}, N_{ideal}) & \text{if affinity;} \\ R_{work} & \text{Otherwise.} \end{cases} \quad (4.7)$$
where N_{local} are jobs that have their data within the partition-reach of the child node or jobs that are assigned to the clusters based on the nearest sibling. R_{work} is the number of jobs requested by the child scheduler.
 3. When the load on a given cluster exceeds the base load level, the push computation is initiated as follows:
 - (a) Push a job to the least loaded cluster
 - (b) The local cluster forward the data of the job to the remote cluster.
-

Figure 4.7: Adaptive local scheduling policy

load levels, respectively, J_8 will be automatically forwarded to the system scheduler.

When the load of a given node in the cluster falls below its base load level, the node pulls its parent for jobs. This triggers the Pull Computation step of the ALS algorithm. The scheduler will give priority to jobs that have their data within the partition reach of the child scheduler. Two cases are possible. First, there is a job waiting to be scheduled with its data in the requesting cluster. This case is illustrated in Figure 4.8b. Here job J_1 that was running at $cluster_1$ completes execution. This

Figure 4.8: An illustration of the adaptive local scheduling policy: (a) Job submission. (b) Self-scheduling and job assignment.

event triggers self-scheduling by both CS_1 and CS_5 , which leads to the transfer of J_8 from SS to CS_5 and J_5 from CS_5 to CS_1 as depicted in Figure 4.8b.

The second case is that there is no job that has its data in the requesting cluster. In this case, the scheduler will try to allocate a job with least common ancestor. Figure 4.9a illustrates this process. Suppose two local jobs (i.e., J_8 and J_{10}) are being executed in $cluster_4$. Since the current load level of CS_4 is zero, it initiates self-scheduling and sends a request for work to its parent which is CS_6 , which in turn pulls the system scheduler. This results in the transfer of job J_7 and its data from CS_2 to CS_4 where it will subsequently be executed. Note that if the current load level of CS_6 is greater than zero at the time of the request arrival from CS_4 , this means that there are jobs at CS_3 waiting to be scheduled. In this case, a job along its data are transferred from CS_3 to CS_4 .

Finally, when the number of jobs for a given cluster exceeds a given threshold, the system scheduler reassigns the job to the least loaded cluster (i.e., push computation step) and at the same time instructs the cluster where the data of the job initially exists to send a copy of the data to the new home of the job. Figure 4.9b illustrates

Figure 4.9: An illustration of the adaptive local scheduling policy.

this case. Assume that two jobs (i.e., J_1 and J_2) are being executed at $cluster_1$ while six jobs with their data stored at $cluster_1$ are waiting (i.e., J_3 and J_4 are waiting at CS_5 and J_5, J_6, J_9 and J_8 are waiting at the systems scheduler) to be scheduled. In this case, the number of jobs waiting to be executed on $cluster_1$ is greater than the threshold for the CS_1 . This triggers the job transfer activity by CS_1 to off-load some of the jobs to other clusters.

4.3.3 Baseline Scheduling Policies

It would be good to compare the scheduling policies developed in this chapter against an optimal scheduling policy. However, due to the fact that the complexity and dynamic nature of the cluster computing system makes it infeasible to compare the proposed policy against an optimal schedule. For instance, the differences in processor load and available memory across the machines in the testbeds change frequently. Finding the optimal schedule requires a way to predict these changes in order to choose the optimal node for each task. Such a prediction in even a simple cluster system is difficult, if not impossible. Therefore, we compare our scheduling policies

with two existing policies called the *static space sharing time-sharing (SSS)* policy [99] and *static space sharing with time-sharing (SST)* policy [99]. The following two subsections briefly describes these two policies along the modifications we introduced to make them suitable for cluster computing environments.

4.3.3.1 Static space sharing Policy

The *static space (SSS)* policy is a two-level scheduling policy. At the top-level there is a *global scheduler* with a FCFS infinite-capacity queue used to hold all jobs that are waiting to be allocated resources. At the second-level, there are m disjoint clusters, $S=C_1, \dots, C_m$, each cluster with p processors and d disks. Jobs arrive into the system scheduler, which assigns the parallel jobs to a processor-disk pair partition on a FCFS basis. Since the original policy did not consider data locality as well as heterogeneity, we slightly modified the original SST policy such that affinity scheduling and heterogeneity of the systems are taken into account when making the job placement decisions. In the modified SSS policy, two events activate the scheduler: the arrival of new jobs and the completion of a job. If the job cannot be assigned to a partition upon arrival, it is placed in the FCFS queue until a partition becomes free. Otherwise, the scheduler maps jobs to clusters as follows:

1. If a partition where the job's data is located is available, assign the job to it.
2. Otherwise, the job is mapped to an available cluster and if there are more than one such clusters, the fastest cluster is chosen.

A cluster is exclusively allocated to a job whereas the disk partitions are shared. Every job is executed to completion on the assigned partition. Upon job departure, the freed partition is allocated to a job that has its data located in the cluster if there is one. Otherwise, the cluster is allocated to a job at the head of the wait queue.

4.3.3.2 Static Space Sharing with Time-Sharing

The *static space sharing with time-sharing (SST)* policy is also a two-level scheduling policy and partitions the set of P processors and N disks in the system into a set of disjoint clusters and assigns parallel jobs to a processor-disk pair partition such that at most MPL number of jobs can be present in a given processor-disk pair partition at any given time. Note that MPL represents the multiprogramming level.

Since the original policy did not consider data locality as well as heterogeneity, we slightly modified the original SST policy such that affinity scheduling and heterogeneity of the systems are taken into account when making the job placement decisions. The modified SST policy works as follows: When the scheduler is activated by the arrival of new jobs or completion of a job, a partition where the data associated with the job is located is chosen first. If this fails, a job is assigned to processor-disk pair partition where both partitions are idle (i.e., no jobs bound to them). If there are more than one such pairs, the fastest partition is chosen. Last but not least, if there is no idle processor-disk pair, a pair where at least one of the two partitions is idle is selected. Otherwise, if there is a scheduling slot where both partitions are already busy with less than MPL jobs each, then the job is scheduled onto it. If the job cannot be assigned to a partition upon arrival, it then waits in the FCFS queue until a partition becomes less than MPL.

4.4 Performance Evaluation

We used discrete event simulation written in C programming language to study the performance of the scheduling policies discussed in the previous section. Table 4.1 shows the default system and workload parameters used in the simulation.

Performance Metrics: We use the mean response time (MRT) and average utiliza-

Parameters	Values
Parallel job mean service demand	20.0
I/O Factor	50%
Local job mean service demand	2.0
Network Latency	15 μ sec
Parallel job arrival CV	1.0
Bandwidth	100 Mbps
Number of clusters	8
Disks in a cluster	1
Service demand CV	3.5
Multiprogramming level	2

Table 4.1: Default parameters and values used in the experiments.

tion as chief performance metric as in [124, 99]. The MRT is defined as the sum of all job response time divided by the number of completed jobs. The average utilization is defined as the job arrival rate times the mean service demand of the jobs divided by the number of processors in the system.

Validation: We used the independent replication method to validate the results. For every mean value, a 95% confidence level was computed. All confidence levels were within 5% of the mean values.

4.4.1 System Model

The system consists of 64 workstations partitioned into 8 clusters, each with 8 workstations. We create a 4-levels cluster tree with the root as the system scheduler, four second-level cluster schedulers that act as children of the system scheduler, and 64 local schedulers at the bottom. Each workstation is connected to a 100Mbps fast Ethernet switch, which in turn is connected via gigabit Ethernet switches. We used 200μ s for the context switch overhead and 100-ms for the time-slice, which is consistent with most of modern operating system for workstations. Also, each workstation is equipped with a disk with performances characteristics similar to an IBM Ultra-

star 9ZX disk delivering a transfer rate of approximately 204.8 Mbites/sec with the rotational Speed (RPM) of 10,020, average read seek time and latency of 6.3 ms and 2.99 ms, respectively.

We used the SPECfp2000 results for a family of Intel Pentium 3 and 4 processors with different clock speeds to model the processor speeds. As in [97], the communication time (T_{comm}) is modeled as the time required to transfer data across the available network bandwidth as follows:

$$T_{comm} = \text{Startup} + \frac{\text{Message size}}{\text{Bandwidth}} \quad (4.8)$$

It has been shown that a communication model that accounts for message startup time and network bandwidth gives adequate result [130]. In all the experiments discussed here, the communication network latency to be 15 μ sec with the transfer rate of 100Mbites/sec. These values are typical of modern light-weight messaging layers running on top of gigabit switched LANs [34].

4.4.2 Workload Model

Background Workload: Each workstation services a stream of background tasks that arrive at individual workstations independently. We used the same approach as in Chapter 3 to model the utilization of a given workstation by background jobs.

Parallel Workload Models: We assume that a set of n jobs, $\mathbf{J}=\{J_1, \dots, J_n\}$ arrives into the system in a stochastic manner. Each job, J_i , can be decomposed into t tasks, $\mathbf{T}=\{T_1, \dots, T_t\}$. Each task T_i executes sequential code and is fully preemptable. Parallel jobs arrive to the system scheduler with Poisson inter-arrival times [72, 124] and α percentage of the arriving jobs are CPU-I/O intensive while the remaining

$(1 - \alpha)$ fraction of the jobs are CPU-bound jobs, where α is uniformly distributed over the range of 0 to 1. The maximum parallelism of the jobs is uniformly distributed between 1 and 32 [124], while the service demands of the jobs are generated using hyper-exponential distribution with mean 20.0 and CVs of 3.5 [72, 124]. Each task of the job begins execution by reading its input data from an I/O device. The execution proceeds by alternating cycles of computation and I/O. The last operation of each task is writing its data to an I/O device and when all the tasks of the job have finished their last writes, the job terminates.

We assume that a job requires a single input file as the work of [21] shows that usually most file operations are performed by a process on a single file. Any data required to run a job is fetched before the task is run if it is not already present locally. The location of the data for a given job is determined uniformly between 1 to the number of clusters in the system. Note that when W1 and W2 jobs are split into n tasks, the number of I/O's do not change, but the amount of data being written/read changes. This is captured in our model by increasing the service requirements at the I/O center. A lot of parallel scientific codes work this way [99], and this is the same approach used in [99].

4.5 Simulation Results and Discussions

In this section, we present and discuss results of the experiments. The results for the dedicated heterogeneous setup is not include here as it is similar to the previous chapter. In all graphs, we append *HOMO* and *HETR* to the names of the workloads (e.g., W1-HOMO) to denote the performance of the workload in shared homogeneous (HOMO) environments and shared heterogeneous environments (HETR) respectively. Also, parameters are set to their default values shown in Table 4.1 unless stated otherwise.

4.5.1 Validation of the Baseline Policies

In order to validate the correctness of the implementation of the original static space sharing (SSS) policy and static space sharing with time-sharing (SST) policy as well as the modified versions of the static space sharing (MSSS) policy and static space sharing with time-sharing (MSST) policy, we conducted several experiments for the workload and experimental setup discussed above. In the experiments, the number of workstations in a cluster is set to 8 and the number of disks per cluster is kept at 1.

Figure 4.10: Validation of the baseline and modified policies.

Figure 4.10 shows that the original versions have the general shape that agrees with the results given in [99]. Figure 4.10 also shows that the proposed optimizations do improve the mean response time. In the rest of the chapter, when we use static space sharing (SSS) and the static space sharing with time-sharing (SST) policies, we mean the modified versions of these two policies.

4.5.2 Unified versus Separate Resource Scheduling

In this section, we use Adaptive Global Scheduling and the Adaptive Hierarchical Scheduling policies. The objective of this experiment is to quantify performance benefits of taking into account the location of data when making scheduling decisions. Figure 4.11 shows the relative performance of the Adaptive Hierarchical Scheduling (AHS) policy and the Adaptive Global Scheduling (AGS) policy as a function of the system load. From this data, we observe that AGS performs substantially better than the AHS policy. This is due to the fact that the number of remote I/O requests in AGS policy are substantially reduced due to its job and data co-allocation capability.

These results demonstrate that performance trends are significantly different from those previously reported where application I/O characteristics were not explicitly considered in the job scheduling policy. The conclusion from this experiment is that when scheduling I/O bound applications, explicit support for scheduling both the processor and I/O resources is required.

4.5.3 Relative Performance of Unified Scheduling Policies

4.5.3.1 Shared Homogeneous Environments

Figure 4.12 shows the mean response time (MRT) of the proposed policies against the baseline policies in shared homogeneous environments. The data show that the proposed policies perform better than the baseline policies. The ALS, AGS and SST perform better than the SSS policy since the space-sharing policy has the disadvantage that when processes perform I/O, the processors on which the process is running remain idle. In contrast, a process ready to run in the ready queue is immediately scheduled in the other three policies. The result also shows that the ALS policy performs better than all the other policies. This is mainly due to the fact that the ALS optimizes locality of file accesses to reduce the amount of communication for

Figure 4.11: Relative performance of AGS and AHS policies.

remote accesses. This is in addition to the overlapping of computation and I/O bursts from different jobs, which results in the most efficient utilization of the resources.

4.5.3.2 Shared Heterogeneous Environments

With respect to shared heterogeneous environments, Figure 4.13 shows that the performance trend is similar to that in the previous case. The proposed scheduling policies perform substantially better than the baseline policies at all utilization lev-

Figure 4.12: Relative performance of the four policies in shared homogeneous environments.

els. One of the reasons for this is that in the baseline policies, a partition may be shared by up to a maximum multiprogramming level of jobs. The problem with this is that it may lead to system fragmentation, which consequently degrades system performance as resources can sit idle while there are jobs to be processed. It also increase the waiting time of the jobs. This is not the case in the proposed policies as we use self-scheduling, which leads faster processors executing more jobs. Another observation from the graph is that the ASL, GSL and SST policies perform better

than the space-sharing policy because they overlap the I/O demands of some jobs with the computational demands of other jobs, which offer a potential improvement in performance. Finally, the data on Figure 4.13 shows that the ALS policy outperforms the other three policies as it optimizes locality of file accesses to reduce the amount of communication for remote accesses. Remote I/O holds CPUs idle while waiting for I/O, yielding a poor efficiency. This situation worsens in cases where the time necessary to transfer the dataset from the remote site is greater than the execution time of the job performing remote I/O.

4.5.4 Sensitivity to I/O Frequency

Note that W1 and W2 workloads mainly differ in the number of CPU-I/O phases and by comparing the performance of the two workloads, we can study the impacts of the number of I/Os in a job on the performance of the two workloads. Therefore, we can observe from the data on Figure 4.11 that reducing the number of I/Os can lead to lower mean response under both scheduling policies, which agrees with the results reported in [80].

The degradation in the performance as the number of job I/Os increase can be explained by the fact that as the number of I/Os in a job increase, there will be more contention for I/O resources, which may lead to longer execution times of the jobs. The longer a job experiences delays the more likely that a job can be interrupted by a background workload leading to even longer completion times of the jobs. In summary, this result suggests that although reducing number of I/Os per job is important in reducing response time, the impact of the job scheduling policy used is more important than simply reducing the number of I/Os.

Figure 4.13: Relative performance of the four policies in shared heterogeneous environments.

4.5.5 Sensitivity to Service Demand Variations

Figure 4.14 and Figure 4.15 show the performance of the four scheduling policies as a function of the service demand coefficient of variations (CVS). In the experiments, we varied the values of the CVS parameter (x-axis) while the utilization is set to 80% and all other parameters are set to their default values as given in Table 4.1.

Note that, higher variance in service demand implies a larger number of small

Figure 4.14: Sensitivity to service demand CV in shared homogeneous environments.

jobs and fewer large jobs. As can be seen from the graph of Figure 4.14, the mean response times of the two workloads increase with the increases with the CVS for all policies. However, the increase in mean response time is at a faster pace in the baseline policies specially the space-sharing policy. This is because in a mixture of jobs consisting of many small jobs and a few large jobs, there is a tendency for the smaller jobs to be enqueued for long durations because of larger jobs in front of them.

Figure 4.15: Sensitivity to service demand CV in shared heterogeneous environments.

4.5.6 Sensitivity to Arrival Time Variations

Figure 4.16 and Figure 4.17 show the performance of the four scheduling policies as a function of the parallel job arrival time coefficient of variations (CVA). In the experiments, we varied the values of the CVA parameter (x-axis) while the utilization is set to 80% and all other parameters are set to their default values shown in Table 4.1. As in [124], we used a two-stage hyper-exponential distribution for $CVA > 1$. Note that, higher variance in arrival time implies clustered job arrivals into the

system.

Figure 4.16: Sensitivity to arrival time CV in shared homogeneous environments.

From the graph of Figure 4.16 and Figure 4.17, it can be seen that ALS and AGS policies are less sensitive to arrival time variations than the baseline policies. This is because of the fact that both AGS and ALS policies limit the partition-size of the jobs through space sharing policy. This behavior leads the AHS to reduce the waiting time of the jobs at the same time decreasing the mean response time. In contrast, no more than MPL jobs can be run under SST policies in a cluster which can lead to

Figure 4.17: Sensitivity to arrival time CV in shared heterogeneous environments.

longer waiting time of unscheduled jobs subsequently increasing the mean response time.

4.5.7 Sensitivity to I/O Factor

Figure 4.18 and Figure 4.19 show the performance of the four scheduling policies as a function of the I/O factor (x-axis). In the experiments, the utilization is set to 70% while all other parameters are set to their default values shown in Table 4.1. From the

data in Figure 4.18 we observe that as the I/O factor increases, the mean response time of the two workloads under all policies also increases. This is because of the fact that with the increase of I/O factor, the jobs become more I/O-bound and less compute-bound. As jobs become more I/O-bound, the competition for I/O devices also increases and there is a high probability of background workload interference in the execution of the parallel jobs.

Figure 4.18: Sensitivity to I/O factor in shared homogeneous environments.

Figure 4.19: Sensitivity to I/O factor in shared heterogeneous environments.

4.5.8 Sensitivity to Background Workload Arrival Variations

Figure 4.20 shows the performance of the four scheduling policies as a function of the background job workstation utilization variations (x-axis). The results show that as the utilization increases, the performance of all policies deteriorate. This is because of the fact that the idle cycles of the workstations available to parallel jobs decreases. Also, the interference from the background jobs on the execution of parallel jobs substantially increases.

Figure 4.20: Sensitivity to background workload utilization: shared homogeneous environments (top) and shared heterogeneous environments (bottom).

4.5.9 Sensitivity to Number of Disks

Figure 4.21 shows performance sensitivity of the scheduling policies as a function of the number of disks when the system load is fixed at 70% while all other parameters are set to their default values for W1 workload. The results for the other cases are qualitatively the same. The data show that as the number of disks increases the mean response time of all scheduling policies substantially decreases. This graph

also captures the relationship between the mean response time and the number of I/O devices in that there is a distinct knee approximately at $D=4$. Initially, as the number of I/O devices is increased a large reduction in mean job response time is achieved. However, increasing the I/O devices beyond this knee produces only a limited improvement in performance.

Figure 4.21: Sensitivity to the number of disks.

4.5.10 Sensitivity to Multiprogramming Level

In this section, we investigate the impact of multiprogramming level on the performance of the system. The multiprogramming level that provides the best performance depends in part on the ratio of the per-phase computation and I/O demands. In the ideal case the scheduling policy will be able to interleave the execution of jobs such that this ratio is maintained very close to unity, thus achieving a total overlapping of computation and I/O requests.

Figure 4.22 shows the performance sensitivity of the scheduling policies to the multiprogramming level when the system load is fixed at 70%. We observe that as the MPL increases so does the mean response time of all policies. This is because

Figure 4.22: Sensitivity to the multiprogramming level.

of the fact that contention for I/O resources increases. Moreover, the context switch overhead increases increasing the response time of the jobs.

4.5.11 Sensitivity to Network Bandwidth

The performance of the scheduling policies can be affected by network parameters (i.e., communication latency, overhead, and bandwidth). In this section, we investigate the sensitivity of the scheduling policies to network bandwidth. In order to observe the effects of the bandwidth on the performance of the policies, the bandwidth is varied between 10 to 100 Mbits/sec while the system load is fixed at 80%.

The results of the experiment are shown in Figure 4.23. The data show that the proposed policies are less sensitive to bandwidth as compared to the other policies. This is due to the fact that our policies try best to overlap computation and communications.

Figure 4.23: Sensitivity to the network bandwidth.

4.6 Summary

The I/O behavior of parallel workloads can have a dominating effect on system performance. To this end, we studied the problem of combined CPU and I/O resource scheduling and proposed two policies referred to as AGS and ALS policies. These policies explicitly co-schedule both the processors and the I/O resources on demand by moving computation across the I/O bottleneck and closer to the data it uses. We compared performance of these two policies with a static space sharing and a static space-sharing and time-sharing policies. We showed that the proposed policies perform substantially better than the existing policies under all system and workload parameters.

Chapter 5

Fault-Tolerant Job Scheduling

With the momentum gaining for the cluster computing systems, the issue of deploying support for integrated scheduling and fault-tolerant approaches becomes paramount importance. In this chapter, we address this problem. We propose a reconfigurable hierarchical infrastructure that provides fault tolerant scheduling services across a network of unreliable clusters of workstations¹. Based on this infrastructure, we develop a mechanism that couples dynamic job scheduling with job replication approach such that parallel jobs submitted to the system are efficiently and reliably executed in the presence of node and link failures².

¹J. H. Abawajy and S.P. Dandamudi, "A Reconfigurable Multi-Layered Grid Scheduling Infrastructure," *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, Pages: 138 - 144, June 23 - 26, 2003, Las Vegas, Nevada, USA.

²J. H. Abawajy and S.P. Dandamudi, "Fault-Tolerant Multi-Cluster Resource Management Infrastructure," To appear in a *Special Issue on Grid Computing of the Journal of Neural, Parallel and Scientific Computations* with Guest Editors: H.R. Arabnia, G.A. Gravvanis and M.P. Bekakos

5.1 Background and Problem Statement

5.1.1 Motivation

Cluster computing systems can potentially furnish large computational and storage resources to solve large-scale problems. As a result, they are increasingly used for applications requiring high levels of performance and reliability [38]. However, as the users may switch off the workstations as well as load the system heavily, cluster computing systems are highly susceptible to a variety of failures including node failure, interconnection network failure, scheduling middleware failure, and application failure. Also, as systems become larger and more complex, the challenge for attaining dependable distributed systems only escalates.

Due to these vulnerabilities, achieving large-scale distributed computing in a seamless manner on cluster computing systems introduces not only the problem of efficient utilization and satisfactory response time but also the problem of fault-tolerance. Therefore, the ability to tolerate failures while effectively exploiting the variably sized pools of cluster computing resources in an scalable and transparent manner must be an integral part of cluster computing systems. This requires that handling faults and resource allocation to work in concert without compromising the high-performance objectives of cluster computing systems as well as the reliability requirements of the end-user applications.

Achieving integrated scheduling and fault-tolerance goal is a difficult proposition as the job scheduling and fault-tolerance are difficult problems to solve in their own right. The research described in this chapter presents the approach we used to realize this goal. Specifically, we address the problem of survivability of the scheduling middleware in the presence of failures. We present a strategy to enhance the survivability of the scheduling middleware. We also address the problem of integrated scheduling and fault-tolerance and propose a scheme that can be used to solve this problem.

In the next section, a formal definition of this problem is given. This section also establishes the fact that, to a large extent, the problem considered in this chapter has not been fully addressed in the literature.

5.1.2 Problem Statement and Related Work

We define a fault-tolerant scheduling (FTS) policy as an on-line, highly available and scalable policy that allows jobs to be efficiently and reliably executed in the presence of failures and repairs. The fault-tolerant scheduling (FTS) problem addressed in this chapter can be formally stated as shown in Figure 5.1. Note that by on-line scheduling we mean that the scheduler has no knowledge of: (1) the service time of the jobs or the tasks; (2) the job arrival times; (3) how many processors each job needs until the job actually arrives; (4) and the set of processors available for scheduling the jobs.

Fault-Tolerant Scheduling (FTS)

Given: A set of n jobs, $\mathbf{J}=\{J_1, \dots, J_n\}$, where each job, J_i , arrives in a stochastic manner into a system composed of m independent clusters, $\mathbf{C}=\{C_1, \dots, C_m\}$. Each cluster is composed of m shareable (i.e., community-based) processors, $C_j = \{P_1, \dots, P_m\}$. Each processor, P_1 , may fail with probability f , $0 \leq f \leq 1$, and be repaired independently.

FTS Problem: Our goal is to design a highly available on-line scheduling policy such that mean response time is reduced while applications are reliably executed to their logical termination.

Figure 5.1: Definition of fault-tolerant parallel job scheduling problem.

Scheduler Organization: We define a scalable and highly available scheduling infrastructure as one that is able to deliver continuous service to users despite system size and component failures. Three commonly used scheduling infrastructures are the centralized, hierarchical and distributed architectures. The distributed framework can provide high availability and scalability of the scheduling middleware. However, due to the inherent overheads associated with distributed framework as well as its

complexity in implementation and maintenance, most cluster systems employ a centralized framework (e.g., [23, 106]). A key disadvantage of using a central manager is that it becomes a critical, single-point of failure; in addition, it does not scale well. Hierarchical approach (e.g., [19]) makes use of a layered structure of the job scheduling wherein different levels of schedulers execute scheduling policies over their own relevant plane. However, although hierarchical organizations can scale, they are also susceptible to failures at multiple points (e.g., the root node).

Therefore, techniques to improve reliability and availability of the scheduling middleware are clearly needed. However, as noted in [107], most of the existing scheduling policies neglect the need for a solution to this problem. In fact, scheduling middleware is commonly assumed to be a failure-free (e.g., [88, 136]) component of resource management infrastructures. This assumption is quite unrealistic in networked distributed systems such as cluster computing environments, where the system is exposed to a variety of failures.

Fault-tolerance: Research in fault-tolerance has focused on reliability issues, which has resulted in various application fault-tolerance approaches [75, 88, 87]. In checkpoint-recovery approach [75], applications instrument themselves with the ability to checkpoint their computation states to stable storage at a regular interval. When a failure occurs, the application may be restarted from the most recent checkpoint. The job replication approach [88] allows multiple copies of a job to be scheduled on different sets of processors either simultaneously (i.e., active replication (AR)) or one replica, called the primary replica, runs while other replicas, called backups, are in a standby mode (i.e., passive replication (PR)). A scheme that requires embedding of failure detection and recovery mechanisms into the application codes is discussed in [87]. However, as noted in [95], these fault-tolerant approaches typically ignore the issue of processor allocation. This can lead to a significant degradation in response time of the

applications [95] and to counter this effect an efficient job scheduling policy is required.

Scheduling: Research in scheduling has focused on efficiency and many different scheduling approaches for cluster computing environments have been proposed in the literature [106, 66, 59, 121]. However, reliability has rarely been factored into the design of most of these scheduling policies. Also, the primary goal of nearly all scheduling algorithms developed for cluster computing systems so far has been high performance by exploiting as much parallelism as possible while assuming that the resources are 100% reliable. As a result, there is very little research on fault-tolerant job scheduling as noted in [88].

Some studies have addressed scheduling and fault-tolerance jointly, but under some unrealistic assumptions such as requiring a particular programming paradigm [88], a membership services [73], and ample available processors for replica scheduling [135]. The problem with some of these approaches such as the state machine approach (which has embodied in toolkits like ISIS [27]) is that they make heavy demands, do not scale well and have proven hard to apply to networked environments [73]. Also, most of these approaches require a specialized and complex software layer that must be installed at each computation node (e.g., [73]). In addition, these systems apply to one cluster and one job situation. We have not found any literature for multi-programmed and multi-cluster environments. Also, all these systems use static scheduling policy whereas we focus here on the dynamic fault-tolerant scheduling approach.

5.1.3 Contributions

Designing and managing distributed systems with availability is generally difficult. Combining efficiency through job scheduling with fault tolerance further complicates the problem. This is because fault tolerance requires the use of redundancy in some

form while efficiency requires the elimination of redundancy. Also, these two areas have largely developed independent of one another each focusing on a different aspects of computing. Given that failures are inevitable, it is essential that attention be paid to how failures are handled and, in particular, how the system's users will be served during these times. To this end, we make the following contributions:

1. We present a strategy for survivability of the scheduling middleware in the face of node and link failures. The proposed approach is based on a general notion of reconfiguration at the system and application levels together with a framework that implements a monitor-analyze-respond approach to detection, identification and treatment of the faults. We show that the proposed scheme enhances the survivability of the scheduling middleware in the face of node and link failures.
2. In order to address the problem of combining efficiency and dependability through job scheduling with fault tolerance, we propose a fault-tolerant dynamic scheduling policy that loosely couples dynamic job scheduling with job replication fault-tolerant mechanism such that parallel jobs submitted to the system are efficiently and reliably executed. The novelty of the proposed algorithm is that it uses passive replication approach under high system load and active replication approach under low system loads. The switch between these two replication methods is also done adaptively.
3. Performance evaluation of the proposed fault-tolerant scheduler and a comparison with similar fault-tolerant scheduling policy is presented and shown that the proposed policy performs better than the existing approach. Also, the results of our experiments show that efficient fault tolerance can be achieved by replicating parallel applications on clusters of computers.

5.2 System Architecture and Failure Modes

In this section, we describe the system model and the basic components of the model as well as the failure modes assumed in this chapter.

5.2.1 Failure Modes

We define failure as anything that affects functional performance of the system components or user applications. We assume that the system components may fail and can be eventually recovered from failure. At any given time, the system components are said to be in one of faulty, alive or dead states. With respect to hardware and system programs, an alive component provides correct service (for user tasks it means that the task is running). In contrast, a dead component will cease functioning (running in the case of software components) completely and the fact that it has failed is eventually detected by all alive components. This is known as fail-stop failure mode [108]. For example, a dead link can stop transmitting messages to and from the system nodes. When a given node in the system is dead, the services or application tasks running on that node are also said to be in a *dead* state. Finally, a faulty component deviates from the normal functioning. This may be due to software bugs or malicious attacks, which is commonly known as Byzantine faults [108].

In this chapter, we assume that both hardware and software components of the system failures obey the fail-stop [108] failure mode. Techniques for dealing with failure modes other than fail-stop are important but are beyond the scope of this thesis and will not be addressed here. We also assume, as in [75], that component failures are fully dynamic (i.e., faults can occur on-line at any point in time) and the total number of faulty processors in a given cluster may never exceed a known fraction. We also assume that node failures are independent from each other [136].

5.2.2 System Configuration

Figure 5.2 shows the scheduling middleware model used along three fault management components, namely the *Configuration Manager (CM)*, the *Health Monitor (HM)* and the *Response Manager (RM)*. These three fault management components collectively construct system configuration information by monitoring the schedulers and user processes and restarting them if need be or reconfigure the system to provide continuous availability of services. The following subsections describe each of these components in detail.

Figure 5.2: Reconfigurable system architecture

5.2.2.1 Health Monitor

The health monitor (HM) is responsible for monitoring the state of the system resources and application processes and in case they deviate from their normal state, it

immediately notifies the change to the appropriate agent for necessary actions to be taken. The HMs are arranged in a hierarchy, as shown in Figure 5.3. In the figure, processors are represented by circles and HMs are represented by rectangles. Each HM is responsible to monitor its immediate descendants (i.e., children) and send a summary of the state of the nodes under its control to its parent. For example, HM_{CS1} monitors the health of P_1 and P_2 and periodically forwards the result to HM_{CS5} , which in turn merges the results from HM_{CS1} and HM_{CS2} and forwards it to HM_{SS} .

Figure 5.3: Health monitor architecture

The HMs detect a failure based on missing heartbeats. Figure 5.4 illustrates how the failure detection works. Node P_k periodically sends to the health monitor (HM) a small message indicating that it is alive. When the HM fails to receive the message from the node and after it has waited for timeout period it declares the node as failed. Once the HM detects the failure, it then notifies the Configuration Manager to take the appropriate action, which is discussed in the next section. This approach substantially decreases the rate of false failures although it increases the detection time.

There is a large body of work on heartbeat-based mechanism for detecting failures of processes or processors (e.g., Globus toolkit [120]). As noted in [98], existing failure detection services scale badly in the number of members that are being monitored. The proposed failure detection service scales well. Also, it reduces the communication overhead as it requires the processes or processors to prove that they are alive to the HM instead of the usual two-way communication where the failure detection service sends a message to each process or processors and in turn each of these entities reply. In addition, the proposed service reduces the potential of false failures as it uses a two step process where the monitored entity is first suspected as failed and then a second phase to prove or disprove the suspicion is entered. Also, it provides timely detection as it uses only one way compared to similar services. For example in [120], every host in the system runs a single status detector process. The status detector receives heartbeats from the locally running processes in order to detect process failures and it exchanges the heartbeats with other detectors to detect processor failure. This can take a long time to detect failures. Last but not least, the proposed service can also be used to identify, in addition to processes and processors, network failures as will be discussed shortly.

Figure 5.4: Failure detection with health monitor.

5.2.2.2 Scheduling Middleware

The scheduling middleware performs several important functions such as scheduling tasks, controlling access to shared resources, facilitating job submission and monitoring. We use a hierarchical scheduling infrastructure in order to realize virtual parallel processing system across multiple clusters. The scheduling infrastructure is fully described in Chapter 3. Since the applications are dependent on scheduling middleware for all aspects of their execution, the survivability of this critical middleware in the face of failures is an important issue that deserves immediate attention. By survivability we mean the ability of the scheduling middleware to continuously provide scheduling services, though possibly alternate or degraded, in the presence of various types of failure and disruption.

In order to achieve high availability of the scheduling middleware, we must make no basic system services to depend on the availability of any single node. To address this problem for the hierarchical organization we are using, each parent scheduler is complemented with a configurable number of backup schedulers selected from its children schedulers. These backup schedulers are ranked as first backup, second back,..., $(k-1)$ backup. The ranking of the backup schedulers can be based on several factors such as the distance from the parent scheduler or the capacity of the child scheduler. The highest ranking backup scheduler is said to be a *hot backup scheduler (HBS)*. In other words, every state change in the primary scheduler is propagated to the HBS right away. The other backup schedulers are *cold backup schedulers (CBS)*, which means that there is no propagation of the primary scheduler state.

The HMs on the children monitor the state of the network between them and their parents as well as the proper functionality of their parents. As shown in Figure 5.5, the HM running on the parent scheduler (i.e., P_k), at regular intervals sends an *alive* message to at least two children schedulers (i.e., C_i and C_j) indicating that it is alive. In Figure 5.5, the dotted lines indicate a missing alive messages from P_k .

Assume that C_i does not receive the alive message from P_k at the scheduled time. In this case, C_i consults C_j asking it to confirm whether or not it has received the *alive* message from P_k in the current monitoring period. We have two cases to consider:

1. C_j received the alive message: In this case, C_i assumes that the *link* between it and P_k may have failed or very slow. Therefore, C_i waits for another round of monitoring period for the *alive* message from P_k to arrive. If the *alive* message arrives before the waiting time expires, then the link and the P_k are presumed alive. Otherwise, C_i notifies the *RM* the fact that it has not received communication from P_k while C_j did.
2. C_j has not received the alive message: In this case, C_i and C_j send to P_k , the "Are you alive?" messages. If no response is received by both children (i.e., C_i and C_j) from P_k within a configurable time period, then P_k is declared dead. If one of the two children, say C_j , receives the response from P_k , then the link between C_i and P_k is declared dead.

Figure 5.5: Failure detection architecture. The dotted lines indicate the missing *alive* messages.

5.2.2.3 Response Manager

The Response Manager (RM) is responsible for determining the action to be taken in response to failure as well as recovery events. The RM is reactive in the sense that it takes corrective action on failure or obvious threat of failure. For example, it decides when and how the system is reconfigured or the replicas are instantiated. It is invoked by the notification messages from the health monitor, where the failure of a process or a node or a link is delivered. It is also activated by repair of a failed node or link.

For example, consider the state shown in Figure 5.5. Assume that the RM receives a message from C_i indicating that its not able to communicate with P_k while C_j could. In this case a link between C_i and P_k is declared *dead*. However, if both C_i and C_j report that they could not communicate with P_k , RM will decide that P_k is failed. The RM notifies the *Resource Fault Management (RFM)* component to reconfigure the system such that it will deliver a continuous scheduling service to applications.

Note that it is possible neither the link nor the node hosting the scheduler is down. Instead, HM or the scheduler or both can be down. We handle this situation as follows: When the *Are you alive?* message arrives at the parent node, it is examined by both the HM and the scheduler. The later simply checks to see if the HM on the node has failed. If so, it sends the response to the children and restarts the failed HM locally. Otherwise, it simply ignores the message. Similarly, if HM discovers that the scheduler is dead, the local HM attempts to restart it on the same host as part of local recovery. If this fails, the system switches from primary target to backup. The case where both the HM and the scheduler fail is considered as a node failure and both the scheduler and the HM have to be restarted manually. How this is automated will be considered in the future.

Upon the arrival of the notify message, the RM determines what actions to be taken by specifically authorizing the Configuration Manager component, which is

discussed next.

5.2.2.4 Configuration Manager

The Configuration Manager (CM) is activated by the RM and is composed of two components: Application Fault Management (AFM) and Resource Fault Management (RFM). The AFM handles the fault-tolerance issues related to user applications while the RFM deals with issues related to making the scheduling middleware resilient to both node and link failures. These two components are activated to remedy the type of the failure/recovery reported by HM and analyzed by RM. Each of these components are explained later in detail.

In the next two sections, we use the fault management components described in this section to enhance scheduling middleware survivability as well as to loosely couple job replication-based fault-tolerance mechanism with dynamic scheduling policy.

5.3 Dynamic Fault-Tolerant Scheduling Policy

When all computers are up and running, we would like the load to be evenly distributed among the computers. When one or more computers break down the load on these computers must be redistributed to other computers in the cluster. To achieve this goal, we propose a dynamic fault-tolerant scheduling (DFTS) policy. The DFTS policy uses the adaptive hierarchical scheduling (AHS) policy described in Chapter 3. Here, we modify AHS to provide fault-tolerance to applications by automatically replicating jobs and tasks over several clusters and processors, keep track of the number of replicas, instantiate them on-demand and delete the replicas when the primary copies of the jobs and tasks successfully complete execution.

Figure 5.6 shows the DFTS policy. In the algorithm, we assume that users indicate whether or not they want fault-tolerance at the time of job submission. The

motivation for asking users to indicate if fault-tolerance is required for the application is that fault-tolerance entails overhead both in terms of storage and processing, which could result in large delays for applications that require fault-tolerance. Some users may not be interested in paying for the fault-tolerance cost while others do. Many systems such as Condor-G[67] provide library routines which allow users interested in fault-tolerance to instrument their applications. The system would automatically provides fault-tolerance to such applications while those that are not linked to the system library are not considered for fault-tolerance.

Note that the parameter N_{jobs} in the algorithm denotes the number of jobs transferred from the parent to a child node. The FINISHED () and FAILED () functions are boolean functions that return true or false depending on the state of jobs, tasks, links and nodes, respectively. The policy maintains some state information for failure and recovery detections in Application Status Table (AST).

We now describe the main components of the DFTS policy in detail.

5.3.1 Application Scheduling and Fail Management

Replica Creation and Placement (RCP): The user can specify the number of job and task replicas at the time of job submission. Jobs are replicated over clusters while tasks are replicated over processors as discussed below:

1. **Job Replication:** When a job with fault-tolerance requirement arrives into the system, DFTS undertakes the following steps: (1) create a replica of the job; (2) keep the replica and send the original job to a child that is alive and reachable; and (3) update the application status table (AST) to reflect where the job replicas are located. This process recursively follows down the cluster tree until we reach the lowest level cluster scheduler (LCS) at which point the replica placement process terminates.

DFTS algorithm

1. When a new job that requires fault-tolerance arrives:
 - (a) WHILE child is not at the leaf-level DO
 - i. Create a replica of the job and keep it.
 - ii. Send the original to a child that is alive and reachable.
 - iii. Update the application status table and backup scheduler.ENDWHILE
 2. When a request for computation arrives at a scheduler:
 - (a) Get a set of jobs and send it to the requesting node as follows:
WHILE (($N_{jobs} > 0$) AND (!EMPTY(JWQ))) DO
 - i. Get a job that has its replica in the partition-reach of child
 - ii. If there is no such job, then select a job within the closest node
 - iii. Set $N_{jobs} = N_{jobs} - 1$ENDWHILE
 - (b) Update the backup scheduler and the application status table.
 - (c) Monitor the execution of the job/task
WHILE (!FINISHED(job) AND (!FAIL(node))) DO
 - i. Every REPORT-INTERVAL, leaf nodes send report to their parent
 - ii. Every SUMMARY-INTERVAL non-leaf nodes send summary to parentENDWHILE
 - (d) If completed, inform all remote CMs to terminate their replica
 3. When a failure is detected, recovery is initiated as follows:
 - (a) IF (FAIL(non-leaf node) OR FAIL(link)) THEN
Invoke Switchover strategy
 - (b) ELSE
Move failed tasks to the head of the task queue
ENDIF
 4. When a node or link is repaired, invoke Switchback strategy
-

Figure 5.6: Dynamic fault-tolerant scheduling policy

Figure 5.7: Application replica placement algorithm

Figure 5.8: Application replica placement algorithm

Figure 5.7 illustrates the job replica placement algorithm. Here, $J3$ arrives at the system scheduler, which creates a copy of $J3$ (i.e., $J3R1$), keeps $J3R1$ and sends $J3$ to $CS10$. When $CS10$ receives $J3$, it creates a new replica of $J3$ (i.e., $J3R2$), keeps $J3R2$ and forwards $J3$ to $CS1$. The replica placement stops here

as there are no more cluster scheduler exists below CS1 (i.e., CS1 is the lowest level cluster scheduler).

2. **Task Replication:** Figure 5.8 shows how tasks are replicated. Here, we assume that the user has specified 2 as the degree of task replication. Hence, J1 at CS1 is partitioned into 4 constituent tasks (i.e., $T_1, \dots, T_4$). For each task, T_i , the replica manager creates 2 replica of T_i tasks, (i.e., $tir1$ and $tir2$), and then distribute these replicas over P_1 and P_3 , respectively.

This process ensures that a job and a task are stored in a number of locations in the cluster tree. Note that the decision as to which child should receive a replica of the job and task can be done in a round robin fashion or based on the load factor of the child.

Job Transfer: Step 2 of the DFTS is concerned with how many jobs and which jobs are transferred from a parent to a child scheduler. The exact number of jobs dispatched from a parent to a child is discussed in Chapter 3 and will not be discussed here. With respect to which jobs are dispatched, DFTS favors jobs that have their replicas within the partition reach of the requesting schedulers. If there are no such jobs, then jobs belonging to the left sibling of the requesting node is searched. If this fails the jobs of the right sibling of the requesting node are selected. This process continues until the exact number of jobs to be sent to the requesting node is reached. The motivation for this job selection scheme is that we minimize replica management overheads (e.g., the replica instantiation latency) in case the original job fails. We also reduce the job transfer latency as we have to only send control messages to the child scheduler if the replica is already located there. Finally, it reduces the time that a child scheduler waits for the jobs to arrive, which increases system utilization. After dispatching the jobs to a child, the parent informs the backup scheduler about the assignment and then updates the application status table (AST) to reflect the

new assignment.

Replica Monitoring: The DFTS monitors applications at job-level (between non-leaf nodes and their parents) and at task-level (between leaf nodes and their parents) as shown in Figure 5.9. A monitoring message exchanged between a parent and a leaf-level node is called a *report* while that between non-leaf nodes is called a *summary*. A report message contains status information of a particular task running on a particular node and sent every *REPORT-INTERVAL* time units. In contrast, the *summary* message contains a collection of many reports and sent every *SUMMARY-INTERVAL* time periods such that $REPORT-INTERVAL < SUMMARY-INTERVAL$.

Figure 5.9: Monitoring Jobs and Tasks.

When a processor completes execution of a task, the report message contains a *FINISH* message. In this case, the receiving scheduler deletes the corresponding replica and informs the backup scheduler to do the same. When the last replica of a given job is deleted, the job is declared as successfully completed. In this case, the cluster scheduler immediately sends a summary message that contains the *COMPLETED* message to the parent scheduler, which deletes the copy of the job and

forward the same message to its parent. This process continues recursively until all replicas of the job are deleted.

Failure Detection and Recovery: After each assignment, the children periodically inform their parents the health of the computations as discussed above. If the parent does not receive any such message from a particular child in a given amount of time, then the parent suspects that the child has failed. In this case, it notes this fact in the AST and sends a request for report message to the child. If a reply from the child has not been received within a specific time frame, the child is declared dead.

Figure 5.10: Illustration of failure detection and recovery.

An example illustrating the failure detection mechanism for a task is shown in Figure 5.10. The cross on P_1 denotes that the node has failed. The first and second rows of the application status table indicates task T_i assignment to processors (P_i). If a task has not been scheduled, then the column corresponding to this task is empty (see for example T4). The 0 in the third row of the AST denotes that the report message from P_1 has not been received in the current monitoring period. The dashed

line between the Health Monitor (HM) and P_1 indicates that a request for the report (i.e., RFTS) has been dispatched to P_1 . On the other hand, the reports for T3 and T2 have been received (marked 1). In the next round of the task status check time (i.e., *REPORT-INTERVAL*), the *HM* discovers that P_1 has not responded to its request for task status (RFTS) message and declares P_1 as failed.

When a failure is detected, a recovery procedure is initiated to handle the failure. The recovery mechanism restarts a replica of the failed primary task as soon as possible. For example, suppose we have the failure case shown in Figure 5.10. When P_2 becomes available, instead of T4 being assigned to it, the scheduler assigns a backup copy of T_1 to P_1 . This completes the detection and recovery for T_1 .

5.3.2 Switchover and Switchback Strategies

A fail-over strategy is used when a link or a non-leaf node failure is detected. We refer to this process as *switchover*. When a node or link is repaired, we restore the system state as before. We refer to this process as *switchback*. In this section, we describe these two strategies. In order to aid us explain these algorithms, we use Figure 5.11 through Figure 5.12, which show different network and node failure scenarios in the cluster tree. In the figure, a cross indicates a failed link while a *dashed line* indicates the reconfigured route in response to the link failure between a parent and one of its child nodes in the cluster tree.

Link Failure: Link failure is addressed by rerouting traffic from the failed part of the network to another portion of the network. For non-root nodes, the child scheduler is informed to communicate through its closest sibling of the parent. If the failed node is the root, then we choose the closest functional and reachable sibling.

Figure 5.11 shows a link failure between two cluster schedulers (Figure 5.11a) and between a system scheduler and a cluster scheduler (Figure 5.11b), respectively. In

Figure 5.11: Link failures: (a) Between two cluster schedulers; (b) Between a cluster scheduler and the system scheduler

both cases, the RM bypasses the faulty link by making the child scheduler to reroute its requests to its parent through an intermediary node.

Node Failure: Figure 5.12 and Figure 5.13 show various possible cases of scheduler failures.

1. **System scheduler:** Figure 5.12 shows the case where the *grid scheduler* is failed. In this case, the RM will designate the highest ranking member of the *backup scheduler* as the *grid scheduler*. This is done by assigning the UID of the failed scheduler. The RM will then instruct the new primary scheduler to start the recovery procedure. Finally, the RM will promote the highest ranking child scheduler to be the hot backup scheduler. Note that recovery process and the promotion process are done in parallel. Finally, the newly promoted schedulers start providing the necessary services to the users.
2. **Backup scheduler:** Figure 5.13a shows the cases where a backup scheduler is failed. In this case, the failed backup scheduler is simply replaced by the

next highest ranking backup scheduler. The primary scheduler will then send information on active assignments (i.e., which processors are running which tasks) as well as replicas of tasks that are waiting to be assigned to the new backup node.

3. **Cluster scheduler:** Figure 5.13b shows the cases where the cluster scheduler failed. In this case, the hot backup scheduler (HBS) is designated as the primary scheduler and the highest ranking cold backup scheduler (CBS) is promoted to become the *HBS*. The new primary scheduler then receives update data from its parent and also store the same at the newly promoted *HBS*. Finally, the newly promoted schedulers start providing the necessary services to the users. Thus, a faulty system or cluster scheduler is replaced promptly so as to avoid denial of services to users. From the user's perspective, the system never goes down and all that the user perceives is a slight slowdown in responsiveness.

Figure 5.12: Repair: (a) System scheduler failure scenarios; (b) Repaired system scheduler

Link Repaired: When the failed link is repaired, traffic is rerouted over the primary

Figure 5.13: Node Failures: (a) Backup scheduler fails; (b) Primary scheduler fails.

route. When *RFM* receives the link repaired message, it notifies the appropriate nodes about the restoration of the link. Then the RFM performs the switchback, which restores the original link and traffic starts flowing over the primary path.

Node Repaired: When a node is repaired, it sends request to join message to the configuration manager. If the node used to be the primary scheduler, then the node that replaced it is told to send to the recovered node and all former children are also notified to update their parent id. The node then rejoins the system and provides services immediately after recovery. If it was a backup node, recovery is not necessary.

5.4 Baseline Scheduling Policies

The common approach to study the performance of a fault-tolerant scheduling policy is to compare it with another fault-tolerant policy (if one is available) or to compare it with a non-fault tolerant policy. In this chapter, we compare the DFTS policy with a modified version of the fault-tolerant scheduling algorithm (FTSA) originally

proposed in [136, 135]. We also use the AHS policy described in Chapter 3 as the non-fault-tolerant baseline policy for comparing the performance of the two fault-tolerant scheduling (i.e., DFTS and FTSA) policies.

5.4.1 Fault-Tolerant Scheduling Algorithm

The fault-tolerant scheduling algorithm (FTSA) [136, 135] exploits the possibility that a wide-area collection of machines may be highly underutilized and uses an active job replication strategy in which each replica is independently scheduled in a different site to achieve fault tolerance. One of the problems with the original policy is that the local site manager (SM) is not resilient to failure. The application may simply fail if the local SM goes down. Also, the policy assumes that the set of replicas are static and unchanging. This means, when a replica fails it is not restarted nor is a new replica is scheduled elsewhere. In addition, a job starts running as soon as there are n functional sites willing to accept the replicas. This means, if there are less than n sites, the job will be queued even if there are sites with known good reliability. Finally, the original policy can lead to a race condition as each local SM tries to recruit remote SMs for job replica placement.

We modified the original policy to address these shortcomings. To address the problem of halting jobs when home SM (HSM) fails, we employ peer-to-peer fail-over strategy such that every SM is a backup of another SM in the system. The HSM and the backup SM (BSM) also communicate periodically such that the BSM assumes the responsibility of the HSM in the event that the latter fails. When a HSM recruits n SMs on which the job executes, it also sends the identity of the BSM to these n sites managers. Also, a job and its replica are submitted to the HSM and the BSM, respectively.

In order to solve the race problem, we introduced a timer-based reservation scheme such that once a remote SM offers its resources to execute a job, it will not accept

FTSA algorithm

1. FOR each new job that requires fault-tolerance DO
 - (a) Poll all sites for availability information
 - i. IF ($R \geq n$) THEN
 - A. Choose the best n sites
 - B. Designate one of them as a backup home SM and notify the n sites the backup
 - ii. ELSE
 - A. Reserve $n-R$ site that are expected to finish soon.
 - (b) Send a replica of the job to each site and update job table and backup scheduler.
 - ENDFOR
 2. Monitor the job execution
FOR replica-interval DO
 - (a) Prompt all remote SMs for job status
 - (b) Determine the number of healthy replicas (i.e., h)
 - (c) IF any replica is done THEN
 - i. Tell all remote SMs to terminate their replica
 - ii. Update job table
 - (d) ELSEIF ($h > n$) THEN
 - i. Select last replica from set to terminate
 - ii. Update job table
 - (e) ELSE
 - i. Pick next site from set to terminate
 - ii. Inform the remote SM to execute the job
 - iii. Update job table
 - ENDIF
 - ENDFOR
-

Figure 5.14: Fault-tolerant scheduling algorithm

any request from another SM until such time that it is released by the HSM that has reserved it or the time limit expires or the job assigned to it completes.

In the case of a link failure between HSM and remote SMs, we let the BSM to assume the monitoring of job progress and inform the HSM. In case this is not possible, we let one of the remote SMs monitor the execution of the job.

Note that in the original policy, if the local SM cannot find n candidate sites for scheduling a job, the job is queued until such time that this condition is met. We remove this constraint such that if there is at least one site ready to run a job replica, we schedule the job.

Figure 5.14 shows the FTSA algorithm implemented in this chapter. FTSA is triggered by the arrival of a job for scheduling to the home site SM. In the algorithm, the parameter R denotes the number of sites that have responded to the poll message of the home SM, *replica-interval* is the time-interval between checking the health of the replicas and h is the number of healthy replicas of a running job at any given point in time. At the time of job submission, the user also specifies the number of desired replicas, n , and the replica threshold, k , where $n \geq k$. In addition, the user provides sufficient information about the job, which the scheduler uses to estimate the run-time of the job using different candidate schedules. There are two main components of the FTSA policy which are briefly discussed next.

Job and Replica Placement: When a job arrives, FTSA chooses a set of n candidate sites for job execution (including the home site) and orders them for an estimate of the job completion time. In the current implementation, we simply choose those sites that can complete the jobs fast and that have not been already reserved. When these sites become available, they contact the SM that has reserved them.

Once the candidate sites are selected, the home SM then sends a replica of the job to each site and marks each replica with a time-stamp to indicate the replica start

time. Finally, it updates the job table and backup scheduler with this information. In case that all replicas of the job have not been scheduled, the home SM monitors the availability of those sites it reserved by listening to them. As soon as one becomes available, SM schedules a replica to the site. If a job successfully completes, the home SM sends release message to each site it had reserved such that these sites can be used for running other jobs.

Replica Management: The local SM periodically queries all remote SMs to obtain machine and network status to monitor the health of any job replicas running within the site. If a failure of a replicas is detected, a replacement site is sought immediately. Specifically, every *replica-interval*, the local SM determines the number of healthy replicas, h , and compares it with the replica threshold, k . If $h > k$, then the last replica to be scheduled is terminated. However, if $h < k$, a replacement site is sought by polling all sites on which the replica of the job are not running. If a site is found, a replica is forwarded to the site and both the backup SM and the job table are updated. However, if no site is available, one of the sites on which the replica of the job is not running is reserved. When this site becomes available, it contacts the SM that has reserved it. If $h < k$, a replica of the job is then sent to this SM. When a replica of the job successfully completes, the algorithm informs all sites on which a replica is running to terminate. Also, any site that has been reserved is notified to cancel the reservation.

5.5 Performance Analysis

We used discrete event simulation written in C programming language to study the performance of the integrated scheduling and fault-tolerance approach. The system, network and workload models are the same as those used in Chapter 3. Table 5.1

shows the parameters and associated default values used in the experiments. Note that the number of job and task replicas required depends on the number and severity of faults to be tolerated as well as on the available resources. We configure the system with two replicas initially, but this default value may change adaptively depending on the reliability of the cluster-computing environment. This will be a subject to be addressed in the future.

Parameters	Values
Multiprogramming level	2
Service demand coefficient of variation	3.5
Parallel job arrival coefficient of variation	1.0
Parallel job mean service demand	14.06
Background job mean service demand	2.0
Number of clusters in the system	8
Number of processors in each cluster	8
Number of task replicas	2
Report and replica intervals	2 seconds
Summary interval	10 seconds

Table 5.1: Default parameters and values used in the experiments.

In the experiments, the three policies are subjected to the same system and workload traces except that in FTSA and DFTS cases the processor trace contains both processor failure and repair time events. We set processor and link time-to-failure to 0.00321 hours and 0.005 hours, respectively. The time-to-repair the link is set to 30 seconds while that of the processors is set to 0.00321 hours. The link failure and recovery figures are based on the work of Cui et. al. [40] while the processor failure and recovery figures are based on the data collected on the experimental assessment of workstation failures in [95]. As in [135, 96, 95], we assume that inter-occurrence times of failures for each processor are independent and identically distributed as exponential random variables with the same failure rate. Also, we assume that the times to failure of workstations and their repair times are mutually independent random variables [95].

Performance Metrics: We use the mean response time (MRT) and average utilization as chief performance metric as in [124, 99]. The MRT is defined as the sum of all job response time divided by the number of completed jobs. The average utilization is defined as the job arrival rate times the mean service demand of the jobs divided by the number of processors in the system.

Validation: A batch strategy is used to compute confidence level (at least 30 batch runs were used for the results reported in this chapter). This strategy produced 95% confidence levels that were less than 1% of the mean response times when the system utilization is low to moderate and less than 5% for moderate to high system utilization (in most cases, up to 90%).

5.6 Results and Discussions

We studied algorithm reliability by testing various failure scenarios. In the following subsections, we present and discuss the results of the experiments.

5.6.1 Deterministic Node Failure Environment

The objective of this experiment is to model transient failures such as workstation rebooting by users. In a deterministic node failure environments, selected nodes in a given cluster are made to fail at specific times. The nodes remain in failed state for a period of time and then become operational (i.e., repaired). We assume that the probability of multiple node failures in the same cluster at a given time interval is much smaller than the chance of a single node failure. This assumption is reasonable as Gray [61] reported less than 1% of the node failures were double node failures. In Section 5.6.3, we examine the performance of the policies when this assumption is

relaxed.

Figure 5.15: Performance of the policies in dedicated heterogeneous environments

Figure 5.16: Performance of the policies in shared homogeneous environments

Figure 5.15, Figure 5.16 and Figure 5.17 show the mean response time of the jobs for the three scheduling policies for dedicated heterogeneous, shared homogeneous and shared heterogeneous environments. In the experiments, we assumed that the probability of network failure is zero and the case in which network failure is an issue

Figure 5.17: Performance of the policies in shared heterogeneous environments

will be addressed later. Also, every 3000 seconds, a non-idle node is randomly selected in each cluster and made to fail for 6 seconds.

From the data on these graphs, we observe that at low system load, FTSA and DFTS are marginally better than AHS policy. This is because of the fact that at this load level, there are many idle processors. This means both FTSA and DFTS can schedule a job on several clusters and at least one of the replicas could finish without being interrupted by a node failure.

However, as the load increases, performance of FTSA deteriorates in all three environments while DFTS performs marginally worse (about 4%) than the AHS policy in most instances. This can be explained by the fact that as load increases the number of idle processors decreases. As a result, finding n idle clusters for FTSA to schedule jobs becomes harder. In contrast, DFTS uses demand-driven approach as in AHS and multiple replicas of a job are only scheduled when there are ample free processors. For the workload type we studied, it seems that the DFTS approach is better than the active replication approach used in FTSA.

5.6.2 Non-Deterministic Node Failure Environment

In this section, we study the performance of DFTS and FTSA policies when the nodes in the system are subjected to random failures and repairs. Both failures and repairs are modeled as an exponential distribution as in [135, 96, 95].

Figure 5.18: Performance of the policies in non-deterministic dedicated heterogeneous environments.

Figure 5.18 shows the mean response time of the jobs for dedicated heterogeneous environments as the trends for the shared heterogeneous and shared homogeneous environments are similar to that shown in Figure 5.16 and Figure 5.17. In the experiments, we assumed that the probability of network failure is zero. Furthermore, the number of nodes that can fail in a cluster at any given time is set to one. Idle and allocated processors are equally likely to fail. The data in the figure shows that DFTS is quite robust to random node failures. This is because DFTS performs marginally worse than AHS while FTSA performs poorly compared to the other policies.

When parallel processing is combined with fault tolerance, overheads are compounded, which must be paid even in the absence of failures. The DFTS overhead consists primarily of job monitoring which will have only a minimal impact on the

finishing time of the application. In contrast, FTSA relies on the expectation that, of the initial set of replicas, the probability is very high that one will complete. If this is not the case, then performance may suffer, particularly for long-running jobs. For example, if all replicas were to fail around the same time and after running for a long time, then a new replica would not be scheduled until after a long time has passed. The end-user would observe a large slowdown in the completion of the application.

5.6.3 Multiple Node Failure in a Cluster

Figure 5.19 shows the case where we relaxed the assumption that no multiple failures occur in the same cluster. Also, both idle and allocated processors are equally likely to fail. In the figure, FTS-2 and FTS-3 denote the performances of the policies when the number of processors that can fail at the same time in a cluster is 2 and 3, respectively. In the experiments, we assumed that the probability of network failure is zero and both failures and repairs of the nodes are modeled as an exponential distribution [135, 96, 95].

Figure 5.19: Impact of multiple processor failures in dedicated heterogeneous environments.

By comparing the data on Figure 5.19 to Figure 5.18, we observe that as the number of nodes that fail in a cluster increases so does the mean response time of the two policies. This is because jobs that experience one or more failures require re-scheduling. However, the results from Figure 5.19 show that DFTS is robust to multiple processor failures as compared to the FTSA. The performance degradation under FTSA is the consequence of many jobs requiring simultaneous migration. Also, as the load increases, FTSA spends a lot of time fruitlessly polling remote SMs to place jobs. This is because at high loads, there is hardly any idle clusters. Also, FTSA suffers from inter-cluster communications. In contrast, DFTS is based on multilevel hierarchy of the cluster-based system model that allows intra-cluster communication to be carried out independent of other clusters and hence provides parallelism in communication of cluster nodes. In addition, DFTS combines the status of multiple jobs into a single message (i.e., summary) as such reducing the number of messages transferred between two clusters. In contrast, every job in FTSA policy sends a message to its home cluster every replica-interval time.

5.6.4 Sensitivity to Mean Times Between Failures

The mean times between failures (MTBF) measures the average amount of time between failures, where failure is defined as a departure from acceptable service of an application. Figure 5.20 shows the performance of the policies as a function of the mean failure time. In the experiment, all parameters are set to their default values while the mean time between failures is varied.

The results show that even in the case of frequent failures, DFTS overhead is low enough to provide a good response time for the applications. Although FTSA detects node failures faster than DFTS, it experiences greater false failure rates, which lead to multiple re-runs of the same job. In addition, when the number of active replicas is found to be more than the replica threshold (i.e., $n \geq k$), one of the replicas

Figure 5.20: Performance of the policies as a function of mean time between failures/hour.

is terminated. The problem is that in some cases the best performing replica is terminated leading to job completion delay.

Also, DFTS policy uses a job selection scheme that minimizes replica management overheads (e.g., the replica instantiation latency) in case the original job fails. The scheme also reduces the job transfer latency as it has to only send control messages to the child scheduler if the replica is already located there. Finally, it reduces the time that a child scheduler waits for the jobs to arrive, which increases system utilization. In contrast, the job placement in FTSA is based on polling each remote SM and reservation. This coupled with the time taken for each local SM to make placement decision as to which clusters to use, the system resources remain idle, which reduces the utilization substantially.

5.6.5 Sensitivity to Link Failures

In this section, we study the impact of link failures on the performance of the two fault-tolerant policies. Each link in a given network continuously cycles between two

states (i.e., alive and dead) independently of all other links. Once alive, the time a link remains alive was determined randomly based on the exponential distribution. The mean of the distribution (i.e., mean time to failure, $\frac{1}{\lambda}$) was an input parameter of the simulation. Once dead, the time a link remained dead was also determined randomly using the exponential distribution. The link failures and repair were modeled using the exponential distribution with a failure rate of 0.005 and recover after 30 sec [40]. We assumed that nodes do not fail.

Figure 5.21: Sensitivity to link failures.

Figure 5.21 shows the mean response time of the jobs as the number of links fail in the system increases from 0 to 3 (i.e., DFTS-0, DFTS-1, DFTS-2 and DFTS-3) in a shared homogeneous settings. Note that when DFTS-0 and FTSA-0, the system do not experience neither link nor node failures. In this settings, the data shows that the mean response time under DFTS-0 is slightly higher (less than 1%) than the mean response time of AHS as shown in Figure 3.12. The reason for the increase is due to fault-tolerance related operations such as job monitoring. Nevertheless, the result shows that DFTS is robust under non-faulty environment as its overhead is not significant.

For the cases when the link failure > 0 , the data in Figure 5.21 shows that DFTS performs substantially better than FTSA. This is because FTSA uses distributed approach and the communication overhead is more than the DFTS. For example, sending work reports less frequently decreases communication time, but may increase detection time.

The data also show as the quality of the network decreases the mean response time of both policies increases. This can be attributed to the fact that the number of redundantly executed jobs increase when communication conditions are poor. Under DFTS, the failure recovery mechanism is activated (decides that a job is dead and recreates it) less often and the overhead introduced is lower than FTSA.

For example, in FTSA the local SM and backup SM (BSM) can be partitioned when the link between them fails or very slow. In this case, the local SM tries to replace a BSM by creating a replica of the job on an idle cluster and then nominate one of the replicas as the BSM. On the other hand, BSM elevates itself to be a local SM for the jobs. It then creates another replica to replace the local SM and designate one of the replicas as a new BSM. This means up to n replicas of a job can be executing under FTSA, where n is the number of clusters in the system. In contrast, this number can never exceed 2 under the DFTS. However, because the communication is poor, the cluster managers suspect failure and send more work reports under DFTS, which explains the performance degradation of DFTS.

5.7 Summary

In this chapter, we designed a survivable job scheduling middleware that is able to provide useful services despite faults. We proposed a scalable resource state monitoring algorithm that reduces communication requirements for distributed monitoring as information about multiple resources can be sent in the same message. We also pre-

sented a scalable framework that loosely couples the dynamic job scheduling approach with the hybrid (i.e., passive and active replications) approach to schedule jobs efficiently while at the same time providing fault-tolerance. The main advantage of the proposed approach is that fail-soft behaviour (i.e., graceful degradation) is achieved in a user-transparent manner. Furthermore, being a dynamic algorithm estimations of execution or communication times are not required. An important characteristic of our algorithm is that it makes use of some local knowledge like faulty/intact or busy/idle states of nodes and about the execution location of jobs. Another important characteristic of the proposed approach is that they are applicable for a wide variety of target machines including Grid computing. The experimental results show that the proposed policy performed at low cost in order to support fault-tolerance. Thus the results we obtained encourage us to continue our research in this direction.

Chapter 6

Conclusions and Future Directions

In the thesis, we addressed several issues relevant to parallel job scheduling on cluster computing systems. This chapter summarizes the results of the thesis and state future directions.

6.1 Processor Scheduling

Efficient job-scheduling strategies are important to improve performance and usability of high performance clusters. The question is: How can we design an on-line parallel job scheduling system for heterogeneous networks of workstations, where computing power varies among the workstations, and local and parallel jobs may interact with each other in execution? To answer this question, we proposed and experimentally evaluated a dynamic scheduling policy that manages multiple job streams across both single and multiple-cluster computing systems with the objectives of optimizing the mean response time and system utilization. The proposed policy is based on a virtually rooted tree structure that employs a pull-push scheme for scheduling and load balancing parallel jobs over multiple clusters. The proposed policy allows multiple job streams to share the system concurrently in an environment where the actual load distribution is not completely predictable in advance and the scheduling decisions are

made on-line. The proposed policy has the following features:

1. It combines the merits of centralized and distributed coordination mechanisms into a single framework suitable for shared cluster computing environments.
2. It integrates space-sharing and time-sharing policies into a single framework such that the virtues of the two policies are realized while their shortcomings are minimized or/avoided.
3. It integrates dynamic load distribution and re-distribution that smooths out imbalance that can results due to system and workload changes.

We studied the performance of the proposed scheduling policy through simulation. We found that applications can experience large slowdown in time-shared systems or high wait times in space-shared systems, due to contention for these limited valuable resources. Policies that combine space-sharing and time-sharing perform much better than the time-sharing alone or space-sharing alone. We also found that job scheduling policies that do not reconsider prior scheduling decisions do worse than those that do re-distribute load. We observed that both heterogeneity and resource contention can cause performance of an application vary dynamically over time. Consequently, the overall performance achievable by the application at the start of execution may also vary as the application executes. Therefore, it is not just enough to design a good scheduling policy to get the best performance possible.

6.2 CPU-I/O Intensive Parallel Job Scheduling

Advanced general-purpose parallel systems should be able to support diverse applications with different resource requirements without compromising effectiveness and efficiency. However, with the increasing processor performance, increased pressure has been placed on the performance of accessing external data in order to keep up with

the demand. The I/O subsystem and related software is unable to meet this demand and valuable CPU resources are left underutilized. This problem is especially acute for many scientific applications which use large data sets and require high performance. The question then is: How can we design coordinated allocation of processor and I/O resources in a system composed of multiple heterogeneous clusters?

To address this question, we proposed two new scheduling policies with considerations for dual (i.e., CPU and I/O) resource allocation to competing parallel jobs. The proposed scheduling policies try to co-schedule the data and the jobs; if this is not possible, they use dynamic prefetching of data to reduce the wait time of applications.

6.3 Fault-Tolerant Resource Allocation

We also addressed the question of: How to achieve execution and utilization efficiency (i.e., job scheduling) while ensuring reliable execution environments (i.e. fault-tolerance) in cluster computing systems. This is an important question that needs to be solved as the potential for faults in large distributed computing systems is a significant and complicating factor for users. Moreover, distributed systems are especially vulnerable to failures because of the inter-dependency of their processing nodes.

We proposed a framework that allows a dynamic job-scheduling policy and the fault-tolerance algorithm to work in unison. We also addressed the problem of making the scheduling middleware survivable in the face of disruptions. The proposed fault-tolerant distributed framework provides distributed system users and parallel programmers with an integrated processing environment, where they can reliably execute their concurrent (distributed) applications despite errors that might occur in the underlying hardware. The framework does not require code changes in the application or operating system. Also, it provides graceful performance degradation in the event of computing resource failures.

6.4 Extending Previous Approaches

We also extended previous scheduling policies such that they are ported to a heterogeneous and non-dedicated environments. We identified several issues that impact the performance of the conventional adaptive space-sharing scheduling policies in shared heterogeneous cluster computing environments. We proposed several solutions to these shortcomings and validated the viability of the proposed approaches through simulation analysis.

6.5 Future Directions

We are extending the coordinated CPU and I/O scheduling algorithm by using appropriate I/O scheduling strategies. To this end, we have proposed two new parallel I/O scheduling algorithms [3, 4] and currently studying the performances of these policies. We are also investigating dynamic data replication to reduce bandwidth consumption and access latency where users require remote access to files that are large: on average, around two Gigabytes.

In the proposed fault-tolerant distributed framework, the latency of detecting the errors might be affected by message traffic in the communication network. To address this problem, we intend to develop an on-line mechanism to dynamically measure the round-trip time of the underlying network and calculate the error latency accordingly.

We are also planning to study the impact of interconnection network on the performance of the proposed policy. In addition, we plan to investigate the viability of the proposed algorithms using workload traces as well as implement the policies on an actual system and evaluate their performance.

Last but not least, we plan to study the impact of the proposed policies on a larger system spanning larger geographic areas. Moreover, we plan to investigate how the hierarchical framework can be used for data distributions such that the overhead of

data movement can be reduced.

Appendix A

Glossary

Average response time is defined as the sum of all job response times divided by the number of completed jobs.

Branching factor is defined as the number of children of a node in the cluster tree.

Cluster schedulers coordinate scheduling of the components of a parallel program (i.e., jobs and tasks) at the cluster level.

Dedicated heterogeneous environment In this environment, there is no background workload. However, the processors in the system may differ with respect to their processing speed.

Degree of multiprogramming is defined as the number of applications that are running in parallel on a system at a given time.

Evolving jobs are jobs that can change their processor requirements during execution.

Grid computing is a type of parallel and distributed system that enables sharing, selection, and aggregation of geographically distributed autonomous resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements.

I/O factor is the ratio between the mean total I/O demand and the mean total CPU

demand for jobs.

Local schedulers coordinate the execution of parallel tasks and local tasks at a workstation.

High Throughput Computing (HTC) is defined as computing environments that can deliver large amounts of processing capacity over long periods of time.

High Performance Computing (HPC) is a computing platform that encompasses a collection of hardware systems, software tools, and programming approaches which collectively provide solutions to problems that require significant computational power.

Malleable jobs are jobs that can change their processor requirements during execution.

Maximum parallelism refers to the maximum number of tasks of a job that can be run in parallel, if an unlimited number of processors is provided.

Mean service demand is the service time needed to execute a job in a dedicated environment.

Moldable jobs are jobs that are flexible in the number of processors at the time the job starts, but cannot be reconfigured during execution.

Multiprogramming level (MPL) control the number of tasks among which a processor is time-shared at any given time.

Rigid job is a parallel job that requires a certain number of processors at arrival time and executes on these processors exclusively until completion.

Partition-reach is defined as all the processors reachable from a given node in the cluster tree.

Processor utilization is the percentage of time a processor spends executing useful tasks.

Response time is defined as the time between the job completion time and its arrival time.

Shared heterogeneous environment: In this setting, processors differ in speed. Also, each workstation may receive background workload at any time.

Shared homogeneous environment: In this setting, all processors run at the same speed. However, each workstation may receive background workload at any time.

Space-sharing is a class of scheduling policy that shares the processors in space by partitioning them among different parallel jobs.

System scheduler coordinates the assignment of parallel jobs to clusters.

System utilization measures the percentage of time a processor spends executing useful tasks.

Time-sharing is a class of scheduling policy that shares the processors by rotating them among a set of jobs in time.

Appendix B

Dynamic/Static Hybrid Policies

B.1 Dedicated Heterogeneous Environments

Figure B.1: Relative performance under dedicated heterogeneous environments

Figure B.2: Relative performance under dedicated heterogeneous environments

B.2 Shared Heterogeneous Environments

Figure B.3: Relative performance under the shared heterogeneous environment.

B.3 Shared Homogeneous Environments

Figure B.4: Relative performance under shared homogeneous environments .

Bibliography

- [1] J. H. Abawajy. An opportunistic scheduling policy for high throughput computing. In *Proceedings of IEEE International Conference on Parallel and Distributed Systems (ICPADS-2002)*, pages 605–610, 2002.
- [2] J. H. Abawajy. Timesharing opportunistic scheduling policy for high throughput computing. In *Proceedings of Workshop on Hardware/Software Support for Parallel and Distributed Scientific and Engineering Computing (SPDSEC-02)*, pages 91–110, 2002.
- [3] J. H. Abawajy. Parallel I/O scheduling in multiprogrammed cluster computing systems. *The International Conference on Computational Science 2003 (ICCS 2003)* , *Lecture Notes in Computer Science*, 57:223–229, 2003.
- [4] J. H. Abawajy. Performance analysis of parallel I/O scheduling approaches on cluster computing systems. In *Proceedings of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, pages 724–729, 2003.
- [5] J. H. Abawajy and S. Dandamudi. A reconfigurable multi-layered grid scheduling infrastructure. In *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, pages 138–144, 2003.

- [6] J. H. Abawajy and S. Dandamudi. Fault-tolerant multicluster resource management infrastructure. *Journal of Neural, Parallel and Scientific Computations*, 2004. To appear.
- [7] J. H. Abawajy and S. P. Dandamudi. An approach to parallel application scheduling on networks of workstations. In *Proceedings of the 1999 International Parallel and Distributed Techniques and Applications (PDTA'99)*, pages 2580–2586, 1999.
- [8] J. H. Abawajy and S. P. Dandamudi. Distributed hierarchical coordination of workstation clusters. In *Proceedings of the IEEE International Conference on Parallel Computing in Electrical Engineering (PARELEC'00)*, pages 111–115, 2000.
- [9] J. H. Abawajy and S. P. Dandamudi. Parallel job scheduling on workstation cluster environment. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2000) Conference*, pages 351–354, 2000.
- [10] J. H. Abawajy and S. P. Dandamudi. Time/space sharing distributed job scheduling policy in a workstation clusters environment. In *Proceedings of the IEEE International Conference on Parallel Computing in Electrical Engineering (PARELEC'00)*, pages 116–120, 2000.
- [11] J. H. Abawajy and S. P. Dandamudi. Parallel job scheduling on multi-cluster computing systems. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2003)*, Hong Kong, China, December 1-4 2003.
- [12] J. H. Abawajy and S. P. Dandamudi. Scheduling parallel jobs with CPU and I/O resource requirements in cluster computing systems. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation*

of *Computer Telecommunications Systems (MASCOTS 2003)*, pages 336–351, 2003.

- [13] J. H. Abawajy and S. P. Dandamudi. A unified resource scheduling approach on cluster computing systems. In *Proceedings of the ISCA 16th International Conference on Parallel and Distributed Computing Systems (PDCS-2003)*, 2003.
- [14] A. Acharya, G. Edjlali, and J. Saltz. The utility of exploiting idle workstations for parallel computation. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS) Conference*, pages 225–236, 1997.
- [15] V. K. P. Ammar H. Alhusaini and C. S. Raghavendra. A unified resource scheduling framework for heterogeneous computing environments. In *Heterogeneous Computing Workshop*, pages 156–165, 1999.
- [16] T. E. Anderson, D. E. Culler, D. A. Patterson, and N. Team. A Case for NOW (Network of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.
- [17] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'95) Conference*, pages 267–278, May 1995.
- [18] A. Arpaci-Dusseau and D. Culler. Extending proportional-share scheduling to a network of workstations. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, June 1997. Available at <http://now.cs.berkeley.edu/Implicit/>.

- [19] S. Ayachi and S. P. Dandamudi. Hierarchical processor scheduling policy for multiprocessor systems. In *Symposium of Parallel and Distributed Processing*, pages 100–109, 1996.
- [20] J. Baldeschwieler, R. Blumofe, and E. Brewer. Atlas: An infrastructure for global computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996. Available at <http://www.cs.utexas.edu/users/rdb/papers.html>.
- [21] A. Barak, O. La’adan, and A. Shiloh. Scalable Cluster Computing with MOSIX for LINUX. In *Proceedings of the 5-th Annual Linux Expo.*, pages 95–100, 1999.
- [22] A. Baratloo, P. Dasgupta, V. Karamcheti, and Z. M. Kedem. Metacomputing with MILAN. In *Heterogeneous Computing Workshop*, pages 169–183, 1999.
- [23] J. Basney and M. Livny. Managing network resources in condor. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing*, pages 2–9, 1987.
- [24] J. Basney and M. Livny. Deploying a high throughput computing cluster. In R. Buyya, editor, *High Performance Cluster Computing*, volume 1. Prentice Hall, Inc., 1999.
- [25] A. Batat and D. G. Feitelson. Gang scheduling with memory considerations. In *14th International Parallel Distributed Processing Symposium*, pages 109–114, May 2000.
- [26] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. Technical Report CS96-482, Champaign - Urbana, IL, 1996.

- [27] K. P. Birman. The process group approach to reliable distributed computing. Technical report, Department of Computer Science, Cornell University, July 1991.
- [28] G. Bosilca, A. Bouteiller, F. Cappello, S. Djailali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. Mpi-ch-v: Toward a scalable fault tolerant MPI for volatile nodes. In *Proceedings of the IEEE/ACM Supercomputing (SC2002) Conference*, pages 29–36, 2002.
- [29] T. B. Brecht. An experimental evaluation of processor pool-based scheduling for shared-memory NUMA multiprocessors. In *Job Scheduling Strategies for Parallel Processing*, pages 139–165. Springer Verlag, 1997.
- [30] A. I. D. Bucur and D. H. J. Epema. The influence of the structure and sizes of jobs on the performance of co-allocation. In *Job Scheduling Strategies for Parallel Processing (JSSPP'00)*, pages 154–173, 2000.
- [31] Y. Chan, S. P. Dandamudi, and S. Majumdar. Performance comparison of processor scheduling strategies in a distributed-memory multicomputer system. *Journal of Computers and Their Applications*, 9(4):227 – 240, 2002.
- [32] S. J. Chapin, D. Katramatos, J. Karpovich, and A. S. Grimshaw. The Legion resource management system. In *Job Scheduling Strategies for Parallel Processing*, pages 162–178. Springer Verlag, 1999.
- [33] S.-H. Chiang and M. K. Vernon. Dynamic vs. static quantum-based parallel processor allocation. In *Job Scheduling Strategies for Parallel Processing*, pages 200–223. Springer-Verlag, 1996.
- [34] G. Chiola and G. Ciaccio. Gamma: a low cost network of workstations based on active messages. In *5th Euromicro Workshop on Par-*

- allel and Distributed Processing (Euro-PDP '97)*, 1997. Available at <http://citeseer.nj.nec.com/chiola97gamma.html>.
- [35] Y. E. Cho, M. Winslett, S.-W. Kuo, J. Lee, and Y. Chen. Parallel I/O for scientific applications on heterogeneous clusters: a resource-utilization approach. In *International Conference on Supercomputing*, pages 253–259, 1999.
 - [36] L. Chung, J. Gray, B. Worthington, and R. Horst. Windows 2000 disk IO performance. Technical report, Microsoft Research-Redmond WA, April 2000.
 - [37] W. Cirne. Using moldability to improve the performance of supercomputer jobs. Technical report, Computer Science and Engineering, University of California San Diego, 2001.
 - [38] D. Collins, A. George, and R. Quander. Achieving scalable cluster system analysis and management with a gossip-based network service. In *Proceedings of Local Computer Networks (LCN)*, pages 14–16, 2001.
 - [39] A. R. A.-C. A. A. Crandall, P. E. and D. A. Reed. Characterization of a suite of Input/Output intensive applications. In *Proceedings of Supercomputing '95*, 1995. Available at <http://www.supercomp.org/sc97/proceedings/TECH/MADHYAST/INDEX.HTM>.
 - [40] W. Cui, I. Stoica, and R. H. Katz. Backup path allocation based on a correlated link failure probability model in overlay networks. In *Proceedings of 10th IEEE International Conference on Network Protocols (ICNP 2002)*, pages 236–245, November 12–15 2002.
 - [41] K. Czajkowski, I. Foster, N. Karonis, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Job Scheduling Strategies for Parallel Processing*, pages 62–82. Springer Verlag, 1998.

- [42] S. P. Dandamudi and P. S. P. Cheng. Hierarchical task queue organization for shared-memory multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(1):1–16, Jan 1995.
- [43] J. G. David Abramson and L. Kotler. High performance parametric modeling with Nimrod/G: Killer application for the global grid? In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2002 (IPDPS 2000)*, pages 520–528, 2000.
- [44] J. G. David Abramson, Rok Sasic and B. Hall. Nimrod: A tool for performing parameterised simulations using distributed workstations. In *Proceedings of IEEE High-Performance Distributed Computing (HPDC 1995)*, pages 112–121, 1995.
- [45] G. S. Dipak Ghosal and S. K. Tripathi. The processor working set and its use in scheduling multiprocessor systems. In *IEEE Transactions on Software Engineering*, volume 17 (5), pages 443–453. IEEE, 1991.
- [46] A. A.-D. R. A.-D. Douglas Thain, John Bent and M. Livny. Gathering at the well: Creating communities for Grid I/O. In *Proceedings of Supercomputing*, pages 58–68, Denver, Colorado, November 2001.
- [47] L. Dowdy, E. Rosti, G. Serazzi, and E. Smirni. Scheduling issues in high-performance computing. *ACM Performance Evaluation Review*, 26(4):60–69, March 1999.
- [48] C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'96) Conference*, pages 25–36, 1996.

- [49] D. G. E. Freeman and D. Kaminsky. Adaptive parallelism and piranha. *Computer Journal*, 28(1):40–49, 1995.
- [50] D. G. Feitelson. Packing schemes for gang scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 89–110. Springer-Verlag, 1996.
- [51] D. G. Feitelson. Memory usage in the LANL CM-5 workload. In *Job Scheduling Strategies for Parallel Processing*, pages 78–94. Springer Verlag, 1997.
- [52] D. G. Feitelson and B. Nitzberg. Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860. In *Job Scheduling Strategies for Parallel Processing*, pages 337–360. Springer-Verlag, 1995.
- [53] D. G. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.
- [54] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer Verlag, 1997.
- [55] D. G. Feitelson and A. M. Weil. Utilization and predictability in scheduling the ibm sp2 with backfilling. In *12th International Parallel Processing Symposium*, pages 542–546, Apr 1998.
- [56] I. Foster and J. M. David Kohr, Jr., Rakesh Krishnaiyer. Remote I/O: Fast access to distant storage. In *Proceedings of the 5th Workshop on Input/Output in Parallel and Distributed Systems*, pages 14–25. ACM Press, 1997.
- [57] K. C. J. D. I. F.-D. G. L. J. K. K. C. K. J. M.-C. D. R. L. T. Francine Berman, A.C. and R. 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.
- [58] J. Gehring and T. Preiss. Scheduling a metacomputer with uncooperative sub-schedulers. In *Job Scheduling Strategies for Parallel Processing*, pages 179–201. Springer Verlag, 1999.
- [59] G. Ghare and S. T. Leutenegger. Improving small job response time for opportunistic scheduling. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'00) Conference*, pages 557–564, 2000.
- [60] F. Giné, F. Solsona, P. Hernández, and E. Luque. Coscheduling under memory constraints in a NOW environment. In *Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science*, volume 2221, pages 41–65. 2001.
- [61] J. Gray. Why do computers stop and what can we do about it? In *Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems*, pages 141–154, 1986.
- [62] R. L. Henderson. Job scheduling under the portable batch system. In *Job Scheduling Strategies for Parallel Processing*, pages 279–294. Springer-Verlag, 1995.
- [63] F. B. Henri Casanova, Graziano Obertelli and R. Wolski. The apples parameter sweep template: User-level middleware for the Grid. In *Proceedings of Super Computing Conference*, pages 75–76, 2000.
- [64] I. Foster and C. Kesselman. Globus: A Toolkit-Based Grid Architecture. In *The Grid: Blueprint for a Future Computing Infrastructure*, pages 259–278. MORGAN-KAUFMANN, 1998.

- [65] N. Islam, A. Prodromidis, and M. S. Squillante. Dynamic partitioning in different distributed-memory environments. In *Job Scheduling Strategies for Parallel Processing*, pages 244–270. Springer-Verlag, 1996.
- [66] D. Jackson, Q. Snell, and M. Clement. Core algorithms of the Maui scheduler. In *Job Scheduling Strategies for Parallel Processing (JSSPP'01)*, pages 87–102. Springer Verlag, 2001.
- [67] I. F. M. L. James Frey, Todd Tannenbaum and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, August 7-9 2001. Available at <http://www.cs.wisc.edu/condor/condorg/>.
- [68] Z. Kalbarczyk, S. Bagchi, K. Whisnant, and R. Iyer. Chameleon: A software infrastructure for adaptive fault tolerance, June 1999.
- [69] T. Kielmann, H. E. Bal, J. Maassen, R. van Nieuwpoort, L. Eyraud, R. Hofman, and K. Verstoep. Programming environments for high-performance grid computing: the albatross project. Available at <http://citeseer.nj.nec.com/500235.html>.
- [70] K. Krauter, R. Buyya, and M. Maheswaran. A survey and taxonomy of resource management systems for grid computing systems. *Software Practice and Experience*, 32:135–164, 2002.
- [71] P. Kwong and S. Majumdar. Study of data distribution strategies for parallel I/O management. In *Proceedings of 3rd International ACPC Conference with Special Emphasis on Parallel Databases and Parallel I/O*, pages 12–23, 1996.
- [72] P. Kwong and S. Majumdar. Scheduling of I/O in multiprogrammed parallel systems. *Informatica (Slovenia)*, 23(1):104–113, 1999.

- [73] K. M. L. Alvisi. Waft: Support for fault-tolerance in wide-area object oriented systems. In *Proceedings of the 2nd Information Survivability Workshop*, pages 5–10, 1998.
- [74] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of I/O for gang scheduled workloads. In *Job Scheduling Strategies for Parallel Processing*, pages 215–237. Springer Verlag, 1997.
- [75] J. Leon, A. L. Fisher, and P. Steenkiste. Fail-safe PVM: A Portable Package for Distributed Programming with Transparent Recovery. Technical report, CMU, Department of Computer Science, Feb 1993.
- [76] S. T. Leutenegger and X.-H. Sun. Limitations of cycle stealing for parallel processing on a network of homogeneous workstations. *Journal of Parallel and Distributed Computing*, 43(2):169–178, 1997.
- [77] S. T. Leutenegger and M. K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'90) Conference*, pages 226–236. IEEE Computer Society, 1990.
- [78] G. Lynch. Parallel job scheduling on heterogeneous networks of multiprocessor workstations. Technical report, University of Toronto, 2002.
- [79] S. Majumdar, D. Eager, and R. Bunt. Scheduling in multiprogrammed parallel systems. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'88) Conference*, pages 104–113, 1988.
- [80] S. Majumdar and F. Shad. Characterization and management of I/O on multiprogrammed parallel systems. In *Proceedings of the 1995 IEEE Symposium on Parallel and Distributed Processing*, pages 502–510, October 1995.

- [81] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, 1993.
- [82] C. McCann and J. Zahorjan. Processor allocation policies for message-passing parallel computers. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'94) Conference*, pages 19–32, 1994.
- [83] E. L. Miller and R. H. Katz. Analyzing the I/O behavior of supercomputer applications. *11th IEEE Symposium on Mass Storage Systems*, pages 388–397, 7-10 1991.
- [84] M. Mutka and M. Livny. Scheduling remote processing capacity in a workstation processor bank network. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 298–299, 2000.
- [85] M. M. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 12(4):269–84, Jul 1991.
- [86] V. K. Naik, S. Setia, and M. S. Squillante. Processor allocation in multiprogrammed distributed-memory parallel computer systems. *Journal of Parallel and Distributed Computing*, 46(1):28–47, 1997.
- [87] A. Nguyen-Tuong. Integrating fault-tolerance techniques in grid applications. Technical report, University of Vergina, 2000.
- [88] A. Nguyen-Tuong, A. S. Grimshaw, and J. F. Karproovich. Fault-tolerance via replication in coarse grain data-flow. Technical Report CS-95-38, Department of Computer Science, University of Virginia, 1995.

- [89] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of 3rd International Conference on Distributed Computing Systems*, pages 22–30. IEEE Computer Society, 1982.
- [90] J. D. Padhye and L. Dowdy. Dynamic versus adaptive processor allocation policies for message passing parallel computers: An empirical comparison. In *Job Scheduling Strategies for Parallel Processing*, pages 224–243. Springer-Verlag, 1996.
- [91] E. W. Parsons and K. C. Sevcik. Multiprocessor scheduling for high-variability service time distributions. In *Job Scheduling Strategies for Parallel Processing*, pages 127–145. Springer-Verlag, 1995.
- [92] E. W. Parsons and K. C. Sevcik. Coordinated allocation of memory and processors in multiprocessors. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'96) Conference*, pages 57–67, 1996.
- [93] B. K. Pasquale and G. C. Polyzos. Dynamic I/O characterization of I/O intensive scientific applications. In *Supercomputer Conference*, pages 660–669, SC 1994.
- [94] V. G. Peris, M. S. Squillante, and V. K. Naik. Analysis of the impact of memory in distributed parallel processing systems. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'94) Conference*, pages 5–8, 1994.
- [95] J. S. Plank and W. R. Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. In *Symposium on Fault-Tolerant Computing*, pages 48–57, 1998.

- [96] J. S. Plank and M. G. Thomason. Processor allocation and checkpoint interval selection in cluster computing systems. *Journal of Parallel and Distributed Computing*, 61(11):1570–1590, 2001.
- [97] K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Symposium for High Performance Distributed Computing*, pages 352–361, 2002.
- [98] R. V. Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical Report TR98-1687, 28 1998.
- [99] E. Rosti, G. Serazzi, E. Smirni, and M. S. Squillante. The impact of I/O on program behavior and parallel scheduling. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'98) Conference*, pages 56–65, 1998.
- [100] E. Rosti, G. Serazzi, E. Smirni, and M. S. Squillante. Models of parallel applications with large computation and I/O requirements. *IEEE Transactions on Software Engineering*, 28(3):286–307, 2002.
- [101] E. Rosti, E. Smirni, L. W. Dowdy, G. Serazzi, and B. M. Carlson. Robust partitioning policies of multiprocessor systems. *Performance Evaluation*, 19(2-3):141–165, 1994.
- [102] E. Rosti, E. Smirni, L. W. Dowdy, G. Serazzi, and K. C. Sevcik. Processor saving scheduling policies for multiprocessor systems. *IEEE Transactions on Computers*, 47(2):178–189, 1998.
- [103] E. Rosti, E. Smirni, G. Serazzi, and L. W. Dowdy. Analysis of non-work-conserving processor partitioning policies. In *Job Scheduling Strategies for Parallel Processing*, pages 165–181. Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

- [104] K. Ryu and J. Hollingsworth. Exploiting Fine-Grained Idle Periods in Networks of Workstations. *IEEE Transactions on Parallel and Distributed Systems*, 11(7):683–698, 2000.
- [105] K. D. Ryu, J. K. Hollingsworth, and P. J. Keleher. Mechanisms and policies for supporting fine-grained cycle stealing. In *International Conference on Supercomputing (SC'1999)*, pages 93–100, 1999.
- [106] J. W. S. Zhou, X. Zheng and P. Delisle. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. *Software – Practice and Experience*, 23(2):1305–1336, 1993.
- [107] A. Sarangi, A. Shriram, and A. Shankar. A scheduling model for grid computing systems. In *Proceedings of GRID 2001*, pages 111–123, 2001.
- [108] F. B. Schneider. Byzantine generals in action: Implementing failstop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, 1984.
- [109] S. Setia, M. S. Squillante, and V. K. Naik. The impact of job memory requirements on gang-scheduling performance. *Performance Evaluation Review*, 26(4):30–39, 1999.
- [110] S. K. Setia. The interaction between memory allocation and adaptive partitioning in message-passing multicomputers. In *Job Scheduling Strategies for Parallel Processing*, pages 146–164. Springer-Verlag, 1995.
- [111] S. K. Setia and S. K. Tripathi. A comparative analysis of static processor partitioning policies for parallel computers. In *Proceedings of IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS'93)*, pages 283–286, 1993.

- [112] D. L. E. Shikharesh Majumdar and R. B. Bunt. Characterisation of programs for scheduling in multiprogrammed parallel systems. *Performance Evaluation*, (2):109–130, 1991.
- [113] J. Skovira, W. Chan, H. Zhou, and D. Lifka. The EASY - LoadLeveler API project. In *Job Scheduling Strategies for Parallel Processing*, pages 41–47. Springer-Verlag, 1996.
- [114] E. Smirni and D. A. Reed. Lessons from characterizing the Input/Output behavior of parallel scientific applications. *Performance Evaluation*, 33:27–44, 1998.
- [115] Q. Snell, M. Clement, D. Jackson, and C. Gregory. The performance impact of advance reservation meta-scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 137–153. Springer Verlag, 2000.
- [116] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, editors. *MPI: The Complete Reference*. The MIT Press, 1996.
- [117] P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien. Dynamic coscheduling on workstation clusters. In *Job Scheduling Strategies for Parallel Processing*, pages 231–256. Springer Verlag, 1998.
- [118] M. S. Squillante. On the benefits and limitations of dynamic partitioning in parallel computer systems. In *Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science*, pages 219–238. 1995.
- [119] M. S. Squillante. On the benefits and limitations of dynamic partitioning in parallel computer systems. In *Job Scheduling Strategies for Parallel Processing*, pages 219–238. Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

- [120] P. Stelling, I. Foster, C. Kesselman, and G. v. L. C. Lee. A fault detection service for wide area distributed computations. In *Proceedings of 7th Symposium on High Performance Computing*, pages 268–278, 1998.
- [121] A. V. Stergios and K. C. Sevcik. Parallel application scheduling on networks of workstations. *Journal of Parallel and Distributed Computing*, 43(1):1159–1166, 1997.
- [122] G. E. Suh, L. Rudolph, and S. Devadas. Effects of memory performance on parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science*, volume 2221, pages 116–132. Springer Verlag, 2001.
- [123] J. V. Sumanth, X. Qin, A. Qadi, and H. Jiang. Performance evaluation of parallel I/O. Technical report, Department of Computer Science, Nebraska-Lincoln, May 2002.
- [124] T. Thanalapathi and S. Dandamudi. An efficient adaptive scheduling scheme for distributed memory multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 12(7):758–768, July 2001.
- [125] Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. *Operating Systems Review*, 23(5):1159–66, 1989.
- [126] J. Turek, U. Schwiegelshohn, J. L. Wolf, and P. S. Yu. Scheduling parallel tasks to minimize average response time. In *Proceedings of the Fifth Annual ACM SIAM Symposium on Discrete Algorithms (SODA)*, pages 112–121, 1994.
- [127] S. Vadhiyar and J. Dongarra. A metascheduler for the grid. In *Proceedings of the 11th IEEE Symposium on High-Performance Distributed Computing*, pages 343–351, July 2002.

- [128] J. D. Vaidy S. Sunderam, Al Geist and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–545, 1994.
- [129] R. van Nieuwpoort, J. Maassen, H. E. Bal, T. Kielmann, and R. Veldema. Wide-area parallel programming using the remote method invocation model. *Concurrency: Practice and Experience*, 12(8):643–666, 2000.
- [130] H. Wabnig and G. Haring. Performance Prediction of Parallel Systems with Scalable Specifications-Methodology and case Study. *Performance Evaluation Review*, 22(2-4):46–62, 1995.
- [131] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, Sep 1995.
- [132] F. Wang, H. Franke, M. Papaefthymiou, P. Pattnaik, L. Rudolph, and M. S. Squillante. A gang scheduling design for multiprogrammed parallel computing environments. In *Job Scheduling Strategies for Parallel Processing*, pages 111–125. Springer-Verlag, 1996.
- [133] F. Wang, M. Papaefthymiou, and M. Squillante. Performance evaluation of gang scheduling for parallel and distributed multiprogramming. In *Job Scheduling Strategies for Parallel Processing*, pages 277–298. Springer Verlag, 1997.
- [134] J. B. Weissman. Gallop: The benefits of wide-area computing for parallel processing. *Journal of Parallel and Distributed Computing*, 54(2):183–205, 1998.
- [135] J. B. Weissman. Fault-tolerant wide area parallel computation. In *Proceedings of Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 1214–1225, 2000.

- [136] J. B. Weissman and D. Womack. Fault-tolerant scheduling in distributed networks. Technical Report CS-96-10, Department of Computer Science, University of Virginia, Sep. 25 1996.
- [137] D. Wright. Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with condor. In *Proceedings of the Linux Clusters: The HPC Revolution conference*, June 2001.
- [138] Ying Chen and Marianne Winslett. Automated tuning of parallel I/O systems: An approach to portable I/O performance for scientific applications. *IEEE Transactions on Software Engineering*, 26(4):362–383, April 2000.
- [139] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam. An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. In *Job Scheduling Strategies for Parallel Processing*, pages 133–158. Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.