Application-Oriented Scheduling in Multicluster Grids

Ömer Ozan SÖNMEZ

Application-Oriented Scheduling in Multicluster Grids

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op maandag 7 juni 2010 om 15:00 uur

door **Ömer Ozan SÖNMEZ**

Master of Science in Computer Engineering, Koç University, Turkey
geboren te Istanbul, Turkey

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. H.J. Sips

Samenstelling promotiecommissie:

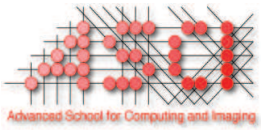
Rector Magnificus	voorzitter
Prof.dr.ir. H.J. Sips	Technische Universiteit Delft, promotor
Dr.ir. D.H.J. Epema	Technische Universiteit Delft, copromotor
Prof.dr. K.G. Langendoen	Technische Universiteit Delft
Prof.dr.ir. H.E. Bal	Vrije Universiteit Amsterdam
Prof.dr. D. Trystram	Grenoble Institute of Technology, France
Dr. E. Deelman	University of Southern California, USA
Dr. A. Gürsoy	Koç University, Turkey
Prof.dr. C. Witteveen	Technische Universiteit Delft, reservelid

Published and distributed by: Ömer Ozan SÖNMEZ
E-mail: osonmez@gmail.com

ISBN: 978-90-79982-07-3

Keywords: Grids, clusters, scientific applications, scheduling, co-allocation, predictions, experimentation, performance

Printed in The Netherlands by: Wöhrmann Print Service



The work described in this thesis has been carried out in the ASCI graduate school. ASCI dissertation series number 201.
This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl), which is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W), and which is part of the ICT innovation program of the Dutch Ministry of Economic Affairs (EZ). Part of this work is also carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

To my parents,

Contents

Chapter 1

Introduction

The term 'grid computing' was coined in the mid 1990s to describe the seamless, secure, and coordinated sharing of geographically distributed computer systems in order to solve computationally demanding problems in science, engineering, and industry [77]. The name grid has been given as an analogy to the power grids with the idea that computational resources should be obtained as swiftly as electrical energy by only plugging into a grid. Over the last decade, numerous national and international grid computing systems have been deployed worldwide[1], typically joining multiple, geographically distributed, autonomous cluster systems with high-speed wide-area interconnections. The EGEE Grid [63] across Europe, TeraGrid [194] in the USA, Grid'5000 [90] in France, and the DAS system [52] in the Netherlands, which will be described in detail in Section 1.4.1, are some examples of such systems. Various scientific communities −from fields as diverse as high energy physics, earth sciences, and life sciences− use these grid systems to run their applications, which have widely different characteristics that pose unique resource requirements to the grid.

The underlying challenges of grid computing, including resource transparency, security, application execution and file management, have been well addressed by grid middleware solutions [88, 89, 92, 183]. Grids also need high-level scheduling (meta-scheduling) systems [33, 38, 91, 148] that use grid middleware in order to map application tasks to resources and then manage their execution on behalf of users. While vast numbers of powerful computers exist in grids, the problem of assigning computations and data to them in such a way as to optimize application performance is a challenging task due to the complex nature of the grid systems, as well as to the communication and structural characteristics of the applications. Further complicating this situation are the competing needs of users. In this thesis we address the challenge of designing and analyzing realistic and practical application-oriented scheduling mechanisms in multicluster grid systems. Application-oriented scheduling in grids aims to optimize user-centric performance cri-

[1] A broad list of grid projects is available at: http://www.gridcomputing.com/

teria, such as application execution time, with methods specialized for different types of applications. However, the vast amount of research on application-oriented scheduling methods in grids does not propose practical solutions due to its unrealistic assumptions about grid environments. The grid community still needs realistic grid-level scheduling solutions −preferably deployed and evaluated in real systems− that would improve the execution performance of certain types of applications. In order to address this need, in this thesis, we propose various application-oriented scheduling policies, most of which we have implemented and evaluated in a real multicluster grid environment, in addition to simulation-based experiments in which we use realistic scenarios.

The remaining part of this chapter is organized as follows. In Section 1.1, we give an overview of scheduling in multicluster grids, while in Section 1.2, we describe the common application types that we encounter in grids. In Section 1.3, we present the problem statement of this thesis. In Section 1.4, we describe the multicluster grid system, the grid scheduler, and the simulation environment that we have used in our implementations and experiments. Finally, in Section 1.5, we present the research contributions and outline the structure of this thesis.

1.1 An overview of scheduling in multicluster grids

Today's grid computing systems vary widely in the technologies and standards they use, as well as in their structure and usage scenarios. However, the aim is very similar in all of these cases, that is, exploiting a diverse set of resources together for the sake of the efficient execution of applications. Therefore, resource management, and in particular scheduling, plays an important role in obtaining better execution performance for the applications. In this section, we present the main concepts of scheduling in multicluster grids through a generic scheduling framework.

1.1.1 Key aspects and organization

Grid scheduling can be defined as the process of assigning jobs to grid resources that span multiple administrative sites. This process is typically done with the goal of minimizing the turn around time of a job. We refer to a job as the application that a user wants to execute in a grid. In addition, we refer to a site as a set of processing and data resources, as well as a local resource manager (scheduler), which runs on the front-end machine of the site, in a single administrative unit. Figure 1.1 depicts a generic scheduling framework for multicluster grids. Below, we describe the main elements, the local resource manager, the grid middleware, and the grid scheduler, which together make up this framework.

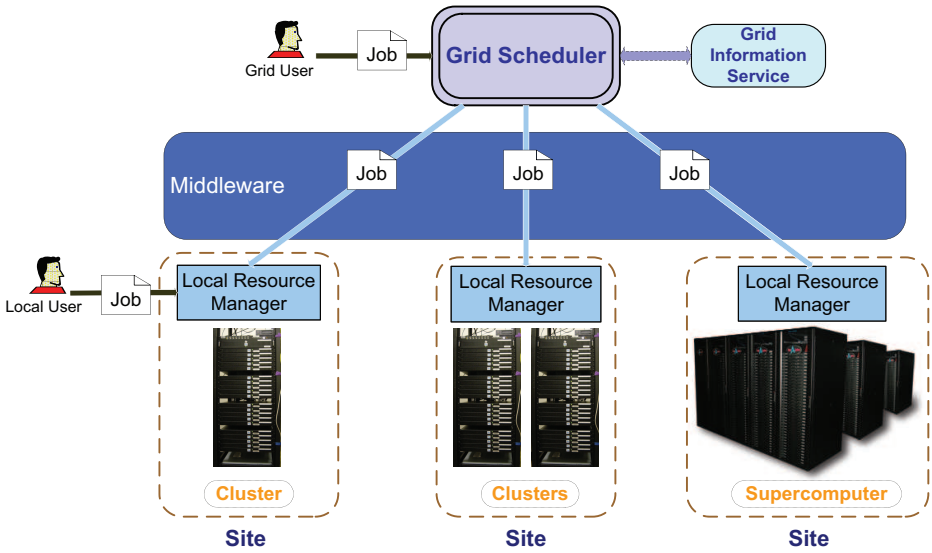


Figure 1.1: A generic scheduling framework for multicluster grids.

Local resource manager

Resources in a single site are managed by a Local Resource Manager (LRM), which provides low-level resource allocation and scheduling for the jobs of both local and grid users. A local user can use the resources of the local site she is affiliated with, while a grid user can use resources from multiple such sites to which she has granted access. An LRM has exclusive control over its resources, that is, no jobs can be scheduled on those resources without using the LRM. Jobs that are submitted to an LRM are initially placed into queues until there are enough idle computing nodes (or processors) to execute the jobs. After that, the LRM dispatches the jobs to the assigned nodes, and then manages the job execution until its completion, and finally, returns the results to the submitting party. LRMs can be queried to retrieve information about the properties of the system that they manage, and information about the queued and scheduled jobs. However, the level of detail of the information provided can be limited if system policy restrictions apply.

Most LRMs, including the Portable Batch System (PBS-Pro) [165], the Sun Grid Engine (SGE) [87], LoadLeveler [1], and the Load Sharing Facility (LSF) [136], focus on maximizing processor throughput and utilization, and minimizing the average wait time and response time of jobs. Typical installations of such LRMs use the first-come first-served (FCFS) with backfilling [70, 132] as the scheduling policy. FCFS simply considers jobs for dispatch in their arrival order, while backfilling is an optimization of FCFS that tries to balance the goals of utilization and maintaining FCFS order. Backfilling requires users to provide information in advance on the maximum job execution time. While the job at the head of the queue is waiting, backfilling allows jobs with smaller processor

requirements to be dispatched, provided that they would not delay the execution of the job at the head of the queue. In addition to traditional job scheduling policies, some of the LRMs (e.g., PBS-Pro [165] and Maui [140]) also support more advanced techniques, including advance reservations, cycle scavenging, peer scheduling, and job checkpointing and restart.

LRMs are typically designed for single administrative domains, and are usually configured in a way to give priority to the jobs of the local users over the jobs of grid users. The agreements on the usage of the resources among the parties that are involved in the grid are usually defined by Service Level Agreements (SLAs) [62]. These agreements define specific policies that state who can access the resources, how and when the resources can be used, and other such criteria.

Grid middleware

Grid computing often requires the use of a middleware software to mitigate the complexity of integrating distributed autonomous systems. Grid middleware intermediates between grid applications and the grid hardware infrastructure, and therefore, it hides the underlying physical infrastructure from the users and from the vast majority of programmers. In doing so, grid middleware offers transparent access to a wide variety of distributed resources to users and simplifies the collaboration between organizations.

Grid middleware such as the Globus Toolkit [76, 89], Legion [92], UNICORE [183], and gLite [88] have contributed a lot to the growth of grids by simplifying grid access and usage. Among these middlewares, the Globus Toolkit is the best known, and it is the one that we use in some of the work of this thesis. Globus comprises a set of modules each of which defines an interface that users and higher-level services can use to invoke that module's mechanisms [89]. These modules implement fundamental services, such as resource management, security, data management, and communications. For instance, the Globus Resource Allocation Manager (GRAM) allows interfacing to different LRMs for locating resources, and for submitting, monitoring, and canceling jobs on remote compute resources. The Globus Security Infrastructure (GSI) module provides basic authentication mechanisms that can be used to validate the identity of both users and resources. GSI supports delegation of credentials for computations that involve multiple resources and/or sites. This allows a user to sign-on only once (single sign-on) to use grid resources at multiple sites. The Monitoring and Discovery System (MDS) is the information service component of the Globus Toolkit, and it provides information about the available resources on the grid and their status. Finally, Globus provides GridFTP, which is a high-performance data transfer tool that is optimized for high-bandwidth wide-area networks.

Grid middleware solutions provide a set of services that facilitate application execution on grids; however, these grid middleware services do not make any scheduling de-

cisions that require application-specific knowledge [51]. Instead, users and higher-level services such as grid schedulers are expected to make any scheduling decisions that require application knowledge, and they use those middleware services to allocate resources and run their applications according to scheduling decisions.

Grid Scheduler

For a common grid user it might be difficult and tedious to manually find and allocate all the resources needed to execute an application. To automate this process, grids need high-level scheduling systems [2]. A typical grid scheduling system provides an interface to users for expressing job requirements, for submitting jobs, for scheduling jobs across the grid, for launching jobs for execution, for error handling, and for recovery during the execution of the job. It uses grid middleware services to interface to different LRMs.

Grid-level scheduling involves three main phases [175]. The first phase, *resource discovery*, involves creating a list of potential execution sites that the user submitting the job has access to. All of the sites in this list must also meet the job requirements specified by the user. The set of possible job requirements can be very broad and can vary among jobs. It may include static details such as an operating system or a specific hardware architecture, as well as dynamic details such as number of processors or a minimum bandwidth allocation.

The second phase, *site selection*, involves determining the execution site, among the potential sites, on which the job will run. In order to attain better job execution performance, efficient scheduling methods are required to identify the proper site for a job based on the information obtained from a grid information service. The grid information service gathers information from individual sites through its software sensors. The scheduler queries the information service to get static (e.g., resource capabilities) or dynamic information (e.g., existing reservations, queue lengths, scheduled jobs, future resource availabilities) about the clusters, to be used in decision making. A grid information service can be provided by the underlying grid middleware, e.g., Globus MDS [89], as well as in the form of a third-party tool, e.g., Ganglia [84], or a custom implementation. In general, scheduling jobs in a distributed system is an NP-complete problem [74], and therefore, the proposed grid scheduling methods are mainly heuristics except for some special cases [11, 43]. In practice, a grid scheduler may be required to handle different types of jobs using different scheduling policies. For instance, some jobs may require Quality of Service (QoS) support while others may require best effort service. Other examples can be given by considering the structural properties of the jobs. In Section 1.2, we elaborate on the characteristics of the applications that we encounter in grids, and that grid schedulers should take into account in order to provide better performance. In addition, the scheduler also has to consider local site policies; a site may specify a maximum

percentage of the resources, in terms of number of resources and time, to be allocated for grid use.

In the third phase, *job execution*, the job is submitted to the LRM of the selected site. The preparation phase of the submission may include setup, file staging, reservation claiming, or other such tasks that are required to prepare the resource to execute the job. Submitting a job in grids can be very complicated because of a lack of standards for job submission since grid middleware services mostly rely on local-parameter fields [175]. The ongoing work in the Global Grid Forum [160] addresses this need for common job submission APIs with the Distributed Resource Management Application API (DRMAA) [59]. After the job is submitted, the status of the job (e.g., failed, queued, running, or finished) is communicated to the scheduler by the job submission service. Finally, after the job is executed, the output files −if there are any− associated with the job are transfered to the designated locations.

Most jobs in grids use the resources of only one site. However, some types of jobs, e.g., jobs that run parallel applications, may take advantage from running on resources in multiple sites. Therefore, jobs may require co-allocation, i.e., the simultaneous or co-ordinated allocation of resources at multiple sites [47, 145, 186]. However, co-allocation presents a challenge to the grid scheduler, that is, guaranteeing the availability of resources in different sites at the job's start time. The most straightforward strategy to do so is to reserve processors at each of the selected sites. If the LRMs do support reservations, this strategy can be implemented by having the scheduler obtain a list of available time slots from each LRM, reserve a common time slot for all job components, and notify the LRMs of this reservation. In the absence of processor reservation mechanisms, alternative solutions are required in order to achieve co-allocation [146].

Scheduling in grids can be done at the application level, globally, or as a combination of the two. Application-level grid schedulers, such as Nimrod/G [33] and AppLeS [15, 38], schedule an application on submission, based on the information available regarding the resources. A separate scheduler instance runs for each application submitted, and information about other applications that are already running, or being simultaneously submitted, are ignored due to the lack of a central control. Global schedulers, such as GridWay [91] and GrADS [201], on the other hand, consider information about other jobs. Since they acquire more knowledge about the status in the system, they can make more informed decisions. Our KOALA grid scheduler [144, 148], which will be described in Section 1.4.2, allows a combination of global and application-level scheduling. Once resources are allocated for a job by the grid-level policies −depending on the job type as we will explain in Section 1.2− the scheduling of the tasks that make up the job can be delegated to the application-level policies.

1.1.2 Challenges in grid scheduling

In traditional parallel computing systems, scheduling is a well-studied problem. The schedulers of such systems are tightly coupled with the system, and have full control of the resources that they manage. In contrast, grid schedulers have no control over the resources that are dispersed across multiple administrative domains. Therefore, the scheduling methodologies and policies proposed for the traditional parallel systems cannot directly be applied to grid environments. Below we identify the main challenges in grid scheduling, which make it more difficult than traditional scheduling, and which we take into account in the scheduling mechanisms and policies that we have designed and implemented in the work of this thesis.

1. **Lack of control over resources.** Grid schedulers have to make resource selection decisions in an environment where they have no control over the local resources; they have to interface to information services about resource availability, and to LRMs to schedule jobs. Each individual site making up a grid may have a different owner, has its own user community, and has its own autonomous LRM. The site owners are often not willing to give up the autonomy of their sites, but will only allow access to their resources through a grid scheduler that interfaces to their LRM according to specific usage rules. Moreover, the LRMs may have different properties and capabilities. For instance, some LRMs may support advance resource reservation, while others support queuing-based scheduling, or some may support job checkpointing and migration, while others do not.

2. **Characteristics of grid resources.** Typically, resources in a grid system are heterogeneous in terms of hardware, e.g., processor architecture, disk space, network, software, e.g., operating system, libraries, and systems management, e.g., security set-up, usage SLAs. Moreover, the availability of resources in a grid system varies frequently. In addition to failures, resources may be allocated (or released) by concurrent users, and resource owners may add or withdraw their resources to/from the resource pool at any time. This dynamic nature of the resources, together with the heterogeneity, makes it difficult for grid schedulers to predict the behavior of applications on grids.

3. **Lack of complete control over jobs.** Grid schedulers do not have full control over the entire set of jobs in a grid; local jobs and jobs submitted by multiple grid schedulers have to co-exist in a grid. The jobs that are executed in a single site in a grid may be submitted through the local scheduler or through any of a number of grid schedulers. This means that a grid scheduler has to take into account jobs from multiple sources when deciding on where a particular job should run.

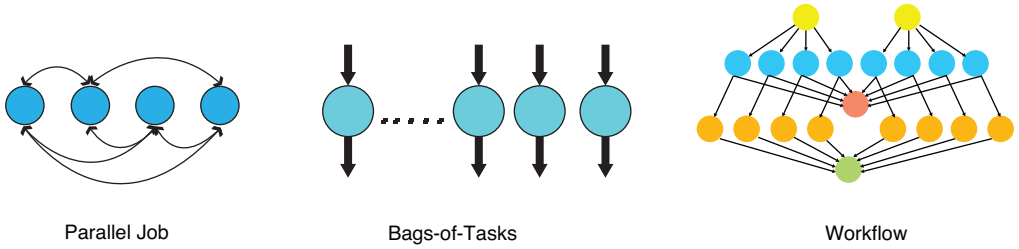


Figure 1.2: Common job models in grids.

1.2 Job models in grids

Ever more scientists use grid computing to meet the resource needs of their applications. For instance, many applications of grand challenge problems [86], such as protein folding, financial modeling, earthquake simulation, and climate/weather modeling, are being executed on grid systems in order to make use of the vast number of resources. Such applications in grids can be implemented using different software libraries (the well-known grid programming libraries are presented in the work of Lee and Talia [127]), such as MPI [152], Ibis [12, 205], JavaSpaces [80], and ProActive [10]. In addition, grid applications often need to be represented with a job model, such as parallel, bags-of-tasks, and workflow (see Figure 1.2), to facilitate their execution in grids. In this section we describe the common job models in grids, which we also use throughout this thesis.

The description of a job varies according to the job description language being supported by the grid scheduler or the grid middleware. A job may comprise several tasks, which can be scheduled independently or together depending on the job model or the scheduling policy being used. A grid scheduling system needs to implement appropriate mechanisms to be able to handle application scheduling and execution with regard to the different job models that we explain below.

A parallel job is composed of several tasks which can be executed in parallel. Tasks run on a number of computational nodes in parallel, and exchange information using some underlying library, such as MPI [152] or Ibis [12, 205]. Parallel jobs can be classified as rigid, moldable, and malleable [71]. A rigid job requires a fixed number of processors. When the number of processors can be adapted only at the start of the execution, the job is called moldable. Similar to rigid jobs, the number of processors for moldable jobs cannot be changed during runtime. Jobs that have the flexibility to change the number of assigned processors during their runtime, that is, that can grow or shrink, are called malleable. Parallel applications can be made malleable using specific programming models, such as Ibis-Satin [203], and the DYNACO Framework [28,30]. For a parallel job, a grid scheduler has to deal with the job's programming library, including assembling the list of resources the job is executed on for an MPI job, or adding the location of the nameserver to the parameters of an Ibis job.

A bags-of-tasks (BoT) is composed of independent tasks that can be scheduled and executed in any order without needing inter-task communication. Parameter sweep applications (PSAs) are a special type of BoT with tasks that each execute the same program but with different parameters. There are various important BoT applications in grids, including data mining, massive searches, Monte Carlo simulations, fractal calculations (such as Mandelbrot), and image processing applications (such as tomographic reconstruction) [207].

Workflows constitute another commonly used job model in many complex grid applications. In general, workflows are represented as directed acyclic graphs (DAGs), where the nodes represent tasks to be performed and the edges represent dependencies between tasks. For a workflow job, the scheduler needs to take into account issues like task interdependencies, advanced reservations, and fault-tolerance, besides the job's programming model. Important workflow applications in grids include Montage (astronomy application) [149], CyberShake (earthquake analysis application) [46], and SIPHT (bioinformatics application) [135].

Irrespective of their models, applications may pose different communication requirements. For instance, some applications are communication-intensive, requiring large volumes of data movement among the tasks, while other applications are computation-intensive, requiring minor, or no data movement at all. In this respect, a grid scheduler should take into account the communication characteristics of the applications in order to improve their execution performance.

1.3 Problem statement

Grid scheduling is challenging due to the heterogeneity and the dynamic nature of the grid resources as well as to the lack of control of those resources. The wide variety in the structural and the communication characteristics of the applications submitted to grids further complicate grid scheduling, and may lead to poor or unpredictable performance unless these characteristics are taken into account. Therefore, we need efficient scheduling mechanisms and policies in grids that are specialized for different common grid application types such as parallel applications, bags-of-tasks, and workflows.

In this thesis our aim is to propose and implement scheduling mechanisms and policies for different application types, and to investigate their performance in a real multicluster grid system, the DAS, using our KOALA multicluster grid scheduler, as well as with simulations using realistic scenarios. We study grid scheduling for a wide range of grid application types including parallel applications that may need co-allocation or malleability, bags-of-tasks that can benefit from cycle scavenging, and workflows. More specifically, in this thesis we aim to answer the following research questions:

How beneficial is processor co-allocation in multicluster grids? In multicluster grid systems, parallel applications may benefit from processor co-allocation, that is, the

simultaneous allocation of processors in multiple clusters. Although co-allocation allows the allocation of more processors than available in a single cluster, it may severely increase the execution time of applications due to the relatively slow wide-area communication. Despite various simulation-based performance evaluation studies on co-allocation [25, 27, 66, 111, 172], and efforts to support co-allocation in a real multicluster grid scheduler [144,145,147,148,187], we still lack a complete understanding of to which extent processor co-allocation is beneficial in real grids. To this end, we will perform a comprehensive investigation on the benefit of processor co-allocation in multicluster grids.

How to schedule malleable applications in multicluster grids? Application malleability, that is, the property of parallel applications to use varying amounts of resources such as processors during their execution, is potentially a very versatile and beneficial feature. Malleability facilitates dealing with the dynamic nature of multicluster grid systems. Despite several approaches that have been proposed to build parallel malleable applications [28, 114, 198, 199], virtually no existing multicluster or grid infrastructure is able to benefit from this property. Most of the previous work on scheduling malleable applications does not meet the challenges that appear in the context of multicluster systems. We will address such issues as the selection of a suitable execution site for each malleable application, resilience to background load due to local users, and prioritizing malleable applications for dynamic resource allocation operations (i.e., growing and shrinking in terms of number of resources being used).

How to run large-scale cycle-scavenging applications politely yet efficiently in multicluster grids? Cycle scavenging is the underlying technology of desktop grids that enables harnessing idle CPU cycles to solve large-scale scientific problems. The same concept can be applied to multicluster grid environments to give users the opportunity of executing such large-scale computation intensive applications at a low priority without being in the way of regular grid users or local users. Although various cycle scavenging systems [7, 44, 134] exist, they all necessitate additional software installations on the compute nodes or modifications to the LRM of the clusters, both of which would be administrative obstacles in multicluster grid systems. We will address the question of how to incorporate/integrate cycle scavenging into grid schedulers in a seamless way without the need for modifying LRMs or for (pre-)installing any additional software on the compute nodes.

What is the performance of bags-of-task scheduling in multicluster grids? In contrast to the workloads of tightly coupled parallel computing systems, a considerable part of the workloads submitted to multicluster grids consists of bags-of-tasks (BoT) [98]. Therefore, a realistic performance analysis of scheduling BoTs in multicluster grids is important. Although many scheduling policies for BoTs have been proposed for a variety of systems, most of these policies have been proposed for tightly-coupled systems, or at

least, they do not consider heterogeneous resources. The few solutions that can be applied in practice in multicluster grid systems [39, 82, 130, 180] assume that the scheduler has either no information or mostly accurate information at its disposal. We will tackle the question of what is the performance of BoT scheduling in multicluster grids by proposing realistic scheduling policies and evaluating them with detailed realistic scenarios.

What is the performance of dynamic workflow scheduling in multicluster grids? Scientists increasingly rely on grid workflows for executing complex mixtures of tasks with data dependencies. Previous research on workflow scheduling [20, 21, 93, 125, 196] considers usually static scheduling methods in which workflow tasks are mapped to resources before their execution. Static scheduling, however, does not consider the dynamic nature of multicluster grids, and besides, it assumes that perfectly accurate information is available about the system resources and the workflow tasks, which is an unrealistic assumption for grids. Motivated by this fact, we will present a realistic investigation on the performance of dynamic workflow scheduling in multicluster grids.

What is the performance of job runtime and queue wait time predictions in grids? Grids bring about not only the advantages of an economy of scale, but also the challenges of resource and workload heterogeneity. A consequence of these two forms of heterogeneity is that job runtimes and queue wait times are highly variable, which generally reduces application performance and makes grids difficult to use by the common scientist. Predicting job runtimes and queue wait times has been widely studied for traditional parallel environments [23, 56, 57, 107]. We will address the question of how the proposed prediction methods perform in grids, whose resource structure and workload characteristics are very different from those in parallel systems. Such an analysis points the grid research community in the right direction to improve the predictability of job runtimes and queue wait times in grids.

1.4 The testbeds

In this section we present the background information regarding the DAS system [52], the KOALA grid scheduler [144, 148], and the Delft Grid Simulator [104], which we have used for our implementations and experiments in this thesis.

1.4.1 The DAS system

The Distributed ASCI Supercomputer (DAS) [52] is an experimental computer testbed in the Netherlands that is exclusively used for research on parallel, distributed, and grid computing. The system was built for the Advanced School for Computing and Imaging (ASCI) [8], a Dutch research school in which several universities participate. The DAS system is now in its third generation (DAS-3) after the first and second generations have

Table 1.1: The distribution of the nodes over the DAS-2 clusters.

Cluster Location	Nodes
Vrije University	72
U. of Amsterdam	32
Delft University	32
Utrecht University	32
Leiden University	32

Table 1.2: The properties of the DAS-3 clusters.

Cluster Location	Nodes	Speed	Structure	Interconnect
Vrije University	85	2.4 GHz	dual-core	Myri-10G & GbE
U. of Amsterdam	41	2.2 GHz	dual-core	Myri-10G & GbE
Delft University	68	2.4 GHz	single-core	GbE
MultimediaN	46	2.4 GHz	single-core	Myri-10G & GbE
Leiden University	32	2.6 GHz	single-core	Myri-10G & GbE

proven to be successes. Part of our research in this thesis has been conducted on the second (DAS-2) and on the third (DAS-3) generation of the DAS system. Therefore, we describe only the last two generations of the DAS system.

The DAS-2, which was in use between 2002 and 2006, consisted of 200 nodes, organized into five dual-processor clusters of identical 1 GHz Intel Pentium III processors. The distribution of the nodes over the clusters is given in Table 1.1. The clusters were interconnected by the Dutch university backbone (100 Mb/s), and the nodes within a cluster were connected by Myrinet LAN [157] (1.2 Gb/s). On each of the clusters, the Sun Grid Engine (SGE) [87] was used as the local resource manager. SGE was configured to run applications on the nodes in an exclusive fashion, i.e., in space-shared mode.

The DAS-3 was installed late 2006. The main difference between the DAS-3 and the previous DAS systems is the degree of heterogeneity present in the system in terms of processor speed, processor structure, and network. The DAS-3 consists of 272 nodes organized into five dual-processor clusters as shown in Table 1.2 with a mixture of single-core and dual-core AMD Opteron processors. As the same table shows, the DAS-3 has a minor level of processor speed heterogeneity. All DAS-3 clusters have 1 Gb/s and 10 Gb/s Ethernet, as well as a high speed Myri-10G [157] interconnect, except for the cluster in Delft, which has only Ethernet interconnects. As in the DAS-2, each of the clusters is managed by SGE in space-shared mode.

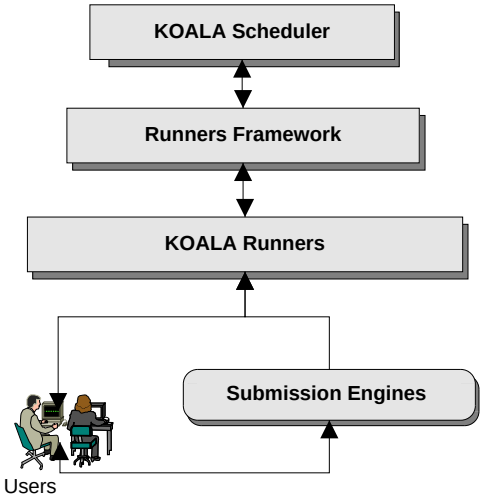


Figure 1.3: The layered architecture of the KOALA grid scheduler.

In the DAS systems, each of the DAS clusters is an autonomous system with its own file system. Therefore, in principle files (including application executables) have to be moved explicitly between users' working directories in different clusters. In addition, simple usage rules are enforced in the DAS systems that users or schedulers have to consider. The most important of these are that any application cannot run for more than 15 minutes from 08:00 to 20:00, and that execution must be performed on the compute nodes, never on the head nodes. The DAS systems can be seen as a fast prototyping computational grid environment, with its structure and usage policies designed to enable grid research.

1.4.2 The KOALA grid scheduler

The KOALA grid scheduler [144, 148][2] is designed for multicluster systems such as the DAS, with the aim of implementing mechanisms and policies for scheduling various grid application types. KOALA served scientists in the DAS-2 between 2005 and 2007, and since May 2007, it has been in operation in the DAS-3 system.

KOALA was initially designed to schedule parallel applications that may need co-allocation, that is, the simultaneous allocation of processors in multiple clusters. In the study of this thesis, we have extended KOALA with support for scheduling parallel malleable applications [31], parameter sweep applications [185], and workflow applications [188]. Below, we describe the main building blocks of KOALA. For further details we refer the reader to [148] and [144].

[2]The KOALA grid scheduler project page: http://www.st.ewi.tudelft.nl/koala/

Architecture of KOALA

KOALA has a layered architecture that allows us to develop distinct layers independently, which can then work together. The KOALA layered architecture consists of four layers: the *scheduler*, the *runners framework*, the *runners*, and the *submission engines*, as shown in Figure 1.3.

The KOALA scheduler is responsible for scheduling jobs received from the KOALA runners or any third-part job submission tools with its scheduling policies that are used to map jobs on suitable execution sites. In general, the choice of which scheduling policy to be used is initiated by the runners and therefore, it can be selected by the users for every submitted job separately. The scheduler is supported by an information service, which monitors the status of resources by means of processor and network information providers and a file replica location service.

To address the challenge of grid application deployment, KOALA has the concept of runners, which are job submission and monitoring tools. Different runners can be written to support the unique characteristics of different application types by using the KOALA runners framework. The runners framework hides the heterogeneity of the grid by providing to the runners a runtime system and its corresponding set of APIs for commonly used job submission operations such as interfacing with the KOALA scheduler for job scheduling, the transfer of input files, deploying jobs on grids, monitoring and responding to failures, and the transfer of output files back to the submission site. In addition, the runners framework has fault tolerance mechanisms that deal with the reliability issues of the grid infrastructure. Relying on the runners framework, the runners provide the environment for users to develop per-application type schedulers that can be specially tailored to match the needs of applications such as PSAs and workflows. The runners have complete freedom to implement their own mechanisms for the application level operations, or alternatively, to use the default implementations that are provided by the runners framework.

The last layer consists of the submission engines, which are third-party tools that use the runners to submit jobs to KOALA. These tools include workload generation and submission tools (e.g., Grenchmark [99]), workflow engines (e.g., Karajan [206]), and user scripts.

KOALA is not tied to any particular operating system or grid middleware. For instance, the current implementation of KOALA can use both Globus GRAM [89] and DRMAA [59] services for job submission and monitoring operations on the DAS-3 system. The job management operations in the KOALA scheduler, including scheduling policies, queue operations, interactions with the runners and middleware services, differ according to the type of the application submitted, and hence the type of the runner used. In Chapters 2, 3, 4 and 6, we present the additional necessary information on the operation of KOALA with regard to the subjects covered in those chapters.

1.4.3 The Delft grid simulator

In grid research, scientists often use simulations in order to conduct performance evaluation experiments. In real grid systems, experiments can be very time consuming and very difficult to reproduce, since it is hard to attain the same circumstances in such a dynamic environment. Therefore, simulations are critical as they facilitate the assessment large numbers of experimental settings in a reasonable amount of time. Despite many efforts, today's grid simulators [34, 35, 61, 112, 124] still lack important features in system modeling, experiment setup, and experiment management. To this end, we have implemented our own grid simulator, the Delft Grid Simulator (DGSim) [104]. When compared to previous simulation tools, DGSim focuses more on the simulation process, with better support for synthetic trace generation and use, and for exploring large design spaces [104]. Besides, DGSim enables us to implement scheduling policies at different levels, including the cluster, the application, and the user level. We tested and validated DGSim in [104] as part of our previous work. In Chapters 2, 5, 6, and 7, we extend and use DGSim for our performance evaluation experiments.

1.5 Research contributions and thesis outline

In this thesis we propose and investigate application-oriented scheduling solutions in multicluster grids. The main chapters aim to answer the research questions formulated in Section 1.3, respectively. In Chapters 2, 3, and 4, we perform mainly real experiments with KOALA at DAS, and in Chapters 5, 6, and 7, we perform mostly simulation-based experiments with our DGSim tool. Below we describe the structure and the main contributions of this thesis.

Assessing the benefit of processor co-allocation (Chapter 2). We investigate the performance of processor co-allocation in multicluster grids for parallel applications that range from computation-intensive to communication-intensive under various system load settings. Further, we compare the performance of several scheduling policies that we have specifically designed for parallel applications that may use co-allocation. We demonstrate that considering latency in the resource selection phase improves the performance of co-allocation, especially for communication-intensive parallel applications. The content of this chapter is based on our research published in [186, 187].

A scheduling framework for malleable applications (Chapter 3). We present an architecture and an actual implementation of the support for malleability in multicluster grid schedulers with the help of the DYNACO framework [28] for application malleability. We propose two policies to manage dynamic resource allocation for malleable applications that already started; one which spreads any additional processors to the malleable jobs that have been running the longest, and one that spreads them equally over all run-

ning malleable jobs. Our experimental evaluation shows that higher system utilization and shorter job execution times can be achieved when malleability is used. We also find that the relative performance of our two malleability management policies varies according to the design choice as to when to initiate a malleability management policy. The content of this chapter is based on our research published in [31].

A scheduling framework for cycle scavenging applications (Chapter 4). We present an integrated design of cycle scavenging support into a scheduling architecture for multicluster grids. The implemented cycle scavenging mechanism runs alongside the regular grid scheduling, being unobtrusive to the jobs of higher priority (both local and grid jobs). Our mechanism obviates the need for additional software installations on the compute nodes or any modifications to the resource managers of the clusters. We exclusively target Parameter Sweep Applications (PSAs) to run as cycle scavenging jobs. In addition, we propose two best-effort cycle scavenging policies that try to achieve fair-share resource allocation among cycle-scavenging users. The content of this chapter is based on our research published in [185].

The performance of bag-of-tasks scheduling (Chapter 5). We propose a taxonomy of scheduling policies for bags-of-tasks that focuses on information availability and accuracy, we also present two new task scheduling policies. We explore the large design space of bags-of-task scheduling in multicluster grids along five axes: the task selection policy, the input workload, the information policy, the task scheduling algorithm, and the resource management architecture. Notably, we find that task selection policy is important only in busy systems. In addition, we find that a centralized resource management architecture achieves the best performance for bag-of-tasks applications. The content of this chapter is based on our research published in [103].

The performance of dynamic workflow scheduling (Chapter 6). We introduce a framework for dynamic workflow scheduling that includes a novel scheduling taxonomy to which we map seven policies that cover the full spectrum of information use. We explore the performance of these seven policies in a comprehensive study that also distinguishes between single and multiple (concurrent) workflow submissions. Notably, we find that there is no single grid workflow scheduling policy with good performance across all the investigated scenarios, and we find that task throttling, that is, limiting the per-workflow number of tasks dispatched to the system, prevents the head-nodes from becoming overloaded while not unduly decreasing the performance. The content of this chapter is based on our research published in [188].

The performance of prediction methods for grid scheduling (Chapter 7). We present an analysis of the performance and benefit of predicting job execution times and queue wait times in multicluster grids based on traces gathered from various research and production grid environments. We find that time series methods for predicting job execution times, and prediction methods that give upper-bounds for job queue wait times,

yield low accuracy due to the common occurrence of burst job submissions in grids. In addition, we have investigated whether prediction-based grid-level scheduling policies can have better performance than policies that do not use predictions. We find that a better accuracy of the predictions does not imply a better performance of grid scheduling. The content of this chapter is based on our research published in [189].

Chapter 8 summarizes the thesis, presents its conclusions, and indicates several research directions originating from this thesis.

Chapter 2

Co-Allocation for parallel applications

In multicluster grids, parallel applications may benefit from using processors in multiple clusters simultaneously, that is, they may use processor co-allocation. This potentially leads to higher system utilizations and lower queue wait times by allowing parallel jobs to run when they need more processors than are available in a single cluster. Despite such benefits, with processor co-allocation, the execution time of parallel applications may severely increase due to wide-area communication overhead and processor heterogeneity among the clusters. In this chapter, we investigate the benefit of processor co-allocation (hereafter, we use 'co-allocation' to refer to 'processor co-allocation'), despite its drawbacks, through experiments performed in the DAS-3 multicluster grid environment, using our KOALA grid scheduler. In addition, we perform simulation-based experiments using our DGSim tool to extend our findings obtained in the real environment.

Our investigation on the benefit of co-allocation in multicluster grids involves the following components: First, we present an analysis of the impact of the inter-cluster communication technology of a system on the co-allocation performance of parallel applications. Secondly, we investigate when co-allocation in multicluster grids may yield lower average job response times through experiments that run workloads of real MPI applications as well as synthetic applications which vary from computation-intensive to communication-intensive. Finally, we compare the performance of co-allocation policies that we have designed and implemented in KOALA, which are the Communication Aware (CA) policy that takes either inter-cluster bandwidth or latency into account when deciding on co-allocation, and the Flexible Cluster Minimization policy (FCM) that only takes the numbers of idle processors into account when co-allocating jobs. Notably, we find that for parallel applications whose slowdown due to the inter-cluster communication is low, co-allocation is advantageous when the resource contention in the system is moderate. However, for very communication-intensive parallel applications, co-allocation is disadvantageous since it increases execution times excessively. In addition, we demonstrate that considering latency in the resource selection phase improves the performance

of co-allocation, especially for communication-intensive parallel applications.

The remaining part of this chapter is organized as follows. Section 2.1 presents a job model for parallel applications that may run on co-allocated resources. In Section 2.2, we explain the parallel job management and the main mechanisms to realize co-allocation in our KOALA grid scheduler. In Section 2.3, we present the scheduling policies of KOALA for co-allocation. In Sections 2.4, 2.5, and 2.6, we present the results of our experiments. In Section 2.7, we discuss the challenges and issues of realizing co-allocation in real multicluster grids. Section 2.8 reviews related work on co-allocation. Finally, Section 2.9 summarizes the chapter.

2.1 A job model for parallel applications

In this section we present our job model for parallel applications that may need processor co-allocation. In our model, a job comprises either one or multiple *components* that can be scheduled separately (but simultaneously) on potentially different clusters, and that together execute a single parallel application. A job specifies for each component its requirements and preferences, such as its size (the number of processors or nodes it needs) and the names of its input files. A job may or may not specify the execution sites where its components should run. In addition, a job may or may not indicate how it is split up into components. Based on these distinctions, we consider three job request structures, *fixed* requests, *non-fixed* requests, and *flexible* requests, as depicted in Figure 2.1.

In a fixed request, a job specifies the sizes of its components and the execution site on which the processors must be allocated for each component. On the other hand, in a non-fixed request, a job also specifies the sizes of its components, but it does not specify any execution site, leaving the selection of these sites, which may be the same for multiple components, to the scheduler. In a flexible request, a job only specifies its total size and allows the scheduler to divide it into components (of the same total size) in order to fit the job on the available execution sites. With a flexible request, a user may impose restrictions on the number and sizes of the components. For instance, a user may want to specify for a job a lower bound on the component size or an upper bound on the number of components. By default, this lower bound is one and this upper bound is equal to the number of execution sites in the system. Although it is up to the user to determine the number and sizes of the components of a job, some applications may dictate specific patterns for splitting up the application into components, hence, complete flexibility is not suitable in such a case. So, a user may specify a list of options of how a job can be split up, possibly ordered according to preference.

These request structures give users the opportunity of taking advantage of the system considering their applications' characteristics. For instance, a fixed job request can be submitted when the data or software libraries at different clusters mandate a specific way

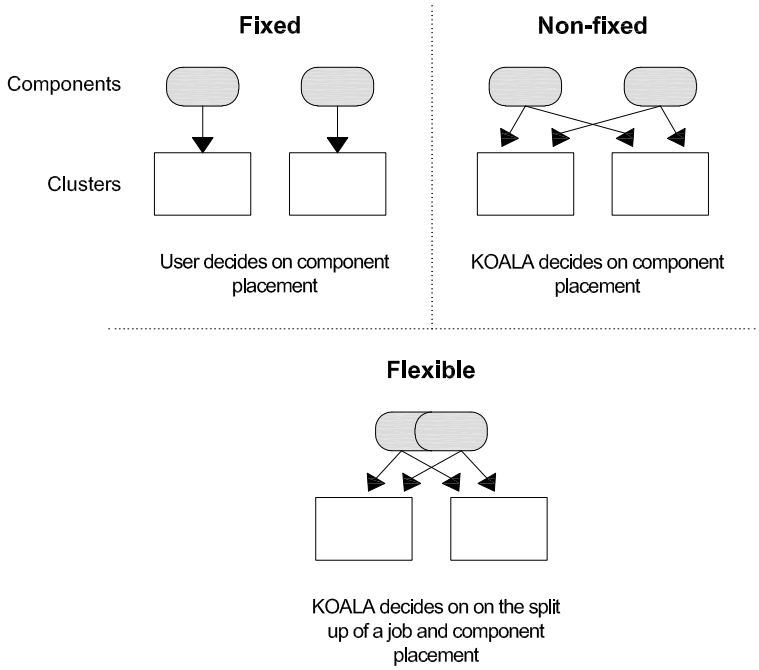


Figure 2.1: The job request types supported by KOALA.

of splitting up an application. When there is no such affinity, users may want to leave the decision to the scheduler by submitting a non-fixed or a flexible job request. Of course, for jobs with fixed requests, there is nothing a scheduler can do to schedule them optimally; however, for non-fixed and flexible requests, a scheduler should employ scheduling policies (called *job placement policies* in the context of KOALA) in order to optimize some criteria.

2.2 Parallel job management and co-allocation support in KOALA

The KOALA grid scheduler is capable of scheduling and co-allocating parallel jobs employing either the Message Passing Interface (MPI) or Ibis [12, 205] parallel communication libraries. In this chapter we only consider MPI jobs, which have to be compiled with the Open-MPI [83] library. Open-MPI, built upon the MPI-2 specification, allows KOALA to combine multiple clusters to run a single MPI application by automatically handling both inter-cluster and intra-cluster messaging.

The parallel jobs that may need co-allocation are handled in KOALA as follows. Upon

submission of a parallel job, the KOALA scheduler uses one of its job placement policies (see Section 2.3) to try to place job components on suitable execution sites, which requires the sites having enough idle processing nodes that can accommodate these job components. If the placement of the job succeeds and input files are required, the scheduler informs the runner, that is, the job submission tool, to initiate the third-party file transfers from the selected file sites to the execution sites of the job components. If a placement try fails, KOALA places the job at the tail of the placement queue, which holds all jobs that have not yet been successfully placed. The scheduler regularly scans the queue from head to tail to see whether it is able to place any job. For each job in the queue its number of placement tries is recorded, and when this number exceeds a certain threshold value, the submission of that job fails.

In order to realize co-allocation, KOALA uses an atomic-transaction approach [47] in which job placement only succeeds if all the components of a job can be placed at the same time. This necessitates the simultaneous availability of the desired numbers of idle nodes in multiple clusters. Since the local schedulers in the DAS-3, which are all SGE [87], do not support advance reservations, KOALA employs an on-spot node allocation mechanism. In order to allocate nodes for the job components, KOALA uses its Component Manager (KCM), which is a daemon process that runs on the head node of a cluster, and that interfaces the SGE through the DRMAA [59] interface.

For each component that is mapped to a cluster by the scheduler, the runner first launches a KCM on the head node of that cluster. The runner provides this KCM the number of nodes to claim, and for how long they should be claimed. Then, the KCM submits one or more placeholder scripts to the SGE [87] through the DRMAA interface. The SGE schedules the placeholder script(s) to a set of its nodes, and then each of these placeholder scripts reports back the *hostname* of its node to the KCM. Subsequently, the KCM compiles a list of the node hostnames it receives from the placeholder scripts and sends them to the runner. Finally, the runner uses the node hostnames, received from all KCMs, to launch the job on multiple clusters by using the corresponding OpenMPI command and parameters[1]. The runner maps two application processes per node, since all the clusters in our testbed comprise nodes of dual processors. Upon job completion, the runner gathers the results and presents them to the user.

2.3 Job placement policies

The KOALA job placement policies are used to decide where the components of non-fixed and flexible jobs should be sent for execution. In this section we present three job placement policies of KOALA, which are the Worst Fit, the Flexible Cluster Minimization, and

[1] An example on how to execute an OpenMPI application manually in the DAS-3 system is presented at http://www.cs.vu.nl/das3/openmpi-tcp.shtml

the Communication-Aware placement policy. Worst Fit is the default policy of KOALA which serves non-fixed job requests. Worst Fit also makes perfect sense in the absence of co-allocation, when all jobs consist of a single component. The two other policies, on the other hand, serve flexible job requests and only apply to the co-allocation case.

2.3.1 The Worst Fit policy

The Worst Fit (WF) policy aims to keep the load across clusters balanced. It orders the components of a job with a non-fixed request type according to decreasing size and places them in this order, one by one, on the cluster with the largest (remaining) number of idle processors, as long as this cluster has a sufficient number of idle processors. WF leaves in all clusters as much room as possible for later jobs, and hence, it may result in co-allocation even when all the components of the considered job would fit together on a single cluster.

2.3.2 The Flexible Cluster Minimization policy

The Flexible Cluster Minimization (FCM) policy is designed with the motivation of minimizing the number of clusters to be combined for a given parallel job in order to reduce the number of inter-cluster messages. FCM first orders the clusters according to decreasing number of idle processors and considers component placement in this order. Then FCM places on clusters one by one a component of the job of size equal to the number of idle processors in that cluster. This process continues until the total processor requirement of the job has been satisfied or the number of idle processors in the system has been exhausted, in which case the job placement fails (the job component placed on the last cluster used for it may be smaller than the number of idle processors of that cluster).

Figure 2.2 illustrates the operation of the WF and the FCM policies for a job of total size 24 in a system with 3 clusters, each of which has 16 idle processors. WF successively places the three components (assumed to be of size 8 each) of a non-fixed job request on the cluster that has the largest (remaining) number of available processors, which results in the placement of one component on each of the three clusters. On the other hand, FCM results in combining two clusters for a flexible job of the same total size (24), splitting the job into two components of sizes 16 and 8, respectively.

2.3.3 The Communication-Aware policy

The Communication-Aware (CA) placement policy takes either bandwidth or latency into account when deciding on co-allocation. The performance of parallel applications that need relatively large data transfers are more sensitive to bandwidth, while the performance of parallel applications which are dominated by inter-process communication are more

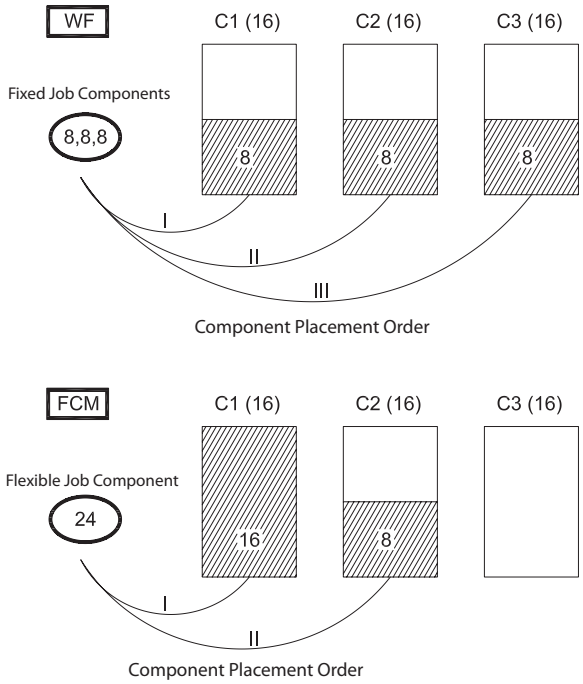


Figure 2.2: An example comparing the WF and the FCM placement policies.

sensitive to latency. In this thesis we only consider the latter case, and we run the CA policy with the latency option.

The latencies between the nodes of each pair of clusters in the system are kept in the information service of KOALA and are updated periodically. CA first orders the clusters according to increasing intra-cluster latency, and checks in this order whether the complete job can be placed in a single cluster. If this is not possible, CA computes for each cluster the average of all of its *inter*-cluster latencies, including its own *intra*-cluster latency, and orders the clusters according to increasing value of this average latency. As in the FCM policy, CA then splits up the job into components of sizes equal to the numbers of idle processors of the clusters in this order (again the last component of the job may not completely fill up the cluster on which it is placed).

In fact, the CA policy does not guarantee the best solution to the problem of attaining the smallest possible execution time for a co-allocated parallel application, since this problem is NP complete. However, it is a reasonable heuristic for small-scale systems. For larger systems, a clustering approach can be considered, in which clusters with low inter-cluster latencies are grouped together, and co-allocation is restricted to those groups separately.

2.4 The impact of system properties on co-allocation performance

In this section, we evaluate the impact of the inter-cluster communication characteristics and the processor speed heterogeneity of a multicluster system on the execution time performance of a single parallel application that runs on co-allocated processors.

2.4.1 The impact of inter-cluster communication

In a multicluster grid environment, it is likely that the inter-cluster communication is slower than the intra-cluster communication in terms of latency and bandwidth, which are the key factors that determine the communication performance of a network. This slowness, in fact, depends on various factors such as the interconnect technology that enables the inter-cluster communication among the processes of a parallel application, the distance between the clusters, the number and capabilities of the network devices, and even the network configuration. Therefore, depending on the communication requirements of a parallel application, the inter-cluster latency and bandwidth may have a big impact on its execution time performance.

In this section, we first present the results of experiments for measuring the communication characteristics of our testbed, and then we present the results of experiments for assessing the impact of inter-cluster communication on execution time performance.

With the DAS-3 system, we have the chance to compare the performance of the Myri-10G [157] and the Gigabit Ethernet (GbE, 1Gb/s) interconnect technologies. When the cluster in Delft is involved in the co-allocation of a parallel job, GbE is used for the entire inter-cluster communication, since it does not support the faster Myri-10G technology. For all other cluster combinations, for co-allocation Myri-10G is used, even though they all support GbE. Table 2.1 shows the average intra-cluster and inter-cluster bandwidth (in MB/s) and the average latency (in ms) as measured between the compute nodes of the DAS-3 clusters (the values are diagonally symmetric). These measurements were performed with an MPI ping-pong application that measures the average bi-directional bandwidth, sending messages of 1 MB, and the average bi-directional latency, sending messages of 64 KB, between two (co-)allocated nodes. The measurements were performed when the system was almost empty. With Myri-10G, the latency between the nodes is lower and the bandwidth is higher in comparison to the case with GbE. The measurements also indicate that the environment is heterogeneous in terms of communication characteristics even when the same interconnection technology is used. This is due to characteristics of the network structure such as the distance and the number of routers between the nodes. For example, the clusters Amsterdam and MultimediaN are located in the same building, and therefore, they achieve the best inter-cluster communication.

Table 2.1: The average bandwidth (in MB/s, top numbers) and latency (in ms, bottom numbers) between the nodes of the DAS-3 clusters (for Delft-Leiden see text).

Clusters	Vrije	Amsterdam	Delft	MultimediaN	Leiden
Vrije	561	185	45	185	77
	0.03	0.4	1.15	0.4	1.0
Amsterdam	185	526	53	512	115
	0.4	0.03	1.1	0.03	0.6
Delft	45	53	115	10	-
	1.15	1.1	0.05	1.45	-
MultimediaN	185	512	10	560	115
	0.4	0.03	1.45	0.03	0.6
Leiden	77	115	-	115	530
	1.0	0.6	-	0.6	0.03

We were not able to perform measurements between the clusters in Delft and Leiden due to a network configuration problem; hence, we excluded either the cluster in Delft or the cluster in Leiden in all of our experiments.

The synthetic parallel application that we use in our execution-time experiments performs one million *MPI_AllGather* all-to-all communication operations each with a message size of 10 KB. The job running this application has a total size of 32 nodes (64 processors), and we let it run with fixed job requests with components of equal size on all possible combinations of one to four clusters with the following restrictions. We either exclude the cluster in Delft and let the inter-cluster communication use the Myri-10G network, or we include the cluster in Delft, exclude the one in Leiden, and let the inter-cluster communication use GbE.

Figure 2.3 shows the execution time of the synthetic application averaged across all combinations of equal numbers of clusters. Clearly, the execution time increases with the increase of the number of clusters combined. However, the increase is much more severe, and the average execution time is much higher, when GbE is used—co-allocation with Myri-10G adds much less execution time overhead. These results indicate that the communication characteristics of the network are a crucial element in co-allocation, especially for communication-intensive parallel applications. However, the performance of co-allocation does not solely depend on this aspect for all types of parallel applications, as we will explain in the following section.

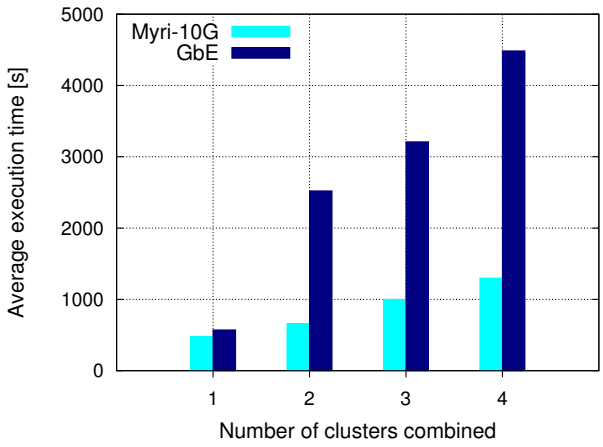


Figure 2.3: The execution time of a synthetic co-allocated MPI application, depending on the interconnect technology used and the number of clusters combined.

2.4.2 The impact of heterogeneous processor speeds

Unless an application developer does take into account processor speed heterogeneity and optimizes his applications accordingly, the execution time of a parallel application that runs on co-allocated clusters may be limited by the speed of the slowest processor, due to the synchronization of the processes. This is a major drawback of co-allocation especially for computation-intensive parallel applications which do not require intensive inter-cluster communications.

In order to investigate the impact of heterogeneous processor speeds on co-allocation performance, we have run a synthetic parallel application combining the cluster in Leiden (which has the fastest processors, see Table 1.2) with each of the other clusters in the DAS-3 system and quantified the increase in the execution time over running the application only in Leiden. The synthetic parallel application performs ten million floating point operations without any I/O operations and inter-process communications except the necessary MPI initialization and finalization calls. As the results in Table 2.2 indicate, there is a slight increase in the execution time ranging from 7% to 17% due to the minor level of processor speed heterogeneity in DAS-3. Therefore, in this study we do not consider the slowdown due to heterogeneous processor speeds in our policies. Nevertheless, the FCM policy can easily be enhanced such that it does consider the processor speeds when co-allocating in systems where this slowdown can be high.

Table 2.2: The execution time of a synthetic application when co-allocating the cluster in Leiden with each of the other clusters.

Cluster combination	Leiden	Leiden-Vrije	Leiden-Delft	Leiden-MultiMediaN	Leiden-Amsterdam
Execution time[s]	30	32	32	32	35
Percentage of increase	-	7%	7%	7%	17%

2.5 Co-Allocation versus no co-allocation

In this section, we investigate when co-allocation for parallel applications may be beneficial over disregarding co-allocation. In Section 2.5.1, we present the applications that we have used in our experiments. In Section 2.5.2, we present and discuss the results of the experiments conducted in the DAS-3 system. We have performed additional experiments in a simulated DAS-3 environment, in order to investigate the performance of co-allocation for a wide range of situations. We present and discuss the results of these simulation-based experiments in Section 2.5.3.

2.5.1 The applications

For the experiments, we distinguish between computation- and communication-intensive parallel applications. We have used three MPI applications, *Prime Number* [166], *Poisson* [148], and *Concurrent Wave* [78], which vary from computation-intensive to communication-intensive.

The Prime Number application finds all the prime numbers up to a given integer limit. In order to balance the load (large integers take more work), the odd integers are assigned cyclically to processes. The application exhibits embarrassing parallelism; collective communication methods are called only to reduce the data of the number of primes found, and the data of the largest prime number.

The Poisson application implements a parallel iterative algorithm to find a discrete approximation to the solution of a two-dimensional Poisson equation on the unit square. For discretization, a uniform grid of points in the unit square with a constant step in both directions is considered. The application uses a red-black Gauss-Seidel scheme, for which the grid is split up into “black” and “red” points, with every red point having only black neighbors and vice versa. The parallel implementation decomposes the grid into a two-dimensional pattern of rectangles of equal size among the participating processes. In each iteration, the value of the each grid point is updated as a function of its previous value and the values of its neighbors, and all points of one color are visited first followed by the ones of the other color.

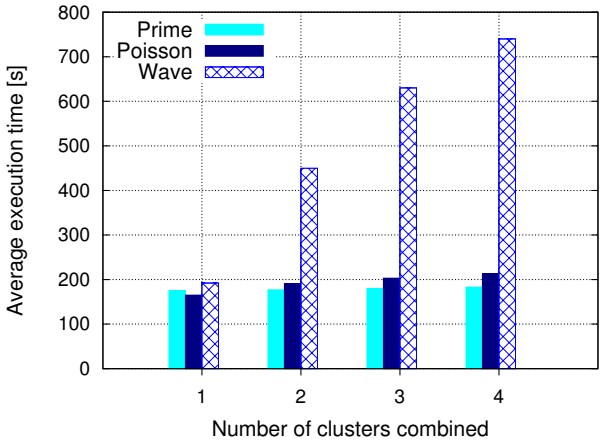


Figure 2.4: The average execution times of the applications depending on the number of clusters combined.

The Concurrent Wave application calculates the amplitude of points along a vibrating string over a specified number of time steps. The one-dimensional domain is decomposed by the master process, and then distributed as contiguous blocks of points to the worker processes. Each process initializes its points based on a sine function. Then, each process updates its block of points with the data obtained from its neighbor processes for the specified number of time steps. Finally, the master process collects the updated points from all the processes.

The runtimes of these applications in the DAS-3 are shown in Figure 2.4. Each application has been run several times on all combinations of clusters (excluding the cluster in Delft; the interconnect technology is Myri-10G) as fixed job requests with a total size of 32 nodes and components of equal size (except for the case of 3 clusters, in which we submit components of sizes 10-10-12 nodes), and the results have been averaged. The results demonstrate that as the Concurrent Wave application is a communication-intensive application, its execution time with multiple clusters increases markedly, from 200 seconds as a single cluster to 750 seconds when combining four clusters. The Poisson application suffers much less from the wide-area communication overhead, while the Prime Number application is not affected by it at all, since it is a computation-intensive parallel application.

2.5.2 Experiments in the real environment

In this section we present our experiments on the DAS-3. We first explain our experimental setup and then discuss the results.

Experimental setup

In our experiments, we use three workloads that each contain only one of the applications presented in Section 2.5.1. In the experiments in which no co-allocation is employed, the workloads are scheduled with the WF policy, and in the experiments in which co-allocation is used, the workloads are scheduled with the FCM policy (and all job requests are flexible).

We consider jobs with total sizes of 8, 16 and 32 nodes, so that the jobs can fit on any cluster in the system in case of no co-allocation; the total size of a job is randomly chosen from this considered set of total sizes. For every application, we have generated a workload with an average inter-arrival time determined in such a way that the workload is calculated to utilize approximately 40% of the system on average. The real (*observed*) utilization attained in the experiments depends on the policy being used, since the theoretical calculation of the utilization (i.e., the *net utilization*) is based on the average single-cluster execution times of the applications. When there is no co-allocation, there is no wide-area communication, and the real and the net utilizations coincide. The job arrival process is Poisson.

We use the tools provided within the GrenchMark project [99] to ensure the correct submission of our workloads to the system, and run each workload for 4 hours, under the policy in question. We have excluded the cluster in Delft, and the interconnect technology is Myri-10G.

In the DAS-3 system, we do not have control over the background load imposed on the system by other users. These users submit their (non-grid) jobs straight to the local resource managers, bypassing KOALA. During the experiments, we monitored this background load and we tried to maintain it between 10% and 30% across the system by injecting or killing dummy jobs to the system. We consider our experimental conditions no longer to be satisfied when the background load has exceeded 30% for more than 5 minutes. In such cases, the experiments were aborted and repeated.

In order to describe the performance metrics before presenting our results, we first discuss the timeline of a job submission in KOALA as shown in Figure 2.5. The time instant of the successful placement of a job is called its *placement time*. The *start time* of a job is the time instant when all components are ready to execute. The total time elapsed from the submission of a job until its start time is the *wait time* of a job. The time interval between the submission and the placement of a job shows the amount of time it spends in the placement queue, i.e., the *queue time*. The time interval between the placement time and the start time of a job is its *startup overhead*.

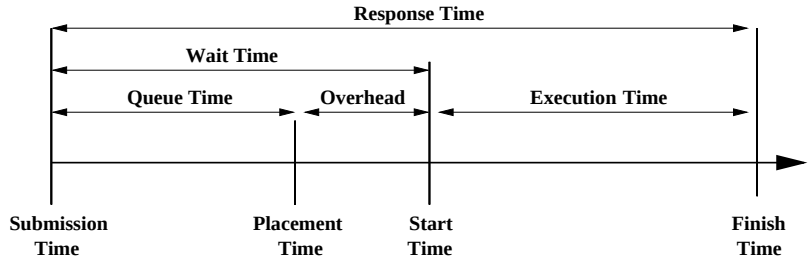
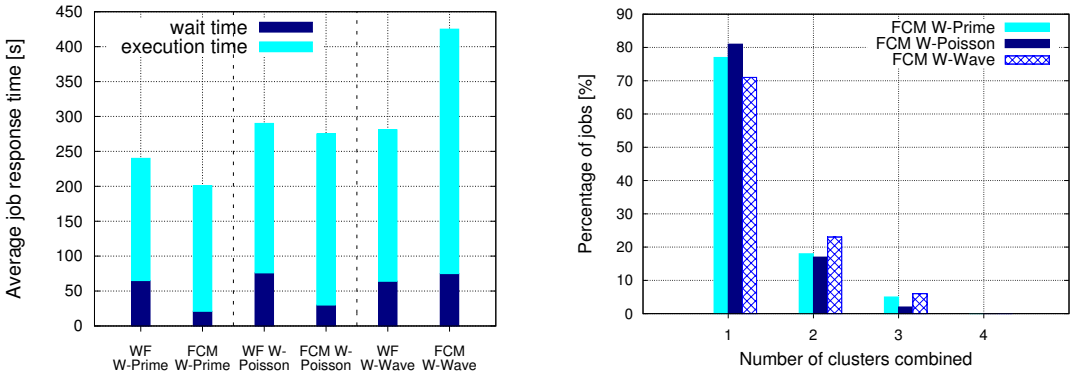


Figure 2.5: The timeline of a job submission in KOALA.



(a) (b)

Figure 2.6: Real Experiments: The average job response times (a), and the percentages of co-allocated jobs (b) of the workloads (denoted by *W-Application Name*).

Results

We will now present the results of our experiments for comparing the performance with and without co-allocation with the WF and FCM policies, respectively, for workloads of real MPI applications. Figure 2.6(a) shows the average job response time broken down into the wait time and the execution time for the workloads of all three applications, and Figure 2.6(b) shows the percentages of co-allocated jobs.

First of all, we have observed in our experiments that the startup overhead of jobs is 10 seconds on average regardless of the number of clusters combined for it, and hence, from the values of the wait time shown in Figure 2.6(a), we conclude that the wait time is dominated by the queue time. Compared to what we have observed with Globus DUROC [89] and the MPICH-G2 [155] library for co-allocation [187], the DRMAA-SGE [59] interface and the Open-MPI [83] library for co-allocation yield a much lower startup overhead, by a factor of 5 on average.

Figure 2.6(a) indicates that for the workloads of the Prime and Poisson applications, the average job response time is lower when the workloads are scheduled with FCM compared to when they are scheduled with WF; however, the average job response time is

higher for the workload of the Wave application with FCM. The FCM policy potentially decreases the job wait times since it is allowed to split up jobs in any way it likes across the clusters. Given that the execution times of the Prime Number and Poisson applications only slightly increase with co-allocation, the substantial reduction in wait time results in a lower average job response time.

For the Wave application, co-allocation severely increases the execution time due to high inter-cluster communication. As a consequence, the observed utilization also increases, causing higher wait times. Together, this leads to higher response times. As Figure 2.6(b) indicates, a relatively small fraction ($<$30%) of co-allocation is responsible for the aforementioned differences in the average job response times between no co-allocation and co-allocation.

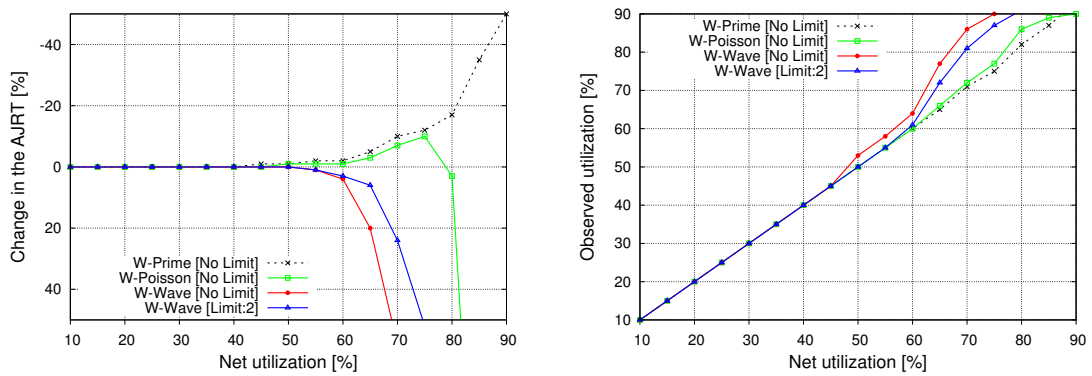
We conclude that in case of moderate resource contention (i.e., 40% workload + 10-30% background load), co-allocation is beneficial for computation-intensive parallel applications (e.g., Prime) and for communication-intensive applications whose slowdown due to the inter-cluster communication is low (e.g., Poisson). However, for very communication-intensive parallel applications (e.g., Wave), co-allocation is disadvantageous due to the severe increase in the execution time. In the next section, we further evaluate the performance of no co-allocation vs. co-allocation under various workload utilization levels using simulations.

2.5.3 Experiments in the simulated environment

In this section, as in the previous section, we first explain the experimental setup and then present and discuss the results of our simulations.

Experimental setup

We have used the DGSim grid simulator [104] for our simulation-based experiments. We have modeled the KOALA grid scheduler with its job placement policies, the DAS-3 environment, and the three MPI applications based on their real execution times in single clusters and in combinations of clusters. We have also modeled a synthetic application whose communication-to-computation ratio (CCR) can be modified. We define the CCR value for a parallel application as the ratio of its total communication time to its total computation time, when executed in a *single cluster*. We set the total execution time of the application to 180 s. in a single cluster irrespective of its CCR. For instance, for a CCR value of 1.0, both the communication and the computation part of the application take 90 s; for a CCR value of 0.5, these values are 60 s., and 120 s. When the application runs on co-allocated clusters, the communication part is multiplied by a specific factor that is calculated from the real runs of the synthetic application on the corresponding co-allocated clusters, and the total execution time of the application increases accordingly.



(a) (b)

Figure 2.7: Simulation Experiments: Percentage of change in the average job response time in reverse scale (a) for the workloads of the MPI applications when they are scheduled with FCM in comparison to when they are scheduled with WF, and the observed utilization versus the net utilization (b) when the workloads are scheduled with FCM.

As in Section 2.5.2, we use workloads that each contain only one of MPI or the synthetic applications. In the experiments in which no co-allocation is employed, the workloads are scheduled with the WF policy, and in the experiments in which co-allocation is used, the workloads are scheduled with the FCM policy. For the workloads of the Wave application, we also consider the case in which FCM is limited to combine two clusters at most.

We consider jobs with total sizes of 8, 16 and 32 nodes, so that the jobs can fit on any cluster in the system in case of no co-allocation; the total size of a job is randomly chosen from this set of considered total sizes. For every application, we have generated seventeen workloads with net utilizations ranging from 10% to 90% in steps of 5%. The job arrival process is Poisson. We assume that there is no background load in the system. Each workload runs for 24 simulated hours, under the policy in question, and we have again excluded the cluster in Delft.

Results

Figure 2.7(a) shows the percentage of change in the average job response time (AJRT) for the workloads of the MPI applications when they are scheduled with FCM in comparison to when they are scheduled with WF. Figure 2.7(b) illustrates the observed utilization vs. the net utilization for the same workloads when they are scheduled with FCM. In Table 2.3, for each policy-workload pair, we present the net utilization interval in which saturation sets in and jobs are stacked in the queue and the wait times constantly increase without bounds.

When the resource contention is relatively low (up to 40%), with the job sizes included

in the workloads, most jobs are placed in single clusters without a need for co-allocation, hence we observe no difference in the average job response times. For the computation-intensive Prime application, the performance benefit of co-allocation increases with the increase of the contention in the system, since jobs have to wait longer in the placement queue in case of no co-allocation. In addition, as Table 2.3 shows the workload of the Prime application causes saturation at lower utilizations when co-allocation is not considered; the saturation point is in between 85-90% net utilization for WF W-Prime, and between 90-95% for FCM W-Prime.

We observe that for the Poisson application, co-allocation is advantageous up to 75% net utilization, since the lower wait times compensate for the increase of the execution times. However, beyond this level, saturation sets in and consequently, the average job response times increase.

For the Wave application, the extreme execution time increase of the jobs with co-allocation increases the observed utilization in the system as shown in Figure 2.7(b), which as a result, causes an early saturation (see also Table 2.3). In addition, we see that limiting co-allocation to two clusters yields a better response time performance than in case of no limit. However, the benefit is minor.

In order to compare real and simulation experiments, in Table 2.4 we present the net utilizations imposed by the workloads in the real and the simulation experiments where the percentages of change in the average job response times match. It turns out that the net utilization in the real experiments is lower than the net utilization in the corresponding simulation experiments, which is probably due to the background load in the real experiments having different characteristics than the workloads of MPI applications.

Figure 2.8 shows the change in the average job response time for the workloads of the synthetic application with various CCR values. Comparing the results to those of the real MPI applications, we see that W-Prime matches CCR-0.1, W-Poisson matches CCR-0.25, and W-Wave matches CCR-4. The results with the workloads of the synthetic application exhibit the following. First, parallel applications with very low CCR values (i.e., 0.10) always benefit from co-allocation. Secondly, for applications with CCR values between 0.25 and 0.50, co-allocation is beneficial to a certain extent; with the increase of the contention in the system, the performance benefit of co-allocation decreases and after some point it becomes disadvantageous. Finally, for applications with CCR values higher than 0.50, co-allocation is disadvantageous since it increases the job response times severely.

2.6 The performance of the placement policies

Although we have observed that it would be really advantageous to schedule communication-intensive applications on a single cluster from the perspective of the exe-

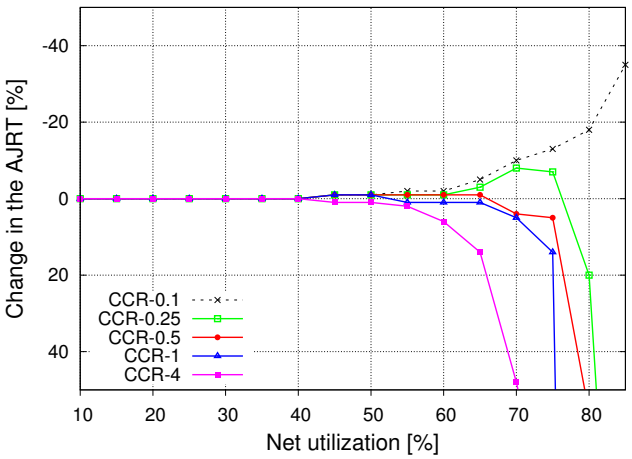


Figure 2.8: Simulation Experiments: Percentage of change in the average job response time for the workloads of the synthetic application (with different CCR values) when they are scheduled with FCM in comparison to when they are scheduled with WF.

Table 2.3: The net utilization intervals in which the policy-workload pairs induce saturation.

Policy-Workload	Utilization Interval
WF {W-Prime, W-Poisson, W-Wave}	85-90%
FCM W-Prime	90-95%
FCM W-Poisson	80-85%
FCM W-Wave [No Limit]	70-75%
FCM W-Wave [Limit:2]	75-80%

cution time (see in Figure 2.4), users may still prefer co-allocation when more processors are needed than available on a single cluster. In this section, we compare the FCM and CA policies in order to investigate their co-allocation performance for communication-intensive parallel applications.

2.6.1 Experiments in the real environment

In this section we present our experiments in the DAS-3. We first explain our experimental setup and then discuss the results.

Experimental setup

In our experiments in this section, we use workloads comprising only the Concurrent Wave application [78], with a total job size of 64 nodes (128 processors). We have gen-

Table 2.4: The net utilizations in the real and the simulation experiments where the changes in AJRTs match.

Workload	Change in the AJRT	Net Utilization (in Real Experiments)	Net Utilization (in Simulations)
W-Prime	-16%	40% + BG Load	75-80%
W-Poisson	-5%	40% + BG Load	65%
W-Wave	+50%	40% + BG Load	70%

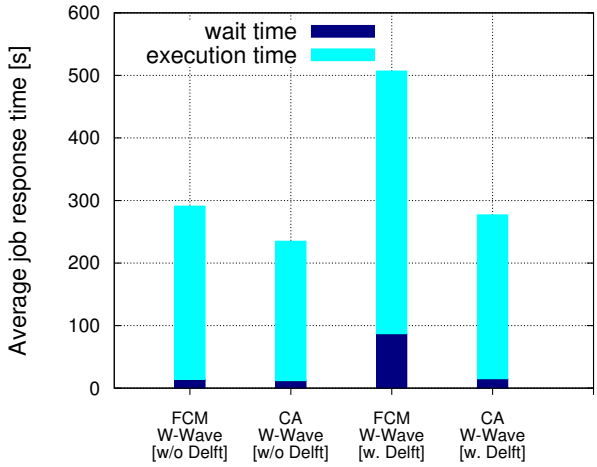


Figure 2.9: Real Experiments: Performance comparison of the FCM and CA policies.

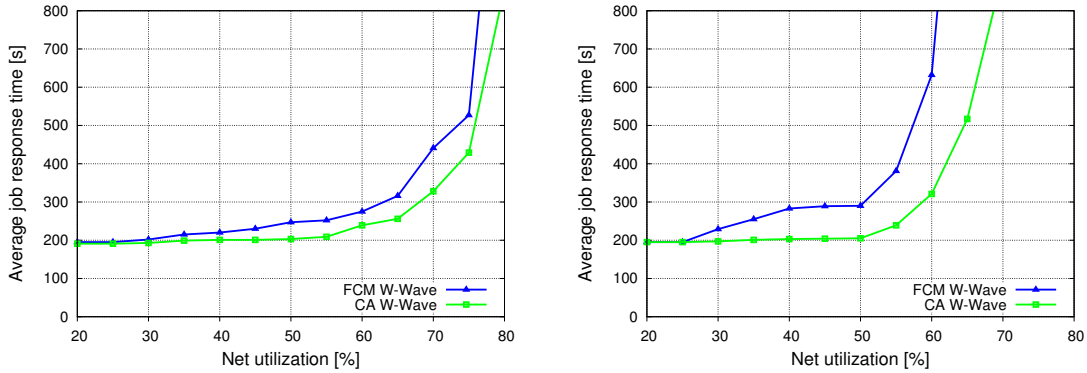
erated a workload with an average inter-arrival time determined in such a way that the workload is calculated to utilize approximately 40% of the system on average. The job arrival process is Poisson.

We handle the background load in the way mentioned in Section 2.5.2. We run the workload for 4 hours, under the policy in question. In the first set of experiments, we have excluded the cluster in Delft, and in the second set of experiments we have excluded the cluster in Leiden and included the one in Delft; the interconnect technology used by a job is GbE when the cluster in Delft is involved in its co-allocation, and Myri-10G otherwise.

Results

Figure 2.9 shows the performance of the FCM and CA policies when scheduling the workload of the Wave application on the sets of clusters without and with the one in Delft.

In terms of the average job response time, the CA policy outperforms the FCM policy, irrespective of the involvement of the cluster in Delft, which has a slow inter-cluster communication speed. The difference in response time is moderate (50 s.) or major (230



(a) Cluster in Delft excluded (b) Cluster in Delft included

Figure 2.10: Simulation Experiments: Performance comparison of the FCM and CA policies.

s.) depending on whether the cluster in Delft is excluded (communication speed has a low variability across the system) or included in the experiments (communication speed has a high variability across the system), respectively.

The CA policy tries to combine clusters that have faster inter-cluster communication (e.g., the clusters in Amsterdam and MultimediaN). However, as it is insensitive to communication speeds, the FCM policy may combine clusters with slower inter-cluster communication, which consequently increases the job response times. The increase is more severe when the cluster in Delft is included in the experiments, since it is involved in many of the co-allocations for the jobs due to its large size.

We conclude that considering inter-cluster latency in scheduling communication-intensive parallel applications that require co-allocation is useful, especially when the communication speed has a high variability across the system. In the following section, we extend our findings in the real environment by evaluating the performance of the FCM and CA policies under various resource contention levels in a simulated DAS-3 environment.

2.6.2 Experiments in the simulated environment

In this section, again, we first explain the experimental setup and then present and discuss the results of our simulations.

Experimental setup

In our simulations, we use workloads comprising only the Concurrent Wave application, with total job sizes of 32, 48, and 64 nodes; the total size of a job is randomly chosen

from this set. We have generated thirteen workloads with net utilizations ranging from 20% to 80% in steps of 5%. The job arrival process is Poisson. We assume that there is no background load in the system. Each workload runs for 24 simulated hours, under the policy in question.

In the first set of experiments, we have excluded the cluster in Delft, and in the second set of experiments we have included the cluster in Delft and excluded the one in Leiden.

Results

Figure 2.10(a) and 2.10(b) illustrate the average job response time results of the FCM and CA policies scheduling the workloads of the Wave application on the set of clusters either excluding the one in Delft or including it, respectively.

The CA policy outperforms the FCM policy for almost all utilization levels in both sets of experiments. As the utilization increases, the gap between the results of the two policies becomes wider. When the cluster in Delft is excluded, the system is saturated between 75-80% net utilization level; however, when it is included, the system is saturated between 60-70% net utilization, which is much less. The reason is that co-allocating the cluster in Delft increases the job response times more severely.

We also see that the simulation results are consistent with the real experiments as the difference in the performance of the two policies is much larger when the cluster in Delft is included than when the cluster in Delft is excluded. This fact supports our claim that taking into account inter-cluster communication speeds improves the performance, especially when the communication speed has a high variability across the system.

To conclude, the results provide evidence that we should omit clusters that have slow inter-cluster communication speeds when co-allocation is needed. In other words, in large systems we should group clusters with similar inter-cluster communication speeds, and restrict co-allocation to those groups separately.

2.7 Challenges with supporting co-allocation

Although we have demonstrated a case for supporting co-allocation in a real environment with our KOALA grid scheduler, there are still many issues to be considered before processor co-allocation may become a widely used phenomenon in multicluster grids and grid schedulers. In this section, we discuss some of these issues, related to communication libraries, processor reservations and system reliability.

2.7.1 Communication libraries

There are various communication libraries available [83, 153–155, 203] that enable co-allocation of parallel applications. However, all these libraries have their own advantages and disadvantages; there is no single library we can name as the most suitable for co-allocation. Some include methods for optimizing inter-cluster communication, some include automatic firewall and NAT traversal capabilities, and some may depend on other underlying libraries. Therefore, it is important to support several communication libraries as we do with the KOALA grid scheduler (e.g., MPICH-G2 [155], OpenMPI [83], and Ibis [203, 205]).

2.7.2 Advance processor reservations

The challenge with simultaneous access to processors in multiple clusters of a grid lies in guaranteeing their availability at the start time of an application. The simplest strategy is to reserve processors at each of the selected clusters. If the local schedulers of the clusters do support advance reservations, this strategy can be implemented by having a grid scheduler obtain a list of time slots from each LRM, reserve a common time slot for all job components, and notify the LRMs of this reservation. Unfortunately, a reservation-based strategy in grids is currently limited due to the fact that only few local schedulers support advance reservations (e.g., PBS-pro [165], Maui [140]). In the absence of an advance reservation mechanism good alternatives are required, such as the mechanism explained in Section 2.2, in order to achieve co-allocation.

2.7.3 System reliability

The single most important distinguishing feature of grids as compared to traditional parallel and distributed systems is their multi-organizational character, which causes forms of heterogeneity in the hardware and software across the resources. This heterogeneity, in turn, makes failures appear much more often in grids than in traditional distributed systems. In addition, grid schedulers or resource management systems do not actually own the resources they try to manage, but rather, they interface to multiple instances of local schedulers in separate clusters who are autonomous and who have different management architectures, which makes the resource management a difficult challenge.

We have experienced in our work on KOALA that even only configuring sets of processors in different administrative domains in a cooperative research environment is not a trivial task. Due to incorrect configuration of some of the nodes, during almost all our experiments, hardware failed and jobs were inadvertently aborted. To accomplish the experiments that we have presented in this study, we have spent more than half a year and we have submitted more than 15,000 jobs to get reliable results. We claim that co-allocation

in large-scale dynamic systems such as grids requires good methods for configuration management as well as good fault tolerance mechanisms.

2.8 Related work

Various advance reservation mechanisms and protocols for supporting processor co-allocation in grid systems have been proposed in the literature [9, 40, 131, 168, 174, 184]. Performance studies on co-allocation, however, mostly studied in simulated environments; only a few studies investigate the problem in real systems. In this section, we discuss some of the studies that we find most related to our work.

Bucur et al. [25–27] study through simulations processor co-allocation in multiclusters with space sharing of rigid jobs for a wide range of such parameters as the number and sizes of the job components, the number of clusters, the service-time distribution, and the number of queues in the system. Parallel to our results, they find that co-allocation is beneficial as long as the number and sizes of job components, and the slowdown of applications due to the wide-area communication, are limited.

Ernemann et al. [66] present an adaptive co-allocation algorithm that uses a simple decision rule to decide whether it pays to use co-allocation for a job, considering the given parameters such as the requested run time and the requested number of resources. The slow wide-area communication is taken into account by a parameter by which the total execution time of a job is multiplied. In a simulation environment, co-allocation is compared to keeping jobs local and compared to only sharing load among the clusters, assuming that all jobs fit in a single cluster. One of the most important findings is that when the application slowdown does not exceed 1.25, it pays to use co-allocation.

Röblitz et al. [171, 172] present an algorithm for reserving compute resources that allows users to define an optimization policy if multiple candidates match the specified requirements. An optimization policy based on a list of selection criteria, such as end time and cost, ordered by decreasing importance, is tested in a simulation environment. For the reservation, users can specify the earliest start time, the latest end time, the duration, and the number of processors. The algorithm adjusts the requested duration to the actual processor types and numbers by scaling it according to the speedup, which is defined using speedup models or using a database containing reference values. This algorithm supports so-called fuzziness in the duration, the start time, the number of processors, and the site to be chosen, which leads to a larger solution space.

Jones et al. [111] present several bandwidth-aware co-allocation meta-schedulers for multicluster grids. These schedulers consider network utilization to alleviate the slowdown associated with the communication of co-allocated jobs. For each job modeled, its computation time and average per-processor bandwidth requirement is assumed to be known. In addition, all jobs are assumed to perform all-to-all global communication

periodically. Several scheduling approaches are compared in a simulation environment consisting of clusters with globally homogeneous processors. The most significant result is that co-allocating jobs when it is possible to allocate a large fraction (85%) of on a single cluster, provides the best performance in alleviating the slowdown impact due to inter-cluster communication.

The Grid Application Development Software (GrADS) [50,201] enables co-allocation of grid resources for parallel applications that may have significant inter-process communication. For a given application, during resource selection, GrADS first tries to reduce the number of workstations to be considered according to their availabilities, computational and memory capacities, network bandwidth and latency information. Then, among all possible scheduling solutions the one that gives the minimum estimated execution time is chosen for the application. Different from our work, GrADS assumes that the performance model of the applications and mapping strategies are already available or can be easily created. While Dail et al. present the superiority of the approach within GrADS over user-directed strategies, we handle the co-allocation problem for various cases, and present a more in-depth analysis.

In addition to the benefit of co-allocation from a system's or users' point of view, various study also addresses the performance of a single co-allocated parallel application [121, 164, 204]. A study by Seinstra et al. [176] present a work on the co-allocation performance of a parallel application that performs the task of visual object recognition by distributing video frames across co-allocated nodes of a large-scale grid system, which comprises clusters in Europe and Australia. The application has been implemented using the Parallel-Horus [177] tool, which allows researchers in multimedia content analysis to implement high-performance applications. The experimental results show the benefit of co-allocation for such multimedia applications that require intensive computation and frequent data distribution.

2.9 Summary

In this chapter we have investigated the benefit of processor co-allocation in a real multicluster grid system, DAS-3, using our KOALA grid scheduler as well as in a simulated environment using our DGSim tool. Initially, we have assessed the impact of inter-cluster communication characteristics of a multicluster system on the execution time performance of a single co-allocated parallel application. Then, we have evaluated the co-allocation performance of a set of parallel applications that range from computation- to communication-intensive, under various utilization conditions. Finally, we have evaluated two scheduling policies for co-allocating communication-intensive applications. We conclude the following.

First, the execution time of a single parallel application increases with the increase

of the number of clusters combined. This increase depends very much on the communication characteristics of the application, and on the inter-cluster communication characteristics and the processor speed heterogeneity of the combined clusters. Secondly, for computation-intensive parallel applications, co-allocation is very advantageous provided that the differences between the processor speeds across the system are small. For parallel applications whose slowdown due to the inter-cluster communication is low, co-allocation is advantageous when the resource contention in the system is moderate. However, for very communication-intensive parallel applications, co-allocation is disadvantageous since it increases execution times too much. Finally, in systems with a high variability in inter-cluster communication speeds, taking network metrics (in our case the latency) into account in cluster selection increases the performance of co-allocation for communication-intensive parallel applications.

Although there is a large opportunity for many scientific parallel applications to benefit from co-allocation, there are still many issues that need to be overcome before co-allocation can become a widely employed solution in future multicluster grid systems. The difference between inter- and intra-cluster communication speeds, efficient communication libraries, advance processor reservations, and system reliability are some of these challenges.

Chapter 3

Malleability for parallel applications

In multicluster grids, resource availability may vary because of resource failures, because resources may be allocated or released by concurrent users, and because organizations may add or withdraw parts of their resources to/from the resource pool at any time. In any of these cases, application malleability, that is, the property of applications to deal with a varying amount of resources during their execution, allows applications to benefit from appearing available resources, while gracefully releasing resources that are reclaimed by the environment. Allowing resource allocation to vary during execution, malleability gives a scheduler the opportunity to revise its decisions even after applications have started executing. Increasing the flexibility of applications by shrinking their resource allocations, malleability allows new applications to start sooner, possibly with resources that are not going to be usable during their whole execution. Making applications able to benefit from the resources that appear during their execution by growing their resource allocations, malleability also helps applications terminate sooner. In addition to these very general advantages, malleability makes it easier to deal with the dynamic nature of large-scale distributed execution environments such as multicluster grid systems. Several approaches have been proposed to build parallel malleable applications [28,114,198,199], and to schedule them in traditional parallel systems [95, 96, 141, 151]. However, previous work does not address how to build mechanisms to schedule parallel malleable applications in the context of a multicluster grid scheduler.

In this chapter we present an architecture and an actual implementation of the support for malleability in our KOALA grid scheduler with the help of the DYNACO framework [28, 30] for application malleability. We propose two policies to manage malleability in the scheduler for malleable jobs that have already been running in the system; one which hands out any additional processors to the malleable jobs that have been running the longest, and one that spreads them equally over all malleable jobs. Each of these policies can be coupled with one of two approaches which either favor running or queued malleable jobs when additional resources become available. Then, we evaluate these

policies and approaches in combination with the worst fit load-sharing scheduling policy of KOALA with experiments in the DAS-3 environment. These experiments show that a higher utilization and shorter job execution times can be achieved when malleability is used.

The rest of this chapter is organized as follows. Section 3.1 presents the aspects of supporting malleability to be considered by a multicluster grid scheduler. Section 3.2 describes how we support malleability in KOALA, and details the malleability management approaches and policies that we propose. Section 3.3 presents the experimental setup, and Section 3.4 discusses our experimental results. Section 3.5 reviews related work on scheduling malleable applications. Finally, Section 3.6 summarizes the chapter.

3.1 Aspects of supporting malleability

In this section we state the aspects of supporting malleability that should be taken into account by a grid scheduler. The aspects that we consider are specification of malleable jobs, initiative of change, and obligation to change, respectively.

3.1.1 Specification of malleable jobs

A malleable job may specify the *minimum* and *maximum* number of processors it requires. The minimum value is the minimum number of processors a malleable job needs to be able to run; the job cannot shrink below this value. The maximum value is the maximum number of processors a malleable job can handle; allocating more than the maximum value would just waste processors. We do not assume that a *stepsize* indicating the number of processors by which a malleable application can grow or shrink is defined. We leave the determination of the amount of growing and shrinking to the protocol between the scheduler and the application (see Section 3.2).

3.1.2 Initiative of change

Another aspect that we consider is the party that takes the initiative of changing the size of a malleable job (shrinking or growing). Either the application or the scheduler may initiate grow or shrink requests. An application may do so when the computation it is performing calls for it. For example, a computation can be in need of more processors before it can continue. On the other hand, the scheduler may decide that a malleable job has to shrink or grow based on the availability of free processors in the system. For example, the arrival of new jobs to a system that is heavily loaded may trigger a scheduler to requests currently running malleable jobs to shrink.

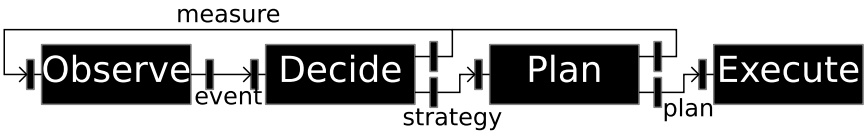


Figure 3.1: Overview of the architecture of the DYNACO framework for adaptability.

3.1.3 The obligation to change

Requests for changing the size of a malleable job may or may not have to be satisfied. A *voluntary* change means that the change does not have to succeed or does not necessarily have to be executed; it is merely a guideline. A *mandatory* change, however, has to be accommodated, because either the application cannot proceed without the change, or because the system is in direct need of the reclaimed processors.

3.2 Designing support for malleability in KOALA

In this section, we present our design for supporting malleable applications in KOALA. First, we describe the DYNACO framework that we use to implement malleable applications. Then, we explain how we include the DYNACO framework into the KOALA multicluster scheduler, and finally, we present our approaches and policies for managing the execution of malleable applications, respectively.

3.2.1 The DYNACO framework and its use for malleability

DYNACO [28][1] is a generic framework for building dynamically adaptable applications. As its architecture shows in Figure 3.1, DYNACO decomposes adaptability into four components, similarly to the control loop suggested in [120]. The *observe* component monitors the execution environment in order to detect any relevant change; relying on this information, the *decide* component makes the decision about adaptability. It decides when the application should adapt itself and which strategy should be adopted. When the strategy in use has to be changed, the *plan* component plans how to make the application adopt the new strategy; finally, the *execute* component schedules actions listed in the plan, taking into account the synchronization with the application code. Being a framework, DYNACO is expected to be specialized for each application. In particular, developers must provide the decision procedure, the description of planning problems, and the implementation of adaptation actions.

[1] DYNACO is available at the following website: http://dynaco.gforge.inria.fr

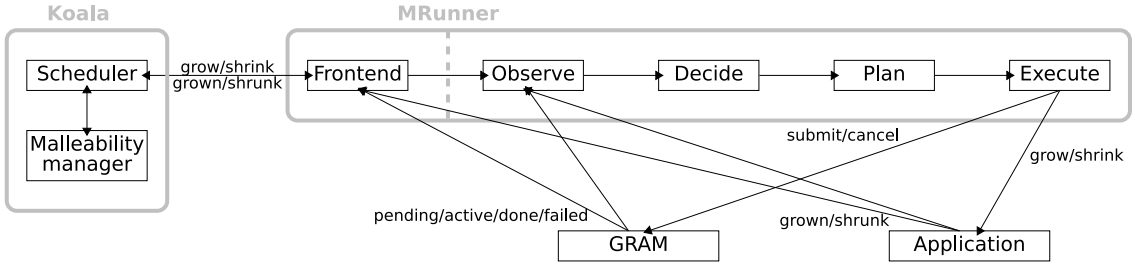


Figure 3.2: The architecture of the Malleable Runner with DYNACO in the KOALA multicluster scheduler.

In addition, developers of the DYNACO framework proposed AFPAC [29] as an implementation of the *execute* component that is specific to SPMD applications. As reported in [28], DYNACO and AFPAC have been successfully used to make several existing MPI-based applications malleable. While not being restricted to this class of applications, DYNACO contributes to reduce the cost of transforming existing parallel applications into malleable ones when it is combined with tools such as AFPAC.

3.2.2 Supporting DYNACO applications in KOALA

In order to support DYNACO-based applications in KOALA, we have designed a specific runner called the Malleable Runner (MRunner); its architecture is shown in Figure 3.2. In the MRunner, the usual control role of the runner over the application is extended in order to handle malleability operations. For that purpose a complete instance of DYNACO is included in the MRunner on a per-application basis. A frontend, which is common to all of the runners, interfaces the MRunner to the scheduler. We add a malleability manager in the scheduler, which is responsible for triggering changes of resource allocations.

In the DYNACO framework, the frontend is reflected as a monitor, which generates events when it receives *grow* and *shrink* messages from the scheduler. Resulting events are propagated throughout the DYNACO framework and translated into the appropriate messages to Globus GRAM [89], which is the job submission daemon that we use in this study, and to the application. The frontend catches the results of adaptations in order to generate acknowledgments back to the scheduler. It also notifies the scheduler when the application voluntarily shrinks below the amount of allocated processors. Since GRAM is not able to manage malleable jobs as discussed in [30], the MRunner manages the malleable job as a collection of GRAM jobs of size 1.

Upon growth, the MRunner submits new jobs to GRAM. When it receives *active* messages from GRAM, it transmits the new collection of active GRAM jobs (i.e. the collection of held resources) to the application. In order to reduce the impact on the execution time, interactions with GRAM overlap with the execution of the application and

suspension of the application does not occur before all the resources are held. To do so, GRAM submissions launch an empty stub rather than the application's program. The stub is turned into an application process during the process management phase, when resources are recruited by the application. That latter operation is faster than submitting a job to GRAM as it is relieved from tasks such as security enforcement and queue management. Conversely, upon shrink, the MRunner first reclaims processors from the application; then when it receives *shrunk* feedback messages, it releases the corresponding GRAM jobs. Again, interactions with GRAM overlap the execution, which resumes immediately.

3.2.3 Job management

In this study we assume that every malleable job is executed in a single cluster, and so, no co-allocation takes place. Therefore, contrary to our parallel job model described in Section 2.1, all the components of a job are scheduled on a single cluster.

Upon submission of a parallel job to KOALA, whether it is rigid or malleable, the initial placement is performed by one of the existing placement policies as described in Section 2.2. In the placement phase of malleable jobs, the initial number of processors required is determined considering the number of available processors in the system. Specifically, given a malleable job, the placement policies place it if the number of available processors is at least equal to the minimum processor requirement of the application.

In the job management context, the malleability manager is responsible for initiating malleability management policies that decide on how to grow or shrink malleable applications. Below, we propose two design choices as to when to initiate malleable management policies, which give Precedence to Running Applications over waiting ones (PRA) or vice versa (PWA), respectively.

In the PRA approach, whenever processors become available, for instance, when a job finishes execution, first the running applications are considered. If there are malleable jobs running, one of the malleability management policies is initiated in order to grow them; any waiting malleable jobs are not considered as long as at least one running malleable job can still be grown. In this approach malleable jobs are never shrunk unless a priority setting is considered between rigid and malleable jobs.

In PWA approach, when the next job in the queue cannot be placed, the scheduler applies one of the malleability management policies for shrinking running malleable jobs in order to obtain additional processors. Those shrink operations are mandatory. If it is however impossible to get enough available processors in order to place that queued job, taking into account the minimum sizes of the running jobs, then the running malleable jobs are considered for growing by one of the malleable management policies. Whenever processors become available, the placement queue is scanned in order to find a job to be placed.

In both approaches, in order to trigger job management, the scheduler periodically polls the KOALA information service. In doing so, the scheduler is able to take into account dynamically the background load due to other users even if they bypass KOALA. In addition, in order not to stress execution sites when growing malleable jobs, and therefore, in order to leave always a minimal number of available processors to local users, a threshold is set over which KOALA never expands the total set of the jobs it manages.

3.2.4 Malleability management policies

The malleability management policies, which we will describe below, determine the means of shrinking and growing of malleable jobs during their execution. Since each malleable job is executed in a single cluster the policies are applied for each cluster separately.

Favor Previously Started Malleable Applications (FPSMA)

The FPSMA policy favors previously started malleable jobs whenever the policy is initiated by the malleability manager. FPSMA starts growing from the earliest started malleable job and starts shrinking from the latest started malleable job. In the *grow* procedure, first, malleable jobs running on the considered cluster sorted in the increasing order of their start time, then the value of the number of processors to be allocated on behalf of malleable jobs is offered to the subsequent job in the sorted list. In reply to this offer (the job itself considers its maximum number of processors requirement), a desired number of processors are allocated on behalf of that job. Then, the policy updates the number of idle processors and continues to offer processors to the subsequent jobs in the list. The *shrink* procedure runs in a similar fashion; the differences with the *grow* procedure are that the jobs are sorted in the decreasing order of their start time, and rather than allocation, the compromised number of processors are waited to be released.

Equi-Grow & Shrink (EGS)

The EGS policy attempts to balance processors over malleable jobs. When it is initiated by the malleability manager, it distributes available processors (or reclaims needed processors) equally over all of the running malleable jobs. In case the number of processors to be distributed or reclaimed is not divisible by the number of running malleable jobs, the remainder is distributed across the least recently started jobs, or reclaimed from the most recently started jobs, respectively.

The EGS policy is similar to the well-known equipartition policy [141], which has originally been proposed as a dynamic processor allocation scheme for malleable parallel applications running in traditional parallel systems. The two policies, however, differ

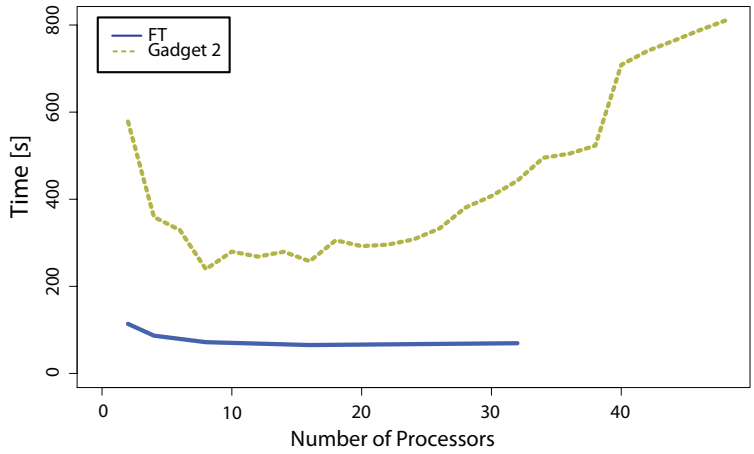


Figure 3.3: The execution times of the two malleable applications depending on the number of used processors.

in the following points. While our EGS policy distributes equally available processors among running jobs, the equipartition policy distributes equally the whole set of processors among running jobs. Consequently, EGS is not expected to make at each time all of the malleable jobs have the same size, while equipartition does. But equipartition may combine grow and shrink messages, while EGS consistently either grows or shrinks all of the running malleable jobs.

3.3 Experimental setup

In this section we describe the setup of the experiments that we have conducted in DAS-3 in order to evaluate the support and the scheduling policies for malleable jobs in KOALA. First, we will present the malleable applications, and then we will describe the details of the workloads that we have used in our experiments.

3.3.1 Malleable applications

For the experiments, we rely on two applications that have been made malleable with DYNACO. These applications are the NAS Parallel Benchmark FT [202], which is a benchmark for parallel machines based on a fast Fourier transform numerical kernel, and GADGET [191], which is a legacy n-body simulator. Further details on how these applications were made malleable can be found in [28]. Figure 3.3 shows how the execution times of the two applications scale with respect to the number of processors on the Delft cluster (see table 1.2). With two processors, GADGET takes ten minutes, while FT lasts two

minutes. The best execution times are respectively four minutes for GADGET and one minute for FT.

While GADGET can execute with an arbitrary number of processors, FT only accepts powers of two. As we have already stated, we propose that the scheduler does not care about such constraints, in order to avoid to make it implement an exhaustive collection of possible constraints. Consequently, when responding to grow and shrink messages, the FT application accepts only the highest power of two processors that does not exceed the allocated number. Additional processors are voluntarily released to the scheduler. In addition, the FT application assumes processors of equal compute power, while GADGET includes a load-balancing mechanism.

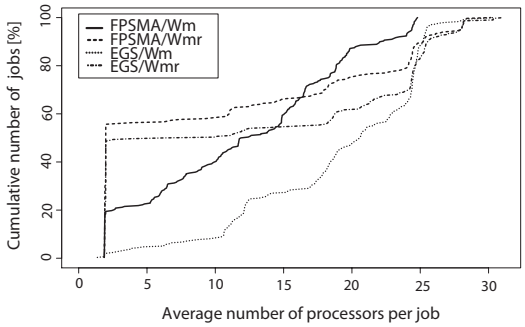
3.3.2 The workloads

The workloads that we employ in our experiments combine the two applications of Section 3.3.1 with a uniform distribution. Their minimum size is set to 2 processors, while the maximum size is 46 for GADGET and 32 for FT. In both cases, 300 jobs are submitted. Jobs are submitted from a single client site; no staging operation is ordered even when processors are allocated from remote sites.

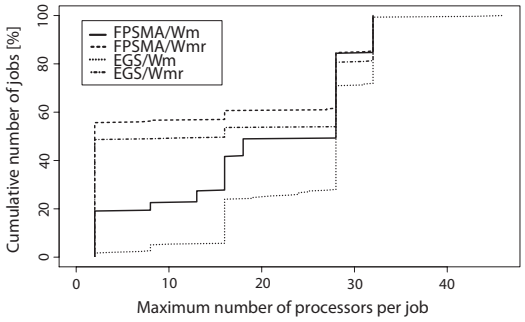
Regarding Figure 3.3, the maximum sizes we have chosen are greater than the sizes for which we have observed the minimum execution times. This deliberate choice comes from the following. Applications are not supposed to scale the same in all of the clusters, which may be heterogeneous. In addition, users may not be aware of the speedup behavior of their applications. Hence, the maximum size of a malleable job should not be the size that gives to the best execution time of the application in any particular cluster.

For the PRA-based experiments, we have used two following workloads. Workload W_m is composed exclusively of malleable jobs, while workload W_{mr} is randomly composed of 50% of malleable jobs and 50% rigid jobs. In both cases, inter-arrival time is 2 minutes. Rigid jobs are submitted with a size of 2 processors, and malleable jobs with an initial size of 2 processors. In our experiments, KOALA employs the Worst-Fit policy.

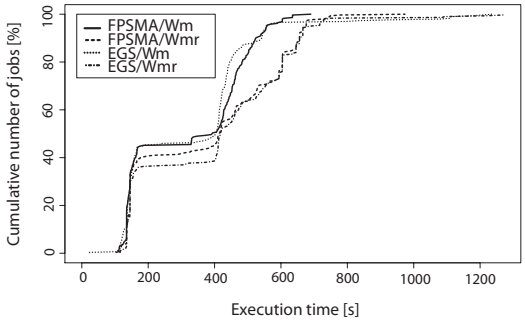
Apart from workload W_m or W_{mr}, the only background load during the experiments is the activity of concurrent users. This background load does not disturb the measures. When analyzing the PWA approach, we have used two workloads W'_m and W'_{mr}, which derive respectively from W_m and W_{mr}. In these workloads, inter-arrival time is reduced down to 30 seconds in order to increase the load of the system.



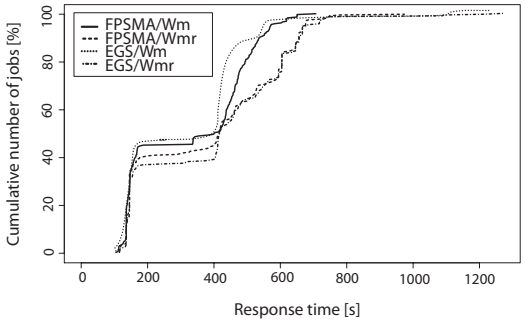
(a) The cumulative distribution of the number of processors per job averaged over the execution time of jobs.



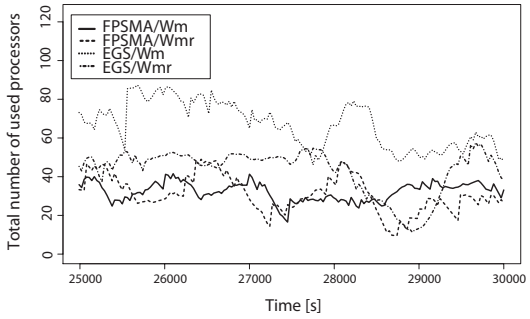
(b) The cumulative distribution of the maximum number of processors reached per job during its execution.



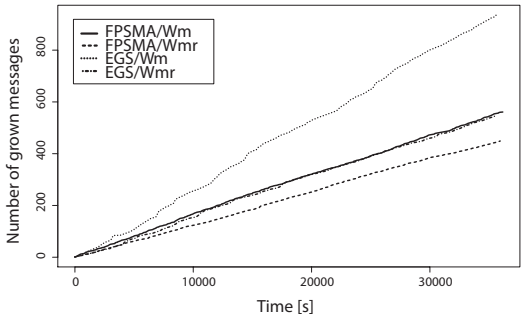
(c) The cumulative distribution of the job execution times.



(d) The cumulative distribution of the job response times.



(e) Utilization of the platform during the experiment.



(f) Activity of the malleability manager.

Figure 3.4: Comparison between the FPSMA and EGS policies with the Precedence to Running Applications (PRA) approach of job management (no shrinking).

3.4 Experimental results

In this section we will present the results of our experiments for both the Precedence to Running Applications (PRA) and the Precedence to Waiting Applications (PWA) approaches.

3.4.1 Analysis of the PRA approach

Figure 3.4 compares the FPSMA and EGS policies for malleability management in the context of the PRA approach for job management, i.e., when jobs are never shrunk. For this experiment, we have done 4 runs for each combination of a malleability management policy (one of FPSMA or EGS) and a workload (either W_m or W_{rm}).

Figures 3.4(a) and 3.4(b) show for each combination how jobs are distributed with regard to their average and maximum size. In both figures, with workload W_{mr}, which has 50% rigid jobs with only 2 processors, relatively few malleable jobs retain their initial size of 2 during their execution. In addition, we observe that among the policies, EGS tends to give more processors to the malleable jobs than FPSMA, both in terms of average and maximum number of processors per job. On one hand, with FPSMA, short applications (like FT in our experiments) may terminate before it is their turn to grow, i.e., before previously started jobs terminate. They are thus stuck at their minimal size. On the other hand, EGS makes all jobs grow every time it is initiated. Hence, even jobs that have been started recently do grow, and only few jobs do not grow beyond their minimal size.

Figures 3.4(c) and 3.4(d) show the distributions of the execution time and the response time, respectively. Two groups of jobs appear clearly: those with execution times and response times less than 200 s, and those for which these times are greater than 400 s. Those two groups correspond to the two applications in the workloads (respectively FT and GADGET). In both cases, we observe that the W_m workload results in better performance than the W_{mr} workload, which means that malleability makes applications actually perform better. Furthermore, with the FPSMA policy, for both of the workloads, the tail of the performance distribution cuts off far before than that of with the EGS policy.

Figure 3.4(e) shows the utilization of the DAS-3 during a part of the experiments. With workload W_m, which includes only malleable jobs, the EGS policy leads to a higher utilization. In fact, as we have already observed, this policy tends to make jobs larger in terms of their size. For the same reason, the utilization is higher with workload W_m than with W_{mr}.

Finally, Figure 3.4(f) shows the activity of the malleability manager. As expected, the number of grow operations is much higher when all jobs are malleable (workload W_m). It is also higher with the EGS policy than with FPSMA. Each time the policy is triggered, EGS makes all of the running malleable jobs grow, while FPSMA only does so with the ones that have started earlier.

3.4.2 Analysis of the PWA approach

Figure 3.5 compares the FPSMA and EGS policies in the context of the PWA approach for job management, i.e., when the scheduler can also shrink malleable jobs. With the PWA approach, the load of the system has a direct impact on the effectiveness of the malleability manager. If on the one hand the system is overloaded, all of the jobs are stuck at their minimal size and malleability management becomes ineffective, while if on the other hand the system load is low, no job is shrunk and PWA behaves exactly the same as PRA. We have therefore used workloads W'_m and W'_{mr}, which increase the load of the system.

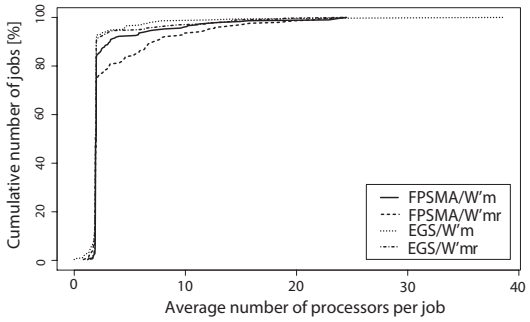
Figure 3.5(f) shows that beyond a certain time, the malleability manager becomes unable to trigger any other change than initial placement of jobs. Similarly, Figures 3.5(a) and 3.5(b) show that many of the jobs are stuck at their minimal size, irrespective of the workload and the malleability management policy being used. This phenomenon is more pronounced with the EGS policy, which means that load balancing is achieved as expected.

Figure 3.5(c) shows that the execution time is almost the same for the four cases. Most of the GADGET jobs have an execution time of 600s, 30% higher than with PRA. This difference results from what we observe about the size of the jobs. Figure 3.5(d) shows that the response time is far worse for the combination of the EGS policy and W'_m workload due to higher wait time. This result confirms the system overload observed on Figure 3.5(e) as a high utilization. Favouring long-running jobs, FPSMA has reduced enough the execution time of GADGET jobs to maintain the load sufficiently low.

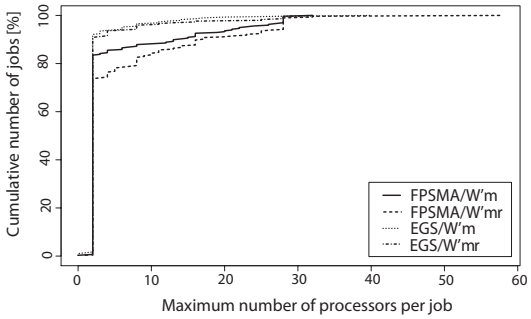
3.5 Related work

Simulation-based studies [32, 109, 110, 151] on the scheduling of malleable applications in cluster systems often neglect the issues that arise in real environments such as the effective scalability of applications and the cost of growing or shrinking in terms of resource allocation at runtime. Besides, the previous research have not considered the combination of malleability management and load sharing policies across clusters, which is an issue specific to multicluster grids. In this section we discuss several implementations of malleable applications and their scheduling in cluster systems.

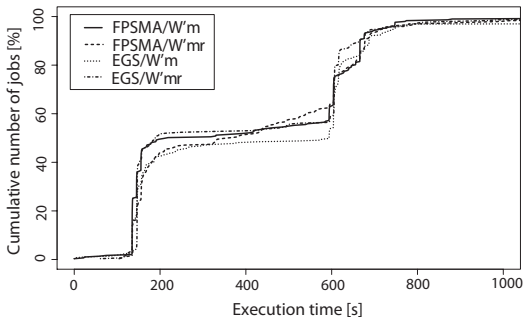
Several approaches have been used to make parallel applications malleable. While GrADS [201] relies on the SRS [200] framework, AppLeS [16] and ASSIST [3] propose to build applications upon intrinsically malleable skeletons. With AMPI [114], malleability is obtained by translating MPI applications to a large number of CHARM++ objects, which can be migrated at runtime. Utrera et al. [198] propose to make MPI applications malleable by folding several processes onto each processor.



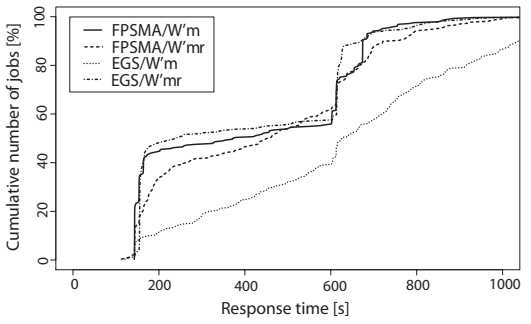
(a) The cumulative distribution of the number of processors per job averaged over the execution time of jobs.



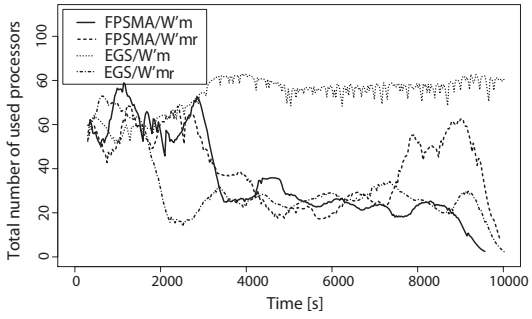
(b) The cumulative distribution of the maximum number of processors reached per job during its execution.



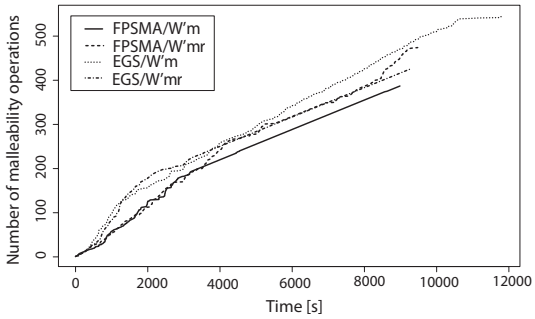
(c) The cumulative distribution of the job execution times.



(d) The cumulative distribution of the job response times.



(e) Utilization of the platform during the experiment.



(f) Activity of the malleability manager.

Figure 3.5: Comparison between the FPSMA and EGS policies with the Precedence to Waiting Applications (PWA) approach of job management (both growing and shrinking).

A couple of works have studied how to schedule malleable applications in combination with making parallel applications malleable. Among them, AppLeS [16] and GrADS [199] are somewhat specific as they propose that applications are responsible to schedule themselves on their own. However, this approach raises the question of how the system behavior and performance would be, in case several concurrent malleable applications compete for resources. Furthermore, as those approaches rely on checkpointing, it is unclear how an application gets its resources back when it accepts to try a new allocation. System-wide schedulers, such as KOALA do not suffer from these drawbacks.

Some other approaches rely on a system-wide scheduler. Corresponding to the underlying execution model, AMPI uses an equipartition policy, which ensures that all jobs get almost the same number of processors; while the policy in [198] is based on folding and unfolding the jobs (i.e., doubling or halving the number of allocated processors). However, those two approaches rely on the properties of their underlying execution model. For instance, equipartition assumes that any application can be executed efficiently with any number of processors, as it is the case with AMPI; while folding restricts the number of processes to be divisible by the number of processors (often a power of 2 for practical reasons), which is the only way to fold efficiently non-malleable applications. A more general approach such as the one we propose is more appropriate in the context of multicluster grids. McCann and Zahorjan [141] further discuss the folding and equipartition policies. According to their experiments, folding preserves well efficiency; while equipartition provides higher fairness. They also propose a rotation policy in order to increase the fairness of the folding policy. However, rotation is almost impracticable in the context of multicluster grids.

As fairness does not imply efficiency in terms of allocated processors, a biased equipartition policy is proposed in [96] such that the cumulative speedup of the system is maximized. The policy further considers both malleable and rigid jobs in a single system [95], and it guarantees to allocate a minimum number of processors to each malleable job, such that they are not ruled out by rigid jobs. However, in multicluster grids it is common that some of the users bypass the global grid scheduler. The problem of making the scheduler take into account that incurred background load is not addressed in this study.

3.6 Summary

In this chapter we have presented the design and a real implementation of the support for malleability in our KOALA grid scheduler using the DYNACO framework [28, 30] for application malleability. We have proposed two malleability management policies, Favor Previously Started Malleable applications (FPSMA) and Equi-Grow&Shrink (EGS), for malleable applications. FPSMA distributes idle processors to the malleable jobs starting from the job that have started earliest, while EGS spreads them equally among all running

malleable jobs. We have also proposed two design choices as to when to initiate malleable management policies, which give precedence to running malleable applications over waiting ones (PRA) or vice versa (PWA), respectively. We have evaluated these policies and design choices running experiments with workloads of malleable and rigid parallel applications in DAS-3. Our experiments show that malleability improves the execution time of the parallel applications. In addition, when PRA is preferred the FPSMA policy outperforms the EGS policy, while they perform similarly when PWA is preferred. PWA behaves exactly the same as PRA unless the load in the system is high.

Chapter 4

Cycle scavenging for parameter sweep applications

Cycle scavenging is the underlying technology of desktop grids and volunteer computing projects (such as Seti@Home [178]), which enables harnessing idle CPU cycles of desktop workstations to solve large-scale scientific problems in a variety of research areas. The same concept can be applied to multicluster grid environments to give users the opportunity of executing such large-scale computation-intensive applications at a low priority without being in the way of regular grid users or local users. In this chapter we present scheduling strategies for the support of cycle scavenging in multicluster grid systems, and evaluate the performance of the implemented solutions both from the user and the system perspectives.

Today, many grids exhibit significant job submission bursts between periods of relative idleness [98]. Many users perform observational scheduling, that is, they postpone the submission of relatively low-priority jobs until a cluster becomes (largely) idle. This attitude, however, may lead to resource contention when several such users crowd the same idle cluster, and may delay the execution of more important jobs unless some form of administrative support for job and user priorities is deployed. Cycle scavenging, on the other hand, obviates the need for establishing priority classes, which can be a time-consuming and error-prone administrative operation. Supporting cycle scavenging in a grid system would enable users to execute long-running applications (e.g., 3-D rendering, molecular docking, and game solving applications) at the lowest priority without violating the resource usage rules enforced in the system.

In this chapter we extend our KOALA grid scheduler with cycle scavenging support. The implemented cycle scavenging mechanism runs alongside the regular grid scheduling, being unobtrusive to the jobs of higher priority (both local and grid jobs). Although cycle scavenging infrastructures do exist [7, 44, 134], our mechanism obviates the need for additional software installations on the compute nodes or any modifications to the re-

source managers of the clusters (e.g., SGE [87]), both of which would be administrative obstacles in multicluster grid systems. We exclusively target large-scale applications that can be modeled as Parameter Sweep Applications (PSAs) to run as cycle scavenging (CS) jobs. We enable single PSAs to run across multiple clusters simultaneously, that is, in a co-allocated fashion.

The scheduling architecture for cycle scavenging that we have incorporated into KOALA comprises two levels. At the grid level, scheduling policies run to ensure fair distribution of idle resources among CS users in a dynamic fashion, and at the application level, CS users can customize the scheduling policies in order to improve the performance of their applications. We have designed two best-effort cycle scavenging scheduling policies that enforce fair resource sharing between CS users in a dynamic fashion. The policies do not need to keep track of the past usages of the CS users. The first policy distributes or reclaims the idle nodes evenly among CS users, regardless of the site these idle nodes belong to. The second policy, on the other hand, partitions or reclaims the idle nodes evenly such that each CS user is assigned an equal share of idle nodes on each site. We show with experiments conducted in the DAS-3 multicluster environment that the latter policy outperforms the former in terms of fairness. We also perform experiments to demonstrate the efficiency of the implemented system in terms of scheduling overhead.

The rest of the chapter is structured as follows. Section 4.1 explains the requirements for the support of cycle scavenging in a multicluster grid scheduler. Section 4.2 presents the implemented scheduling architecture and the scheduling policies for cycle scavenging. Section 4.3 presents the experiments that we have performed to assess the performance of the implemented cycle scavenging scheduling policies. Section 4.4 reviews the related work, and finally, Section 4.5 summarizes the chapter.

4.1 Requirements for supporting cycle scavenging

In this section we state the requirements of cycle scavenging (CS) that should be taken into account by a grid resource manager or a grid scheduler to ensure the efficient allocation of idle resources. These requirements are fair-share scheduling, a proper notion of idleness of processing nodes, and unobtrusiveness, respectively.

4.1.1 Fair-Share scheduling

As we know from desktop grids or volunteer computing environments, CS applications are high throughput computing applications that consist of independent sequential tasks, which scale to many thousands of nodes, and require large amounts of computation. Considering the size of these applications, the scheduling mechanism should try to achieve fair-share resource allocation among the users submitting CS tasks, so that it prevents

some users monopolizing all free resources for a considerable amount of time, leaving space for users having relatively light workloads.

Fair-Share scheduling, in fact, was originally proposed for managing resource allocation of processes on time-sharing uniprocessor systems [119]. The application of the fair-share resource allocation to a distributed system is very much dependent on how the system administrators define the fair share. A study investigating this issue in cluster systems [122] shows that unless the jobs on a cluster are flexible in terms of space (number of nodes) and time (checkpointable), fair-share is not able to achieve real-time fairness as it can on a uniprocessor; rather, it becomes a best-effort service.

Due to the fact that tasks of a CS application may be preempted at any time, giving a hard completion time guarantee for a CS application is almost impossible; nevertheless, users submitting CS applications may be more interested in the rate of receiving partial results, that is, the throughput, since their jobs consist of independent sequential tasks. To improve such performance, scheduling at the application level should be considered as a separate layer under the fair-share resource allocation scheme. Such a layered architecture would provide modularity and flexibility.

4.1.2 Notion of idleness

A grid scheduler would possibly schedule CS tasks whenever it is aware of idle nodes on clusters. However, the existence of idle resources may not be the only prerequisite for acquiring those resources to run tasks of a CS application. When to consider a resource as idle may be subject to the additional administrative policies of each site. For instance, a site may only be willing to run such tasks when a certain percentage of its resources are idle and simultaneously there are no local jobs waiting in the queue, or site administrators may want to set time limits such as allowing CS tasks to run only at nights. In addition, site administrators may need to set usage limits due to reasons such as cooling costs, which would increase by running long-lasting CS applications at a high utilization.

4.1.3 Unobtrusiveness

From the system perspective, we need to ensure that placing and running CS tasks are unobtrusive to the jobs of higher priority; a grid scheduler has to make immediate preemption possible without causing significant delays whenever non-CS jobs demand the nodes allocated to CS jobs. These high priority jobs in a multicluster grid can be defined as the non-CS jobs that are directly submitted to local resource managers by local users, and the non-CS jobs that are submitted to a grid scheduler by grid users.

When a CS task is canceled due to a high priority job by the grid scheduler, it has to return back to a task pool so that it can be re-scheduled. A checkpointing mechanism

would be useful not to lose many computations; however, we do not consider checkpointing as it lies outside the focus of this chapter. Instead, we leave users to implement their own application level checkpointing solutions.

4.2 Designing support for cycle scavenging in KOALA

In this section we present our design for supporting cycle scavenging (CS) in KOALA. First, we explain the application model, then we describe the system architecture, and finally, we present the fair-share policies and scheduling at the application level, respectively.

4.2.1 Application model

For this study we consider in particular high-throughput applications that conform to the Parameter Sweep Application (PSA) model. A PSA can be defined as a single executable that is run for a range of parameters for a large number of times. The PSA model is well suited for our problem since on the one hand many large-scale scientific applications are structured in this way, and on the other hand PSAs are very flexible to be run as CS jobs in a multicluster grid environment. We support OGF's standardized Job Description Language (JSDL 1.0 [113]) to which we have added an extension for parameter sweeps such that users can submit PSAs as single entities by specifying input files for parameter extraction and representing the ranges of parameters as loops or lists of comma-separated values.

We treat PSAs as malleable applications that can grow or shrink dynamically in terms of the number of compute nodes they execute on. Moreover, we allow PSAs to run in multiple clusters simultaneously, i.e., in a co-allocated fashion. The KOALA scheduler has been modified to handle PSAs in a different way than regular grid jobs (we explain this in the next section in more detail). A job component, in the cycle scavenging context, represents the set of tasks of a PSA that are running in the same cluster. The job components are dynamically created at run time according to the interactions with the scheduler rather than being statically specified at submission time.

4.2.2 System architecture

In order to support cycle scavenging in KOALA, we have implemented two additional components, a specific KOALA runner called the *CS-Runner* and a glide-in mechanism [81] called the *Launcher*. Figure 4.1 illustrates the interaction between these components and the existing KOALA components that together achieve cycle scavenging. The

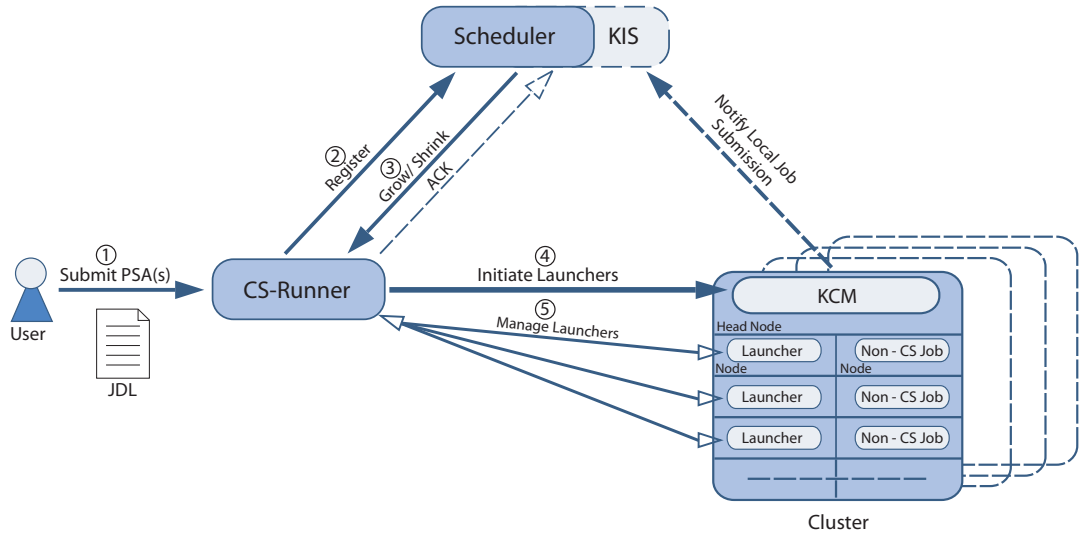


Figure 4.1: The system architecture to support cycle scavenging in KOALA.

fair-share scheduling policies have been incorporated in the existing scheduler component. The KOALA Component Manager (KCM) is our job submission daemon that runs as a separate copy per job component on the head node of the corresponding cluster. The KCM interfaces to a local resource manager (that is SGE [87] in DAS-3) through the standardized DRMAA [59] interface. We have added to the KCM the functionality of notifying the KOALA Information Service (KIS) about local (non-CS) job submissions.

Along with scheduling regular grid jobs, the scheduler is responsible for allocating and reclaiming idle nodes among active CS-Runners, based on the fair-share policy in use. The fair-share policies decide on which CS-Runners are going to be offered additional nodes (i.e., *grow*), or are going to be forced to release nodes (i.e, *shrink*), from which clusters, and how many (see Section 4.2.3). A CS-Runner may receive a grow request whenever the scheduler becomes aware of idle nodes in one or more clusters through the KIS. On the other hand, a CS-Runner may receive a shrink request for one of two reasons. First, a CS-Runner may be forced to release nodes that it occupies whenever a non-CS job (at the local or grid level) demands those nodes. Secondly, depending on the fair-share policy deployed, it may also be asked to release nodes to open up space for other CS-Runners.

The CS-Runner is entitled to manage the scheduling and monitoring of CS tasks (each task refers to a parameter) on behalf of a user on the allocated idle resources. It initiates Launchers on the idle nodes to delegate the execution of the parameters. For simplicity, in this study we restrict users to have a single CS-Runner at a time; hence, each CS-Runner process in the system corresponds to a different user. When it accepts a grow offer, a CS-Runner submits a request to the KCM to initiate Launchers on the idle nodes allocated

by the scheduler, and then the parameter values are submitted to the Launchers, based on the application level policy in use (see Section 4.2.4). Upon successful execution of a parameter, a Launcher sends back the result to the associated CS-Runner. Upon receiving a shrink message, the Launchers are preempted according to the Launcher preemption policy deployed in the CS-Runner (see Section 4.2.4).

The Launcher runs an executable for the given set of parameters in sequential order that otherwise would be submitted one by one to the scheduler. The motivation behind the Launcher mechanism is to reduce the overhead by decreasing the number of job submissions which would put a considerable burden on the head nodes in a parameter study, and to have more control on application level scheduling in the CS-Runner (see Section 4.3.1 for the performance results). If compute nodes have multiple processors, a CS-Runner can optionally initiate a separate Launcher per processor in order to improve the performance (i.e., the throughput).

In fact, the idea of enabling the rapid execution of many jobs on clusters, creating a virtual pool of resources and bypassing the local resource manager, has previously been realized with several implementations such as the Condor Glide-In mechanism [81], MyGrid's virtual cluster approach [45], DIANE [150], and the Falkon framework [169]. The Launcher adopts the same idea; however, it differs from these mechanisms in terms of functionality as it is specialized to run complete PSAs as single entities. For instance, it saves statistical information about the parameters that it has run, and is able to send partial results on completion or periodically.

4.2.3 Fair-Share policies

Our solution to fair-share resource allocation is to partition the idle resource space equally among the active CS-Runners. We have designed two best-effort fair-share policies that are based on the well-known *Equipartition* policy [141], which has originally been proposed as a dynamic processor allocation scheme for malleable parallel applications running in a single cluster.

The first policy, *Equipartition-All*, tries to distribute the idle nodes (or reclaims the required number of nodes) evenly among the active CS-Runners on a grid-wide basis (see Figure 4.2). Whenever some of the nodes are reclaimed for non-CS jobs, the policy does not repartition the idle nodes allocated to the CS-Runners to equalize the numbers of idle nodes they occupy. Instead, the policy gives back the reclaimed nodes to the corresponding CS-Runners after the non-CS jobs are finished.

The second policy, *Equipartition-PerSite*, partitions the idle nodes on a per-cluster basis. It tries to allocate or reclaim nodes evenly in each cluster to or from the active CS-Runners. There is no need for repartitioning with the Equipartition-PerSite policy, since each CS-Runner has the same number of nodes before and after some of the nodes are

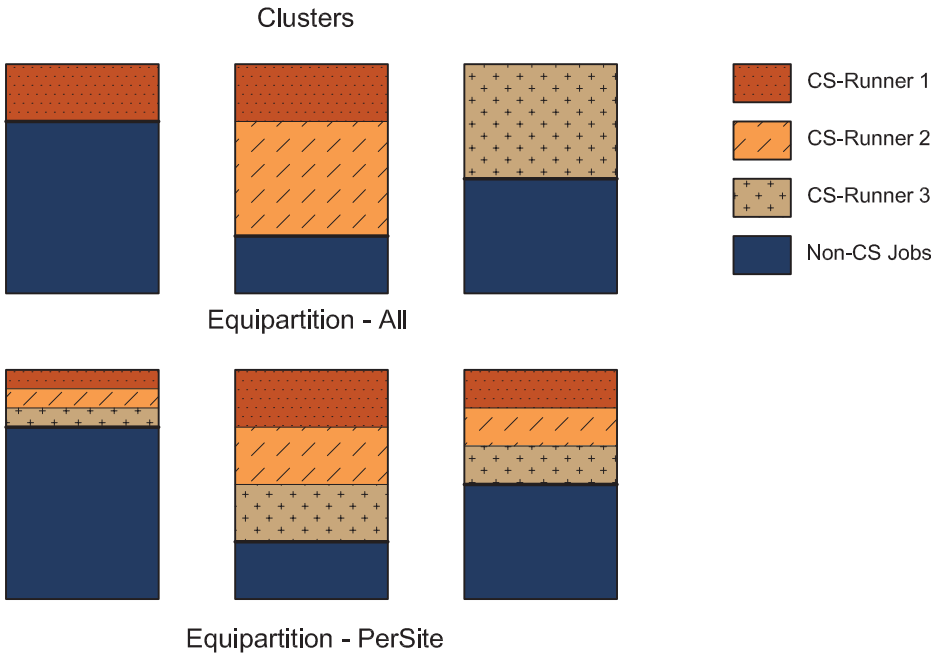


Figure 4.2: The distribution of the idle nodes among the CS-Runners with the Equipartition-All and the Equipartition-PerSite policies.

reclaimed for higher priority jobs.

In comparison to the Equipartition-PerSite policy, the Equipartition-All policy may not treat users equally due to the heterogeneity of the node capabilities and the possibly different background loads on the clusters because of local or grid-level jobs. We investigate this issue and its effects in Section 4.3.2.

In case there are too many CS users competing for the limited numbers of idle resources, the overall throughput can substantially decrease with our dynamic fair-share policies. Therefore, we apply admission control to limit the number of active CS-Runners to improve the overall service quality.

4.2.4 Scheduling at the application level

The default scheduling solution that we have implemented at the application level is based on a pure pull model. Each Launcher requests from its CS-Runner a new parameter to execute when it becomes idle. When the scheduler sends a shrink message, the desired numbers of Launchers are preempted, and the uncompleted parameters are placed back into the parameter pool of the CS-Runner. The Launcher preemption policy preempts the Launchers starting from the one that has pulled a parameter most recently, to the one that has pulled a parameter earliest, with the intention to lose less computation. The reason we prefer the pull model instead of the push model is that the latter necessitates a CS-Runner

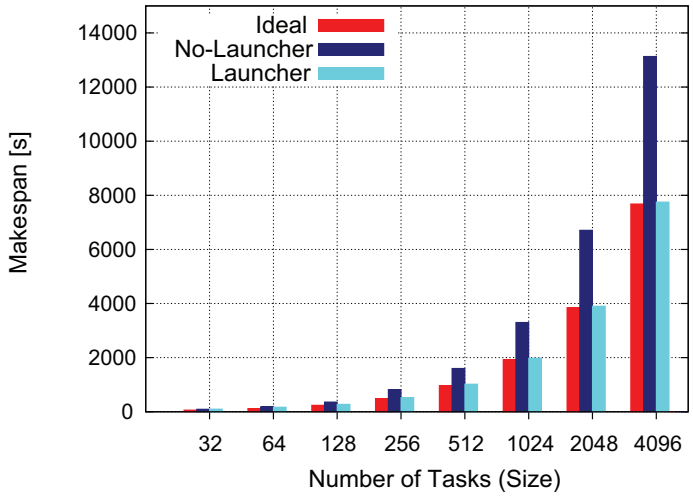


Figure 4.3: The makespan of PSAs for different submission mechanisms.

to frequently poll its Launchers for idleness.

We provide an API with which users can customize the CS-Runner, and the mentioned scheduling solutions according to their applications' characteristics and requirements. One such example might be resubmitting the preempted jobs to the sites where input files already exist (in case the parameters are input files), in order to reduce unnecessary file transfers.

4.3 The performance of the cycle scavenging system

In this section we evaluate, in the DAS-3 testbed, the performance of the cycle scavenging system and the scheduling strategies that we have incorporated KOALA.

4.3.1 The impact of the task submission mechanism

In our first experiment we demonstrate the performance gain of using the Launcher mechanism over submitting the tasks (parameters) of a PSA as separate tasks to a grid system. We use a synthetic application that takes the same time, 60 seconds, to execute each of its parameters. We consider a single cluster with 32 nodes, and vary the application size (number of parameters) as powers of 2 between 32 and 4096. We submit the application using the CS-Runner with and without the Launchers for each size. For the former case, the Launchers are initiated once and pull equal numbers of parameters to execute. For the latter, the parameters are submitted as separate tasks to the grid middleware whenever the CS-Runner is notified of idle nodes by the scheduler. During the experiments we ensure that no other jobs run in the cluster.

Figure 4.3 shows the results in terms of the *makespan* of the application. The makespan of a PSA can be defined as the difference between the time of the earliest submission of one of its tasks, and the time of the latest completion of one of its tasks. The ideal case assumes no overhead in the system, that is, sets of 32 tasks are placed on the nodes and started immediately. With using Launchers, the performance is close to the ideal case, irrespective of the application size. On the other hand, as the size of the application increases, the difference becomes much more visible between the regular submission of tasks and the ideal case. It leads to a difference of approximately one and a half hour when running the application of size 4096. Provided that the executable and the input files reside in the execution site, there are two sources of this difference: the task startup overhead, which is 5 seconds per task on average, and the information delay due to the polling nature of monitoring resources. The monitoring period is 60 seconds in the experiment, which can be considered as a realistic grid monitoring setting (see, e.g., Ganglia [84]).

4.3.2 Performance of the fair-share policies

In our second experiment, we compare the performance of the Equipartition-All and the Equipartition-PerSite policies. We assume that three CS users submit the same application with the same parameter range. The parameter sweep application we use is a program that we have implemented to solve a rewarding puzzle (2M.US$), Eternity-II [68], which is played on a square board with 16X16 spaces. The goal is to place all of the 256 pieces on the board in such a way that the patterns on adjacent sides match. Finding a solution is computationally hard; a brute-force technique requires millions of CPU years. Our solver performs random walks to yield the best solution that it can, based on the parameters given.

Although local job submissions force CS-Runners to release nodes, in this experiment, we mainly attributed preemptions to grid job submissions. We use two synthetic workloads, with different arrival patterns, in order to represent the jobs of regular grid users. The first workload, *WBlock*, periodically imposes for ten minutes a steady load of 40% on the system with a period of 20 minutes; the load is distributed non-uniformly across the clusters. The second workload, *WBurst*, imposes a 40% load (again non-uniform across the clusters) with burst submissions of 1 minute repeated every 10 minutes. The motivation behind using such workloads is to observe the performance of the policies under dynamic loads, which is a typical case for grids.

Each experiment is terminated after 1 hour. We monitor the load due to local jobs in each cluster, and ensure that this background load is steady and does not disturb the experiments. We use the tools provided within the GrenchMark project [99] to create the workloads and to ensure the correct submission of them to the system. We use with

Table 4.1: Performance of the CS policies under the WBlock workload.

	Equipartition-All			Equipartition-PerSite		
	Throughput [tasks/min]	Num. of Preemptions	Avg. Num. of Nodes	Throughput [tasks/min]	Num. of Preemptions	Avg. Num. of Nodes
CS User-1	1.68	346	23	2.55	323	37
CS User-2	3.3	304	45	2.4	321	37
CS User-3	1.92	333	28	2.7	323	37

KOALA the Worst-Fit policy (see Section 2.3) in order to schedule the jobs of these workloads.

Figure 4.4 shows the performance results of the Equipartition-All and the Equipartition-PerSite policies in terms of the number of completed tasks (parameters) during the experiments, and Table 4.1 presents the throughput, the number of preemptions, and the average number of nodes allocated to each CS user for the experiments with WBlock. With the Equipartition-All policy, we observe that the number of tasks completed (as well as the metrics in Table 4.1) per CS user varies considerably. With both workloads, the load is distributed in an unbalanced way across the clusters. As a consequence, with the Equipartition-All policy, CS users who happen to occupy more processors than other users in the heavily loaded clusters, suffer more due to frequent preemptions. This phenomenon, on the other hand, does not affect the CS users when the Equipartition-PerSite policy is used. That policy always allocates an equal number of nodes in each cluster to each CS user, and therefore, each CS user would suffer or benefit equally from the behavior of non-CS jobs in a particular cluster. In this experiment, we have also verified that all the CS users receive almost the same amounts of CPU time with the Equipartition-PerSite policy. The reason for the small differences in the number of completed tasks are due to Launcher failures (the failed Launchers are restarted immediately when noticed by the CS-Runners) and to the execution time variation of the solver application due to the randomness it includes.

4.3.3 Unobtrusiveness of the cycle scavenging system

In order to assess the unobtrusiveness of our CS system, we have performed extensive test runs to quantify the additional delay that local jobs and non-CS grid jobs experience before they start execution due to reclaiming of the nodes occupied by CS-Runners. For non-CS jobs submitted to KOALA, we observe an additional delay between 2 and 8 seconds before they start execution. Local jobs, however, experience an additional delay between 8 and 15 seconds. KCM polls the local resource manager with a period of 10 seconds to be aware of recent local job submissions. This contributes most to the addi-

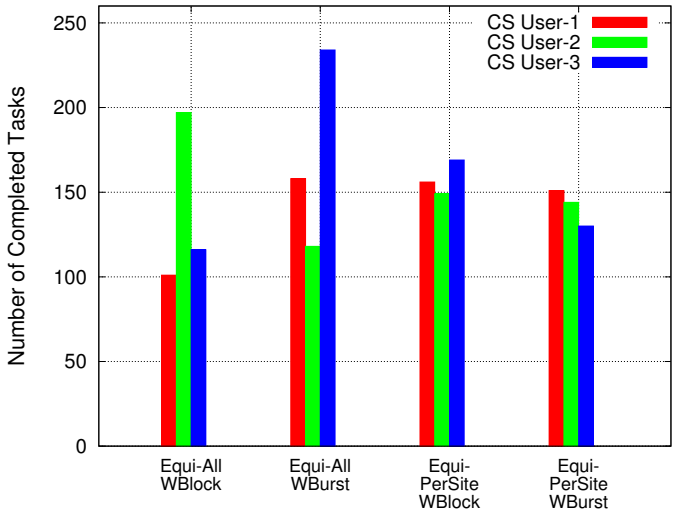


Figure 4.4: Number of tasks completed per CS user for the two fair-share policies.

tional delay that local jobs experience. To decrease the polling period would of course decrease the delay, but we have observed that periods lower than 10 seconds increase the processor load on the head node of the cluster in question considerably.

4.4 Related work

There are various platforms that make desktop grids or volunteer computing possible as well as some that enable cycle scavenging at the organization level to put idle cycles to good use.

The BOINC [7] platform facilitates volunteer computing projects (e.g., Folding@home [75], Rosetta@home [173], and Seti@home [178]). BOINC provides tools that allow participants to remotely install a client software on a large numbers of desktops, and to attach the client software to accounts on multiple projects. With BOINC, desktop owners are able to specify how their resources can be allocated among the projects. Another similar platform is Entropia [42], which can be distinguished from its counterparts by its binary sandboxing technology for ensuring security and unobtrusiveness, and its architecture which incorporates physical node management, resource scheduling, and job management layers. OurGrid [44] is an open platform that enables different research labs to share their idle computational resources when needed. OurGrid relies on a peer-to-peer incentive mechanism, called *Network of Favors*, which aims to make it in each participant's best interest to donate idle cycles, along with preventing free riding. With OurGrid, each user runs a broker-agent which competes with other agents to schedule the jobs over the resources on behalf of the user. Therefore, it does not provide fair-share resource allocation among users. The Condor [134] platform was initially designed to

scavenge compute cycles on large collections of idle desktop machines, but it has also been extended to operate as a batch scheduler on top of a cluster system and as a resource broker on top of Globus [89] based grids [81]. In addition to desktop computers, it is also possible with Condor to scavenge idle nodes in a cluster by configuring each node such that they can execute Condor jobs when no job is running which has been submitted by the local resource manager of the cluster. Condor does not adopt traditional scheduling, rather it uses a central matchmaking mechanism in which jobs and resources are matched according to their requirements specified with so-called ClassAds. As the fair-share policy, Condor runs the *Up-Down* [156] algorithm to protect the rights of light users when a few heavy users try to monopolize all resources. The algorithm relies on the information of past resource usage rates of users.

In this study we have not considered past usage, since our notion of fair-share partitions the idle resources evenly among users dynamically in real time. In addition, to deploy any of the platforms mentioned above in a multicluster grid system requires per-node installations or configurations in the local resource managers of clusters, which could be impossible due to administrative restrictions. In contrast, the system we have presented in this chapter does not require any such installations or modifications, rather, it seamlessly integrates the notion of cycle scavenging into grid-level scheduling.

4.5 Summary

In this chapter we have presented the design and the analysis of the support for cycle scavenging in multicluster grids. We have incorporated scheduling strategies for cycle scavenging in our KOALA grid scheduler. We have implemented two best-effort fair-share polices that dynamically partition the idle nodes among the active CS users. We have compared these policies with experiments conducted in a real multicluster grid system. The results show that a dynamic cycle scavenging policy should distribute the idle nodes from each cluster in equal amounts to the active CS users in order to ensure fairness. In addition, we have performed experiments to show that the implemented system is efficient in terms of scheduling overhead, and, is unobtrusive to higher priority jobs.

Chapter 5

The performance of bags-of-tasks in multicluster grids

Even though the local and wide-area interconnections in multicluster grid systems have improved markedly as in our DAS system, and efficient wide-area communications libraries are now available such as MPI variants and Ibis [205], it turns out that a large fraction of the jobs in the workloads imposed on such systems is due to sequential applications, often submitted in the form of Bags-of-Tasks (BoT) [98, 102] (see also Chapter 7 for a broader discussion). The reasons for this phenomenon are the relatively high network latencies, the complexities of parallel programming models, and the nature of the scientific computational work (e.g., repeated simulations, parameter sweeps).

In this chapter we present a realistic and systematic investigation of the performance of scheduling BoTs in multicluster grids. What distinguishes our study from previous efforts is that we use a workload model for BoTs with which we generate realistic traces for our evaluation, and that we explore the large design space of bag-of-task scheduling in multicluster grids along five axes, which are the task selection policy, the workload, the information policy, the task scheduling policy, and the resource management architecture.

The rest of this chapter is organized as follows. In Section 5.1, we propose a systematic approach to evaluating the scheduling of BoTs in grids; with this approach we identify three scheduling policies that have not been investigated previously, which are the Earliest Completion Time with task runtime Predictions, the Fastest Processor First, and the Shortest Task First with Replication policies. In Section 5.2, we present the workload model for BoTs. In Sections 5.3 and 5.4, we investigate the performance of BoTs in multicluster grids. In Section 5.5, we review the related work on BoT scheduling. Finally, in Section 5.6, we summarize the chapter.

5.1 A scheduling model for BoTs

In this section we present a scheduling model for BoTs in multicluster grids consisting of four components. In Section 5.1.1, we describe the models of the system and of the jobs submitted to it. The system model considers clusters of resources. In Section 5.1.2, we present the resource management architectures, which add structure to the set of clusters as to how their resources are jointly managed. Deciding which tasks to run where is in our model a two-step process: first from the waiting tasks in the system an *eligible set* is created using one of the *task selection policies* (Section 5.1.3), and then the tasks from the eligible set are mapped to resources using one of the *task scheduling policies* (Section 5.1.4).

Our model extends the current state-of-the-art in several ways. The resource management architecture and the task selection policies have not been explicitly included in previous BoT scheduling models. For the task scheduling policies, previous models [39,130] have considered that the resource performance (e.g., speed, SPECInt2006 value [190]) or the task runtime are not (accurately) known by the scheduling policy, but that at least one of them is accurately known. In contrast to these models, our model considers that at the same time the information about both the processor performance and the task runtime may be inaccurate or even missing.

5.1.1 System and job model

In our model we assume that the computing resources (processors) are grouped in clusters. The processors may have different performance across clusters, but within the same cluster they are homogeneous. The workload of the system consists of jobs submitted by various users; each of the jobs is a bag of sequential tasks (possibly only one). We employ the SPEC CPU benchmarks model for application execution time [190], that is, the time it takes to finish a task is inversely proportional to the performance of the processor it runs on (see also Section 5.3). Upon their arrival into the system the tasks are queued, waiting for available resources on which to be executed. Once started, tasks run to completion, so we do not consider task preemption or task migration during execution. Instead, tasks can be replicated and canceled, or migrated before they start.

5.1.2 Resource management architectures

The complete set of clusters is operated as one large-scale distributed computing system using a resource management architecture which dictates how jobs are distributed across the resources in the system. In our model, each of the clusters has its own local resource manager (RM), but other RMs may be employed to build a complete architecture. A

global resource manager (GRM) is an RM that can submit tasks for execution to another RM.

Each RM in the system executes the same scheduling procedure upon the arrival of new BoTs and on the completion of tasks. The scheduling procedure is as follows. First, the RM calls the task-selection policy, which selects from the RMs current queue the *eligible set* of tasks. Then, the scheduler executes the eligible set using the task-scheduling policy, which in turn sorts the eligible set and/or ranks the resources to create a schedule. Only after all the tasks in the schedule are completed is a new eligible set generated.

In this work we use three resource management architectures, one based on independent clusters, one centralized, and one decentralized. Below we describe these architectures:

1. **SEParated Clusters (`sep-c`):** Each cluster operates separately with its own local RM and its own local queue to which jobs arrive. Each user can submit tasks to exactly one RM.

2. **Centralized Scheduler with Processor monitoring (`csp`):** Each cluster operates separately with its own local RM and its own local queue to which jobs arrive. In addition, a GRM with a global queue operates on top of the cluster local RMs. The users submit tasks only to the global system queue. When the global RM observes that a cluster has idle resources, it moves some of the tasks on the global queue to that cluster. The information about the number of free processors is gathered periodically by a monitoring service.

3. **Condor-like, with Flocking (`fcondor`):** This models a Condor-like architecture with flocking capabilities [65]. Similarly to `sep-c`, each cluster operates separately with its own local RM and its own local queue to which jobs arrive. However, here each user can submit tasks to any RMs. A user keeps submitting tasks to the same RM while that RM starts the tasks immediately; when tasks start to be queued, the user will switch to another RM, in round-robin order.

In our model, a GRM may submit at most one task at a time to another RM; the target RM will in this case receive BoTs with one task. This ensures that the GRM can control the order in which its tasks are considered at the remote RM. Note that for `sep-c` there is no GRM, but there exist local users.

5.1.3 Task selection policies

The RM uses its task selection policy to select from the RM's local queue the *eligible set* of tasks. We investigate in this work seven task selection policies, the first two of which do not take into account the user who submits a task:

1. **S-T** The Select-Tasks policy selects all the tasks in the system. For each arriving BoT, the set of its tasks is added to the eligible set.

2. **S-BoT** The Select-BoTs policy selects BoTs in the order of their arrival. When the eligible set has become empty, the set of tasks of the selected BoT are considered the new eligible set.

3. **S-U-Prio** The Select-User-Priority policy assumes that each user in the system has a unique priority. It selects all the tasks of the user with the highest priority. Ideally, each user has a unique priority, which distinguishes the user's resource usage rights from any other's. In practice, a system will be configured with just a few (e.g., up to four) distinct priorities. We use in this work the ideal scenario; in many settings, its performance gives an upper bound of the performance of the practical scenario.

4. **S-U-T** The Select-User-Tasks policy aims at the equal sharing of resources among the system users: It first selects the user with the lowest resource consumption, and then it selects all the tasks of the selected user.

5. **S-U-BoT** Similarly to S-U-T, the Select-User-BoT policy aims at the equal sharing of resources among the system users: It first selects the user with the lowest resource consumption, then it orders the selected user's BoTs in their order of arrival into an ordered set. From this set, the tasks of the first BoT are considered the new eligible set.

6. **S-U-GRR** The Select-User-Global-Round-Robin algorithm selects the next user in round-robin order. It then adds all waiting tasks of this user to the eligible set. If a user submits additional BoTs during his turn, the corresponding tasks will not be considered for selection during this turn.

7. **S-U-RR** The Select-User-Round-Robin policy is a variation of S-U-GRR, where only one task is selected per user at each round. Under S-U-RR, a BoT of size N will receive complete service after exactly N rounds. This algorithm is similar to the WFQ algorithm for scheduling packets over the network [161], with the main difference of tasks not having a known runtime at selection time, as opposed to network packets having a pre-assigned number of bits to transfer.

For all task-selection policies that require it, the resource consumption is simply computed as the sum of the past usage and usage of currently running tasks. The usage of a task is computed as the CPU time spent by it until its completion (the current moment) for tasks that have (not yet) completed. Given the range of the runtimes of the tasks in our traces and experiments, a precision on the order of seconds is enough for computing and accurately measuring these CPU time values.

Table 5.1: An information availability framework. **K**, **H**, and **U** stand for information Known a-priori, based on Historical data, and Unknown, respectively. The scheduling policies marked with $\star$ have not been previously studied in the context of BoT scheduling.

		Task Information		
		K	H	U
Resource Information	K	ECT [138], FPLT [142] MaxMin [39]	ECT-P*	FPF*
	H	DFPLT [48] MQD [130]	-	-
	U	STFR*	-	RR [82] WQR [48]

5.1.4 Task scheduling policies

There exist many scheduling policies for BoT workloads in large scale distributed systems [39, 48, 82, 130]. We categorize such policies according to the information policy used for resources and tasks, where a piece of information can be either fully Known (K), known from Historical records (H), or fully Unknown (U). In this work we consider only two pieces of information (the *information set*): the performance of a processor and the execution time of a task (e.g., on a reference processor). Then, a task-scheduling policy can be characterized in terms of its information usage by a tuple (R, T), with R and T the information policy for resources and for tasks, respectively. We map several scheduling policies to this characterization in Table 5.1. Most of the existing scheduling policies are either (U, U) or (K, K); in practice, Condor [195] uses by default an (U, U) policy, AppLeS supports several (U, U) and (K, K) heuristics [16], MyGrid implements an (U, U) policy [45]. Scheduling policies for BoT of types (U, K), (K, U), (U, H), (H, U), and (H, H) have not been addressed in the literature of BoT scheduling.

To address the main goal of this work, a systematic approach to evaluating BoT scheduling in grids, we propose one simple policy for each of the types (U, K), (K, U), and (K, H). The (U, K) and (K, U) policies give an upper bound of the achievable performance of (U, H) and (H, U) policies, respectively. The scheduling policies used in this chapter are described below:

1. Earliest Completion Time (ECT) (K, K) : The ECT policy assigns each task to the resource (cluster or processor) that leads to the earliest completion time possible. If the resource is a cluster, it also takes into account the cluster's queue when computing the earliest completion time. It is a Gantt chart-based scheduling policy [39].

2. Fastest Processor Largest Task (FPLT) (K, K) : The FPLT policy assigns the largest task to the fastest processor available.

3. **Round-Robin with Replication (RR)** (U, U) : The RR policy first assigns all the tasks to processors, in the initial order of the eligible set. After finishing all tasks in the eligible set, it replicates tasks at most once on the resources that become available, in round-robin order.

4. **Work Queue with Replication (WQR)** (U, U) : The Work Queue with Replication policy differs from RR in that it can replicate tasks several times instead of only once. The number of replicas is appended to name of the scheduling policy, e.g., WQR-1 replicates tasks once (and is identical to RR).

5. **Dynamic Fastest Processor Largest Task (DFPLT)** (H, K) : The DFPLT policy assumes that the resource performance is dynamic over time. On the completion of a task, the performance of the resource on which the task was executed is (re-) computed, and the resource receives a performance rank. This policy assigns the largest task to the resource with the highest performance rank. In this work DFPLT is exactly the same as FPLT since we do not incorporate a performance model for resources in our simulation model.

6. **ECT with task runtime Prediction (ECT-P)** (K, H) : The ECT-P policy operates similarly to ECT, but uses predicted instead of real task runtime values. We refer to Chapter 7 for the details of the prediction scheme that we use with this policy.

7. **Shortest Task First with Replication (STFR)** (U, K) : The STFR policy always assigns the shortest task first. After finishing all tasks in the eligible set, it replicates tasks at most once on the resources that become available, in round-robin order.

8. **Fastest Processor First (FPF)** (K, U) : The FPF policy assigns the tasks in the initial order of the eligible set. Each task is assigned to the fastest available processor.

The information set can be extended to more dimensions than just two. However, the two selected pieces of information already foster non-trivial customization, e.g., the execution time of a task can be extended to include the job setup and removal. If the job execution model of the system does not allow for the decoupling of job data from the job execution, as is the case for many cluster managers used in practice, the execution time of a task can also include the data transfer time to/from the execution place. Similarly, the execution time can include the setup of a virtual environment that is needed for executing a job.

5.2 The workload model for bags-of-tasks

In this section we present the workload model for BoTs in multicluster grids that is proposed in [97, 101], and that we use in this study.

Table 5.2: Characteristics of the seven grid traces used to validate the BoT workload model.

	System		Trace	
Trace ID	Name	Size [CPUs]	Duration [Years]	Size [tasks]
T1	DAS-2	400	1.5	1.1M
T2	Grid5000	∼2500	2.5	1.0M
T3	NGS	378	3	0.6M
T4	AuverGrid	475	1	0.4M
T5	SHARCNET	6,828	1	1.1M
T6	LCG	24,515	0.03	0.2M
T7	NorduGrid	∼2000	2	0.8M

Table 5.3: The parameter values for the best fits of the statistical distributions to the BoT model for the seven studied traces. N, LN, W, and G stand for the normal, lognormal, Weibull, and gamma distributions, respectively. Z stands for the Zipf distribution with two parameters: α and the number of unique users (ranks).

Trace	User	Bag-Of-Tasks			Task	
ID	Ranking	IAT [s.]	Daily Cycle	Size	ART [s.]	RTV [s.]
T1	Z(1.25,333)	W(4.06,7.91)	G(2.62,0.13)	W(1.75,2.91)	N(1.78,3.87)	W(1.64,10.21)
T2	Z(1.39,481)	W(4.27,8.42)	W(1.57,20.54)	G(2.47,1.64)	N(2.31,4.97)	N(5.54,9.56)
T3	Z(1.30,379)	W(4.94,8.48)	W(1.64,25.42)	W(2.02,1.58)	N(3.50,3.51)	G(1.79,0.29)
T4	- (see text)	W(3.87,7.33)	W(1.72,25.49)	W(1.78,1.51)	G(3.55,0.47)	N(6.41,11.73)
T5	Z(1.25,412)	W(4.17,7.65)	W(2.44,28.99)	W(1.37,1.89)	N(3.06,7.45)	W(1.93,13.65)
T6	Z(1.32,216)	W(4.05,6.48)	W(1.71,23.86)	N(1.33,2.71)	LN(1.82,0.34)	W(2.85,14.21)
T7	Z(1.36,387)	N(1.97,8.00)	W(1.62,22.18)	W(1.80,2.17)	N(2.76,9.04)	W(2.86,16.58)
Avg	Z(1.31,368)	W(4.25,7.86)	W(1.79,24.16)	W(1.76,2.11)	N(2.73,6.1)	W(2.05,12.25)

5.2.1 Model overview

The model for BoTs focuses on four aspects: The submitting user (Section 5.2.2), the BoT arrival patterns (Section 5.2.3), the BoT size (Section 5.2.4), and the intra-BoT (task) characteristics (Section 5.2.5).

Seven grid workload traces from the Grid Workloads Archive (GWA) are used to validate the BoT model[1]. Table 5.2 summarizes the characteristics of the seven systems and their traces.

The real (trace) data corresponding to each of the characteristics are fitted to the following candidate distributions, each of which has low complexity [69] and is used extensively in the analysis of computer systems: exponential, hyper-exponential, normal,

[1]The GWA makes (anonymized) grid workload traces available to researchers at: http://gwa.ewi.tudelft.nl/

log-normal, gamma, and Weibull.

The fitting process uses the Maximum Likelihood Estimation (MLE) method [4], which delivers good accuracy for the large data samples specific to workload traces. Then, goodness-of-fit tests are used to assess the quality of the fitting for each distribution, and to establish a best fit for each of the model parameters. For each candidate distribution with the parameters found during the fitting process, the hypothesis is formulated that the data are derived from it (*the null-hypothesis* of the goodness-of-fit test). The Kolmogorov-Smirnov test (KS-test) [133] is used for testing the null-hypothesis. The KS-test statistic D estimates the maximal distance between the CDF of the empirical distribution of the input data and that of the fitted distribution. The null-hypothesis is rejected if D is greater than the critical value obtained from the KS-test table. The KS-test is robust in outcome (i.e., the value of the D statistic is not affected by scale changes, like using logarithmic values). The KS-test has the advantage over other traditional goodness-of-fit tests, like the t-test or the chi-square test, of making no assumption about the distribution of the data. The KS-test can disprove the null-hypothesis, but *cannot* prove it. However, a lower value of D indicates better similarity between the input data and data sampled from the theoretical distributions.

For each model characteristic, the candidate distribution with the lowest D value is selected for each workload trace; the parameters of the best fit distributions are recorded as an instance of the model of the respective trace. The selected distributions for each trace and their parameters are depicted in Table 5.3. Trace T4 does not include user information.

The model produces parameters for an "average system" as the system that has the average properties of the seven systems considered in this work. Using the average system properties we can generate synthetic yet realistic traces, without using a single real system as a reference. The average system properties are built as follows. For each model characteristic, a candidate distribution that has the lowest average D value over all seven traces is selected as the average system fit. When for two candidate distributions the difference of their D values is below 0.01, the distribution closest to the average system fit is selected. The data for each trace are then fit independently to the candidate distribution, resulting in a set of best fit parameters. The parameters of the average system represent the average of this set. The 'Avg' row in Table 5.3 presents the parameters of the average system.

5.2.2 Submitting user

The Zipf distribution was fitted to the user ranking based on their relative job submission frequencies. First, the users are ranked based on number of submitted jobs in descending order (the lowest rank is equal to the number of unique users in the trace). The probabili-

ties associated with each rank are calculated by dividing the number of submitted jobs for each user of that rank by the total number of jobs in the trace.

5.2.3 BoT arrival patterns

The BoT arrival patterns are modeled in two steps: first the inter-arrival time (IAT) between consecutive BoT arrivals during peak hours, and then the IAT variations caused by the daily submission cycle.

Similar to the results in [98], the hours between 8AM and 5PM are found to be "peak hours", with significantly more arrivals than during the rest of the day. Only the data for BoTs arriving during the peak hours are considered when modeling the IAT. According to established modeling practice [137], a logarithmic transformation with base 2 is applied to these data to reduce the range and the effect of extreme values; this does not affect the quality of the data fitting. The Weibull distribution is selected as the average system fit and as the best fit for six of the seven traces.

The daily cycle is modeled similarly to [137]. First, the day is split into 48 slots of 30 minutes each, then the number of BoT arrivals during each of the 48 slots is counted, and finally this data set is fitted against the candidate distributions. The Weibull distribution is selected as the average system fit.

5.2.4 BoT size

Similarly to IAT modeling, a base-two logarithmic transformation is applied before fitting to the batch size (i.e., the number of tasks in a BoT). The Weibull and the normal distributions are tied, followed closely by gamma; the Weibull distribution is selected as the average system fit due to the higher number of best fits for individual traces: five against one.

The average BoT size per trace is between 5 and 50, while the maximum BoT size can be on the order of thousands. Depending on size, nine classes of BoTs are defined: of size 2-4, of size 5-9, of size 10-19, of size 20-49, of size 50-99, of size 100-199, of size 200-499, of size 500-999, and of size 1000 and over. In addition, a Small (Medium) class is defined encompassing the BoTs that fall in the classes 2-4 and 5-9 (10-19 and 20-49).

5.2.5 Intra-BoT characteristics

The modeled intra-BoT characteristics are the average task runtime (ART) and the task runtime variability (RTV), both measured in seconds. Similarly to IAT modeling, a base-two logarithmic transformation is applied before fitting to the task runtime. The normal distribution is the average system fit and the best fit for five of the seven traces.

The intra-BoT task runtime variability is defined as the variance of runtimes of the tasks belonging to the same BoT. The Weibull distribution is the average system fit and the best fit for four of the seven traces.

5.3 Experimental setup

This section describes the setup of the experiments in Section 5.4.

5.3.1 The simulated environment

The experiments are performed in a simulated environment encompassing two multi-cluster environments, the DAS-3 and Grid'5000 grids, for a total of 20 clusters and over 3500 processors, which is motivated by the current work on inter-operating the corresponding real environments. For our simulations, we have extended our DGSim tool [104] with the task-selection and task-scheduling policies described in Section 5.1. In addition, we have extended DGSim to support heterogeneous processing speeds; the processing speed of the resources used in simulation correspond to the SPEC [190] values of the real DAS-3 and Grid'5000 resources, and their relative performance ranges between 1.0 and 1.75.

5.3.2 The performance metrics

To assess the performance of BoT scheduling in multicluster grids we use the following metrics:

Makespan (MS), which for a BoT is defined as the the time elapsed from its submission to the system until the completion of its last task.

Normalized Schedule Length (NSL), which for a BoT is defined as the ratio of its Makespan and the sum of its tasks' runtime on a reference processor. The NSL is the extension of the slowdown used for jobs in traditional computing systems [73]. Lower NSL values are better, in particular NSL values below 1 are desired.

5.3.3 The workloads

Each of the 20 clusters of the combined system receives an independent stream of jobs (input workload). The input workloads used in our experiments are either one month-long traces collected from the individual grids starting at identical moments in time (*real*

traces), or synthetic traces that reflect the properties of grid workloads (*realistic traces*)[2]. The need for realistic traces is twofold. First, traces coming from one system cannot be used unmodified on a different system [67, 79] (e.g., the job submission depends on the original circumstances); the seven traces used in this work originate from seven different grids. Furthermore, modifying the real traces (e.g., by scaling or duplicating their jobs) may lead to input that does not actually represent a realistic trace; scheduling results are highly sensitive to such changes [67]. Second, given the size of the explored design space (see Section 5.4), performing the experiments for each real trace becomes unmanageable. The use of real traces is required for validation purposes, to give evidence that results obtained with real and with realistic traces are similar.

Unless otherwise noted, the realistic traces use the parameter values of the average system given in row "Avg" of Table 5.3. Our workloads have between 20,000 and 175,000 tasks, with an average of over 60,000 tasks (the median is over 64,000 tasks).

5.3.4 Simulation assumptions

We have made the following five assumptions in the experiments. First, we assume that the network between the clusters is perfect and has zero latency. Second, because all tasks are sequential, all sites employ the FCFS policy, without backfilling. Third, multi-processor machines (which do occur in the DAS-3 and Grid'5000) behave and can be used as sets of single-processor machines without a performance penalty. Fourth, all load arrives directly at the RMs of the resource management architecture which is in place, with no jobs bypassing those RMs. Finally, we assume that there are no resource failures, because in light of the model for cluster-based grids [98] and of the model for desktop grids of Kondo et al. [123], an environment with resource failures is equivalent to a smaller environment, provided that the average resource availability duration is not lower than the runtime of the jobs.

Table 5.4: The design space coverage of the experiments presented in this section. The characteristics in bold indicate the main focus of each section.

Section	Res.Mgmt. Architecture	Selection Policy	Scheduling Policy	Workload Characteristics	System Load	Information Inaccuracy
Section 5.4.1	`csp`	`S-T`	**all**	real, realistic	∼25%, 20-95%	no
Section 5.4.2	`csp`	`S-T`	all	**synthetic**	60%	no
Section 5.4.3	`csp`	`S-T`	all	realistic	60%	**yes**
Section 5.4.4	`csp`	**all**	`FPLT`	realistic	20-95%	no
Section 5.4.5	**all**	`S-T`	`FPLT`	realistic	20-95%	no

[2]We use throughout this chapter the formulation *realistic traces* in place of *realistic synthetic traces* to accentuate the difference between the traces used in this work and the unrealistic synthetic traces used in previous studies.

5.4 The performance of bags-of-tasks

In this section we present an investigation of the performance of BoTs in multicluster grids. We cover with over 1200 trace-based simulations a design space with five axes and more than 2 million points: We consider 7 task-scheduling policies, 7^P tuples of values describing the workload (with P the number of parameters in the workload model and 7 the number of values of each), 2 values for information inaccuracy (yes or no), 8 task-selection policies, and 3 resource management architectures; the exploration further uses 3 types of workloads (real, realistic, synthetic), for 7 system loads. To explore this design space efficiently, we assess in the subsections below in turn with a set of experiments for each parameter the impact on performance of varying the parameter along one of the axes of the design space; Table 5.4 shows an overview of what each experiment covers.

Most of the results present average values of the BoT MS and/or NSL metrics under various system loads, and as such are mostly useful to the system administrator or to the user submitting a workload with characteristics closely match those of the average workload. However, Sections 5.4.1 and 5.4.2 present detailed results for different BoT sizes and task runtimes; thus, they are also useful to understanding the performance perceived by users whose workload characteristics are significantly different than the average workload.

5.4.1 The impact of the task scheduling policy

In this section we assess the impact on performance of the scheduling policy using the setup described in the first line of Table 5.4. Our main findings are summarized below.

Small and Medium-Sized BoTs have a high NSL even under low load

We simulate the task-scheduling policies in a `csp/S-T` system under real load of 25%. Figure 5.1 shows that even with this low load, under `csp/S-T` the BoTs of sizes 5-9 and 10-19 have a higher than average NSL, while the average MS increases monotonically with the BoT size. This indicates that their users get higher than expected wait times for their tasks compared to other system users. The main beneficiaries are the users submitting very small (size 2-4) or very large BoTs (size over 200).

Comparison of policies with different information types

Figure 5.2 shows the performance of the scheduling policies under realistic load varying between 20% and 95%. The (K, K) and the (K, U) policies have the best balance

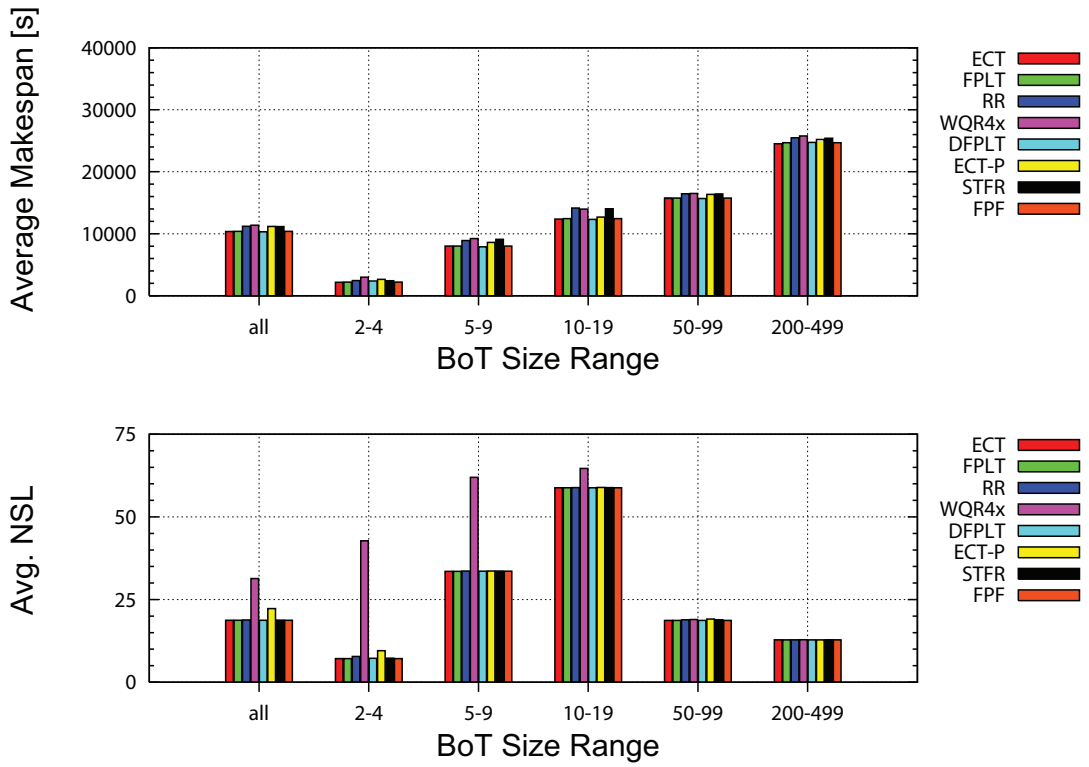


Figure 5.1: The performance of the scheduling policies in multicluster grids with csp/S-T, under real load.

between MS and NSL. In general, ECT (K, K), FPLT (K, K), and FPF (K, U) perform better than the other policies. As the policies that include unknown (U) information models rely on the quality of their uninformed heuristic, their performance ranges from surprisingly good to surprisingly poor. The (U, K) policy STFR has the best overall NSL, but average MS. The (K, U) policy FPF has the lowest overall MS, but poor NSL. The (U, U) policy RR has a good MS, but poor NSL. The WQR4x has poor MS and NSL, especially at high load. The performance of the ECT policy worsens when the task runtime estimations are inaccurate (ECT-P), nevertheless, the performance of ECT-P is better than especially the (U, U) policies.

5.4.2 The impact of the workload characteristics

In this section we assess the impact on performance of the workload characteristics using the setup described in second line of Table 5.4. For each such characteristic, we generate a synthetic workload imposing a system load of 60% using the workload model described in Section 5.2, so that the values of the characteristics have the desired statistical properties (e.g., a median BoT size of 25). Our main findings are summarized below.

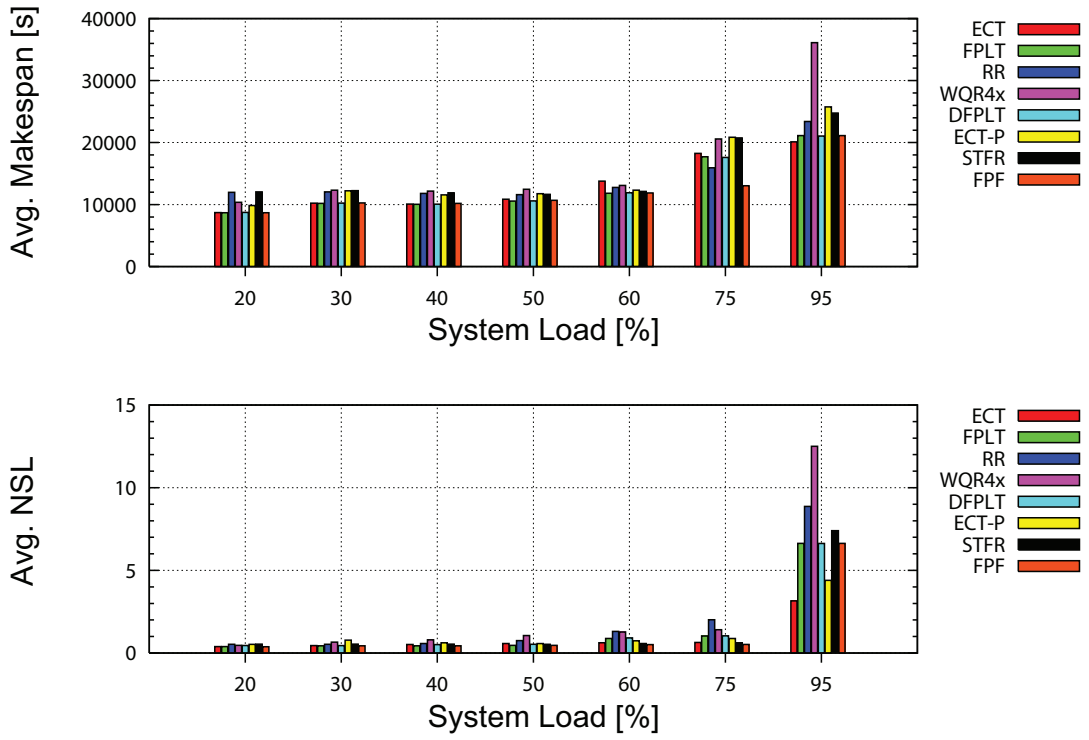


Figure 5.2: The performance of the scheduling policies in multicluster grids with csp/S-T, under realistic load.

Burstiness leads to poor performance in csp/S-T (Figure 5.3) We vary the BoT arrival pattern with as possibilities a daily cycle based on the workload model (Realistic), the pattern with all BoTs arriving at the beginning of the simulation (”All at T=0”), and the pattern with all BoT interarrival times being equal (Evenly Spread). Figure 5.3 shows that the extreme of burstiness (”All at T=0”) leads to much higher average MS and NSL values when compared to the other two patterns. The policies have similar performance results with Evenly Spread and Realistic case.

Impact of the BoT size (Figure 5.4) From the realistic workload model presented in Section 5.2, we vary the BoT size to achieve median values between 10 and 50. Figure 5.4 shows that the MS increases with the increase of the median of the BoT size. Relative to a BoT median size of 10, the MS increase ranges from 37% (35%) for ECT (FPLT) to over 80% for WQR4x, which indicates that for some policies, the average MS is less dependent from the BoT size than for others. The average NSL decreases for STFR with the increase of the BoT size; the other policies do not exhibit this performance improvement trend.

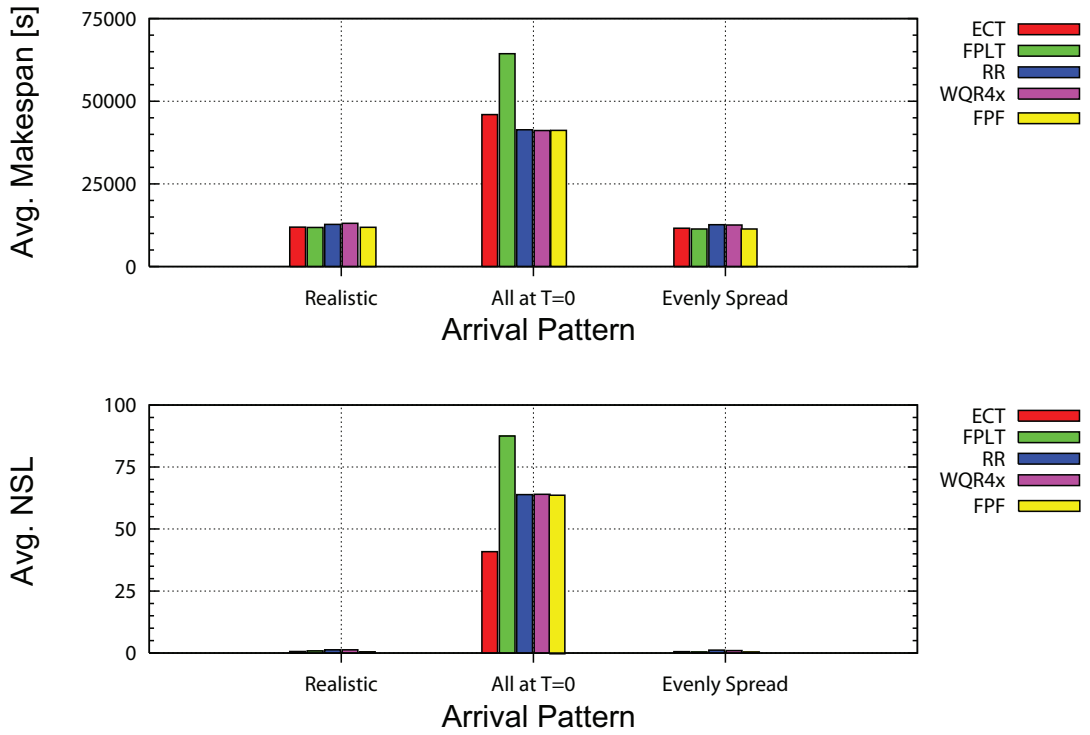


Figure 5.3: The performance of the scheduling policies in multicluster grids with `csp`/`S-T`, under various BoT arrival patterns.

Longer tasks lead to better NSL (Figure 5.5) From the realistic workload model proposed in Section 5.2, we vary the task runtime to achieve mean values between 3 minutes and 3 hours. Figure 5.5 shows that the NSL tends to decrease with the increase of task runtime, since for each BoT the effect of task inter arrival time on the NSL diminishes with the increase of task runtime.

5.4.3 The impact of the dynamic system information

In this section we assess the impact of various inaccuracy values under the assumption of null overall inaccuracy, that is, we make the optimistic assumption that while any individual estimation may be highly inaccurate, the average estimation inaccuracy is 0%.

Under null overall inaccuracy, accurate per-task information is not needed to schedule well (This finding may lead to new types of predictors for BoT scheduling that would be useful in the context of independent tasks). We first set the maximum inaccuracy I to a value between 0% (perfect information) and 10000% (high inaccuracy). Then, for each task runtime estimation we sample the estimation inaccuracy E from the uniform

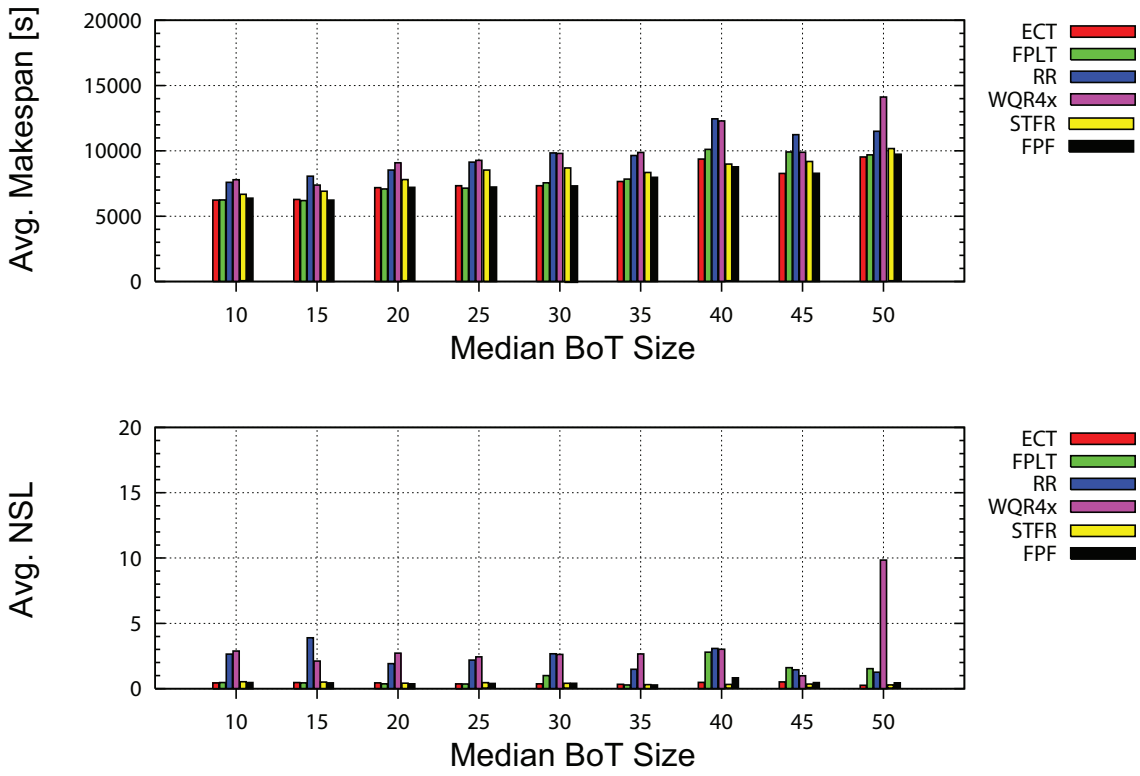


Figure 5.4: The performance of the scheduling policies in multicluster grids with csp/S-T, for various median BoT sizes.

distribution $[-I, +I]$; the task runtime is set to $\max(R + (E/100) \times R, 1)$, where R is the actual task runtime, and any task is at least 1 second long. Figure 5.6 shows that in general, under null overall inaccuracy the MS and NSL vary little with the increase of inaccuracy. We attribute this result to the moderate system load imposed on the system in the experiment, i.e., 60%, and to the independence of tasks in BoTs.

5.4.4 The impact of the task selection policy

In this section we assess the impact of the task-selection policy on the system performance. We use the setup described in the fourth line of Table 5.4: a csp resource management architecture, the FPLT task-scheduling policy, and realistic workloads that subject the system to a load between 20% and 95%. Our main findings are described below.

The task-selection policy is important only in busy systems Figure 5.7 shows that for loads up to 50%, the performance of the system is almost identical for the seven task-selection policies. This is not surprising: with a (K, K) scheduling policy, the

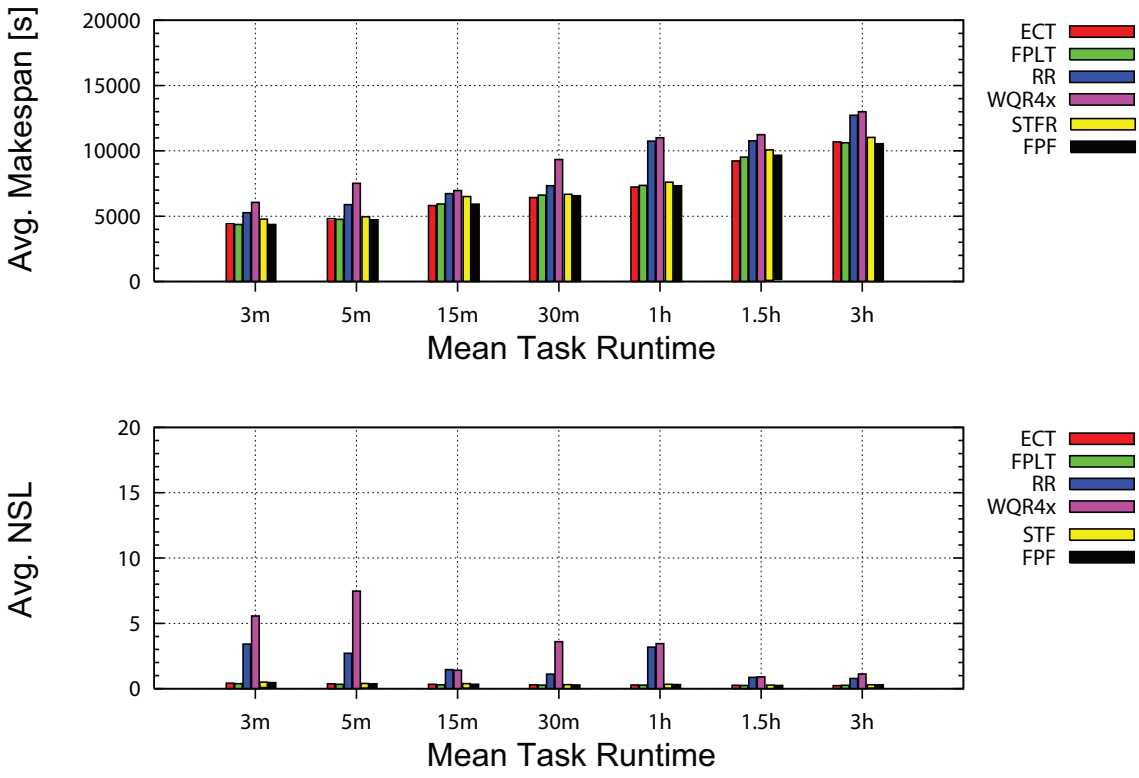


Figure 5.5: The performance of the scheduling policies in multicluster grids with csp/S-T, for various mean task runtimes.

system resources are harnessed efficiently when much spare capacity exists. The fact that FLTP does not use replication is also important in establishing a load of 50% as the threshold beyond which the task-selection policy does have an impact on performance. With a policy that does use replication, this threshold would be lower, in proportion to the average number of replications per task. Figure 5.7 also shows that the task-selection policy has an important impact on performance for loads of 50% and higher; this impact increases with the load.

S-BoT has much better performance than the other task-selection policies Figure 5.7 depicts the NSL (MS) of all the task-selection policies for various system load. For loads of 60% and higher, the average NSL (MS) for S-BoT is up to 16 (2) times lower than the average NSL (MS) of the other policies. In particular, S-BoT outperforms the task-selection policy most commonly used in practice, which is S-T. We attribute the differences to the design of the S-BoT policy, which greatly favors the small BoTs that are common in the workloads of grids. This is confirmed by the set of columns corresponding to 95% load in Figure 5.7.

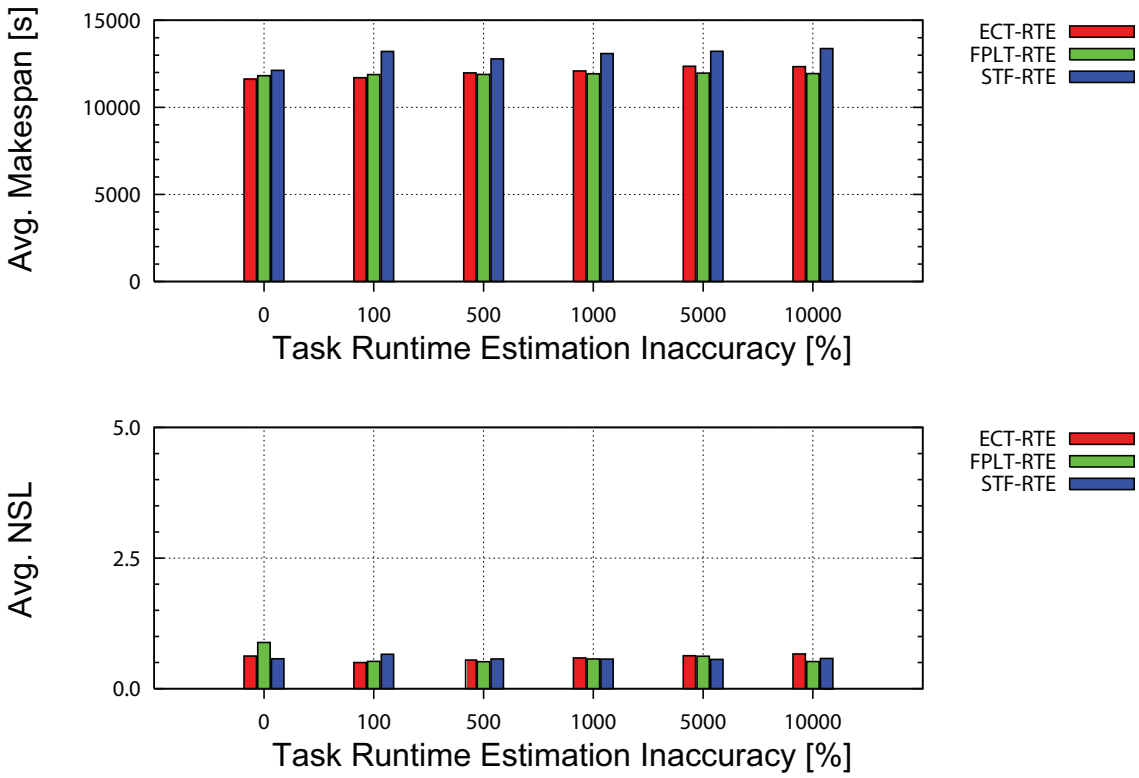


Figure 5.6: The performance of the scheduling policies in multicluster grids with csp/S-T, when the inaccuracy of task runtime estimations varies.

System fairness costs: accounting system or performance (Figure 5.7) Introducing fairness into a resource management system in general reduces the aggregate performance of the system. We compare the four task-selection policies studied in this chapter that consider fairness: S-U-T, S-U-BoT, S-U-GRR, S-U-RR. The first three of these may all lead to one user blocking the system for a long time regardless of what the other users do (e.g., when sending one or more large BoTs and getting selected); S-U-RR does not suffer from this problem if all the users submit equally large BoTs. While system blocking is a possibility, our study shows that this does not happen in practice often enough to be significant. From the four policies, S-U-T performs the best. The ordering of BoTs by arrival time employed by S-U-BoT does not lead to better performance. The round-robin ordering of users employed by S-U-GRR, which does not require an accounting system to be present, leads in turn to up to 15% higher NSL than S-U-T. Finally, S-U-RR, which also does not require an accounting system, has up to 45% higher NSL values than S-U-T. To conclude, to have fairness a system must either setup an accounting system, or it will pay in lower performance.

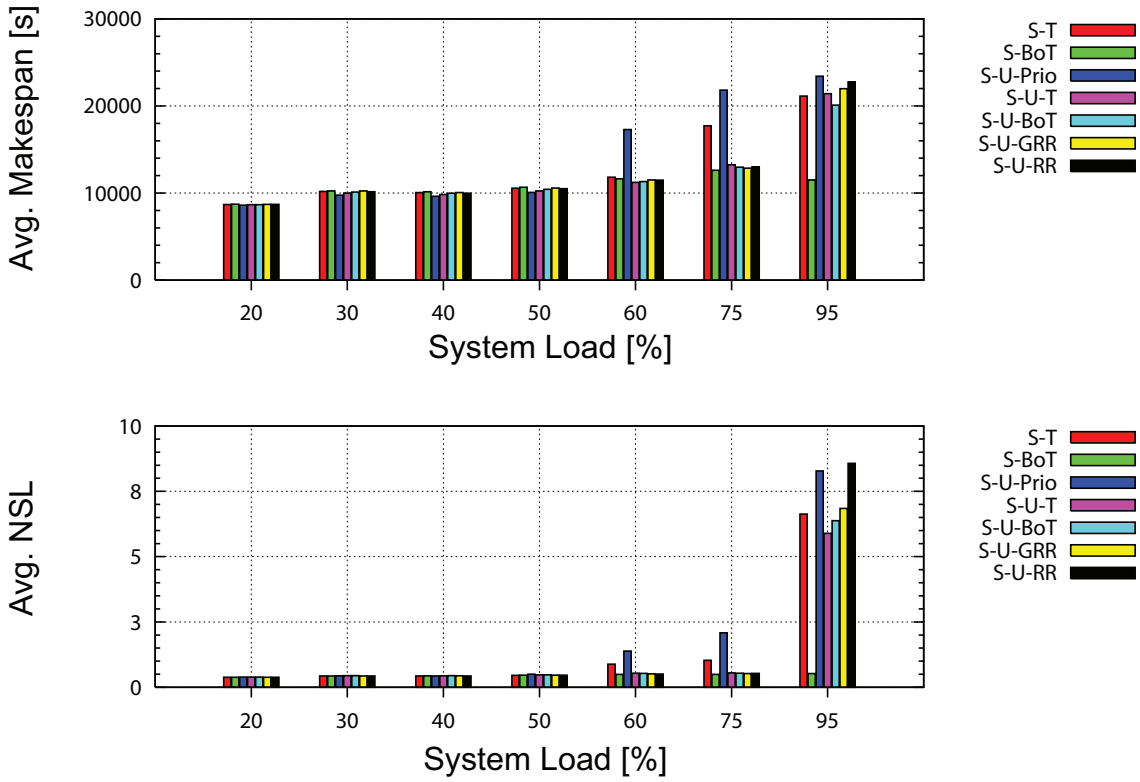


Figure 5.7: The performance of the task selection policies in multicluster grids with csp/FPLT, under realistic load.

System QoS costs: level vs. performance (Figure 5.7) Offering guarantees about the response time of the submitted tasks is expensive in terms of performance in a system that does not support advance resource reservation. We compare the three task-selection policies presented in this chapter that support QoS: `S-U-Prio`, `S-U-GRR`, `S-U-RR`. `S-U-Prio` guarantees that the tasks of the user with the highest priority from the users currently having queued tasks will be selected next. `S-U-GRR` guarantees that a user's task will be selected for execution at most n rounds after submission, where n is the total number of users in the system, both with and without queued jobs. `S-U-RR` guarantees that at least one of a user's queued tasks is selected in the next n rounds; the rounds of `S-U-RR` are on average much shorter than those of `S-U-GRR`, as `S-U-RR` selects only one task per round. `S-U-Prio` and `S-U-RR` have similar performance. Compared to `S-U-GRR`, their performance is lower, by up to 20%. To conclude, there is a trade-off between the weak QoS guarantees but higher performance of `S-U-GRR`, and the stronger QoS guarantees but lower performance of `S-U-Prio` and `S-U-RR`.

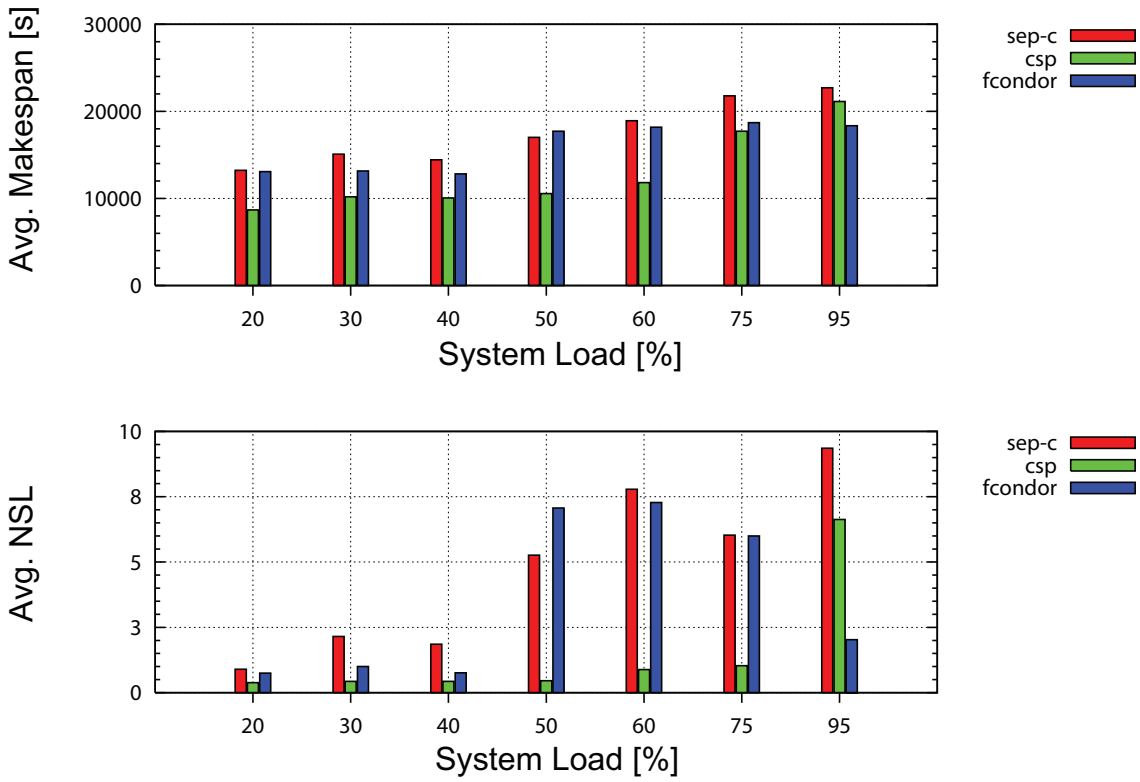


Figure 5.8: The impact of the system resource management architecture on performance under realistic load, with the S-T (FPLT) selection (scheduling) policies.

5.4.5 The impact of the resource management architecture

In this section we assess the impact of the resource management architecture on the system performance. We use the setup described in Table 5.4 (i.e., an S-T selection policy, and the FPLT scheduling policy), and realistic workloads that subject the system to a load between 20% and 95%.

Centralized, separated, or distributed? (Figure 5.8) The centralized policy csp achieves the best performance. Conversely, the independent clusters policy sep-c achieves the worst performance. The distributed architecture fcondor exhibits mixed NSL performance results: for loads below 50% its performance is similar to that of sep-c, and, for loads above and including 50% its performance is similar to that of sep-c. We observe that for loads above and including 50% fcondor and sep-c cannot complete all tasks; for the 95% load they complete only 44% and 53% of the tasks, respectively. We attribute the inability of fcondor to complete more tasks to its fostering of ”natural competition”: most of the tasks that would get blocked in a central architecture (until other tasks are finished) are submitted in the distributed architecture to

the clusters, where they compete with each other for occupying the idle processors.

5.5 Related work

This work stands at the intersection of two directions of research: design of BoT scheduling policies, and design of complete BoT scheduling systems. We have discussed throughout the text the research most closely related for design of BoT scheduling policies [39, 82, 130]. The policies have been previously evaluated through simulation in independent clusters environments (i.e., `sep-c` in Section 5.1) using a per BoT scheduling algorithm (i.e., algorithm `S-BoT`); the workload did not resemble the workloads of real grids. Closest to our work, Casanova et al. [39] compare the performance of several BoT scheduling policies in a `sep-c` environment. They also present one class of selection policies (`S-BoT`). In comparison with these results, our work focuses on the systematic evaluation BoT scheduling in multicluster grids (resource management architecture, scheduling algorithm, scheduling policy), and on using realistic workloads.

In contrast to our simulation-based approach, the theory of divisible loads [17] proposes mathematical analysis tools to assess the performance of various BoT scheduling algorithms for several distributed and centralized resource management architectures [13, 14]. However, work using this theory [13, 14, 36, 85] does not consider the information policies, most of the selection policies, and many of the resource management architectures considered in this work; they also do not consider realistic workloads.

5.6 Summary

In this chapter we have performed a realistic and systematic investigation of the performance of bags-of-tasks scheduling in multicluster grids. We first proposed a taxonomy of scheduling policies that focuses on information availability and accuracy, and we mapped to this taxonomy several task scheduling policies three of which have not been investigated previously. Then we explored the large design space of bags-of-tasks scheduling in multicluster grids along five axes: the task selection policy, the workload, the information policy, the task scheduling policy, and the resource management architecture.

We conclude the following. The task scheduling policies that make use of the available task and resource information perform better. Although the task-selection policy is important only in busy systems, the **Select-BoT** policy has much better performance than the other task selection policies. In order to achieve fairness while preserving the performance, a task selection policy should account the past resource usages of the users. Among the task selection policies that consider fairness, the **Select-User-Tasks** policy yields the best performance. In addition, we find that under null overall inaccuracy, that

is, our assumption that while any individual task runtime estimation may be highly inaccurate, the average inaccuracy is 0, accurate per-task runtime information is not needed to schedule well. Finally, in terms of the resource management architecture, the centralized policy (**csp**) achieves the best performance. Conversely, the independent clusters policy (**sep-c**) achieves the worst performance.

Chapter 6

The performance of scheduling workflows in multicluster grids

For convenience and cost-related reasons, scientists execute scientific workflows [19, 60] in distributed large-scale computational environments such as multicluster grids. However, executing scientific workflows in grids is a dynamic process that raises numerous challenges. One of the most important of these is scheduling with incomplete or dynamic information about the workflows and the resource availability—the runtimes of, and the amounts of data transferred between workflow tasks may be unknown a priori, and the other grid users may impose a background load on the grid resources. It is the purpose of this chapter to present a comprehensive and realistic investigation of the performance of a wide range of dynamic workflow policies in multicluster grids.

A large body of work on workflow scheduling already exists. However, much of this work focuses on parallel computing environments [125], which are very different from grids, or considers static scheduling methods in which all tasks of a workflow are mapped to resources before its execution starts [20, 60, 143]. Recent research [128, 129, 213, 214] does address adaptive approaches in which scheduling decisions are revised at runtime, but it assumes the availability of perfectly accurate information about the workflow tasks and the system resources, which is not true for dynamic systems such as grids (see Chapter 7 for a thorough discussion of this topic). Moreover, few research results have been obtained for the realistic situations in which multiple workflows are submitted simultaneously to the grid, and when there exists contention for the system resources caused by non-workflow (background) system load. Finally, previous investigations have largely been based on either simulations or on real system experiments, but not on both.

In this chapter, we first introduce a framework for dynamic workflow scheduling that includes a novel taxonomy of dynamic workflow scheduling policies based on the amount of (dynamic) information used. We further map to this taxonomy seven workflow scheduling policies that cover the full spectrum of dynamic information use. Secondly, we inves-

tigate the performance of these seven policies in various realistic and dynamic scenarios using both simulations and experiments in real systems. One of the findings from our real-system experiments that does not show in the simulations is that the performance may be severely degraded because the head-nodes of the grid clusters may become overloaded due to the large number of workflow tasks and file transfers they have to manage. To solve the problem of head-node overload, we analyze the performance of task throttling, that is, of limiting the per-workflow number of tasks concurrently present in the system. Our results indicate that task throttling can prevent head-node overload while not unduly decreasing the performance.

The remaining part of this chapter is organized as follows. In Section 6.1, we present the framework for dynamic workflow scheduling, which includes the workflow model, the system model, the scheduling policies, and the task throttling mechanisim. In Section 6.2, we describe the experimental setup, while in Sections 6.3, and 6.4, we present and discuss the results that we have performed in a simulated environment and in the DAS-3, respectively. Section 6.5 reviews related work on workflow scheduling in parallel computing environments and grids. Finally, Section 6.6 summarizes the chapter.

6.1 The scheduling framework

In this section we present a framework for dynamic workflow scheduling in multicluster grids.

6.1.1 Workflow model

We assume a workflow to be represented by a directed acyclic graph (DAG), in which the nodes represent computational tasks and the directed edges represent communication. We call a task having no predecessor tasks an entry task, and a task having no successor tasks an exit task; a DAG may have several entry and exit tasks. We define the size of a workflow as the total number of its tasks.

We assume that a task depends on each of its predecessors by means of a file. An output file of a task can be the input file to several successor tasks. A task can be executed only after all its predecessor tasks are completed and all the corresponding files are available on its execution site.

In this chapter, both in the simulation-based experiments and the real experiments, we use the the DAX (DAG in XML) abstract workflow description language of Pegasus [53] to represent DAGs.

Table 6.1: Mapping scheduling policies to our information availability framework. **U**, **K**, and **R** stand for information that is **U**nknown (or ignored), **K**nown a priori, and obtained at **R**untime by a policy, respectively.

Policy	Resource Information			Task Information	
	Status	Processing Speed	Link Speed	Task Execution Time	File Size
Round Robin	U	U	U	U	U
Single Cluster	R	U	U	U	U
All-Clusters	R	U	U	U	U
All-Cls. File-Aware	R	U	K	U	R
Coarsening	R	U	K	U	K
Cluster Min.	R	K	U	U	U
HEFT(-P)	R	K	K	K	K

6.1.2 Multicluster grid model

In our multicluster grid model, we assume that processors are grouped in clusters. The processors may have different performance across clusters, but within the same cluster they are homogeneous. Clusters are fully connected with network links that can be heterogeneous in terms of bandwidth. Each cluster has its own Local Resource Manager (LRM), and so its own local queue to which tasks arrive. Each LRM in the system executes the same scheduling procedure upon the arrival of new tasks and on the completion of tasks.

We consider a decentralized architecture for workflow scheduling in which each workflow submitted to the system is managed and scheduled by an individual broker/agent. This broker incorporates a workflow execution engine and employs a scheduling policy in order to map tasks to resources. The workflow engine submits tasks to the LRMs of the clusters according to some schedule, and initiates the necessary file transfers. Finally, we assume a monitoring service provides information about the status of the clusters (e.g., the numbers of idle resources and the loads).

6.1.3 Workflow scheduling policies

In this section we first propose a taxonomy for workflow scheduling policies based on the information they consider regarding resources and workflow tasks. We then map to our taxonomy seven dynamic workflow scheduling policies in most of which tasks are assigned to resources only after they become *eligible*, i.e., after all of their predecessor tasks are finished.

In our taxonomy, a particular piece of information regarding resources or workflow tasks can be either unknown (U), known a priori (K), or obtained at runtime (R). We consider the following pieces of information: for resources, their status, processing speed, and inter-cluster link speeds, and for tasks, their execution time and the sizes of their output files. Our taxonomy for workflow scheduling policies extends our taxonomy for bags-of-tasks scheduling policies (see Section 5.1.4) by considering more detailed resource and task information, and it differs significantly from previous efforts [125, 212], since their taxonomies classify policies based on algorithmic approaches rather than on the information the policies take into account.

We consider seven dynamic workflow scheduling policies, and in Table 6.1 we show how these policies map to our taxonomy. We describe these policies below in the order of increasing information usage:

1. Round Robin submits the eligible tasks of a workflow to the system clusters in round-robin order; the order of clusters is arbitrary.

2. Single Cluster maps every complete workflow to the least-loaded cluster at its submission. The load of a cluster is defined as the total processor requirement of all jobs running or queued in the cluster normalized by the cluster size. This policy executes all tasks of a workflow in the same cluster in order to avoid inter-cluster file transfers. However, this policy may increase the makespan of a workflow when the number of eligible tasks, at any moment during its execution, is larger than the number of idle processors in the cluster.

3. All-Clusters submits each eligible task to the least-loaded cluster. This policy can exploit idle resources across the grid; however, the performance may degrade due to inter-cluster file transfers.

4. All-Clusters File-Aware submits each eligible task to the cluster that minimizes the transfer costs of the files on which it depends. The size of an output file can be known only after the task that creates it is completed. This policy gives preference to the clusters with idle processors.

5. Coarsening is, in fact, a technique used in multilevel graph partitioning problems [118, 139] that iteratively reduces the size of a graph by collapsing groups of nodes and their internal edges. In each iteration, a group of nodes of the current graph is selected and is combined into a single coarser node. The edges of the original graph between nodes in the group disappear, but the edges that connect the corresponding coarse nodes remain in the new graph [117]. We use the Heavy Edge Matching (HEM) [117] coarsening technique to group tasks that are connected with heavy edges, i.e., task dependencies

that correspond to relatively large files in a workflow. After coarsening, the remaining edges are conceivably be the task dependencies that correspond to relatively small files. Our approach is to execute all the tasks of a group in the same cluster (where the first task of that group is submitted to) with the aim to minimize the cost of inter-cluster file transfers. After a workflow is coarsened, it is scheduled with the All-Clusters File-Aware policy.

6. Cluster Minimization submits as many eligible tasks as possible to a cluster until the cluster has no idle processors left before considering the next cluster. Therefore, it aims to reduce the number of inter-cluster file transfers by minimizing the number of clusters being used. It considers the clusters in descending order of their processing speeds, and hence, it gives preference to faster resources. If none of the clusters have idle processors, then the tasks are submitted to the cluster where the last task has been scheduled.

7. Heterogeneous Earliest-Finish-Time (HEFT) [196] is a commonly cited list-scheduling heuristic [53,206,208]. This policy initially orders all the tasks of a worklflow in descending order of their *upward rank* values. The upward rank of a task is calculated as the sum of the execution time and the communication time (on a reference processor and with a reference bandwidth) of the tasks that are on this task's critical path, including itself. Then in this order the policy maps each task to a processor which ensures its earliest completion time.

In this chapter, we have modified the original HEFT policy such that it maps tasks to clusters instead of processors, hence it operates at the grid level, and we have changed the static task scheduling process to dynamic scheduling. Our implementation of the HEFT policy operates as follows. Among the eligible tasks, at each step, it selects the task with the highest upward rank value and assigns the selected task to the cluster that ensures its earliest completion time. We assume that an information service runs on each cluster that responds to queries regarding the estimated start times of workflow tasks. Then the completion time of a workflow task on a cluster is estimated as the sum of the estimated start time, the estimated delay of transferring input files on which it depends, to that cluster, and the execution time of the task on that cluster.

Although we assume the execution times of the workflow tasks are known a priori, we consider two kinds of prediction information regarding the execution times of non-workflow tasks (i.e., background load due to local users; see Section 6.2.3 for details), leading to two variations of this policy. The **HEFT** policy makes use of perfectly accurate task completion time predictions. In contrast, **HEFT-P** is provided task completion time predictions that may be inaccurate; the execution times of the non-workflow tasks submitted to a cluster are predicted using the Last-2 method [197], which predicts the

execution time of a task as the average of the last two previously observed task execution times. We refer to Chapter 7 for details of this prediction scheme. The prediction service simulates the scheduling policy of the LRM with the predicted task execution times and the actual execution times of the workflow tasks that have been submitted to the cluster in order to determine when a new workflow task will start. Although perfectly accurate estimations of task information is not realistic in grid settings, we use the HEFT policy in order to observe the performance that can be achieved if such information were available.

6.1.4 Task throttling

In grids, overload conditions may occur due to the bursty nature of task arrivals, as in the case of executing large workflows. As a consequence, the execution performance of the applications may deteriorate to unacceptable levels; response times may grow significantly, and throughput may degrade. Furthermore, as we will demonstrate in Section 6.4.2, the real system may become unstable when executing large workflows, as the load on the head-nodes of the clusters severely increases due to the activity of the workflow execution engine [193], to the number of concurrent task submissions [37, 193], and to the excessive number of concurrent inter-cluster file transfers.

A common way to achieve efficient overload control in traditional distributed systems is to use admission control mechanisms [41, 108]. In grids, however, instead of dropping or rejecting tasks, it may be sufficient to delay the submission of some of the tasks, in the expectation that the sites which to dispatch them will become less overloaded at a later time. For instance, the Condor system [195] allows [1] the system administrator to limit the total number of concurrent tasks in the system. We call this mechanism, *task throttling*, and apply it on a per-workflow basis. We define the *concurrency limit* as the maximum number of concurrent tasks that are dispatched (they can be running or queued) in a multicluster grid to all its clusters and at all times during the execution of the workflow.

Task throttling may have non-trivial effects on scheduling performance. For example, although with task throttling tasks may be delayed by the workflow execution engine, this may also lead to a decrease in the amount of inter-cluster communication, since there will be fewer concurrent tasks, and they will possibly be spread to fewer clusters. Consequently, the execution performance of the workflow may improve despite the additional delay of the tasks.

[1] The Condor Manual: `http://www.cs.wisc.edu/condor/manual/v7.5/condor-V7_5_0-Manual.pdf`

6.2 Experimental setup

In this section, we present the experimental setup used both in the simulations and the real experiments.

6.2.1 Experimental environments

Our goal is to emulate, both in simulations and in practice, realistic scenarios in which we execute workflows on the DAS-3. The properties of the DAS-3 clusters are shown in Table 1.2, and the average inter-cluster bandwidth (MB/s) values are shown in Table 2.1. Other relevant details on DAS-3 have been given in Section 1.4.1.

Simulated environment

In our *computation model*, we employ the SPEC CPU benchmarks model for task execution time [190], that is, the time it takes to finish a task is inversely proportional to the performance of the processor it runs on. We assume that all LRMs of the clusters employ the FCFS policy. Once started, tasks run to completion, so we do not consider task preemption or task migration during execution.

In our *communication model*, we assume that the file transfer delay of sending a file from a predecessor to a successor task is zero if these two tasks are executed on the same cluster. If a task depends on a file created on a different cluster, we model the file transfer delay as the ratio of the file size and the bandwidth of the inter-cluster link. If a task depends on multiple files, then we model the total file transfer delay as the sum of the individual file transfer delays. This means that we consider the worst case scenario in which all file transfers are performed in series, although concurrent inter-cluster file transfers can perform better in reality [5]. Once a file is created in a cluster or transfered from another cluster, it remains in place until the whole workflow is executed; hence, we avoid redundant file transfers.

Real environment

For our experiments in the DAS-3, we have extended KOALA with workflow execution support. To this end, we have designed and implemented two components: a workflow runner (WRunner), which can be considered as a workflow execution engine, and a workflow execution service (WES), which is responsible for interacting with the grid middleware. A separate WRunner is responsible for each workflow submission; it manages the dependencies between the tasks and the scheduling of workflow tasks. In our design, we use a service-based approach for better scalability. Hence, there is a single WES on the

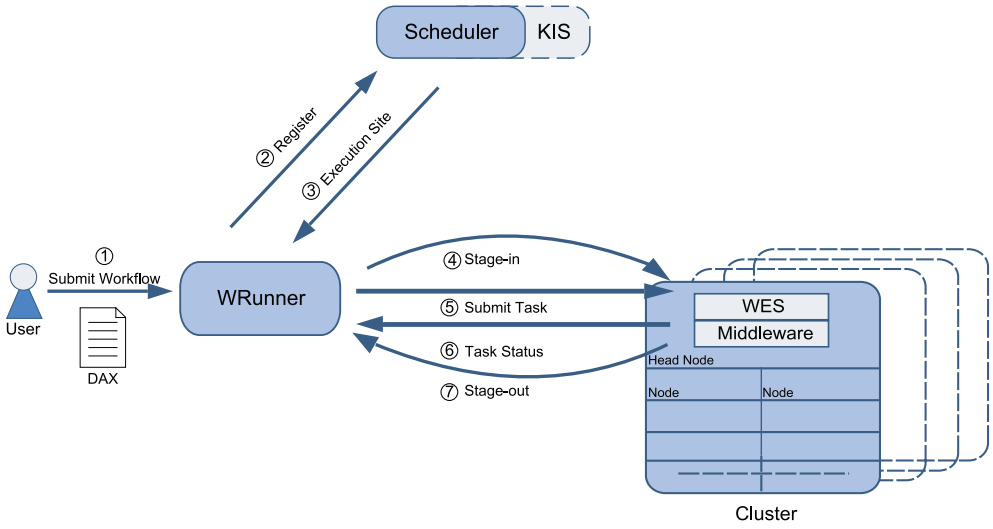


Figure 6.1: The system architecture to support workflow execution in KOALA.

head-node of each of the clusters that initiates the execution of all workflow tasks scheduled to that cluster. Figure 6.1 shows the architecture of the KOALA workflow execution environment, and the interaction between the components.

After registration to the scheduler, a WRunner obtains system-wide status information from the KOALA Information Service (KIS), and determines the execution sites of the tasks according to the scheduling policy it employs. For each task of the workflow, the WRunner copies input files and its executable to the execution site, and then delegates the execution of the task to the WES on this site. The WES is responsible for submitting the task to the middleware using the DRMAA [59] interface, and for monitoring its execution with the callbacks provided by the middleware. The WES provides the task status to WRunner upon its completion. If the task fails, the WRunner handles the failure by resubmitting the task. If the task finishes successfully, upon notification, the WRunner first transfers its output files to the submission site of the workflow, and then it updates the set of eligible tasks. The WRunner continues this process until all tasks of the workflow are completed.

Additional considerations

Although we try to model the DAS-3 multicluster grid as realistically as possible, in our simulations there are some differences between our model and the real system:

- In our simulations, we do not model the head-nodes and the resource contention that may occur on the head-nodes in the real system due to the file transfers. This resource contention and the instability it causes has a significant effect on the performance; as we demonstrate in Section 6.4.2.

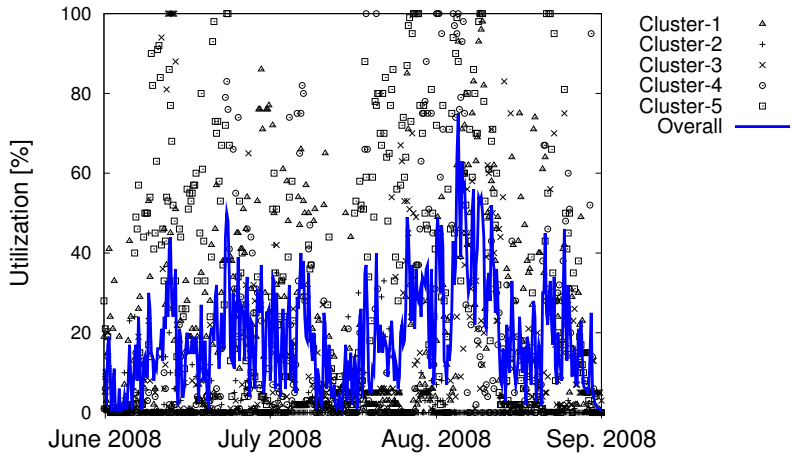


Figure 6.2: The overall utilization as well as the utilization in the individual clusters due to the background load considered in the simulation-based experiments. This load refers to the jobs submitted to the DAS-3 system during a period of four months (June-September 2008).

- In our simulations, we consider a task as finished when its execution completes; then, the eligible set of tasks is updated accordingly. But in the real experiments, we only consider a task as finished when its execution has completed and its output files have been transferred back to the submission site; and only then the eligible set of tasks is updated.

6.2.2 The workflows

We use synthetic but realistic workflow applications that are publicly available in [19]. The applications are based on four real scientific workflows: Montage, CyberShake, Inspiral, and SIPHT. These workflows are composed of several structural components such as pipeline, data distribution, data aggregation, and data redistribution. For each of these workflows, four synthetic applications are available[2], with sizes of 30, 50, 100, and 1000 tasks.

We categorize a synthetic workflow as *small*, if its size is at most 100, and as *large*, if its size is 1000. With small workflows, the number of tasks that become eligible concurrently can all fit in a single cluster in our system; however, with large workflows, more tasks may become eligible concurrently than can be accommodated by even the largest cluster. We further categorize the workflows according to their communication versus computation characteristics. Table 6.2 presents the categorization of the workflows and

[2]Available Pegasus Workflow Types: `http://vtcpc.isi.edu/pegasus/index.php/WorkflowGenerator`

Table 6.2: Categorization of the workflows.

Workload Type	Workflow Name-Size	Avg. Makespan on a Reference Cluster [s]	Avg. Total Size of the Output Files of a Workflow [MB]
wf-small1	CyberShake-30, 50,100	220	2246
wf-small2	Inspiral-30,50 Montage-100	1260	23
wf-large1	CyberShake-1000 Montage-1000	410	2866
wf-large2	SIPHT-1000 Inspiral-1000	3290	1150

their characteristics. While the workflows in *wf-small1* and *wf-large1* are communication-intensive, the workflows in *wf-small2* and *wf-large2* are more computation-intensive.

6.2.3 The workloads

In our experiments, we assume that two workloads are submitted to the system: a grid workload, which comprises the workflow applications submitted by grid users, and a local workload, which comprises the tasks submitted directly to the clusters by local users (background load). In our simulations, depending on the experimental scenario, we impose a background load together with a grid workload in order to attain realistic resource availability conditions. The background load refers to the jobs submitted to the DAS-3 system during a period of four months (June-September 2008). Figure 6.2 illustrates the system utilization of the background load. The corresponding workload trace is obtained from the Grid Workloads Archive [102]. In the simulations, the tasks that belong to the background load are submitted to the LRMs of their original execution locations. In our DAS-3 system, users may submit tasks directly to the LRMs, bypassing KOALA. We keep this non-workflow (background) load under control for performing controlled experiments in the real environment. To this end, during our real system experiments, we monitor the background load, and we maintain it between 30% and 40% in each cluster (which is not the case in our simulations).

We classify our experiments as either single workflow or multi-workflow scheduling. In the simulations of single workflow scheduling with background load, for each policy, each of the workflows is scheduled only once and the average results are presented per workload type (see Table 6.2). In the simulations with the background load, for each workload type, we generate ten traces in each of which randomly selected workflow instances arrive at six-hour intervals during a period of three (simulated) months. For each

policy-workload type pair, we run the ten corresponding experiments and present the average results. In the real system experiments (where there is always background load) of single workflow scheduling, for each policy, each of the considered workflows is executed ten times and we present the average results.

For multi-workflow scheduling, both for simulations and real experiments, several instances of the same workflow application are submitted simultaneously (the exact number varies across the experiments). We do not consider background load in the simulations of multi-workflow scheduling.

6.2.4 The performance metrics

To assess the performance of the workflow scheduling policies, we use the following traditional metrics [125]:

- The **Makespan (MS)** of a workflow in a grid is the time elapsed from its submission to the grid until the completion of its last task. For the multi-workflow scheduling experiments, in which a number of concurrent instances of a workflow are submitted to the grid, we consider the metric **Total Makespan**, which is the time elapsed from the submission of the instances until all the instances are completed.
- The **Normalized Schedule Length (NSL)** of a workflow in a grid is the ratio between its makespan and the time to execute (one of) its critical path(s).
- The **Wait Time** of a task is the time elapsed from when a task becomes eligible until the time it starts execution. It comprises two components: the File Transfer Delay **(FTD)**, which is due to waiting for input files to become available at the execution site, and the Queue Wait Time **(QWT)**, which is the time a task spends in the local queue of the cluster to which it was submitted.

In addition to these metrics, we also consider the Number of inter-cluster File Transfers per workflow **(NFT)**.

6.3 Simulated environment results

In this section we present our experimental results of single workflow scheduling and multi workflow scheduling, respectively.

6.3.1 Single workflow scheduling

We first evaluate the performance of the scheduling policies that we describe in Section 6.1.3 under various experimental scenarios. Then, we investigate the impact of

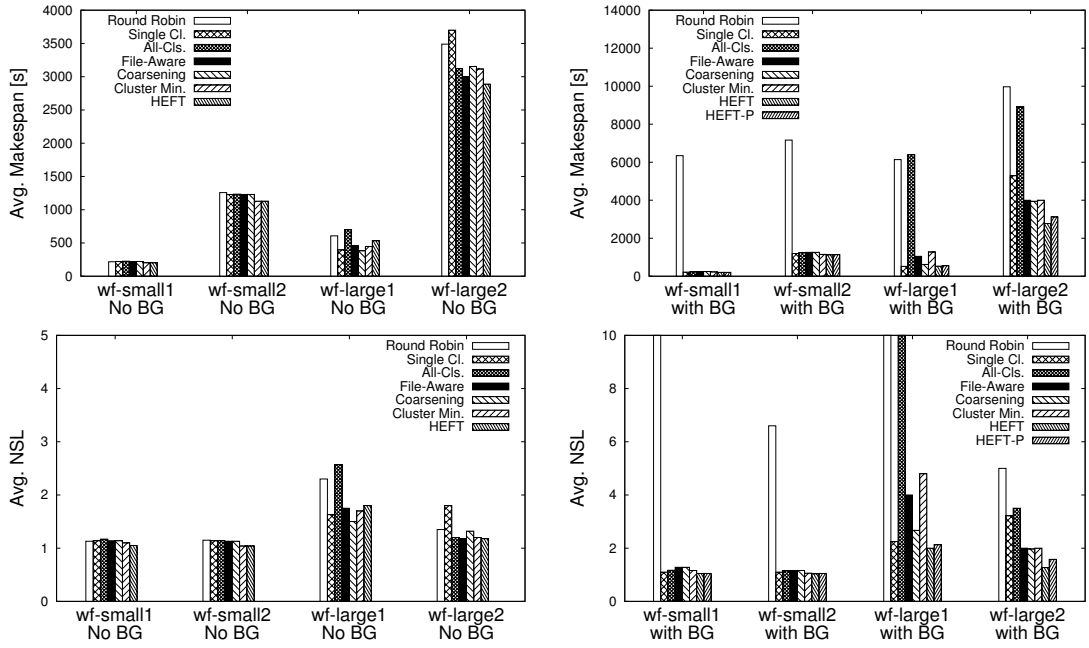


Figure 6.3: **Simulations, single workflow scheduling:** The performance of the workflow scheduling policies in terms of the average makespan and the average NSL without and with background load.

task throttling on the performance of dynamic workflow scheduling when executing large workflows.

The performance of the scheduling policies

In addition to our experimental setup (in Section 6.2), we applied the Coarsening policy only to wf-large1 and wf-large-2; the workflow sizes are shrunk to 100 and then scheduled with the File-Aware policy. For wf-small1 and wf-small2, Coarsening is exactly the same as the File-Aware policy. In addition, we applied the HEFT-P policy only when we impose background load in our experiments.

Figure 6.3 presents the performance of the workflow scheduling policies in terms of the average makespan and the average NSL, without and with background load (denoted by BG). Table 6.3 presents additional metrics such as the average task queue wait time, the average task file transfer delay, and the average number of inter-cluster file transfers performed per workflow. Finally, Figure 6.4 shows the cumulative distribution functions of the makespan and the NSL when background load is imposed (only for wf-small1 and wf-large1). Below, we present a separate discussion for each of the experimental scenarios.

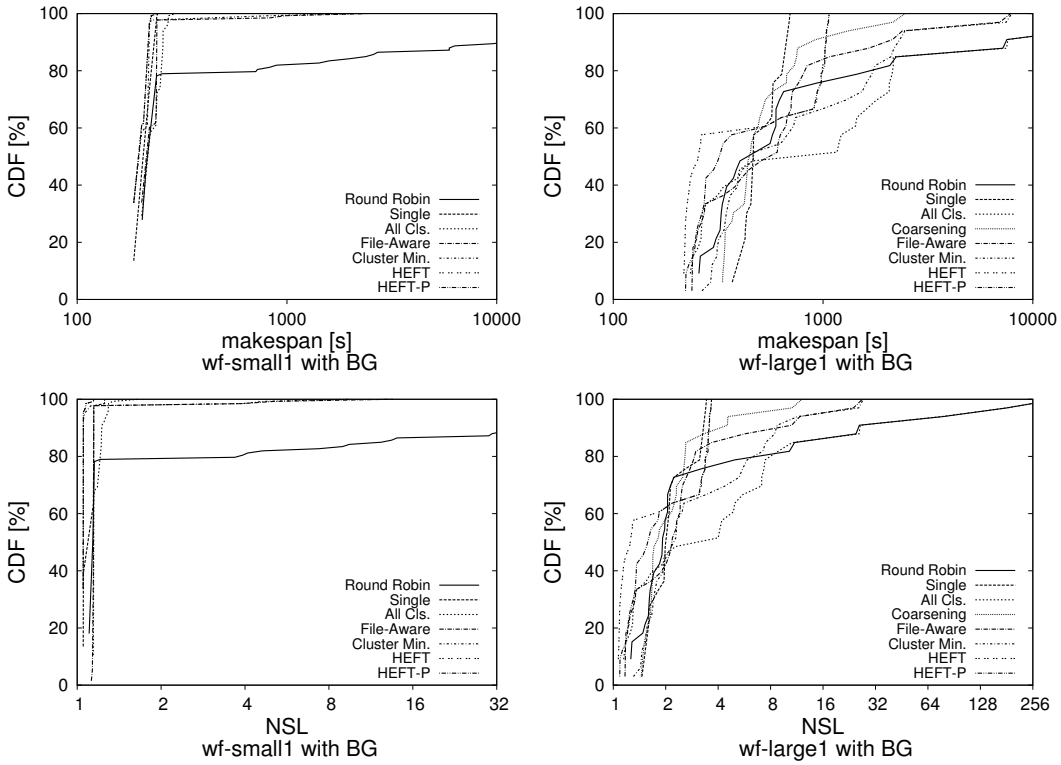


Figure 6.4: **Simulations, single workflow scheduling:** The cumulative distribution functions of the makespan and the NSL with background load. The horizontal axis has a logarithmic scale.

wf-small, without BG Load

The very first noticeable result is that the performance variation is small among the policies. The reason is that, with any of the policies, except Round Robin, tasks are executed mostly in a single cluster. When the system is not loaded, for small workflows, selecting the appropriate cluster (e.g., according to processing speed) is a good strategy to attain a better performance. Since the Cluster Minimization policy takes into account the processing speed of the clusters, it is slightly better than the other policies, and yields almost identical performance to that of HEFT, which is fed by perfectly accurate resource and task information, and which we expect to perform best in any of the scenarios.

wf-small, with BG Load

In this scenario we see that Round Robin performs much worse in comparison to the other policies. Round Robin maps tasks also to the most loaded clusters, and consequently, some tasks experience large queue wait times. As this policy distributes tasks across all clusters, it causes more inter-cluster file transfers than the other policies (see Table 6.3). The other policies perform similarly to each other, although Cluster Minimization, HEFT and HEFT-P yield slightly better performance.

Table 6.3: **Simulations, single workflow scheduling:** The performance of the policies without and with background load, in terms of the average task queue wait time (QWT [s]), the average task file transfer delay (FTD [s]), and the average number of inter-cluster file transfers performed per workflow (NFT). '-' indicates that no experiment has been performed (see text).

	Round Robin			Single Cl.			All-Cls.			File Aware		
Workload Type	QWT	FTD	NFT	QWT	FTD	NFT	QWT	FTD	NFT	QWT	FTD	NFT
wf-small1	0	1.26	25	0	0	0	0	0.4	4	0	0	0
wf-small2	0	0.5	49	0	0	0	0	0.1	10	0	0	0
wf-large1	8	1.1	460	49	0	0	3.2	1.44	366	5	0.43	243
wf-large2	7.5	0.4	778	93	0	0	0.4	0.5	637	2	0.3	354
wf-small1+BG	525	1.23	26	0.03	0	0	0	1.2	6.7	3.3	0.01	0.1
wf-small2+BG	423	0.5	51	0.3	0	0	0.5	0.1	12	3	0.01	1.1
wf-large1+BG	682	1.2	455	113	0	0	573	3.1	392	42	0.4	229
wf-large2+BG	842	0.42	756	386	0	0	598	0.5	636	79	0.4	422

	Coarsening			Cluster Min.			HEFT			HEFT-P		
Workload Type	QWT	FTD	NFT	QWT	FTD	NFT	QWT	FTD	NFT	QWT	FTD	NFT
wf-small1	-	-	-	0	0	0	0	0	0	-	-	-
wf-small2	-	-	-	0	0	0	0	0.2	15	-	-	-
wf-large1	34	0.18	39	3.3	1.3	495	4	0.6	218	-	-	-
wf-large2	9.9	0.01	18	0.4	0.5	638	1.25	0.2	301	-	-	-
wf-small1+BG	-	-	-	1.7	0.1	1.2	0	0	0	0	0	0
wf-small2+BG	-	-	-	3.2	0.01	1.05	0	0.2	15	0.4	0.2	15
wf-large1+BG	93	0.3	58	42	1.4	466	9	0.5	290	9	0.5	290
wf-large2+BG	69	0.04	25.6	146	0.68	694	7.5	0.4	441	28	0.4	501

wf-large, without BG Load

When executing large workflows, many tasks may become eligible simultaneously, which gives more room to make different scheduling decisions. Consequently, when scheduling large workflows, we observe that the performance varies considerably among the policies, and even their relative performance varies with different workloads (Figure 6.3). For the case of wf-large1, Round Robin and All-Clusters are the two worst performing policies, and Single Cluster and Coarsening are the two best performing policies. On the other hand, for the case of wf-large2, Single Cluster performs the worst, while HEFT performs the best. We attribute this difference to the communication characteristics of the workloads (see Table 6.2).

In general, we observe that both Round Robin, All-Clusters, and Cluster Minimization perform many inter-cluster file transfers (see Table 6.3). Coarsening is a trade-off be-

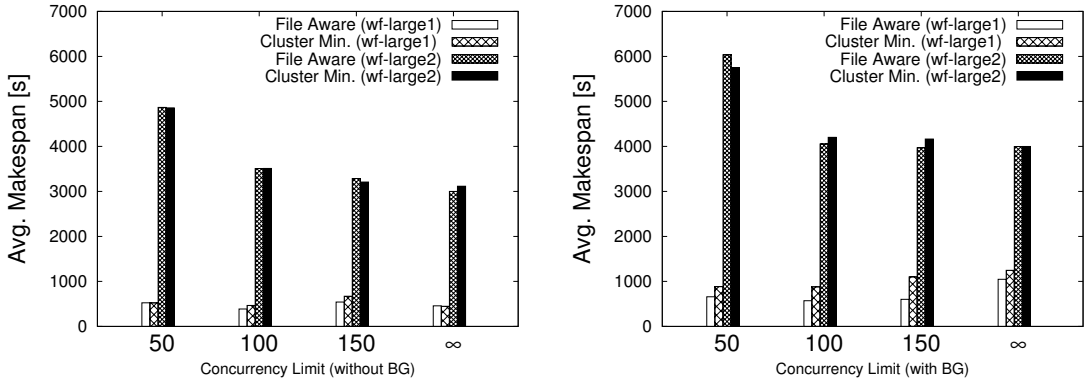


Figure 6.5: **Simulations, single workflow scheduling:** The average makespan for the File-Aware and the Cluster Minimization policies when **task throttling** is applied.

tween low inter-cluster communication and high queue wait times. Although File-Aware is better than Cluster Minimization in terms of reducing inter-cluster communication, selecting faster resources pays off for Cluster Minimization even when the heterogeneity in the system is low (the ratio of the fastest processor to the slowest is 1.25 in our settings). Nevertheless, selecting the faster clusters may not be always an advantage if such clusters are much smaller than a slower cluster when scheduling large workflows, since in such a case tasks may be distributed to more clusters, and as a result the number of inter-cluster communications may increase.

wf-large, with BG Load

When background load is imposed, we observe a substantial increase in the average makespans and NSLs, respectively. The HEFT policy outperforms the other policies for both of the workloads. However, the performance of the HEFT policy worsens when the task completion time information is inaccurate (HEFT-P). File-Aware, Coarsening, and Cluster-Minimization all yield similar performance results when scheduling wf-large2; however, Coarsening is much better than the others when scheduling wf-large1, which includes more communication-intensive workflows. As a consequence of spreading tasks to many clusters, both Round Robin and All-Clusters suffer from file transfer delays as well as from large queue wait times.

The impact of task throttling

In this section we evaluate the scheduling performance of the dynamic workflow scheduling policies when *task throttling* is applied for each workflow submitted to the system.

To conduct our evaluation we modify our default setup as follows. We only consider the workloads that contain large workflows, and run them both with and without background load. As scheduling policies we select, from the policies that perform relatively

Table 6.4: **Simulations, single workflow scheduling:** Percentage of change (Best, Worst) in the performance of File-Aware and Cluster-Minimization, when **task throttling** is applied, relative to the original performance of the policies.

Policy (Workload Type)	Change in the Makespan [%]		Change in the NSL [%]	
without BG	Best	Worst	Best	Worst
File-Aware (wf-large1)	-15	+16	-9	+25
File-Aware (wf-large2)	+17	+60	+18	+116
Cluster-Min. (wf-large1)	-4	+50	-20	+9
Cluster-Min. (wf-large2)	+11	+68	+16	+123
with BG				
File-Aware (wf-large1)	-50	-37	-40	-26
File-Aware (wf-large2)	+4	+51	+6.5	+80
Cluster-Min (wf-large1)	-30	-11	-27	-13
Cluster-Min (wf-large2)	+4	+43	+15	+75

well in the previous section, the File-Aware and the Cluster-Minimization policies. In our simulations, we do not model overload conditions for the head-nodes of the clusters; hence, we assess the impact of throttling on scheduling performance in ideal conditions. We use, in turn, three values for the concurrency limit: 50, 100, and 150. While these values are, as we show below, enough to understand the main impact of task throttling on the performance of scheduling, it is outside the scope of this work to assess the optimal concurrency limit.

Figure 6.5 shows the makespan performance of the policies for all values of the concurrency limit, as well as when no task throttling is applied. Table 6.4 presents the best and the worst performance results of the policies relative to their original performance out of all concurrency limit scenarios (50, 100, and 150). We observe that the performance that can be attained with task throttling is related both to the communication characteristics of the workflows, and to the utilization in the system (see Table 6.4). When scheduling workload wf-large1, which includes communication-intensive workflows, without background load, the makespan (NSL) performance can be improved by 15% (20%), while it can be improved by 50% (40%) when background load is imposed. On the other hand, when scheduling the workload wf-large2, which is more computation-intensive, the makespan (NSL) performance worsens for all the limit values, but the performance degradation is smaller when background load is imposed than without background load.

6.3.2 Multi-Workflow scheduling

In the simulations of multi-workflow scheduling we consider the Single Cluster, the File-Aware (without and with task throttling), the Cluster Minimization (without and with task

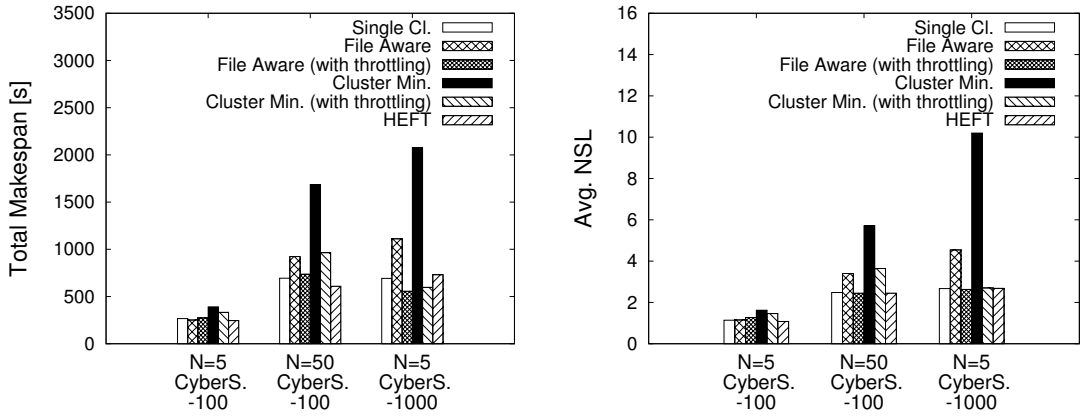


Figure 6.6: **Simulations, multi-workflow scheduling:** The performance of the scheduling policies in terms of the total makespan [s] and the average NSL. N denotes the number of application instances submitted together.

Table 6.5: **Simulations, multi-workflow scheduling:** The performance of the scheduling policies in terms of the average task queue wait time (grid-level + local, QWT [s]), the average task file transfer delay (FTD [s]), and the average number of inter-cluster file transfers performed per workflow (NFT).

	Single Cl.			File Aware			File Aware with Throttling			Cluster Min.			Cluster Min. with Throttling			HEFT		
C.S.-100	QWT	FTD	NFT	QWT	FTD	NFT	QWT	FTD	NFT	QWT	FTD	NFT	QWT	FTD	NFT	QWT	FTD	NFT
N=5	0	0	0	0	0.02	1.8	21+0	0	0	0	29	12	63+0	6.22	12.8	0.24	0.13	7.2
N=50	124	0	0	95	1.63	30	71+28	2	21	198	30	43	97+36	7	28	121	0.49	24
C.S.-1000																		
N=5	162	0	0	182	0.5	400	134+0	0.01	8	406	1	473	138+1	0.8	314	123	4.87	319

throttling), and the HEFT policies. We have used the CyberShake workflow application with sizes of 100 and 1000. For each of the policies that we consider, we submit either 5 or 50 concurrent instances of the CyberShake-100 application, and we submit 5 concurrent instances of the CyberShake-1000 application. When we apply task throttling, the concurrency limit per workflow submitted is set to 15 and 50 for the CyberShake-100, and CyberShake-1000 applications, respectively.

Figure 6.6 presents the performance of the scheduling policies in terms of the total makespan and the average NSL metrics. Table 6.5 presents additional metrics such as the grid-level delay due to the task throttling, average task queue wait time, the average task file transfer delay, and the average number of inter-cluster file transfers performed per workflow. For the small workflow (CyberShake-100) case, HEFT outperforms the other policies. Single Cluster and File Aware with task throttling yield similar performance results and they outperform the rest of the policies. For the large workflow case (CyberShake-1000), File Aware and Cluster Minimization both with task throttling have better total makespan performance than the other policies. In terms of NSL, File Aware

and Cluster Minimization both without task throttling have the worst performance while the other policies achieve a similar performance.

With the Single Cluster policy, workflows are balanced across clusters, and no inter-cluster file transfer takes place. Task throttling decreases the amount of inter-cluster communication (see Table 6.5), hence both the performance of the File-Aware and the Cluster Minimization policies improve, and they even outperform HEFT for the large workflow case. According to the results, for multi-workflow scheduling, a dynamic policy that takes inter-cluster communication into account and that also applies task throttling should be used. Alternatively, the Single Cluster policy may be preferred, depending on whether the application is communication- or computation-intensive and on the background load in the clusters, when the number of simultaneous workflow applications exceeds the number of clusters in the system.

6.3.3 Discussion

Our investigation shows that different system conditions, scenarios and workflow applications need different scheduling approaches in order to attain good application execution performance.

In general, increasing information usage about the workflow tasks and the grid resources improves the performance of the dynamic workflow scheduling policies. Although HEFT, which is the omniscient policy, is unrealistic for grids, HEFT-P, which uses predicted resource availability information, is a good alternative to be used provided that the application characteristics are known a priori, and that the prediction of the cluster that will finish a task first is correct.

The policies that take inter-cluster communication into account achieve better performance than the policies that do not. For instance, the Coarsening policy, and the File-Aware policy with task throttling are good alternatives that can be considered in the absence of complete task and resource information when scheduling communication-intensive large workflows.

When many workflow applications are submitted together, balancing the workflows across clusters separately or applying task throttling improves the performance, since both approaches prevent high inter-cluster network traffic, which increases file transfer times when many tasks are distributed across clusters.

6.4 Real system results

In this section we present the results of the experiments that we performed in DAS-3. We only present the results for the Single Cluster, All-Clusters, and Cluster Minimization policies since according to our simulation results, Round Robin performs the worst,

Table 6.6: **Real system, single workflow scheduling:** The performance of the scheduling policies in terms of **average makespan** (MS [s]), average queue wait time (QWT [s]), average task file transfer delay (FTD [s]), and NSL.

Workload Type	Workflow	Single Cluster				All-Clusters				Cluster Min.			
		MS	NSL	QWT	FTD	MS	NSL	QWT	FTD	MS	NSL	QWT	FTD
wf-small1	CyberShake-30	244.35	1.38	4.98	0.99	251.40	1.42	4.19	1.00	247.46	1.39	5.16	1.01
wf-small2	Montage-100	1380.49	1.29	4.67	2.08	1390.72	1.30	4.62	2.14	1387.43	1.30	5.05	2.15
wf-large1	CyberShake-1000	1321.67	6.51	36.08	0.05	1101.39	5.42	4.96	0.02	718.70	3.54	5.09	0.02

Table 6.7: **Real system, multi-workflow scheduling:** The performance of the scheduling policies in terms of **total makespan** (MS [s]), average queue wait time (QWT [s]), average task file transfer delay (FTD [s]), and NSL.

Workload Type	Workflow	Single Cluster				All-Clusters				Cluster Min.			
		MS	NSL	QWT	FTD	MS	NSL	QWT	FTD	MS	NSL	QWT	FTD
wf-small1	CyberShake-30	328.92	1.85	4.61	1.01	331.89	1.87	5.46	1.02	330.14	1.86	3.57	1.02
wf-small2	Montage-100	2195.24	2.06	15.82	2.21	2377.00	2.23	17.29	2.27	3568.16	3.35	48.16	2.58
wf-large1	CyberShake-1000	1413.15	6.96	26.21	0.06	2859.89	14.08	103.41	0.12	1940.47	9.55	6.56	0.08

File-Aware has similar performance (in many cases) as Cluster Minimization, and HEFT requires accurate resource and task information, which we find unrealistic for grid environments. Finally, we investigate the impact of task throttling with multi-workflow scheduling with the aim of improving system stability and responsiveness.

6.4.1 Single workflow scheduling

We first evaluate the performance of the selected workflow scheduling policies using the CyberShake-30 and the Montage-100 workflows for the wf-small1 and the wf-small2 workload type, respectively, and the CyberShake-1000 workflow for the wf-large1 workload type.

Table 6.6 shows the performance of the policies for single workflow scheduling experiments. We observe that the policies perform similarly for small workflows, confirming the simulations results for small workflows as shown in Figure 6.3 and Table 6.3. For the large workflow, Cluster Minimization has the best performance, significantly outperforming the Single Cluster and All-Clusters policies. Single Cluster has a higher queue wait time than the other policies, unlike the simulation results where All-Clusters has the highest queue wait time as shown in Table 6.3 (wflarge1 + BG). We attribute this difference to the much higher variability of the background load used in the simulations.

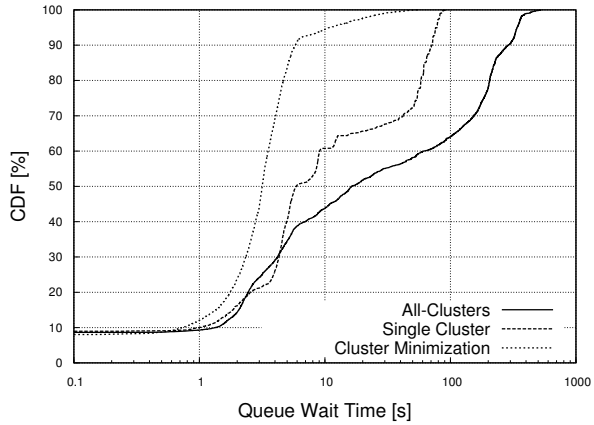


Figure 6.7: **Real system, multi-workflow scheduling:** Cumulative distribution function of the queue wait time of the CyberShake-1000 workflow tasks for all policies. The horizontal axis has a logarithmic scale.

6.4.2 Multi-Workflow scheduling

In this section we evaluate the performance of the selected scheduling policies, and the impact of task throttling on the performance of multi-workflow scheduling.

The performance of the scheduling policies

For the multi-workflow scheduling experiments we submitted five instances of the same workflow application simultaneously and only once. Table 6.7 shows the performance of the policies for these experiments.

For small workflows, the relative performance order of the policies is the same as in the single workflow scheduling experiments, except for the Montage-100 workflow for which the Cluster Minimization policy has the worst performance. The reason is that Cluster Minimization does not balance the load well compared with the other policies, hence increasing the queue wait times.

For the large workflow, we observe that Single Cluster performs the best since all workflows that are submitted simultaneously are mapped to a separate cluster, hence distributing the load better than the other policies, and no inter-cluster file transfers take place. This result confirms the simulation results in Table 6.5 (last row). The All-Clusters policy has worse performance than the other policies. The reason is twofold. First, the All-Clusters policy causes the number of inter-cluster file transfers to increase. Secondly, with the All-Clusters policy, many tasks suffer a large wait time, which is shown in Figure 6.7. Unlike for the other policies, around 37% of the tasks experience queue wait times of more than 100 seconds. We also observe a significant difference in median performance: the performance of the Cluster Minimization policy is roughly twice better

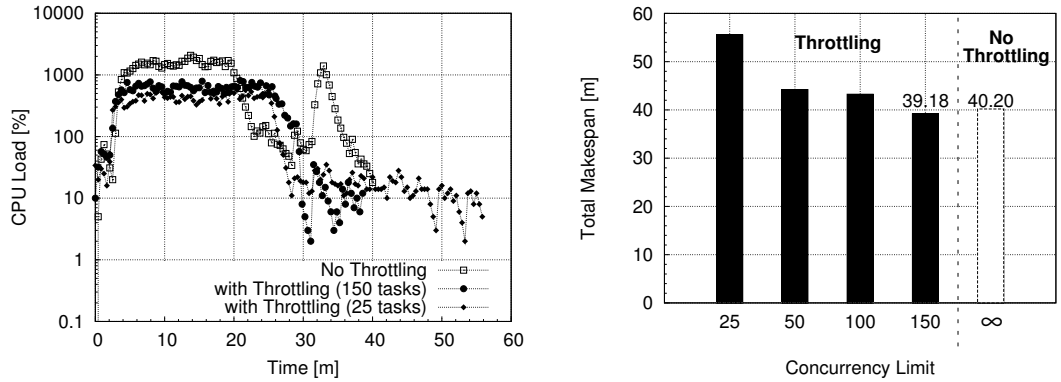


Figure 6.8: **Real system, multi-workflow scheduling:** The average CPU load of the Delft cluster head-node when 5 instances of CyberShake-1000 are submitted simultaneously (left), and the total makespan with and without throttling applied (right). The vertical axis of the left graph has a logarithmic scale.

than that of the Single Cluster policy, and the performance of the Single Cluster policy is roughly twice better than that of the All-Clusters policy. The performance difference between the policies is more significant in the real experiments than in the simulations. We attribute this observation to the resource contention in many layers of the system (e.g., in the network, and file system), which gets even worse for large workflows.

One of the main reasons for poor performance for multi-workflow submissions of large workflows is that the head-nodes become overloaded at such a scale. Figure 6.8 (left) shows the average CPU load of the Delft cluster head-node as reported by the system for the multi-workflow scheduling experiments with the CyberShake-1000 workflow. The CPU load is the value reported by the system's `top` utility. A high CPU load exceeding 100% can result in the incapacity of the system to perform even the simplest operations such as opening a socket or a file. This is the reason why we observe long delays in initiating file transfers (not included in FTD), as the workflow engine connects to the head-node of the execution site which is not responsive in overload situations. To overcome the problems related to head-node overload, we next investigate throttling as a possible solution.

The impact of task throttling

In this section we evaluate the impact of task throttling on the performance of multi-workflow scheduling. To this end, we submit five instances of the CyberShake-1000 workflow application simultaneously and only once. We use 25, 50, 100 and 150 as the concurrency limits. In contrast with the experiments presented in Section 6.3.1, we use here only three of the five DAS-3 clusters, due to the unavailability of the two other clusters.

We first look at the impact of the concurrency limit on performance. Figure 6.8 (right) depicts the total makespan with and without task throttling applied. As the concurrency limit increases from 25 to 150, we observe a decrease in the total makespan, converging as the concurrency limit increases to the performance of the system when no throttling is applied.

Second, we investigate the effects of using task throttling on the head-node. Figure 6.8 (left) shows the average CPU load of the Delft cluster head-node, when the cluster acts both as the submission site and as one of the execution sites. We observe in the figure a stable period where the CPU load is high due to the large number of running tasks in the system. Although the throttled system with a concurrency limit of 150 tasks and the initial system yield a similar total makespan (see Figure 6.8 (right)), the system with task throttling exhibits a factor of 2 improvement in the average CPU load, which consequently improves the system stability and responsiveness. When the concurrency limit is set to 25, the average CPU load shows a further substantial decrease, leading to a factor of 4 improvement over the system without task throttling, but then the total makespan increases noticeably because of the low concurrency limit.

To conclude, task throttling with appropriate concurrency limits prevents head-nodes being overloaded and simultaneously preserves the execution performance of the workflow applications. This fact motivates future research on determining appropriate throttling mechanisms and their associated parameters (e.g., the concurrency limit).

6.4.3 Discussion

For single workflow scheduling, the different policies have similar performance for small workflows. However, for large workflows, the policies have different performance, and in particular, the policies that minimize the inter-cluster communication have better performance than the policies that do not.

For multi-workflow scheduling, selecting a single cluster per workflow for execution yields the best performance. Policies distributing the tasks across clusters have worse performance due to the increased inter-cluster communication. In addition, for large workflows, head-nodes may get overloaded, which consequently threatens the performance; task throttling alleviates this problem.

6.5 Related work

An extensive body of research has focused on scheduling workflows in traditional parallel systems, addressing both homogeneous [125] and heterogeneous [21] sets of processors. The scheduling methods are usually static, that is, all tasks are mapped to processors before execution of the workflow starts, and they assume that perfectly accurate information

is available about the communication and the computation characteristics of the tasks. A classification of static scheduling approaches is presented in [196]. Such scheduling solutions, however, cannot be applied directly to multicluster grids. First, they operate at the processor level, while in grids the tasks are submitted to the local resource managers. Second, they do not consider the dynamic resource availability experienced in grids, which also makes accurate predictions of computation and communication costs difficult. Nevertheless, several studies [128, 129, 213, 214] adapted previous static scheduling methods by revising scheduling decisions at runtime taking the grid dynamics into account.

There are several scientific workflow management systems that can operate in grids. Some of the well-known ones are the Condor DAGMan [49], Pegasus [53], Karajan [206], Kepler [6], and Askalon [208]. They employ various types of static and/or dynamic scheduling methods. For more details we refer to the survey of Yu and Buyya [211].

Although most of the related work in grids deals with single workflow scheduling [20, 60, 93, 143], there are some studies that also address multi-workflow scheduling; by comparison, our work puts forth a more comprehensive investigation. Zhao and Sakellariou [216] present a method that combines several workflows into a single workflow, then prioritizes the tasks and maps them to resources using a static scheduling method. Iverson et al. [106] demonstrate that scheduling each competing workflow with a dynamic policy in a decentralized way improves the overall performance.

In addition, several researchers have addressed data-aware workflow scheduling, in which large data sets associated with scientific workflows are taken into account when scheduling tasks. Park and Humphrey [162] propose a bandwidth allocation technique to speed up file transfers. Bharathi and Chervenak [18] present several data staging techniques for data intensive workflows, and demonstrate that decoupled data staging can reduce the execution time of workflows significantly. Ramakrishnan et al. [170] evaluate a dynamic method that minimizes the storage space needed by workflows by removing data files at runtime when they are no longer needed.

In summary, our study complements and extends previous work in three main ways. First, we consider dynamic policies with various information availabilities. Secondly, we consider both single and multi-workflow scheduling in our performance evaluation. Finally, we perform simulations with realistic scenarios, and we validate our findings through experiments in the DAS-3 multicluster grid.

6.6 Summary

The performance of grid workflow scheduling policies affects an increasing number of scientists. To understand this performance, in this chapter we have conducted a comprehensive and realistic performance evaluation of dynamic workflow scheduling policies in multi-cluster grids. We have first introduced a scheduling taxonomy based on the amount

of information used in the scheduling process, and we have mapped seven scheduling policies that span the full information spectrum to this taxonomy.

Secondly, we have investigated the performance of these policies in realistic scenarios using both simulations and real system experiments. Overall, we found that different system conditions and workflow applications need different scheduling approaches in order to attain good application execution performance. Therefore, we believe it is important in grids, as we do with our KOALA grid scheduler [148], to support various scheduling policies from which the users can benefit considering the characteristics of their applications and the system capabilities. Alternatively, a scheduling mechanism can be implemented that switches dynamically the scheduling policy or the associated parameters, based on the system state and the workflows to be scheduled. We also found that, for scheduling communication-intensive workflows, the scheduling policies that take into account inter-cluster communication achieve better performance than the policies that do not. For example, the Coarsening policy, and the File-Aware policy with task throttling, that is, limiting the per-workflow number of tasks concurrently present in the grid, are two good options that can be considered in the absence of complete task and resource information.

Thirdly, our real system experiments have revealed performance problems that did not show in the simulations. In particular, we found that the head-nodes of real grid clusters may become unstable as the workflow size increases, leading to much lower performance. To solve this problem, we have analyzed the performance of task throttling, and we have shown that this approach keeps the system stable while delivering good performance.

Chapter 7

Evaluating prediction methods for grid scheduling

Although grid systems can be cost-effective and easily scalable, their multi-site and heterogeneous resource structure, and their dynamic and heterogeneous workloads limit the efficient use of the system resources. Moreover, the high variability of the job runtimes and queue wait times make such systems difficult and often frustrating to use for the common user. Prediction methods, and in particular prediction-based scheduling, have been employed to address these problems in parallel production environments, but their use for large-scale distributed systems such as multicluster grids remains largely unexplored. In this chapter we present a systematic investigation of prediction methods with application to grid scheduling.

An extensive body of research has focused on devising and applying prediction methods for such quantities as job runtimes and job queue wait times [38], CPU load [215], resource availability [158], and resource failure rates [115] in (large-scale) computer systems such as parallel systems and grids. The aim of such methods is to aid in the efficient scheduling in such systems and to assist users in selecting resources for their jobs. For instance, runtime predictions have been used to improve the performance of backfilling in batch queueing systems [197], and runtime and queue wait time predictions together can guide the decisions of a grid scheduler as to which grid sites to send jobs for execution. What is missing so far from this research is a detailed investigation of the performance of prediction methods for job runtime and queue wait time in grids, and of the benefit of using predictions in grid scheduling. In this chapter we fill this gap by applying simple and widely used prediction methods to the job runtimes and queue wait times of nine workload traces of research and production grids in the Grid Workload Archive [102], and by assessing the benefit of using predictions in grid level scheduling via trace-based simulations.

Our investigation is based on three guidelines. First, we target multicluster grid sys-

tems in which the processors are managed by space-sharing policies. Secondly, we restrict ourselves to time series prediction methods, which make their predictions based on historical data, usually in the form of the time-ordered set of past observations of, e.g., the job runtimes. Thirdly, we classify jobs in different ways, and apply these methods to the different job classes separately, in the hope to improve the performance of the prediction methods. Among the job classifications we will employ are grouping jobs per grid site, per user, and per user and per site. In this way, we aim to give realistic answers to the following research questions:

- **How accurate are the simple but widely used time series methods in predicting job runtimes in grids, and what is the impact of job classification on the accuracy of these predictions?** We answer these two questions in Section 7.2, where we assess the accuracy of five time series methods under four job classifications.

- **What is the performance of queue wait time predictors in grids?** We answer this question in Section 7.3 by evaluating the performance of a *point-valued* predictor that simulates the local scheduling policy with the predicted job runtimes to predict queue wait times of jobs, and by evaluating the performance of methods that predict *upper bounds* for the queue wait times of jobs.

- **Can prediction-based grid scheduling policies perform better than grid scheduling policies that do not use predictions?** We answer this question in Section 7.4, where we compare three grid-level scheduling policies in a simulated environment. The prediction-based scheduling policy bases its decisions on job runtime and queue wait time predictions (which are either assumed to be perfect or potentially inaccurate), whereas the non-prediction-based policies balance the load on the clusters or prefer faster resources.

7.1 Grid workload traces

For our investigation we use nine grid traces from the Grid Workloads Archive [102]. Each trace consists of ordered job entries according to their submission time which is in UNIX timestamp format. All traces have complete information about the jobs' submission times, runtimes, and requested numbers of nodes. Some of the traces lack other job attributes such as queue wait time, application name, group name, etc. None of the traces contain information about user runtime estimates.

Table 7.1 summarizes the characteristics of the grid traces that we have used in our work, and contrasts them to those of four traces taken from the Parallel Workloads Archive [167]. The grid traces are gathered from four research grids and five production

Table 7.1: The characteristics of nine grid traces taken from the Grid Workloads Archive and of four traces taken from the Parallel Workloads Archive. The sign "-" denotes missing information.

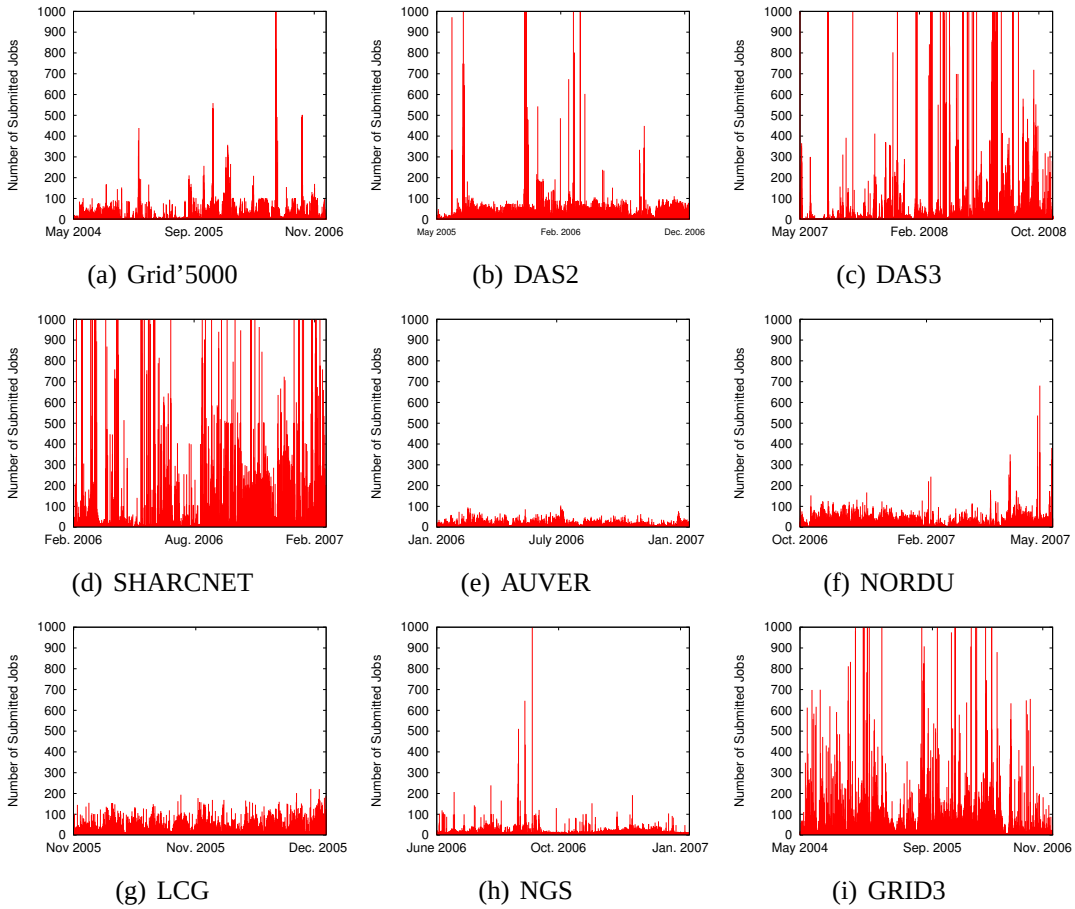
	System			Trace			
	Type	Num. of Clusters	Size [CPUs]	Duration [Months]	Size [Tasks]	% of Parallel Jobs	Num. of Users
Grid Workloads Archive Traces [102]							
DAS2	Research	5	400	18	1.1M	66%	333
Grid'5000	Research	15	∼2500	27	1.0M	45%	470
DAS3	Research	5	544	18	2M	15%	331
SHARCNET	Research	10	6828	12	1.2M	10%	412
AUVER	Production	5	475	12	0.4M	0%	405
NORDU	Production	75	2000	24	0.8M	0%	387
LCG	Production	-	24515	4	0.2M	0%	216
NGS	Production	7	-	6	0.6M	0%	378
GRID3	Production	35	∼3500	18	1.3M	0%	15
Parallel Workloads Archive Traces [167]							
CTC SP2, PWA-6	Production	1	430	11	0.1M	56%	679
SDSC SP2, PWA-9	Production	1	128	24	0.1M	63%	437
LANL O2K, PWA-10	Production	1	2048	5	0.1M	-	337
SDSC DS, PWA-19	Production	1	1664	13	0.1M	100%	460

grids. There are several differences between typical research and production grids. In research grids, the workloads contain both parallel jobs and sequential jobs, and the system utilizations are low (10%-30%), whereas in production grids, the workloads consist solely of sequential jobs, and the system utilizations are much higher (over 60%) [98, 100]. The main differences between grids and paralel processing enviornments (PPEs) as we observe are the following: grid resources are spread across multiple sites, grid workloads include fewer parallel jobs, and over the long term, the grid workloads include many more jobs than PPEs (by a factor of 2-20).

An important assumption of this chapter is that grids exhibit often bursty job arrivals. We show that this is indeed the case in Figure 7.1, which depicts the numbers of job submissions during five-minute intervals. We conclude that in addition to the grid workload characteristics mentioned above, grid workloads are bursty, which provides even more motivation for this study.

7.2 Job runtime predictions

In this section we investigate the performance of job runtime predictions in grids. We first describe in Section 7.2.1 our methodology. Then, we describe in Section 7.2.2 our experimental setup, and last we present and discuss in Section 7.2.3 the experimental results.



(a) Grid'5000 (b) DAS2 (c) DAS3

(d) SHARCNET (e) AUVER (f) NORDU

(g) LCG (h) NGS (i) GRID3

Figure 7.1: The number of job submissions during five-minute intervals in nine grid systems. All systems have bursty periods. The vertical axis is truncated at 1000 for better visibility.

7.2.1 Methodology

The methodology we use for runtime predictions consists of three elements: the classifications of jobs we use, the way we simulate grid traces to compute runtime predictions, and the actual runtime prediction methods, which we now describe in turn.

First, the job classification methods create classes according to job attributes such as the execution site of a job, the user submitting it, etc. We consider the following four classification methods for the jobs in a trace:

1. **Site**: The jobs are classified according to the site where they are executed.

2. **User**: The jobs are classified according to the user who submits them, irrespective of the execution site.

3. **User on Site**: The jobs are classified according to both the user and the execution site.

4. **(User + Application Name + Job Size) on Site**: The jobs are classified according to the user, the application name, the job size (i.e., the number of processors employed by the job), and the execution site.

Secondly, in the simulation of a trace, we go sequentially through the trace and we compute for every next job its predicted runtime with the prediction method in place, based on the history consisting of the runtimes of the jobs of the same class that have submitted and finished before the current job under consideration. This means that during the simulation, the time series for each of the job classes is created.

Thirdly, for each job, the prediction method predicts the runtime using the time series data of the class the job belongs to that has already been created. We consider the following prediction methods, which are applied to the time series of the runtimes of the jobs on a per-class basis:

1. **Exponential Smoothing (ES)** predicts the runtime as a weighted moving average of the observed job runtimes. A parameter α, with $0 \leq \alpha \leq 1$, is used to control the sensitivity of the smoothing. We take α to be equal to 0.5 (e.g., see [54]). We refer to [24] for details.

2. **Running Mean (RM)** predicts the runtime as the mean of all observed job runtimes.

3. **Sliding Median (SM)** predicts the runtime as the median of a sliding window of observed job runtimes. We take 5 as the window size (e.g., see [209]).

4. **Last** predicts the runtime as the last previously observed job runtime.

5. **Last2** [197] predicts the runtime as the average of the last two previously observed job runtimes.

For the first three job classifications introduced above, we evaluate the performance of all prediction methods on all traces. Then, we pick the best method for each trace and run it with the last classification for those traces that include the Application Name attribute (all of the traces except LCG and NORDU). The Job Size attribute is considered only for the research grid traces since only they include parallel jobs.

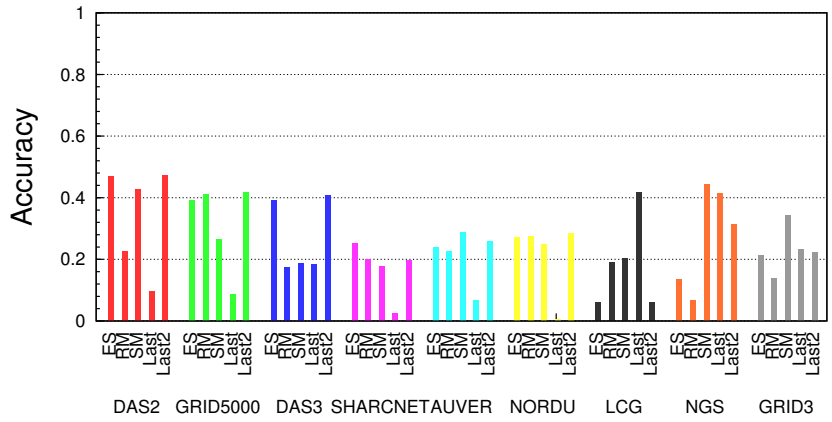


Figure 7.2: The average accuracy of the job runtime prediction methods on the grid traces under the Site classification.

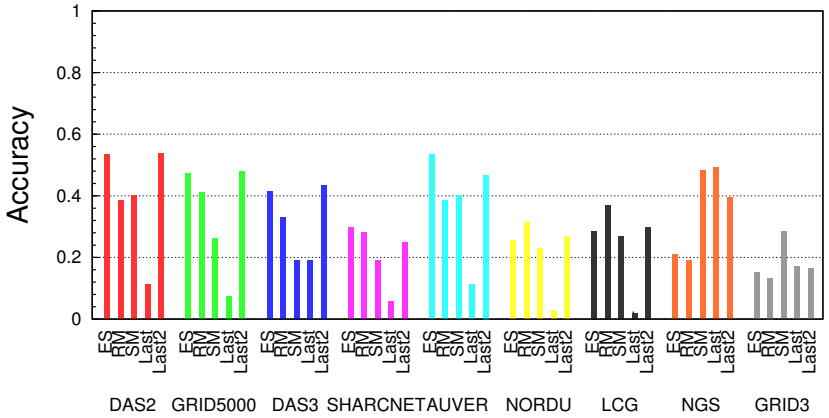


Figure 7.3: The average accuracy of the job runtime prediction methods on the grid traces under the User classification.

7.2.2 Experimental setup

We have used the Grid Workloads Archive tools to process the grid traces. We have extended the tools so that the jobs in the traces are classified according to the classifications described in the previous section.

To evaluate the accuracy of the runtime predictions, we consider the following metrics:

1. The **accuracy**, which is defined as in [197]:

$$accuracy = \begin{cases} 1 & \text{if } P = T_r, \\ T_r/P & \text{if } P > T_r, \\ P/T_r & \text{if } P < T_r. \end{cases} \tag{7.1}$$

where P is the predicted job runtime and T_r is the actual job runtime.

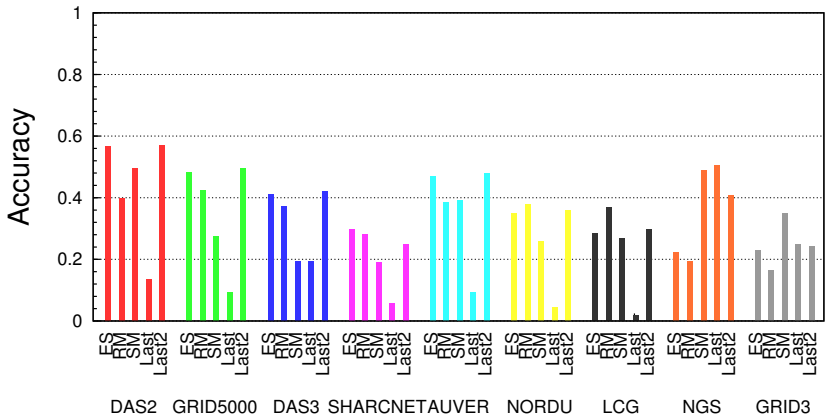


Figure 7.4: The average accuracy of the job runtime prediction methods on the grid traces under the User on Site classification.

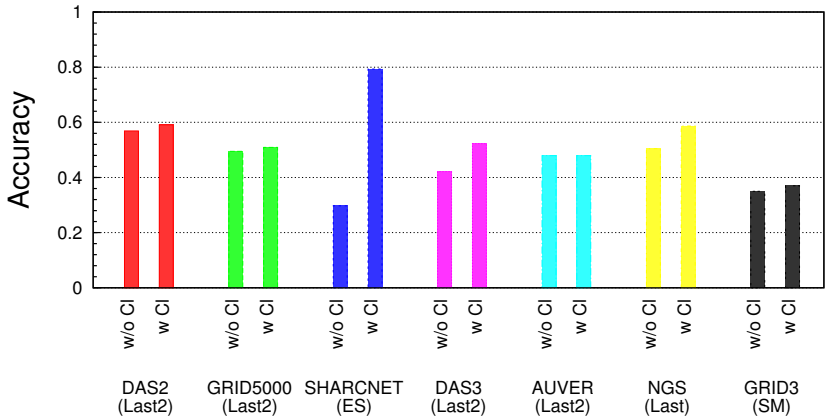


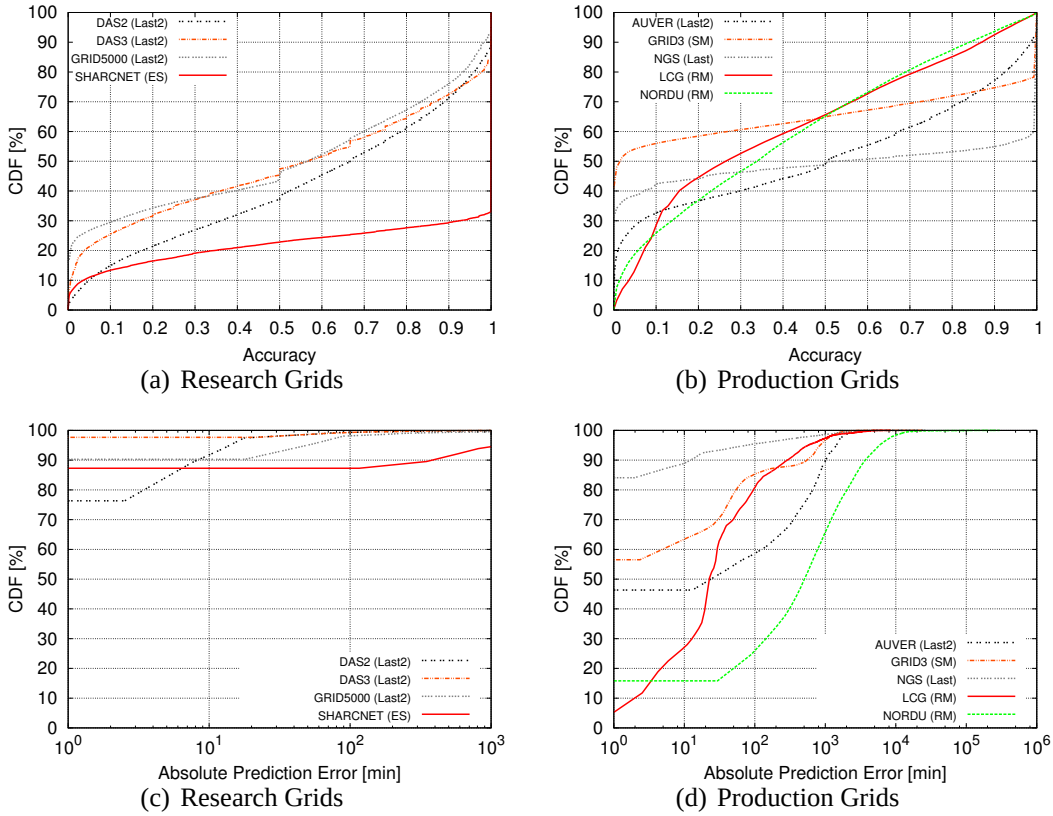
Figure 7.5: The average accuracy of the job runtime prediction methods on the grid traces under the (User+Application Name+Job Size) on Site classification.

2. The **absolute prediction error** is the absolute difference between the predicted and the actual runtime.

7.2.3 Results

For each of the nine traces, Figures 7.2, 7.3, and 7.4 present the average accuracy of the prediction methods under the Site, the User, and the User on Site classifications, respectively. Figure 7.5 shows the average accuracy of the best methods under the (User + Application Name + Job Size) on Site classification (**w/o Cl**: Best result from the other classifications, **w Cl**: Result with this classification.).

As the historical data gets more specific, that is, going from Site to (User + Application Name + Job Size) on Site, the accuracy of the job runtime predictions increases significantly. In particular, SHARCNET yields an outstanding job runtime accuracy with



(a) Research Grids (b) Production Grids

(c) Research Grids (d) Production Grids

Figure 7.6: The cumulative distribution functions of the accuracy and absolute prediction error for the best (method+classification) results for job runtimes. The horizontal axis for the bottom row has a logarithmic scale.

the most specific classification. For the traces of DAS-2, DAS-3, and Grid'5000, Last2 performs better than the other prediction methods for all classifications. For the other traces, we do not observe such a dominant method, and even the best method for a trace differs among the classifications. The results suggest that grid systems or even grid sites should have their own specific prediction methods, since they may have different user behaviors and different job and system characteristics.

Figure 7.6 presents the cumulative distributions of the accuracy and the absolute error only for the best results (the method and classification that give the best accuracy for a trace) for the traces of the research and production grids separately. We observe that in most of the cases, the job runtimes are predicted more accurately and with lower absolute errors in research grids than in production grids. The possible reasons include longer job runtimes and higher utilizations in production grids. For SHARCNET, almost 70% of the predictions have high accuracy (i.e., above 0.9), while for the other traces this percentage ranges between 20 to 30. Among the production grid traces, we see that NGS and GRID3 exhibit a higher prediction accuracy and a lower absolute prediction error.

Table 7.2: The performance of the point-valued predictor of queue wait time for jobs that have non-zero wait times (indicated by “non-zero jobs”).

		Without Correction		With Correction	
Grid-Site	% non-zero jobs	Avg. Accuracy	AWPE [min]	Avg. Accuracy	AWPE [min]
DAS-2 FS1	20	0.60	110	0.64	121
DAS-2 FS3	35	0.54	307	0.55	292
DAS-3 FS3	80	0.30	364	0.35	350
DAS-3 FS4	70	0.51	451	0.60	442
Grid5K G1/S1/C3	60	0.51	322	0.63	256
Grid5K G1/S6/C1	10	0.56	852	0.61	653
AUVER clrlcgce01	20	0.57	203	0.64	180
AUVER clrlcgce03	67	0.63	190	0.69	172

All in all, we find the job runtime prediction accuracy to be low and the absolute prediction error to be high, even for the best results (except for SHARCNET). There are several reasons for this poor performance. The first is the occurrence of burst submissions that we observe in grids (e.g., see Figure 7.1); the same prediction error is made for all the jobs submitted together or relatively close in time. Even though these jobs could be similar in terms of runtime, they do not affect the predictions before they finish execution. A second reason is the (lack of) stationarity of a time series. For good predictability, a time series should be stationary [24], that is, it should have a constant long-term mean and variance. We have performed several experiments using the Augmented-Dickey-Fuller[1] (ADF) test [64] for checking stationarity on some of the time series that we use in this chapter; unsurprisingly, we have found the time series to be non-stationary.

7.3 Queue wait time predictions

In this section we investigate the performance of queue wait time predictions in grids. In Section 7.3.1 we evaluate the performance of a point-valued predictor that simulates the local scheduling policy with predicted job runtimes to predict job queue wait times. In Section 7.3.2 we evaluate the performance of two non-parametric statistical methods that predict upper bounds for queue wait times with a specified confidence level. Such non-parametric methods have the advantage of obviating the need to know the internal operation of local scheduling policies in predicting the queue wait times. We use the traces of the DAS-2, DAS-3, Grid’5000, and AUVER grids. These are the systems/traces of which we know their characteristics to model in our simulations in Section 7.3.1, and that contain the queue wait time data that we need in the simulations in Section 7.3.2.

[1]We have obtained the tool for the ADF-test from http://www.web-reg.de/adf_addin.html

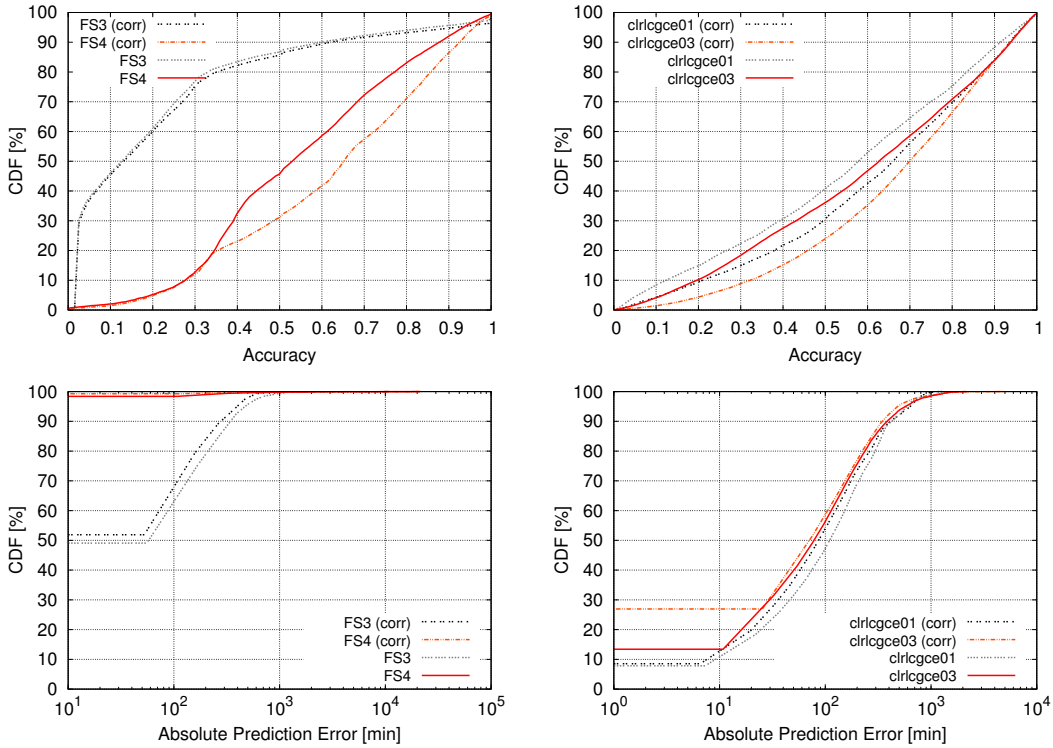


Figure 7.7: The cumulative distribution functions of the accuracy and absolute prediction error of the queue wait time predictions for the DAS-3 (left) and the AUVER (right) traces with the Last2 method for runtime predictions; (corr) denotes that the correction mechanism is applied to the job runtime predictions. The horizontal axis has a logarithmic scale for the bottom row.

7.3.1 Point-Valued predictions

In our simulation model, each site has a local resource manager (LRM) that employs the FCFS policy without backfilling. We assign jobs to their original execution sites. A point-valued predictor runs on each site, and computes a queue wait time prediction for each submitted job in a two-step process. First, the runtimes of all queued and running jobs are predicted with the Last2 method and the (User + Application Name + Job Size) on Site classification that is described in Section 7.2.1. Then, the scheduling policy of the LRM is simulated with the predicted job runtimes to determine when the new job will start. Whenever the runtime of a job turns out to be under-predicted, its predicted runtime is doubled until the predicted value is larger than the actual runtime.

We also consider a prediction correction mechanism in which upon completion of a job, the predicted runtimes of both the queued and the running jobs that belong to the same class as the completed job are updated with the Last2 method to include the runtime of the completed job in the computation. We perform the experiments with and without this correction mechanism.

Table 7.3: The accuracy of queue wait time prediction methods giving upper bounds.

	BMBP				
Grid-Sites	AWPE [min]	Avg. Accuracy	Under-predictions	Perfect-predictions	Over-predictions
DAS-2 FS1	16	0.50	8%	9%	83%
DAS-3 FS4	26	0.41	15%	4%	81%
AUVER clrlcgce01	376	0.20	12%	1%	87%
Grid5K G1/site1/c3	31	0.72	20%	0%	80%

	Chebyshev				
Grid-Sites	AWPE [min]	Avg. Accuracy	Under-predictions	Perfect-predictions	Over-predictions
DAS-2 FS1	52	0.21	8%	0%	92%
DAS-3 FS4	101	0.23	7%	1%	82%
AUVER clrlcgce01	1236	0.10	7%	0%	93%
Grid5K G1/S1/C3	1093	0.24	16%	0%	84%

Figure 7.7 shows the cumulative distribution functions of the accuracy and the absolute prediction error of the point-valued queue wait time predictor for DAS-3 and AUVER; for clarity, we only present the results for the two sites of each grid system to which most of the jobs have been submitted. Table 7.2 presents the average values of the metrics for all grids (AWPE refers to average absolute queue wait time prediction error). In the results, we only consider the jobs that have non-zero wait times.

We find the overall accuracy of the point-valued predictor to be low, and the average absolute queue wait time prediction error to be high due to inaccurate job runtime predictions. The prediction error in the queue wait times is accumulated because the predictor simulates the local scheduling policy with inaccurately predicted runtimes. There is an improvement ranging from 1% to 10% in average accuracy and from 1% to 16% in average absolute prediction error when the prediction correction mechanism is applied.

7.3.2 Upper-Bound predictions

In this section we assess the accuracy of two upper-bound methods for queue wait time predictions by means of trace-based simulations. These methods are the Binomial Method Batch Predictor (BMBP) [23] and a predictor that makes use of Chebyshev's inequality [192]. In our analysis, we use the wait times of the jobs that are completed in order to predict the wait time of a new job. We do not simulate local scheduling policies since we use real wait time and runtime data (from the traces). To evaluate the prediction methods, we use the average accuracy, the average absolute prediction error, and the number of under-predictions, perfect-predictions, and over-predictions as metrics.

BMBP predicts an upper-bound with a specified quantile and confidence level. It uses the history of job queue wait times, and estimates the quantile of the wait time distribution

with the specified confidence level. It employs a change-point detection method in order to take only a stationary part of the time series into account. By detecting the change-points in the time series, BMBP trims the historical wait time data. BMBP also clusters the jobs based on the numbers of processors they request, hence it uses the historical data of similar jobs when making predictions. For further details on BMBP we refer to [23].

Chebyshev's inequality states that regardless of the underlying distribution, the probability of a random variable differing from its mean by more than k standard deviations is less than or equal to $1/k^2$. We have implemented a predictor that uses this inequality; it calculates the mean and the standard deviation of the wait time data of the completed jobs to predict an upper bound for the wait time of a new job with a specified confidence level (e.g., $\mu + 2\sqrt{5}\sigma$ is the predicted wait time with a 95% confidence level, where μ is the mean and σ is the standard deviation of the historical wait time data). We trim and update the historical data in a similar way as explained in [23].

In our analysis of the BMBP method, we use the BMBP trace-based simulator [2], and for Chebyshev's inequality method we use our own tools. For BMBP, we consider a quantile and confidence level of 95%, and we use 10% of the data for training. Similarly, for the predictor that uses Chebyshev's inequality, we consider a confidence level of 95%.

The results are presented for a single site of each trace in Table 7.3. The number of over-predictions when using Chebyshev's inequality is larger than when using BMBP, whereas the accuracy of BMBP is higher. There is a trade-off between the accuracy and the tightness of the upper-bound. Both of these methods fail when the jobs arrive in bursts, as the methods use the same predicted wait time value for all jobs in a burst.

We claim that user runtime estimates, if available, can also be used in predicting upper bounds for queue wait times. To show this, we use a simple model for user runtime estimates that is proposed by Mu'alem and Feitelson [70]. The model assumes that a job's estimate is uniformly distributed within $[R, 5R]$, where R is the job's actual runtime. We use the runtime estimates of users together with the FCFS policy (to guarantee upper bounds) to predict wait times. Figure 7.8 shows the real wait time values and the ones predicted with this approach for a bursty period of the DAS-3 FS4 site. While guaranteeing over-predictions, this approach results in slack estimates of queue wait time.

7.4 The performance of prediction-based grid scheduling

In this section we assess whether it is beneficial to use predictions for scheduling in grids. To this end, we perform a set of simulations using workloads from the DAS-3 and AUVER grids to investigate whether prediction-based grid-level scheduling improves performance over traditional grid-level scheduling policies. In Section 7.4.1 we explain the

[2] We have obtained the simulator from the Network Weather Service website `http://nws.cs.ucsb.edu`.

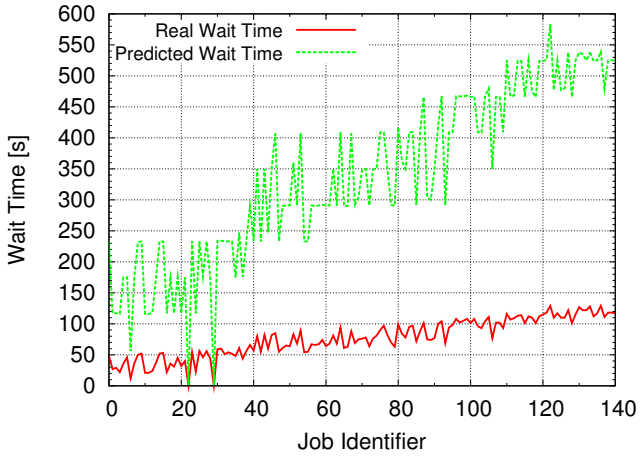


Figure 7.8: Real and predicted queue wait times for a burst submission case taken from DAS-3, site FS4.

experimental setup, in Sections 7.4.2 and 7.4.3 we describe the scheduling policies and the performance metrics we use in our simulations, respectively, and in Section 7.4.4 we present and discuss the experimental results.

7.4.1 The experimental setup

For our experiments we have modeled two multi-cluster grid environments, the DAS-3 (research) and the AUVER (production) grids, using our event-based grid simulator DGSim [104]. Table 7.4 shows the sizes of the clusters of each system. In DAS-3, the processing speeds of the compute nodes differ among clusters, but in AUVER, the nodes are homogeneous in terms of processing speeds. To model the processor heterogeneity across the clusters of the DAS-3, we employ the SPEC CPU benchmark model for job runtimes [190].

In our model, each cluster has its own LRM, and so its own local queue to which jobs arrive, and a global central scheduler with a global queue operates on top of the cluster LRMs. The jobs are submitted to the global scheduler, which decides in which cluster a job is going to run based on one of the scheduling policies that are explained in Section 7.4.2. Irrespective of the policy in operation, the global scheduler considers all jobs in its queue as the eligible set for scheduling. The LRMs of the clusters employ the FCFS policy without backfilling. Once started, tasks run to completion, so we do not consider task preemption or task migration during execution. For the experiments with prediction-based policies, a prediction service runs on each of the clusters in order to respond to the queries issued by the central scheduler regarding the predicted completion time of a job; i.e., the sum of the predicted queue wait time of a job and its predicted runtime.

Table 7.4: The size of the clusters in DAS-3 and AUVER grids.

DAS-3		AUVER	
Cluster (Original Name)	Size	Cluster (Original Name)	Size
C1 (FS0)	85	C1 (clrlcgce01)	112
C2 (FS1)	32	C2 (clrlcgce02)	84
C3 (FS2)	41	C3 (clrlcgce03)	186
C4 (FS3)	68	C4 (iut15)	38
C5 (FS4)	46	C5 (opgc)	55

Table 7.5: The workload characteristics used for assessing the performance of prediction-based grid scheduling.

Trace	Period	Number of Jobs	Avg. Utilization
DAS-3	July-Oct. 2008	∼220,000	∼30%
AUVER	Aug.-Nov. 2006	∼90,000	∼70%

In our simulations we consider the busiest four-month period from the trace of each system and submit it as the workload. Table 7.5 shows the properties of the workloads that we have used in our experiments.

7.4.2 Scheduling policies

We compare the performance of the following policies which we find representative for many other prediction-based and traditional policies proposed in the grid scheduling literature:

- The **Earliest Completion Time** (ECT) [138] is a Gantt chart-based scheduling policy that submits each job to the cluster that leads to the earliest completion time possible, taking into account the clusters' queues. We consider two kinds of prediction information leading to two variations of this policy. **ECT-Perfect** policy is a theoretical omniscient policy whose predictions are always given with perfect accuracy (equal to 1). In contrast, **ECT-Last2** uses the point-valued predictor defined in Section 7.3.1 (with corrections); hence, the predictions of **ECT-Last2** may be inaccurate.

- **Load Balancer** (LB) submits each job to the least-loaded cluster, where load is defined as the total processor requirement of all jobs running or queued in the cluster normalized by the cluster size.

- **Fastest Processor First** (FPF) submits each job to the cluster that has the fastest compute nodes among the clusters that have enough idle nodes to accommodate the job. Different from the policies mentioned above, FPF does not forward jobs as long as there are not enough idle nodes in any cluster; therefore, jobs are only queued in the global queue.

7.4.3 Performance metrics

To assess the performance of the scheduling policies, we use the following traditional metrics:

- The **Queue Wait Time** of a job is the time elapsed from the submission of the job until the start of its execution.
- The **Response Time** of a job is the sum of its queue wait time and its runtime.
- The **Bounded Slowdown** [72] of a job is defined as

$$\max\left(1, \frac{T_w + T_r}{\max(\tau, T_r)}\right), \tag{7.2}$$

  where T_w and T_r denote the queue wait time and the runtime of the job, respectively, and τ denotes the threshold for the job runtimes. The bounded slowdown eliminates the emphasis on short jobs due to having the runtime in the denominator. In our analysis we have used a threshold value of 60 seconds.

Table 7.6: The performance of the three scheduling policies.

DAS-3	ECT-Perfect	ECT-Last2	LB	FPF
Avg. Res. Time [s]	1320	1400	4318	1911
Avg. Wait Time [s]	105	186	3061	681
Avg. Boun. Slowd.	1.7	1.6	80	26

AUVER	ECT-Perfect	ECT-Last2	LB	FPF
Avg. Res. Time [s]	40951	41003	40959	41334
Avg. Wait Time [s]	6515	6574	6534	6898
Avg. Boun. Slowd.	48	43.6	48	50.88

7.4.4 Results

The cumulative distribution functions of the queue wait time, the response time, and the bounded slowdown are shown in Figures 7.9 and 7.10, and their averages are shown Table 7.6.

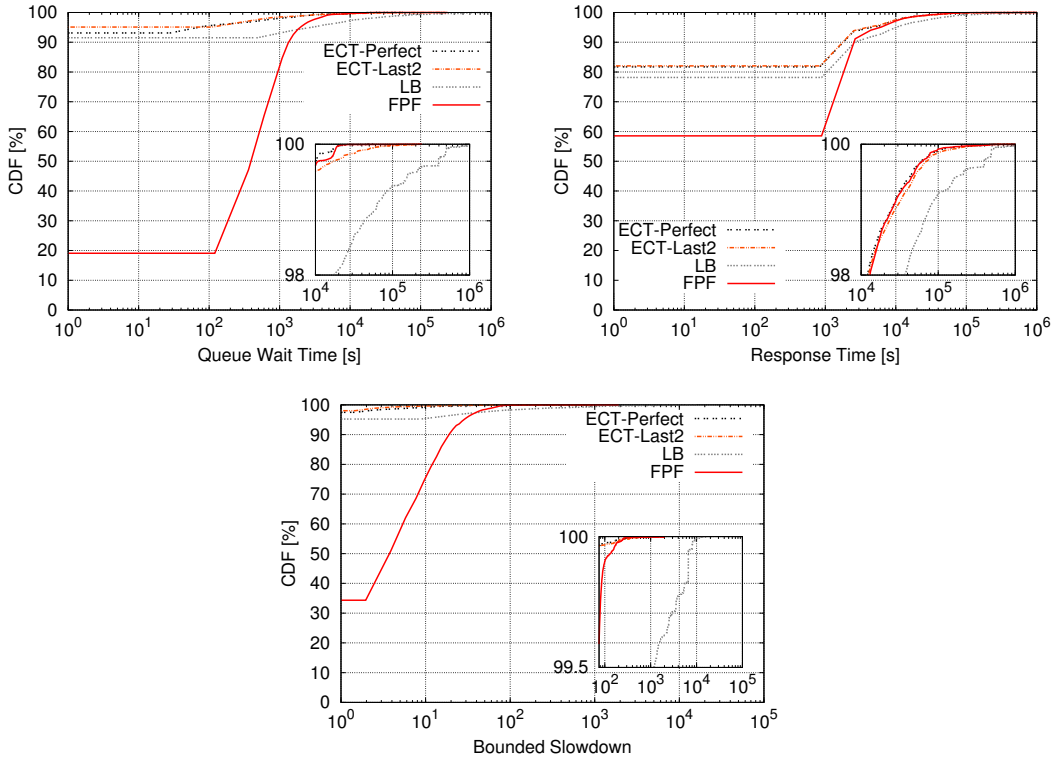


Figure 7.9: The cumulative distribution functions of the queue wait time, the response time, and the bounded slowdown for DAS-3. The horizontal axis has a logarithmic scale.

We find that in DAS-3, the prediction-based scheduling policies (ECT-Perfect and ECT-Last2) perform better than their traditional counterparts (LB and FPF), with especially the LB policy having very poor performance. It turns out that with LB, approximately 5% of the jobs have a queue wait time of more than 10,000 seconds. On the other hand, with the FPF policy, a small number of jobs suffer from queue wait times as high as 10,000 seconds. In contrast, for AUVER, all policies have similar performance. LB seems to perform slightly better than FPF in AUVER, which suggests that it can be a candidate for highly utilized systems when prediction-based policies are not considered. In general, the performance of the considered policies is worse for the AUVER grid, which is probably due to the higher utilization of the considered period compared to the DAS-3 (see Table 7.5).

Since the ECT-Perfect and ECT-Last2 policies have similar performance, both in the DAS-3 and the AUVER experiments, we conclude that more accurate predictions do not necessarily imply a better performance of grid scheduling. A similar result was previously obtained when using predictions for improving backfilling performance in parallel production systems [197]. ECT-Perfect and ECT-Last2 yield similar performance results even when their scheduling decisions differ; in Figure 7.11 we see that the distribution of

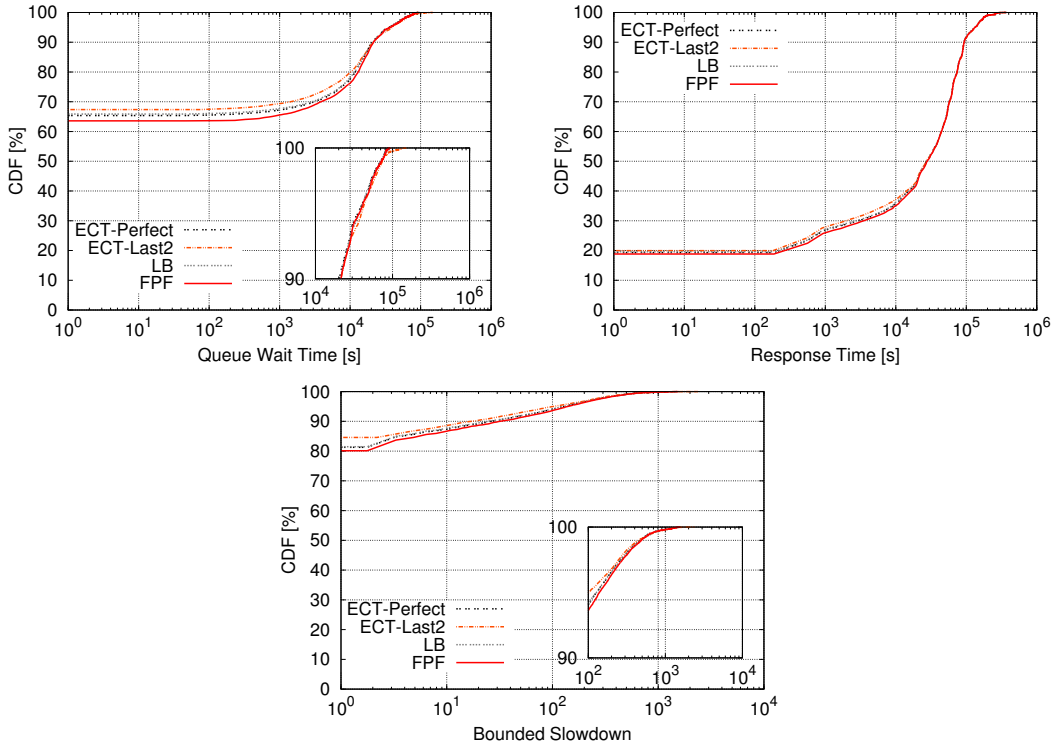


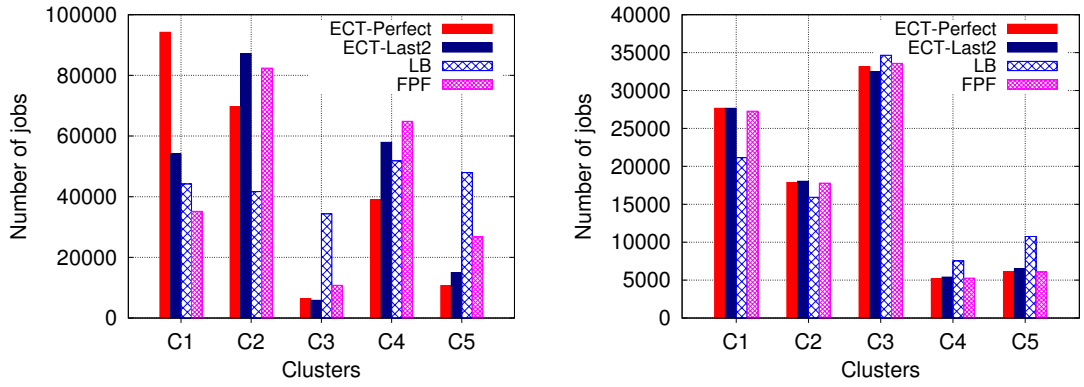
Figure 7.10: The cumulative distribution functions of the queue wait time, the response time, and the bounded slowdown for AUVER. The horizontal axis has a logarithmic scale.

the tasks across clusters is very diverse when the jobs are scheduled with this two policies in DAS-3, while the policies distribute the jobs similarly in AUVER. All in all, we claim that the main concern of a prediction-based grid scheduling policy is the correct prediction of the cluster that will finish a job first, rather than perfect prediction accuracy of the job's completion time.

7.5 Related work

Previous work on predictions has mainly focused on proposing novel prediction methods [23, 56, 57, 182, 209], on enhancing existing methods [55, 179], and on making use of predictions in space-shared parallel environments [126, 197]. In contrast, we have focused on the performance of job runtime and queue wait time predictions in grid environments.

In addition to various complex solutions proposed for job runtime predictions such as analytical benchmarking/code profiling [105, 163, 210], genetic algorithms [181, 182], and instance-based learning [116], simple time series methods have also received great attention from the community due to their advantages such as ease of implementation and speed of delivering prediction results. In [55, 179], exponential smoothing is used



(a) DAS-3 (b) AUVER

Figure 7.11: The distribution of jobs across the clusters of DAS-3 and AUVER.

for predicting the runtimes of jobs. In [126], a prediction method based on linear regression is proposed to predict the runtimes of parallel applications. Dobber et al. [56] present a survey on prediction methods for job runtimes on space-shared processors, and they propose a new prediction method, called Dynamic Exponential Smoothing, which uses exponential smoothing and adapts dynamically to peaks and level changes in the job runtimes. Feitelson et al. [197] show that even a simple time series prediction method improves backfilling performance significantly when system-generated predictions replace user-estimated runtimes. The Network Weather Service (NWS) [209] is a well-known prediction service which is used to predict the performance of computational grid resources, with simple time series prediction methods. NWS tracks the accuracy of all its predictors, and dynamically changes the prediction method to the one that gives the highest accuracy.

The problem of predicting queueing delays of jobs in high performance computing settings has also received constant attention from the research community. In [22], model-fitting is used to model machine availability in desktop and enterprise grid computing environments. To estimate when a cluster of a given size will be available and hence the runtime of the job at the head of the queue, Downey [57, 58] explore a log-uniform distribution to model the remaining lifetimes of the jobs. This work shows that the queue wait time of jobs can be predicted if the runtimes of jobs and the scheduling algorithm are known. Similarly, in [182] the queue wait time of a job is predicted under the conditions that the job runtimes and the local scheduling algorithm are known. In this work, the authors use a template-based approach to cluster the jobs, and then perform searches based on genetic algorithms. Wolski et al. [159] propose QBETS, which is a non-parametric time series method to predict bounds on the queue wait times of individual jobs in space-shared parallel environments.

7.6 Summary

In this chapter we have studied job runtime and queue wait time prediction methods and their application in grid scheduling. First, we have studied through trace-based analysis the accuracy of well-known, simple time series prediction methods when predicting job runtimes, and the impact of job classification on the accuracy of job runtime predictions. We found that the low accuracy of time series prediction methods for grids can be improved significantly by the use of such classifications. Secondly, we have analyzed the performance of queue wait time predictors. We have shown that current solutions for queue wait time predictions that give upper-bounds cannot handle the common grid workload characteristic of burst submissions. Thirdly, we have compared with trace-based simulations several prediction-based and traditional grid scheduling policies in order to investigate the impact of predictions on the performance of grid-level scheduling. We have found that a better accuracy of the predictions does not imply a better performance of grid scheduling, and that for the highly utilized production grids the investigated policies perform similarly to each other.

Chapter 8

Conclusion

In this thesis we have studied application-oriented scheduling in multicluster grids. We have designed, implemented and then evaluated various realistic and practical scheduling policies for different application types in the DAS-3 multicluster grid system using our KOALA scheduler, as well as in a simulated environment using realistic settings. In Chapter 1 of this thesis, we have introduced the challenges addressed by our research. In Chapter 2, we investigated the benefit of co-allocation for parallel applications. We compared the performance of several co-allocation scheduling policies under various system load settings. In Chapter 3, we extended our KOALA grid scheduler with support for malleable parallel applications. We proposed several policies to manage dynamic resource allocation for such applications. In Chapter 4, we extended KOALA with the support for scheduling cycle scavenging applications, of which parameter sweeps are a prime example. We proposed two policies that try to achieve fair-share resource allocation among cycle scavenging users. In Chapter 5, we investigated the performance of scheduling bags-of-tasks in multicluster grids with realistic simulations. We explored the performance impact of several elements such as the workload, the task selection policy, the task scheduling policy, and the resource management architecture. In Chapter 6, we evaluated the performance of dynamic workflow scheduling in multicluster grids with realistic simulation-based experiments as well as experiments conducted in DAS-3. Finally, in Chapter 7, we investigated the performance and benefit of predicting job execution times and queue wait times in multicluster grids based on the traces collected from various research and production grid environments.

In the remainder of this chapter we present the main conclusions of this thesis (Section 8.1) and suggest directions for future work (Section 8.2).

8.1 Conclusions

Based on the research reported in this thesis we have drawn the following major conclusions:

1. Supporting application-oriented scheduling mechanisms and policies in a grid scheduler is the key to achieve good application execution performance in multicluster grids.

2. For the performance evaluation of scheduling policies in grids, if possible, simulations should be supported by experiments in real grid systems, since it is difficult to model detailed system behavior for simulation purposes, which may reveal problems that do not show in simulations.

3. The benefit of co-allocation for parallel applications depends on various factors such as the communication characteristics of the applications, the communication and computation characteristics of the resources, the level of resource contention in the system, and the grid software being used. Nevertheless, in systems where the heterogeneity in inter-cluster communication speeds is high, using network performance metrics in resource selection, such as the latency, increases the performance of co-allocation for communication-intensive parallel applications.

4. Application malleability makes it easier to deal with the dynamic nature of multicluster grid systems by growing or shrinking the resource allocation of applications at runtime. In particular, malleability increases system utilization while decreasing application execution times.

5. Integrating a mechanism for cycle scavenging seamlessly into grid-level scheduling obviates the need for additional software installations on the compute nodes or any modifications to the resource managers of the clusters, both of which are administrative obstacles in multicluster grid systems.

6. The performance of scheduling bags-of-tasks in multicluster grids depends not only on the task scheduling policy but also on the order that tasks are considered for scheduling, and on the resource management architecture being used.

7. When executing large workflows, the head-nodes of real grid clusters may become unstable, which as a result leads to much lower performance. To solve this problem, task throttling can be applied, that is, limiting the per-workflow number of tasks dispatched to the system, since task throttling prevents the head-nodes from becoming overloaded while largely preserving performance, at least for communication-intensive workflows.

8. The simple but widely used time series methods for predicting job execution times, and prediction methods that give upper-bounds for job queue wait times in multicluster grids, yield low accuracy in grids because of the frequent burst job submissions that we observe. In addition, a better accuracy of the predictions does not imply a better performance of grid scheduling. In fact, the main concern of a prediction-based grid scheduling policy is the correct prediction of the cluster that will finish a job first, rather than perfect prediction accuracy of the completion time of a job.

8.2 Suggestion for future work

The research described in this thesis reveals several possibilities for further investigation of application-oriented scheduling in multicluster grids:

1. In the Communication-Aware co-allocation policy, which is described in Chapter 2, we leave the network performance metric selection (either latency or bandwidth) to the users assuming that they know their applications' communication characteristics best. It would be more useful if the users could weigh these metrics depending on their applications. However, to provide a uniform metric combining bandwidth and latency for network evaluation is still an open problem [94].

2. In our design of support for scheduling malleable applications in Chapter 3, we have not considered grow operations that are initiated by the applications. This feature is mainly useful in case the parallelism patterns of the applications are irregular. Incorporating such grow operations is however not straightforward. For instance, it raises the design choice whether such grow operations should be mandatory or only voluntary. Another element that we have not incorporated in our design, yet would be interesting to add, is the malleability of co-allocated parallel applications.

3. The cycle scavenging system that we have presented in Chapter 4 can be a guideline for those who want to develop their own light-weight cycle scavenging system for their multicluster grid environment. In our design we restrict the number of active cycle scavenging users to improve the overall service quality. However, this admission control is administrated manually; implementing a dynamic solution that will incorporate predictions of future availability of idle resources may improve the overall service quality even further.

4. Our investigation of scheduling bags-of-tasks in Chapter 5 can be extended by evaluating additional scheduling heuristics, in particular of the kind that do not assume the a priori availability of workloads and resource information and collect this information dynamically.

5. In our experiments of Chapter 6, we have experienced that it is challenging to determine an ideal limit for task throttling when executing large workflows. If the limit is too low or too high, application performance may degrade due to the queueing and due to the excessive amount of inter-cluster communication, respectively. This fact motivates a future study of methods for dynamic task throttling mechanisms that change the task limit at runtime considering both the application characteristics and the system status.

6. Our evaluation of the performance of time series methods for predicting job runtimes and queue wait times in Chapter 7 does not completely answer the question of how to improve those methods or devise new ones such that they consider burst job submissions. In addition, it would be interesting to repeat such an investigation using more complex methods than time series, such as machine learning.

Bibliography

[1] IBM LoadLeveler: User's Guide, 1993. `http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp`.

[2] Attributes for communication between grid scheduling instances. pages 41–52, 2004.

[3] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance grid programming environments. Technical Report TR-04-09, Universita di Pisa, Dipartimento di Informatica, February 2004.

[4] J. Aldrich. R. A. Fisher and the making of maximum likelihood 1912-1922. *Statistical Science*, 12(3):162–176, 1997.

[5] W. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link. The globus striped GridFTP framework and server. In *Supercomputing*, pages 54–64, 2005.

[6] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. In *Proc. of the Scientific and Statistical Database Management Conference*, pages 21–23, 2004.

[7] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *Proc. of the 4th IEEE/ACM International Conference on Grid Computing (GRID'06)*, pages 4–10, 2004.

[8] Advanced School for Computing and Imaging. `http://www.asci.tudelft.nl/`.

[9] F. Azzedin, M. Maheswaran, and N. Arnason. A synchronous co-allocation mechanism for grid computing systems. *Cluster Computing*, 7(1):39–49, 2004.

[10] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[11] R. Bajaj and D. P. Agrawal. Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):107–118, 2004.

[12] H. E. Bal, N. Drost, R. Kemp, J. Maassen, R. van Nieuwpoort, C. van Reeuwijk, and F. J. Seinstra. Ibis: Real-world problem solving using real-world grids. In *Proc. of the International Symposium on Parallel and Distributed Processing (IPDPS'09)*, pages 1–8, 2009.

[13] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, L. Marchal, and Y. Robert. Centralized versus distributed schedulers for multiple bag-of-task applications. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, 2006.

[14] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, and Y. Yang. Scheduling divisible loads on star and tree networks: Results and open problems. *IEEE Transactions on Parallel and Distributed Systems*, 16(3):207–218, 2005.

[15] F. Berman, R. Wolski, H. Casanova, and W. Cirne. Adaptive computing on the grid using AppLeS. *IEEE Transactions on and Distributed Systems*, 14(4):369–382, 2003.

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

[17] V. Bharadwaj, D. Ghose, and T. G. Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Journal of cluster Computing*, 6(1):7–17, Jan. 2003.

[18] S. Bharathi and A. Chervenak. Data staging strategies and their impact on the execution of scientific workflows. In *Proc. of the International Workshop on Data-Aware Distributed Computing*, page 5, 2009.

[19] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. Characterization of scientific workflows. In *The 3rd Workshop on Workflows in Support of Large Scale Science, in conjunction with Supercomputing*, 2008.

[20] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task scheduling strategies for workflow-based applications in grids. In *Proc. of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, pages 759–767, 2005.

[21] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, June 2001.

[22] J. Brevik, D. Nurmi, and R. Wolski. Automatic methods for predicting machine availability in desktop Grid and peer-to-peer systems. In *Proc. of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'04)*, pages 190–199, 2004.

[23] J. Brevik, D. Nurmi, and R. Wolski. Predicting bounds on queuing delay for batch-scheduled parallel machines. In *Annual Symposium on Principles and Practice of Parallel Programming*, pages 110–118, 2006.

[24] P. J. Brockwell and R. A. Davis. *Introduction to Time Series and Forecasting*. Springer, March 2002.

[25] A. I. D. Bucur and D. H. J. Epema. The maximal utilization of processor co-allocation in multicluster systems. In *Proc. of the 17th International Symposium on Parallel and Distributed Processing (IPDPS'03)*, page 60.1, 2003.

[26] A. I. D. Bucur and D. H. J. Epema. The performance of processor co-allocation in multicluster systems. In *Proc. of the third International Symposium on Cluster Computing and the Grid (CCGRID'03)*, page 302, 2003.

[27] A. I. D. Bucur and D. H. J. Epema. Scheduling policies for processor coallocation in multicluster systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(7):958–972, 2007.

[28] J. Buisson, F. André, and J-L. Pazat. A framework for dynamic adaptation of parallel components. In *Proc. of International Conference on Parallel Computing (ParCo)*, pages 65–72, 2005.

[29] J. Buisson, F. André, and J.-L. Pazat. AFPAC: Enforcing consistency during the adaptation of a parallel component. *Scalable Computing: Practice and Experience (SCPE)*, 7(3):83–95, sep 2006.

[30] J. Buisson, F. Andre, and J-L. Pazat. Supporting adaptable applications in grid resource management systems. In *Proc. of the 8th IEEE/ACM International Conference on Grid Computing (GRID'07)*, pages 58–65, 2007.

[31] J. Buisson, O. O. Sonmez, H. H. Mohamed, L. Wouter, and D. H. J. Epema. Scheduling malleable applications in multicluster systems. In *Proc. of IEEE International Conference on Cluster Computing*, pages 372–381, 2007.

[32] E. K. Burke, M. Dror, and J. B. Orlin. Scheduling malleable tasks with interdependent processing rates: Comments and observations. *Discrete Applied Mathematics*, 156(5):620–626, 2008.

[33] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. In *Proc. of the fourth International Conference on High Performance computing in Asia-Pacific Region*, pages 283–289, 2000.

[34] R. Buyya and M. Murshed. GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience (CCPE,* 14(13):1175–1220, 2002.

[35] D. G. Cameron, A. P. Millar, C. Nicholson, R. Carvajal-Schiaffino, K. Stockinger, and F. Zini. Analysis of scheduling and replica optimisation strategies for data grids using OptorSim. *Journal of Grid Computing*, 2(1):57–69, 2004.

[36] T. E. Carroll and D. Grosu. A strategy proof mechanism for scheduling divisible loads in bus networks without control processors. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, 2006.

[37] H. Casanova. On the harmfulness of redundant batch requests. In *Proc. of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC'06)*, pages 255–266, 2006.

[38] H. Casanova, F. Berman, G. Obertelli, and R. Wolski. The AppLeS parameter sweep template: User-level middleware for the grid. In *Supercomputing*, pages 60–79, 2000.

[39] H. Casanova, D. Zagorodnov, F. Berman, and A. Legrand. Heuristics for scheduling parameter sweep applications in grid environments. In *Proc. of the International Heterogeneity in Computing Workshop*, page 349, 2000.

[40] C. Castillo, G. N. Rouskas, and K. Harfoush. Efficient resource management using advance reservations for heterogeneous grids. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, pages 1–12, 2008.

[41] L. Cherkasova and P. Phaal. Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Transactions on Computers*, 51:669–685, 2002.

[42] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63(5):597–610, 2003.

[43] T. Y. Choe and C. I. Park. A task duplication based scheduling algorithm with optimality condition in heterogeneous systems. In *Proc. of the International Conference on Parallel Processing Workshops*.

[44] W. Cirne, F. Brasileiro, N. Andrade, L. Costa, A. Andrade, R. Novaes, and M. Mowbray. Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246, 2006.

[45] W. Cirne, D. P. da Silva, L. Costa, E. Santos-Neto, F. V. Brasileiro, J. P. Sauvé, F. A. B. Silva, C. O. Barros, and C. Silveira. Running bag-of-tasks applications on computational grids: The MyGrid approach. In *ICPP*, pages 407–, 2003.

[46] Southern California Earthquake Center. `http://www.scec.org`.

[47] K. Czajkowski, I. Foster, and C. Kesselman. Resource co-allocation in computational grids. In *Proc. of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC'99)*, page 37, 1999.

[48] D. P. da Silva, W. Cirne, and F. V. Brasileiro. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Proc. of the European Conference on Parallel Processing (Euro-Par'03)*, pages 169–180, 2003.

[49] Condor DAGMan. `http://www.cs.wisc.edu/condor/dagman/`.

[50] H. Dail, F. Berman, and H. Casanova. A decoupled scheduling approach for grid application development environments. *Journal of Parallel and Distributed Computing*, 63(5):505–524, 2003.

[51] H. Dail, O. Sievert, F. Berman, H. Casanova, A. YarKhan, S. Vadhiyar, J. Dongarra, C. Liu, L. Yang, D. Angulo, and I. Foster. Scheduling in the grid application development software project. In *Grid resource management: state of the art and future trends*, pages 73–98. Kluwer Academic Publishers, 2004.

[52] The Distributed ASCI Supercomputer Project page. `http://www.cs.vu.nl/das/`.

[53] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus:

a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

[54] M. Dobber, G. Koole, and R. Van Der Mei. Dynamic load balancing for a grid application. In *Proc. of the International Conference on High Performance Computing*, pages 342–352, 2004.

[55] M. Dobber, G. Koole, and R. van der Mei. Dynamic load balancing experiments in a grid. In *Proc. of the fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, pages 1063–1070, 2005.

[56] M. Dobber, R. van der Mei, and G. Koole. A prediction method for job runtimes on shared processors: Survey, statistical analysis and new avenues. *Performance Evaluation*, 64(7-8):755–781, 2007.

[57] A. B. Downey. Predicting Queue Times on Space-Sharing Parallel Computers. In *Proc. of the 11th International Symposium on Parallel Processing*, pages 209–218, 1997.

[58] A. B. Downey. Using Queue Time Predictions for Processor Allocation. In *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 35–57, 1997.

[59] Distributed Resource Management Application API (DRMAA). `http://en.wikipedia.org/wiki/DRMAA`.

[60] R. Duan, R. Prodan, and T. Fahringer. Run-time optimisation of grid workflow applications. In *The 7th IEEE/ACM International Conference on Grid Computing*, pages 33–40, 2006.

[61] C. Dumitrescu and I. Foster. GangSim: a simulator for grid scheduling studies. In *Proc. of the fifth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'05)*, pages 1151–1158, 2005.

[62] C. L. Dumitrescu, I. Raicu, and I. Foster. Usage SLA-based scheduling in grids. *Concurrency and Computation: Practice and Experience*, 19(7):945–963, 2007.

[63] Enabling Grids for E-sciencE Project page. `http://eu-egee.com/`.

[64] G. Elliott, T. J. Rothenberg, and J. H. Stock. Efficient tests for an autoregressive unit root. *Econometrica*, 64(4):813–836, 1996.

[65] D. H. J. Epema, M. Livny, R. Dantzig, X. Evers, and J. Pruyne. A worldwide flock of Condors: Load sharing among workstation clusters. *Future Generation Computer Systems*, 12:53–65, 1996.

[66] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. On advantages of grid computing for parallel job scheduling. In *Proc. of the second IEEE International Symposium on Cluster Computing and the Grid (CCGRID'02)*, pages 39–46, May 2002.

[67] C. Ernemann, B. Song, and R. Yahyapour. Scaling of workload traces. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Proc. of the Workshops on Job Scheduling Strategies for Parallel Processing*, volume 2862, pages 166–182, 2003.

[68] The Eternity Puzzle. `http://www.eternityii.com`.

[69] M. Evans, N. Hastings, and B. Peacock. *Statistical Distributions*. Wiley, 3rd edition, 2000.

[70] D. Feitelson and A. Weil. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *Proc. of the 12th International Parallel Processing Symposium*, page 542, 1998.

[71] D. G. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Proc. of Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–26, 1996.

[72] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer Verlag, 1997.

[73] D. G. Feitelson and Larry Rudolph. Metrics and benchmarking for parallel job scheduling. In *Proc. of the 12th International Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processing*, volume 1459, pages 1–24, 1998.

[74] D. Fernandez-Baca. Allocating modules to processors in a distributed system. *IEEE Transactions on Software Engineering*, 15(11):1427–1436, 1989.

[75] Folding@home Project page. `http://folding.stanford.edu/`.

[76] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

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

[78] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving problems on concurrent processors. Vol. 1: General techniques and regular problems*. Prentice-Hall, Inc., 1988.

[79] E. Frachtenberg and D. G. Feitelson. Pitfalls in parallel job scheduling evaluation. In *Proc. of the Workshops on Job Scheduling Strategies for Parallel Processing*, pages 257–282, 2005.

[80] E. Freeman, S. Hupfer, and K. Arnold. *Javaspaces Principles, Patterns, and Practice: Principles, Patterns and Practices*. The Jini Technology Series. Addison-Wesley Longman, June 1999.

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

[82] N. Fujimoto and K. Hagihara. Near-optimal dynamic task scheduling of independent coarse-grained tasks onto a computational grid. In *Proc. of the International Conference on Parallel Processing*, pages 391–398, 2003.

[83] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proc. of the 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, 2004.

[84] Ganglia Monitoring System. `http://ganglia.sourceforge.net/`.

[85] N. Garg, D. Grosu, and V. Chaudhary. An antisocial strategy for scheduling mechanisms. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005.

[86] Federal coordinating council for science, engineering, and technology. A Research and Development Strategy for High Performance Computing, 1987.

[87] W. Gentzsch. Sun Grid Engine: Towards creating a compute power grid. In *Proc. of the first IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01)*, pages 35–42, 2001.

[88] Lightweight middleware for grid computing. `http://glite.web.cern.ch/glite/`.

[89] The Globus Toolkit. `http://www.globus.org/toolkit/`.

[90] Grid'5000 Project page. `http://www.grid5000.org/`.

[91] The Gridway metascheduler. `http://gridway.org/`.

[92] A. S. Grimshaw and Wm. A. Wulf. Legion - the next logical step toward the world-wide virtual computer. *Communications of the ACM*, 40:252, 1996.

[93] L. He, Stephen A. Jarvis, D. P. Spooner, D. Bacigalupo, G. Tan, and G. R. Nudd. Mapping DAG-based applications to multiclusters with background workload. In *Proc. of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, pages 855–862, 2005.

[94] S. Hongjie, F. Binxing, and Z. Hongli. A distributed architecture for network performance measurement and evaluation system. In *Proc. of the Sixth International Conference on Parallel and Distributed Computing Applications and Technologies*, pages 471–475, Washington, DC, USA, 2005. IEEE Computer Society.

[95] J. Hungershofer. On the combined scheduling of malleable and rigid jobs. In *Proc. of the 16th Symposium on Computer Architecture and High Performance Computing*, pages 206–213, 2004.

[96] J. Hungershofer, A. Streit, and J.-M. Wierum. Efficient resource management for malleable applications. Technical Report TR-003-01, Paderborn Center for Parallel Computing, December 2001.

[97] A. Iosup. *A Framework for the Study of Grid Inter-Operation Mechanisms*. PhD thesis, Delft University of Technology, 2009.

[98] A. Iosup, C. Dumitrescu, D. H. J. Epema, H. Li, and L. Wolters. How are real grids used? the analysis of four grid traces and its implications. In *Proc. of the 7th IEEE/ACM International Conference on Grid Computing (GRID'06)*, pages 262–269, 2006.

[99] A. Iosup and D. H. J. Epema. GRENCHMARK: A framework for analyzing, testing, and comparing grids. In *Proc. of the sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, pages 313–320, 2006.

[100] A. Iosup, D. H. J. Epema, C. Franke, A. Papaspyrou, L. Schley, B. Song, and R. Yahyapour. On grid performance evaluation using synthetic workloads. In *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 232–255, 2006.

[101] A. Iosup, D. H. J. Epema, T. Tannenbaum, M. Farrellee, and M. Livny. Inter-Operating grids through delegated MatchMaking. In *Supercomputing*, 2007.

[102] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. H. J. Epema. The grid workloads archive. *Future Generation Computer Systems*, 24(7):672–686, 2008.

[103] A. Iosup, O. O. Sonmez, S. Anoep, and D. H. J. Epema. The performance of bags-of-tasks in large-scale distributed systems. In *Proc. of International Symposium on High Performance Distributed Computing (HPDC'08)*, pages 97–108, 2008.

[104] A. Iosup, O. O. Sonmez, and D. H. J. Epema. DGSim: Comparing grid resource management architectures through trace-based simulation. In *Proc. of the European Conference on Parallel Processing (Euro-Par'08)*, volume 5168 of *LNCS*, pages 13–25, 2008.

[105] M. A. Iverson and G. J. Follen. Run-time statistical estimation of task execution times for heterogeneous distributed computing. In *Proc. of the ACM/IEEE International Symposium on High Performance Distributed Computing*, pages 263–270, 1996.

[106] M. A. Iverson and F. Özgüner. Hierarchical, competitive scheduling of multiple dags in a dynamic heterogeneous environment. *Distributed Systems Engineering*, 6(3):112–, 1999.

[107] M. A. Iverson, F. Ozguner, and L. C. Potter. Statistical Prediction of Task Execution Times Through Analytic Benchmarking for Scheduling in a Heterogeneous Environment. *IEEE Transactions on Computers*, 48:1374–1379, 1999.

[108] R. Iyer, V. Tewari, and K. Kant. Overload control mechanisms for web servers. In *Workshop on Performance and QoS of Next Generation Networks*, pages 225–244, 2000.

[109] K. Jansen and H. Zhang. Scheduling malleable tasks with precedence constraints. In *Proc. of the 17th annual ACM symposium on Parallelism in Algorithms and Architectures*, pages 86–95, 2005.

[110] K. Jansen and H. Zhang. An approximation algorithm for scheduling malleable tasks under general precedence constraints. *ACM Transactions on Algorithms*, 2(3):416–434, 2006.

[111] W. M. Jones, L. W. Pang, W. B. Ligon, and D. Stanzione. Bandwidth-aware co-allocating meta-schedulers for mini-grid architectures. In *Proc. of IEEE International Conference on Cluster Computing*, pages 45–54, 2004.

[112] W. M. Jones, L. W. Pang, D. Stanzione, and W. B. Ligon. Job communication characterization and its impact on meta-scheduling co-allocated jobs in a mini-grid. In *Proc. of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'04)*, volume 15, page 253b, 2004.

[113] Open Grid Forum: JSDL Specification 1.0. `www.ggf.org/documents/GFD.56.pdf`.

[114] L. V. Kalé, S. Kumar, and J. DeSouza. A malleable-job system for timeshared parallel machines. In *Proc. of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, page 230, 2002.

[115] W. Kang and A. Grimshaw. Failure prediction in computational grids. In *Proc. of the 40th Annual Simulation Symposium*, pages 275–282, 2007.

[116] N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive application-performance modeling in a computational grid environment. In *Proc. of International Symposium on High Performance Distributed Computing (HPDC'99)*, pages 47–54, 1999.

[117] G. Karypis and V. Kumar. Multilevel graph partitioning schemes. In *Proc. of the International Conference on Parallel Processing*, pages 113–122, 1995.

[118] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48:71–95, 1998.

[119] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, 1988.

[120] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[121] T. Kielmann, H. E. Bal, S. Gorlatch, K. Verstoep, and R. F. H. Hofman. Network performance-aware collective communication for clustered wide-area systems. *Parallel Computing*, 27(11):1431–1456, 2001.

[122] S. D. Kleban and S. H. Clearwater. Fair share on high performance computing systems: What does fair really mean? In *Proc. of the third IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)*, page 146, 2003.

[123] D. Kondo, G. Fedak, F. Cappello, A. A. Chien, and H. Casanova. Characterizing resource availability in enterprise desktop grids. *Future Generation Computer Systems*, 23(7):888–903, 2007.

[124] K. Kurowski, J. Nabrzyski, A. Oleksiak, and J. Weglarz. Grid scheduling simulations with GSSIM. In *Proc. of the 13th International Conference on Parallel and Distributed Systems*, pages 1–8, 2007.

[125] Y-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[126] B.-D. Lee and J. M. Schopf. Run-Time Prediction of Parallel Applications on Shared Environments. In *Proc. of IEEE International Conference on Cluster Computing*, pages 487–582, 2003.

[127] C. Lee and D. Talia. Grid programming models: Current tools, issues and directions. In G. Fox F. Berman and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 555–578. Wiley, 2003.

[128] J.-H. Lee, S.-H. Chin, H.-M. Lee, T.-M. Yoon, K.-S. Chung, and H.-C. Yu. Adaptive workflow scheduling strategy in service-based grids. In *Proc. of the 2nd International Conference on Grid and Pervasive Computing*, pages 298–309, 2007.

[129] K. Lee, N. W. Paton, R. Sakellariou, E. Deelman, A. A. Fernandes, and G. Mehta. Adaptive workflow processing and execution in pegasus. In *Proc. of the 3rd International Conference on Grid and Pervasive Computing*, pages 99–106, 2008.

[130] Y. L. Lee and A. Y. Zomaya. Practical scheduling of bag-of-tasks applications on grids with dynamic resilience. *IEEE Transactions on Computers*, 56(6):815–825, 2007.

[131] J. Li and R. Yahyapour. Negotiation model supporting co-allocation for grid scheduling. In *Proc. of the 7th IEEE/ACM International Conference on Grid Computing*, pages 254–261, 2006.

[132] D. A. Lifka. The ANL/IBM SP scheduling system. In *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 295–303, 1995.

[133] H. W. Lilliefors. On the Kolmogorov-Smirnov test for the exponential distribution with mean unknown. *Journal of the American Statistical Association*, 64:387–389, 1969.

[134] M. Litzkow, M. Livny, and M. Mutka. Condor - A hunter of idle workstations. In *Proc. of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[135] J. Livny, H. Teonadi, M. Livny, and M. K. Waldor. High-throughput, kingdom-wide prediction and annotation of bacterial non-coding rnas. *PloS one*, 3(9), 2008.

[136] The Load Sharing Facility. `http://www.platform.com/products/LSFfamily/`.

[137] U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105–1122, 2003.

[138] M. Maheswaran, S. Ali, H. Jay Siegel, D. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Proc. of the Heterogeneity in Computing Workshop*, pages 30–44, 1999.

[139] N. Mansour, R. Ponnusamy, A. Choudhary, and G. C. Fox. Graph contraction for physical optimization methods: a quality-cost tradeoff for mapping data on parallel computers. In *Supercomputing*, pages 1–10, 1993.

[140] Maui scheduler. `http://www.clusterresources.com/products/maui/`. Cluster Resources, Inc.

[141] C. McCann and J. Zahorjan. Processor allocation policies for message-passing parallel computers. In *Proc. of the ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 19–32, 1994.

[142] D. A. Menascé, D. Saha, S. C. S. Porto, V. Almeida, and S. K. Tripathi. Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures. *Journal of Parallel and Distributed Computing*, 28(1):1–18, 1995.

[143] L. Meyer, D. Scheftner, J. Vockler, M. Mattoso, M. Wilde, and I. Foster. An opportunistic algorithm for scheduling workflows on grids. In *Proc. of the International Meeting on High Performance Computing for Computational Science*, pages 1–12, 2006.

[144] H. H. Mohamed. *The Design and Implementation of the* KOALA *Grid Resource Management System*. PhD thesis, Delft University of Technology, 2007.

[145] H. H. Mohamed and D. H. J. Epema. An evaluation of the close-to-files processor and data co-allocation policy in multiclusters. In *Proc. of IEEE International Conference on Cluster Computing*, pages 287–298, 2004.

[146] H. H. Mohamed and D. H. J. Epema. The design and implementation of the KOALA co-allocating grid scheduler. In *Proc. of the European Grid Conference*, volume 3470 of *LNCS*, pages 640–650, 2005.

[147] H. H. Mohamed and D. H. J. Epema. Experiences with the KOALA co-allocating scheduler in multiclusters. In *Proc. of the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, pages 784–791, 2005.

[148] H. H. Mohamed and D. H. J. Epema. KOALA: a co-allocating grid scheduler. *Concurrency and Computation: Practice & Experience*, 20(16):1851–1876, 2008.

[149] Montage: An astronomical image engine. `http://montage.ipac.caltech.edu`.

[150] J. T. Moscicki. DIANE - distributed analysis environment for GRID-enabled simulation and analysis of physics data. In *Proc. of the IEEE Nuclear Science Symposium Conference*, pages 1617–1620.

[151] G. Mounie, C. Rapine, and D. Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *Proc. of the 11th ACM Symposium on Parallel Algorithms and Architectures*, pages 23–32, 1999.

[152] The Message Passing Interface (MPI) Standard. `http://www.mcs.anl.gov/research/projects/mpi/`.

[153] Grid ready MPI Library: MC-MPI Project page. `http://www.logos.ic.i.u-tokyo.ac.jp/h_saito/mcmpi/`.

[154] GridMPI Project page. `http://www.gridmpi.org/`.

[155] MPICH-G2 Project page. `http://www3.niu.edu/mpi/`.

[156] M. Mutka and M. Livny. Scheduling remote processing capacity in a workstation-processing bank computing system. In *Proc. of the 7th International Conference on Distributed Computing Systems*, pages 2–9, Berlin, Germany, September 1987.

[157] Myricom Technologies. `http://www.myri.com/`.

[158] F. Nadeem, R. Prodan, T. Fahringer, and A. Iosup. A Framework For Resource Availability Characterization And Online Prediction in the Grids. In *CoreGRID Integration Workshop*, pages 209–224, 2008.

[159] D. Nurmi, J. Brevik, and R. Wolski. QBETS: Queue Bounds Estimation from Time Series. In *SIGMETRICS*, pages 379–380, 2007.

[160] Open Grid Forum. http://www.gridforum.org/.

[161] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Transactions on Networking*, 2(2):137–150, 1994.

[162] S.-M. Park and M. Humphrey. Data throttling for data-intensive workflows. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, pages 1–11, 2008.

[163] D. Pease, A. Ghafoor, I. Ahmad, David L. Andrews, K. Foudil-Bey, T. E. Karpinski, M. A. Mikki, and M. Zerrouki. PAWS: A performance evaluation tool for parallel computing systems. *IEEE Computer*, 24(1):18–29, 1991.

[164] A. Plaat, H. E. Bal, and R. F. H. Hofman. Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects. *Future Generation Computer Systems*, 17(6):769–782, 2001.

[165] The Portable Batch System. http://www.pbspro.com/.

[166] The Prime Number Application. http://www.mhpcc.edu/training/workshop/mpi/samples/C/mpi_prime.c.

[167] The Parallel Workloads Archive Logs. http://www.cs.huji.ac.il/labs/parallel/workload/logs.html.

[168] Changtao Qu. A grid advance reservation framework for co-allocation and co-reservation across heterogeneous local resource management systems. In *Proc. of the 7th International Conference on Parallel Processing and Applied Mathematics*, pages 770–779, 2007.

[169] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a Fast and Light-weight tasK executiON framework. In *Supercomputing*, pages 323–356, 2007.

[170] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi. Scheduling data-intensiveworkflows onto storage-constrained distributed resources. In *Proc. of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'04)*, pages 401–409, 2007.

[171] T. Roblitz and A. Reinefeld. Co-reservation with the concept of virtual resources. In *Proc. of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, pages 398–406, 2005.

[172] T. Röblitz, F. Schintke, and A. Reinefeld. Resource reservations with fuzzy requests: Research articles. *Concurrency and Computing: Practice and Experience*, 18(13):1681–1703, 2006.

[173] Rosetta@home Project page. `http://boinc.bakerlab.org/rosetta/`.

[174] J. Sauer. Modeling and solving multi-site scheduling problems. In W. van Wezel, R. J. Jorna, and A. M. Meystel, editors, *Planning in Intelligent Systems: Aspects, Motivations and Methods*.

[175] J. M. Schopf. Ten actions when grid scheduling: the user as a grid scheduler. In *Grid resource management: state of the art and future trends*, pages 15–23. Kluwer Academic Publishers, 2004.

[176] F. J. Seinstra and J. M. Geusebroek. Color-based object recognition on a grid. In *ECCV Workshop on Computation Intensive Methods for Computer Vision*, May 2006.

[177] F. J. Seinstra, J.-M. Geusebroek, D. Koelma, C. G. M. Snoek, M. Worring, and A. W. M. Smeulders. High-performance distributed video content analysis with parallel-horus. *IEEE MultiMedia*, 14(4):64–75, 2007.

[178] SETI@home Project page. `http://setiathome.ssl.berkeley.edu/`.

[179] K. H. Shum. Adaptive Distributed Computing through Competition. In *Proc. of the 3rd International Conference on Configurable Distributed Systems*, page 220, 1996.

[180] D. P. Da Silva, W. Cirne, F. V. Brasileiro, and C. Grande. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Proc. of European Conference on Parallel Processing*, pages 169–180, 2003.

[181] W. Smith, I. Foster, and V. Taylor. Predicting Application Run Times Using Historical Information. In *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 122–142, 1998.

[182] W. Smith, V. Taylor, and I. Foster. Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance. In *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 202–219, 1999.

[183] D. Snelling. UNICORE and the open grid services architecture. *Grid Computing:Making the Global Infrastructure a Reality*, pages 701–712, 2003.

[184] A. C. Sodan, C. Doshi, L. Barsanti, and D. Taylor. Gang scheduling and adaptive resource allocation to mitigate advance reservation impact. In *Proc. of the 6th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'06)*, pages 649–653, 2006.

[185] O. O. Sonmez, B. Grundeken, H. H. Mohamed, A. Iosup, and D. H. J. Epema. Scheduling strategies for cycle scavenging in multicluster grid systems. In *Proc. of the ninth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'09)*, pages 12–19, 2009.

[186] O. O. Sonmez, H. H. Mohamed, and D. H. J. Epema. On the benefit of processor co-allocation in multicluster grid systems. *IEEE Transactions on Parallel and Distributed Systems*. to appear in 2010.

[187] O. O. Sonmez, H. H. Mohamed, and D. H. J. Epema. Communication-aware job placement policies for the KOALA grid scheduler. In *Proc. of the second IEEE International Conference on e-Science and Grid Computing*, pages 79–86, 2006.

[188] O. O. Sonmez, N. Yigitbasi, S. Abrishami, A. Iosup, and D. H. J. Epema. Performance analysis of dynamic workflow scheduling in multicluster grids. To appear in the Proc. of the International Conference on High Performance and Distributed Computing conference (HPDC'10).

[189] O. O. Sonmez, N. Yigitbasi, A. Iosup, and D. H. J Epema. Trace-based evaluation of job runtime and queue wait time predictions in grids. In *Proc. of International Symposium on High Performance Distributed Computing (HPDC'09)*, pages 111–120, 2009.

[190] SPECCPU Team. SPEC CPU2006. Standard Performance. `http://www.spec.org/cpu2006/`.

[191] V. Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, 2005.

[192] H. Stark and J. W. Woods. *Probability, random processes, and estimation theory for engineers*. Prentice-Hall, Inc., 1986.

[193] C. Stratan, A. Iosup, and D. H. J. Epema. A performance study of grid workflow engines. In *Proc. of the 9th IEEE/ACM International Conference on Grid Computing*, pages 25–32, 2008.

[194] TeraGrid Project page. `http://www.teragrid.org/`.

[195] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[196] H. Topcuoglu, S. Hariri, and M. Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[197] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803, 2007.

[198] G. Utrera, J. Corbalan, and J. Labarta. Implementing malleability on mpi jobs. In *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 215–224, 2004.

[199] S. Vadhiyar and J. Dongarra. Self adaptability in grid computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):235–257, February 2005.

[200] S. S. Vadhiyar and J. Dongarra. Srs: A framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(2):291–312, 2003.

[201] S. S. Vadhiyar and J. J. Dongarra. A metascheduler for the grid. In *Proc. of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC'02)*, page 343, 2002.

[202] R. F. van der Wijngaart. NAS parallel benchmarks version 2.4. Technical Report NAS-02-007, NASA Advanced Supercomputing division, October 2002.

[203] R. van Nieuwpoort, G. Wrzesinska, C. J. H. Jacobs, and H. E. Bal. Satin: a high-level and efficient grid programming model. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(3), 2010.

[204] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *Proc. of the 8th ACM SIGPLAN symposium on Principles and Practices of Parallel Programming*, pages 34–43, 2001.

[205] K. van Reeuwijk, R. van Nieuwpoort, and H. E. Bal. Developing java grid applications with ibis. In *Proc. of the European Conference on Parallel Processing*, pages 411–420, 2005.

[206] G. von Laszewski and M. Hategan. Java CoG Kit Karajan/Gridant Workflow Guide. `http://www.cogkit.org`.

[207] C. Weng and X. Lu. Heuristic scheduling for bag-of-tasks applications in combination with qos in the computational grid. *Future Generation Computer Systems*, 21(2):271–280, 2005.

[208] M. Wieczorek, R. Prodan, and T. Fahringer. Scheduling of scientific workflows in the ASKALON grid environment. *Special Interest Group on Management of Data*, 34(3):56–62, 2005.

[209] R. Wolski. Experiences with predicting resource performance on-line in computational grid settings. In *SIGMETRICS*, pages 575–611, 2006.

[210] J. Yang, I. Ahmad, and A. Ghafoor. Estimation of execution times on heterogeneous supercomputer architectures. In *International Conference on Parallel Processing*, pages 219–226, 1993.

[211] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *ACM Special Interest Group On Management of Data*, 34(3):44–49, 2005.

[212] J. Yu, R. Buyya, and R. Kotagiri. *Workflow Scheduling Algorithms for Grid Computing.* Springer, Berlin Germany, 2008.

[213] Z. Yu and W. Shi. An adaptive rescheduling strategy for grid workflow applications. In *Proc. of the International Parallel & Distributed Processing Symposium (IPDPS'10)*, page 115, 2007.

[214] Y. Zhang, C. Koelbel, and K. Cooper. Hybrid re-scheduling mechanisms for workflow applications on multi-cluster grid. In *Proc. of the 9th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'09)*, pages 116–123, 2009.

[215] Y. Zhang, W. Sun, and Y. Inoguchi. Predict task running time in grid environments based on cpu load predictions. *Future Generation Computer Systems*, 24(6):489–497, 2008.

[216] H. Zhao and R. Sakellariou. Scheduling multiple DAGs onto heterogeneous systems. In *Heterogeneity in Computing Workshop*, April 2006.

Acknowledgments

It is a pleasure to acknowledge the people who have contributed to this thesis. First and foremost, I would like to thank my research supervisor, Dick Epema, who has supported me throughout my doctoral study with his patience and knowledge while allowing me the room to work in my own way.

I offer my sincerest gratitude to Henk Sips, the head of the department of Software Technology and my promoter, who has been always there to listen and give advice.

I also thank the rest of my graduate committee members for their constructive comments on this thesis, and clearing their busy schedules to participate in my defense session.

I am indebted to my colleagues Hashim Mohamed, Alexandru Iosup, and Nezih Yiğitbaşı for their support, contributions, and true friendship. Hashim's and Alex's mentorship, and Nezih's enthusiasm have all facilitated my research. This dissertation could not have been accomplished without them.

It was a pleasure to work with Catalin Dumitrescu, Mathieu Jan, and Jérémy Buisson who were all once visitor researchers in our group. The outcome of our collaborative work constitutes an important part of this thesis. Special thanks go to Bart Grundeken, a former MSc student in our group, for his hard work in our cycle scavenging project. Many thanks to Paulo Anita for solving all our DAS-related problems. I am very grateful to all my colleagues at the PDS floor, who made it a convivial place to work.

My friends Zülküf Genç and Zekeriya Erkin deserve special mention for all their support and friendship that they have given me during my study in TU Delft. I am sure the (non-)scientific chats we had at 3pm tea-hours at the canteen of the EWI building contributed to this thesis one way or another. I wish you guys success with your thesis.

My special thanks go to my friend Fatma İnan (Fatoş), who helped me in translating the summary (samenvatting) of this thesis into Dutch.

Delft is a safe, attractive, and historic town, but boring unless you have good friends who can turn it into a vibrant city. In this sense, I owe my gratitude to my former housemates (aka Ternate Boys), all the good friends I have made during my stay in Delft as well as the ones who visited me, and in particular my football teammates in DIY and Veteranspor.

My deepest gratitude goes to my family for their unflagging support and love through-

out my life. Last but not least, thanks to Beste Ertürk, my beloved girlfriend who has always supported me with love and patience (as well as with delicious food) during the writing of this thesis.

Summary

Grid computing appeared in the mid 1990s with the vision of sharing geographically dispersed large computational resources for executing computation-intensive scientific applications. Today, we can name numerous grid projects that run successfully to solve challenging scientific problems such as the grid project of European Organization for Nuclear Research (CERN), which combines thousands of computers worldwide (over 200 sites in about 30 countries) to store and analyze huge amounts of data, which are produced by the Large Hadron Collider (LHC) at CERN.

The resources in a grid system are typically heterogeneous since they belong to different administrative domains, and they are managed by proprietary policies. To cope with this heterogeneity, a grid relies on a layer of middleware, which offers transparent access to the distributed resources and simplifies the collaboration between organizations. Grids also need high-level scheduling systems that use grid middleware in order to map application tasks to resources and then manage their execution on behalf of users. However, scheduling in grids is challenging due to the dynamic nature of the grid resources as well as to the lack of control of those resources. The wide variety in the structural and the communication characteristics of the applications submitted to grids further complicate grid scheduling, and may lead to poor or unpredictable performance unless these characteristics are taken into account.

In this thesis we address the challenge of designing and analyzing realistic and practical application-oriented scheduling mechanisms in multicluster grid systems. Application-oriented scheduling focuses on the optimization of user-centric performance criteria, such as application execution time, with methods that are specialized for different types of applications. In this thesis we cover a wide-range of grid application types, including parallel applications that may need co-allocation or malleability, bags-of-tasks that can benefit from cycle scavenging, and workflow applications that may push the system to its limits with their computation and data requirements. We investigate the performance of our scheduling mechanisms and policies in a real multicluster grid system, the DAS, using our KOALA multicluster grid scheduler, as well as with simulations using realistic scenarios.

In Chapter 1 of this thesis we give an overview of application-oriented scheduling

in multicluster grids, in particular, we focus on the challenges of application-oriented scheduling addressed by this thesis. In addition, we describe the testbed that we have used in our implementations and experiments.

In Chapter 2 we assess the benefit of processor co-allocation for parallel applications, that is, the simultaneous or coordinated allocation of processing resources at multiple clusters to single applications. We conduct our investigation on DAS-3 using our KOALA grid scheduler, as well as in a simulated environment using our DGSim tool. We evaluate the impact of various factors on the performance of co-allocation such as the communication characteristics of the parallel applications, the communication and computation characteristics on the resources, the level of resource contention in the system, and the scheduling policy that maps the tasks of a parallel application to the resources. Notably, we find that for very communication-intensive parallel applications, co-allocation is disadvantageous since it increases execution times extremely. In addition, we demonstrate that in systems where the heterogeneity in inter-cluster communication speeds is high, using network performance metrics in resource selection, such as the latency, increases the performance of co-allocation for communication-intensive parallel applications.

In Chapter 3 we extend our KOALA grid scheduler with support for malleable parallel applications that are able to use varying amounts of resources such as processors during their execution. We propose two malleability management policies, Favor Previously Started Malleable Applications (FPSMA) and Equi-Grow&Shrink (EGS), to manage dynamic resource allocation in the scheduler for malleable applications that have already been running in the system. FPSMA distributes any additional processors to the malleable applications starting from the application that has started earliest, while EGS spreads them equally among all running malleable applications. Each of these policies can be coupled with one of two approaches which either favor running or queued malleable applications when additional resources become available. Our experiments show that using malleability helps to increase system utilization as well as to decrease the execution times of parallel applications. We also find that the relative performance of our malleability management policies varies according to the design choice as to when to initiate a malleability management policy.

In Chapter 4 we extend KOALA with support for scheduling cycle scavenging applications, of which parameter sweeps are a prime example. The implemented cycle scavenging mechanism in KOALA runs alongside the regular grid scheduling, being unobtrusive to the jobs of higher priority. We propose two policies that try to achieve fair-share resource allocation among cycle scavenging users. The first policy distributes or reclaims the idle processing nodes evenly among cycle scavenging users, regardless of the cluster these idle nodes belong to. The second policy, on the other hand, partitions or reclaims the idle nodes evenly such that each cycle scavenging user is assigned an equal share of idle nodes on each cluster. We show with experiments conducted in the DAS-3 that the

latter policy outperforms the former in terms of fairness.

In Chapter 5 we present a detailed and systematic evaluation of the performance of scheduling bags-of-tasks in multicluster grids through realistic simulations. First, we propose a taxonomy of scheduling policies for bags-of-tasks that focuses on information availability and accuracy. Then, we explore the large design space of bag-of-task scheduling along five axes, which are the task selection policy, the workload, the information policy, the task scheduling policy, and the resource management architecture. We demonstrate that the task scheduling policies that make use of the available task and resource information perform better. We also find that the task selection policy is important only in busy systems, and we find that in terms of the resource management architecture, the centralized policy achieves the best performance.

In Chapter 6 we present a comprehensive and realistic investigation of the performance of a wide range of dynamic workflow policies in multicluster grids. We first introduce a scheduling taxonomy based on the amount of (dynamic) information used in the scheduling process, and we map to this taxonomy seven scheduling policies that cover the full spectrum of dynamic information use. Then we investigate the performance of these policies in realistic scenarios using both simulations and real system experiments. We find that none of the policies delivers good performance across all the investigated scenarios, and we find that task throttling, that is, limiting the per-workflow number of tasks dispatched to the system, prevents the cluster head-nodes from becoming overloaded while not unduly decreasing the runtime performance.

In Chapter 7 we investigate the performance and benefit of predicting job execution times and queue wait times in multicluster grids with simulations using traces collected from various research and production grid environments. Our analysis reveals that the time series methods for predicting job execution times, and prediction methods that give upper-bounds for job queue wait times, yield low accuracy because of the frequent burst job submissions that we observe in grids. In addition, we investigate whether prediction-based grid-level scheduling policies can have better performance than policies that do not use predictions. We find that a better accuracy of the predictions does not imply a better performance of grid scheduling.

Samenvatting

Grid computing ontstond in de jaren 1990 met als visie het delen van grote hoeveelheden geografisch gespreide rekenkracht voor het uitvoeren van rekenintensieve wetenschappelijke applicaties. Tegenwoordig kunnen er meerdere grid-projecten worden genoemd die succesvol complexe wetenschappelijke problemen kunnen oplossen, zoals het Grid-project van de Europese Organisatie voor Kernonderzoek (CERN), dat duizenden computers over de hele wereld met elkaar verbindt (meer dan 200 vestigingen in ongeveer 30 landen) om de grote hoeveelheden data te kunnen opslaan en analyseren die geproduceerd worden door de Large Hadron Collider (LHC) bij CERN.

De recources in een grid-systeem zijn typisch heterogeen omdat ze behoren tot verschillende administratieve domeinen, en ieder worden beheerd volgens hun eigen policy. Om te kunnen omgaan met deze heterogeniteit, zijn grids gebaseerd op een laag van middleware, die transparante toegang biedt tot de gedistribueerde resources en die de samenwerking tussen organisaties vereenvoudigt. Grids hebben tevens hoog-niveau scheduling systemen nodig die gebruik maken van grid middleware om resources te kunnen toewijzen aan de taken van applicaties en vervolgens de uitvoering daarvan te beheren namens de gebruikers. Echter, scheduling in grids is een uitdaging vanwege de dynamische aard van de grid resources alsmede het gebrek aan controle over deze resources. De grote verscheidenheid in de structuur en communicatie-kenmerken van de applicaties aangeboden aan grids maakt grid scheduling nog complexer, en kan leiden tot slechte of onvoorspelbare prestaties, tenzij met deze kenmerken rekening wordt gehouden. In dit proefschrift gaan we de uitdaging aan van het ontwerpen en analyseren van realistische en praktische applicatie-georiënteerde scheduling mechanismen in multicluster grid systemen. Applicatie-georiënteerde scheduling richt zich op de optimalisatie van gebruikersgerichte prestatiecriteria, zoals de verwerkingstijd van applicaties, met methoden die zijn gespecialiseerd voor verschillende soorten applicaties. In dit proefschrift bestrijken we een breed scala van grid applicatietypen, zoals parallelle applicaties die co-allocatie of “kneedbaarheid” nodig kunnen hebben, *bags-of-tasks* die kunnen profiteren van *cycle-scavenging*, en workflow applicaties die het systeem met hun vereisten aan reken- en datacapaciteit tot het uiterste kunnen dwingen. We onderzoeken de prestaties van onze scheduling mechanismen en policies in een echt multicluster grid-systeem, de DAS, met

behulp van onze KOALA multicluster grid scheduler, en met simulaties met behulp van realistische scenario's.

In Hoofdstuk 1 van dit proefschrift geven we een overzicht van applicatie-georiënteerde scheduling in multicluster grids, en richten we ons met name op de uitdagingen van applicatie-georiënteerde scheduling die aangepakt worden in dit proefschrift. Daarnaast beschrijven we het testbed dat we hebben gebruikt in onze implementaties en experimenten.

In Hoofdstuk 2 beoordelen we de voordelen van processor co-allocatie voor parallelle applicaties, dat wil zeggen, de gelijktijdige of gecoördineerde allocatie van processoren in meerdere clusters aan individuele applicaties. We voeren ons onderzoek uit op de DAS-3 gebruikmakend van onze KOALA grid scheduler, evenals in een gesimuleerde omgeving met behulp van onze DGSim simulator. We evalueren het effect van de verschillende factoren op de prestaties van co-allocatie, zoals de communicatiekenmerken van parallelle applicaties, de communicatie- en de computionele kenmerken van de resources, het niveau van de resource contentie in het systeem, en de scheduling policy die resources toewijst aan de taken van een parallelle applicatie. We concluderen dat voor zeer communicatie-intensieve parallelle applicaties co-allocatie nadelig is, omdat de verwerkingstijdtijd er enorm door toeneemt. Bovendien laten we zien dat in systemen waarin de heterogeniteit in inter-cluster communicatiesnelheden hoog is, het gebruik van netwerkprestatiemetrieken in de resource-selectie, zoals de *latency*, de prestaties van co-allocatie voor communicatie-intensieve parallelle applicaties verbetert.

In Hoofdstuk 3 breiden we onze KOALA grid scheduler uit met ondersteuning voor kneedbare parallelle applicaties die in staat zijn om wisselende hoeveelheden resources zoals processoren tijdens hun executie te gebruiken. We stellen hiervoor twee policies voor, Favor Previously Started Malleable Applications (FPSMA) en Equi-Grow&Shrink (EGS), om dynamische resource-allocatie in de scheduler te bewerkstelligen voor kneedbare applicaties die reeds gedraaid hebben in het systeem. FPSMA verdeelt extra processoren over de kneedbare applicaties beginnend met de applicatie die als eerste is gestart, terwijl EGS deze gelijkelijk verdeelt over alle draaiende kneedbare applicaties. Elk van deze policies kan gekoppeld worden aan een van de twee benaderingen die extra vrijkomende resources ofwel aan draaiende ofwel aan wachtende kneedbare applicaties toewijzen. Onze experimenten tonen aan dat het gebruik van kneedbaarheid helpt om de bezettingsgraad van het systeem te verhogen, en tevens om de verwerkingstijd van parallele applicaties te reduceren. We hebben tevens gemerkt dat de relatieve prestaties van onze twee policies varieert naar gelang de ontwerpkeuze van het tijdstip van het initiëren van zo'n policy.

In Hoofdstuk 4 breiden we KOALA uit met ondersteuning voor de scheduling van *cycle-scavenging* applicaties, waarvan *parameter sweeps* een goed voorbeeld zijn. Het in KOALA geïmplementeerde *cycle-scavenging* mechanisme draait naast de reguliere grid

scheduling, onzichtbaar voor de jobs met een hogere prioriteit. Wij stellen twee policies voor die proberen om *fair-share* resource-allocatie tussen de gebruikers van *cycle scavenging* te bereiken. De eerste policy verdeelt of vordert processoren gelijkelijk over de gebruikers van *cycle-scavenging*, ongeacht de cluster waartoe ze behoren. De tweede policy daarentegen partitioneert of vordert de processoren gelijkelijk over alle gebruikers van *cycle scavenging* in iedere cluster. We tonen met experimenten uitgevoerd in de DAS-3 aan dat de tweede policy beter presteert dan de eerste in termen van fairness.

In Hoofdstuk 5 presenteren we een gedetailleerde en systematische evaluatie van de prestaties van de scheduling policies voor *bags-of-taks* in multicluster grids door middel van realistische simulaties. Ten eerste stellen we een taxonomie voor van scheduling policies voor *bags-of-tasks* die is gebaseerd op de beschikbaarheid en nauwkeurigheid van informatie. Vervolgens onderzoeken we de grote ontwerpruimte van bag-of-task scheduling langs vijf assen, te weten de policy voor taakselectie, de werklast, de informatiepolicy, de taak scheduling policy, en de architectuur voor het beheer van de resources. We laten zien dat de taak scheduling policies die gebruik maken van de beschikbare informatie omtrent taken en resources beter presteren. Wij laten tevens zien dat de policy voor taakselectie alleen van belang is in drukke systemen, en we laten zien dat de gecentraliseerde policy in termen van de resource management architectuur zorgt voor de beste prestaties.

In Hoofdstuk 6 presenteren we een uitvoerig en realistisch onderzoek naar de prestaties van een breed scala aan dynamische workflow policies in multicluster grids. We introduceren eerst een scheduling taxonomie gebaseerd op de hoeveelheid (dynamische) informatie die gebruikt wordt in het scheduling proces, en we categoriseren zeven scheduling policies die het volledige spectrum van dynamisch informatiegebruik bestrijken aan de hand van deze taxonomie. Vervolgens onderzoeken we de prestaties van deze policies in realistische scenario's met behulp van zowel simulaties als met echte systeemexperimenten. Wij laten zien dat geen van de policies goede prestaties levert in alle onderzochte scenario's, en dat taak *throttling*, dat wil zeggen het beperken van het aantal taken per workflow dat in het systeem draait, voorkomt dat de *head nodes* van de clusters worden overbelast zonder de verwerkingstijden van de workflows al te zeen te verslechteren.

In Hoofdstuk 7 onderzoeken we de prestaties en de voordelen van het voorspellen van de verwerkingstijden en wachttijden van jobs in multicluster grids met simulaties met behulp van *traces* van verschillende onderzoeks- en productiegrids. Onze analyse toont aan dat tijdreeksmethoden voor het voorspellen van verwerkingstijden en voorspellingsmethoden die bovengrenzen geven voor wachttijden, niet erg nauwkeurig zijn vanwege de frequente *bursts* in de job-aankomsten in grids. Daarnaast onderzoeken we of grid-level scheduling policies gebaseerd op voorspellingen betere prestaties leveren dan policies die geen voorspellingen gebruiken. We vinden dat een grotere nauwkeurigheid van de voor-

spellingen niet leidt tot betere prestaties in grid scheduling.

About the author

Ömer Ozan Sönmez was born in Istanbul, Turkey, on September 6th, 1980. He received a BSc degree in computer engineering from İstanbul Technical University, Turkey, in 2003, and MSc degree in computer science from the Koç University, Turkey, in 2005. In November 2005, he joined the Parallel and Distributed Systems Group of Delft University of Technology, Delft, The Netherlands, as a PhD student.

His research interests focus on resource management and scheduling in multicluster systems and in particular grids. Throughout his academic career, he assisted several courses, gave lectures, and supervised an MSc student. He has several publications in prestigious conference proceedings and journals. Since November 2009, he has been working as a researcher in the same department where he pursued his PhD.

Ozan's main personal enjoyments and interests are throwing barbecue parties, having Belgian beers with friends, traveling, watching movies, and watching&playing football.

List of referred publications

Journal papers

- O. O. Sonmez, H. H. Mohamed, and D. H. J. Epema. On the Benefit of Processor Co-Allocation in Multicluster Grid Systems. **to appear** *in IEEE Transactions on Parallel and Distributed Systems*.
- O. O. Sonmez, and A. Gursoy. A Novel Economic-Based Scheduling Heuristic for Computational Grids. *Journal of High Performance Computing Applications*, 21(1), pages. 21-29, 2007.

Conference papers

- O. O. Sonmez, M. N. Yigitbasi, S. Abrishami, A. Iosup, and D.H.J. Epema. Performance Analysis of Dynamic Workflow Scheduling in Multicluster Grids. **to appear** *in Proc. of the ACM/IEEE International Symposium on High Performance Distributed Computing (HPDC'10)*.

- O. O. Sonmez, M. N. Yigitbasi, A. Iosup, and D. H. J. Epema. Trace-Based Evaluation of Job Runtime and Queue Wait Time Predictions in Grids. *In Proc. of the ACM/IEEE International Symposium on High Performance Distributed Computing (HPDC'09)*, pages. 111-120, 2009.

- O. O. Sonmez, B. Grundeken, H. H. Mohamed, A. Iosup, and D. H. J. Epema. Scheduling Strategies for Cycle Scavenging in Multicluster Grid Systems. *In Proc. of the 9th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'09)*, pages. 12-19, 2009.

- A. Iosup, O. O. Sonmez, and D. H. J. Epema. DGSim: Comparing Grid Resource Management Architectures Through Trace-Based Simulation. *In Proc. of the European Conference on Parallel Processing (Euro-Par'08)*, pages. 13-25, 2008.

- A. Iosup, O. O. Sonmez, S. Anoep, and D. H. J. Epema. The Performance of Bags-of-Tasks in Large-Scale Distributed Computing Systems. *In Proc. of the ACM/IEEE International Symposium on High Performance Distributed Computing (HPDC'08)*, pages. 97-108, 2008.

- A. Iosup, M. Jan, O. O. Sonmez, and D. H. J. Epema. On the Dynamic Resource Availability in Grids. *In Proc. of the IEEE International Conference on Grid Computing*, pages. 26-33, 2007.

- A. Iosup, M. Jan, O. O. Sonmez, and D. H. J. Epema. The Characteristics and Performance of Groups of Jobs in Grids, *In Proc. of the European Conference on Parallel Processing (Euro-Par'07)* pages. 382-393, 2007.

- J. Buisson, O. O. Sonmez, H. H. Mohamed, W. Lammers, and D. H. J. Epema. Scheduling Malleable Applications in Multicluster Systems, *In Proc. of the IEEE International Conference on Cluster Computing*, pages. 372-381, 2007.

- C. Dumitrescu, A. Iosup, O. O. Sonmez, H. H. Mohamed, and D. H. J. Epema. Virtual Domain Sharing in e-Science based on Usage Service Level Agreements. *In Proc. of the CoreGrid Symposium*, pages. 15-25, 2007.

- O. O. Sonmez, H. H. Mohamed, and D. H. J. Epema, Communication-Aware Job Placement Policies for the Koala Grid Scheduler. *In Proc. of the IEEE International Conference on e-Science*, pages. 79-87, 2006.

- O. O. Sonmez, A. Gursoy, Comparison of Pricing Policies for a Computational Grid Market. *In Proc. of the International Conference on Parallel Processing and Applied Mathematics*, pages. 766-773, 2005.