RICE UNIVERSITY

Grid-Centric Scheduling Strategies for Workflow Applications

by

Yang Zhang

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Keith D. Cooper
L. John and Ann H. Doerr Professor of
Computational Engineering,
Computer Science

Charles Koelbel
Research Scientist, Computer Science

Tim Warburton
Associate Professor,
Computational and Applied Mathematics

Rice University, Houston, Texas

September, 2009

Grid-Centric Scheduling Strategies for Workflow Applications

Yang Zhang

Abstract

Grid computing faces a great challenge because the resources are not localized, but distributed, heterogeneous and dynamic. Thus, it is essential to provide a set of programming tools that execute an application on the Grid resources with as little input from the user as possible. The thesis of this work is that *Grid-centric scheduling techniques of workflow applications can provide good usability of the Grid environment by reliably executing the application on a large scale distributed system with good performance.* We support our thesis with new and effective approaches in the following five aspects.

First, we modeled the performance of the existing scheduling approaches in a multi-cluster Grid environment. We implemented several widely-used scheduling algorithms and identified the best candidate. The study further introduced a new measurement, based on our experiments, which can improve the schedule quality of some scheduling algorithms as much as 20 fold in a multi-cluster Grid environment.

Second, we studied the scalability of the existing Grid scheduling algorithms. To deal with Grid systems consisting of hundreds of thousands of resources, we designed and implemented a novel approach that performs explicit resource selection decoupled from scheduling. Our experimental evaluation confirmed that our decoupled approach can be scalable in such an environment without sacrificing the quality of the schedule by more than 10%.

Third, we proposed solutions to address the dynamic nature of Grid computing

with a new cluster-based hybrid scheduling mechanism. Our experimental results collected from real executions on production clusters demonstrated that this approach produces programs running 30% to 100% faster than the other scheduling approaches we implemented on both reserved and shared resources.

Fourth, we improved the reliability of Grid computing by incorporating fault-tolerance and recovery mechanisms into the workow application execution. Our experiments on a simulated multi-cluster Grid environment demonstrated the effectiveness of our approach and also characterized the three-way trade-off between reliability, performance and resource usage when executing a workflow application.

Finally, we improved the large batch-queue wait time often found in production Grid clusters. We developed a novel approach to partition the workow application and submit them judiciously to achieve less total batch-queue wait time. The experimental results derived from production site batch queue logs show that our approach can reduce total wait time by as much as 70%.

Our approaches combined can greatly improve the usability of Grid computing while increasing the performance of workow applications on a multi-cluster Grid environment.

Acknowledgements

I would like to thank my late advisor, Ken Kennedy, for his guidance and support without which this dissertation would not have been possible. Ken not only introduced me to the cutting edge domain of high performance computing research but also inspired me to continue my work goes beyond this dissertation. I am grateful to my advisor, Dr. Keith Cooper. Keith Cooper has been extremely supportive of all my efforts and took me under his wings after Ken passed away. I am also very grateful to Dr. Chuck Koelbel. He has been a constant source of detailed help regarding all areas of my research, from research problem definitions to conference talk presentations, ever since I started. I also would like to thank my committee member, Dr. Tim Warburton, for his support and insights.

I worked in the VGrADS projects throughout my PhD years and I would like to take this opportunity to thank all the members. In particular, I would like to thank Yang-suk Kee, Lavanya Ramakrishnan, Daniel Nurmi and especially Anirban Mandal for their help in implementation at various points of time. Research discussions with several VGrADS PIs like Andrew Chien, Rich Wolski, Henri Casanova and Dan Reed have helped me shape my research. I would like to thank all my friends and research staff in the compiler group for the vibrant work environment Nathan Tallent, David Peixotto, Jason Eckhardt, Jeff Sandval, Yuan Zhao, Rui Zhang, Anshu Dasgupta,Yuri Dotsenko, Apan Qasem, Cheryl McCosh, Alex Grosul, Cristian Coarfa, Zoran Budimlic, Timothy Harvey and Yi Guo. My sincere thanks to all the supporting staffs at the Computer Science department, including Penny Anderson, Darnell Price, Lena Sifuentes, Bel Martinez, Amanda Nokleby, Iva Jean Jorgensen and BJ Smith for being so helpful and supportive.

Finally, my parents and wife are invaluable. They have been with me throughout this long journey and have provided all the support and encouragement that make me who I am. This dissertation is especially for my daughter, Sunny, all I pray for is her happiness.

Contents

Illustrations

Chapter 1

Introduction

Advances in networking technologies have made it possible to use distributed information infrastructures as a computational resource as well as an information resource that we refer to as the Grid [40, 41]. Foster *et al.* [40] described it as a distributed infrastructure that connects computers, databases, instruments, and people in a seamless web of computing and distributed intelligence, that can be used in an on-demand fashion as a problem-solving resource in many fields of human endeavor. Just as the electric power grid provides electricity, the ultimate "Grid" vision is to provide pervasive access to large scale computation and data as an integrated problem-solving resource in diverse fields of science, engineering and commerce.

Since the inception of Grid, it has advanced from an ambitious vision pursued by a small number of academic researchers into a large-scale research and production activity involving hundreds of scientists and engineers. The Grid vision of flexible, large-scale deployment and resource sharing across multiple organizations has spawned not only a wealth of research [4, 82, 3], but also commercial products [36, 112, 77] and large-scale deployment [107, 45] used by both scientific and commercial applications [23, 110, 96, 34, 103, 1, 38, 60, 28, 47]. Those computational and data Grids provide access to software and hardware resources geographically distributed and maintained by different institutions. In more recent development, the Grid is evolving towards the "cloud" computing but Grid technologies will still play a critical role in the new cloud computing.

1.1 Motivation

The availability of Grid resources gave rise to a new computing paradigm: Grid computing. Unlike scalar or parallel computing, Grid computing enables users to share their resources, data and software instead of competing for them. Such collaborations have widespread appeal for the distributed and high performance computing communities. However, as with most new technologies, only part of the Grid's potential is currently a reality.

One of the fundamental challenges in Grid computing is that Grid applications typically involve massive task-parallelism and may include processing of large-scale data. The tera or even peta-scale of the applications not only put stress on the Grid software stacks but also on the hardware infrastructures. Another fundamental challenge is that the Grid applications run on resources that are distributed, heterogeneous, dynamic and sometimes unpredictable. These characteristics of the resources have largely confined the use of Grid computing to engineers and scientists with extensive training and experience. Finally, the synchronous use of shared resources distributed across multiple organization and administrative domains is largely unrealized [9] partly because it requires a form of co-scheduling. The underlying technical challenge is that a Grid application needs to coordinate with several local resource managers to allocate enough resources for itself.

Because the Grid environment is inherently more complex than previous computer systems, applications that execute on the Grid would inevitably reflect some of this complexity. However, we believe that it is possible to encapsulate the complexity of Grid computing away from the application developers. We believe that the key to make the Grid usable lies in sophisticated programming tools that embody major advances in both the theory and practice of building Grid applications. Our ultimate vision is that Grid application developers will write component-based workflow applications with the help of workflow generator tools. Workflow applications are an important class of applications that consist of multiple sub-tasks linked to each

other by dependences. The developers can be Grid-oblivious as long as they express the general software and hardware requirements for the application. Then, a user will submit the Grid application through a user portal and a Grid runtime system will automatically execute it on the matching Grid resources. A Grid runtime system usually consists of one global resource manager and one application manager per application. Finally, the Grid application finishes and automatically stages all the results to a user designated place for the user to collect. Unfortunately, the reality is, most Grid application developers have been experts in distributed computing and the users run the application by directly invoking remote procedure calls through the Grid middleware. Therefore, instead of our vision that users can submit an application onto Grid and leave it unattended, users need to continuously monitor the status of the application during its execution.

In an effort to fulfill our ambitious vision, we focused on the development of a good application manager since it plays a critical role in the automation process. A fully automated application manager identifies the application requirements, selects Grid resources for the applications, coordinates with the global resource manager to allocate enough resources, schedules the application, executes the application in the right order, monitors the execution and reschedules or relaunches the application in case an unexpected event happens. We also believe that a good application manager is key to the performance of the application. It is already known that the distributed, heterogeneous and dynamic characteristics of the Grid resources can unexpectedly hurt the performance of the application [129]. Thus, one important task for the application manger is to select and allocate the right resources and schedule the application onto these resources in a way to minimize the execution time which we usually refer to as the *turn-around time*. At the heart of this procedure is the scheduling problem which is known to be NP-complete except in the simplest scenarios [43]. Traditional scheduling assumes that the performance of the application on a certain resource is not only known but also invariant. In a heterogeneous dynamic Grid environment,

these assumptions do not hold. Thus, Grid application scheduling poses even larger challenges than does scheduling in a static homogenous environment.

1.2 Thesis

The thesis of this work is that *Grid centric scheduling techniques of workflow applications can provide good usability of the Grid environment by reliably executing the application on a large scale distributed system with good performance.* To support this thesis, we designed and implemented several workflow application scheduling mechanisms in the context of the Virtual Grid Application Development Software (VGrADS) project. Our researches have provided the necessary techniques to reliably execute a workflow application on a Grid environment and achieve good performance (turn-around time) and scalability.

1.3 Research Contributions

The main contribution this thesis work is a set of published Grid-centric scheduling techniques for workflow applications.

- We studied the performance of existing scheduling approaches in the Grid environment. I analyzed the results and introduced a new measurement called effective aggregated computing power (EACP) that could improve the results of some scheduling algorithms by as much as 20 fold [127].

- We studied the scalability of existing scheduling approaches and designed and implemented a decoupled two-level approach that performs explicit resource selection decoupled from scheduling. The experimental results confirmed that our approach can be scalable in a large Grid environment without sacrificing the quality of the schedule by more than 10%. [130].

- We proposed a new cluster-based hybrid scheduling mechanism that dynamically executes a top-down static scheduling algorithm using the real-time feed-

back from the execution monitor. The experimental execution results showed that this approach produces programs running 30% to 100% faster than the other scheduling approaches we implemented on both reserved and shared resources [129].

- We incorporated fault-tolerance and recovery mechanisms into workflow application scheduling and execution that improve the reliability of Grid computing by at much as 250% when the resources are unreliable [131].

- We proposed a new approach to aggregate a workflow application into several groups and submit them according to the batch queue wait time estimation to reduce the workflow's waiting time in the batch queues on production sites by as much as 80% [128].

1.4 Organization

The thesis is organized as follows. In Chapter 2, we present the background of our research and related works. In Chapter 3, we present our study of the various scheduling algorithms on a multi-cluster Grid environment and propose and evaluate our new EACP approach. In Chapter 4, we present our two-level scheduling strategy that addresses the scalability issues for Grid applications. In Chapter 5, we present a novel two-level cluster based hybrid rescheduling technique and its evaluation on a real multi-cluster Grid. In Chapter 6, we present our work on incorporating the fault-tolerance and recovery mechanism with workflow application scheduling. In Chapter 7, we present a novel workflow aggregation algorithm that can reduce a workflow application's wait time in batch queue controlled resources. Finally, we conclude our dissertation in Chapter 8.

Chapter 2

Background and Related Work

In this chapter, we first describe Grid computing from the perspective of our research. Then we list related works on Grid middleware systems including the GrADS project, its successor the VGrADS project on which this thesis work is built, the Globus project and other Grid projects. Secondly, we present the background of workflow applications and list some existing workflow application management systems. Finally, we present workflow application scheduling techniques and related works on scheduling strategies for homogeneous, heterogeneous and Grid platforms. We will also present comparisons with our work when appropriate.

2.1 Grid Computing

2.1.1 Overview

The Grid is a distributed infrastructure that connects computers, databases, instruments, and people into a seamless web of advanced capabilities [40]. There are many types of Grid, such as computational Grid, desktop Grid, data Grid and utility Grid, to name a few. In our research, we focus on multi-cluster computational Grid. A multi-cluster Grid composes of several clusters that are physically located in a geographically distributed manner and its main purpose is to provide enough computational resources to accommodate applications with large computational needs. Grid computing is a new computing paradigm that could harness the computing power of a Grid. In our multi-cluster Grid environment, Grid computing means a user submits a large application, usually a workflow application, through an application manager onto one or more clusters and the user can collect the results later without further

intervention. In other words, we see Grid computing as an automation for a user to harness the computing power of multiple clusters or a large number of distributed resources in general.

A flurry of research projects has been proposed on different aspects of the Grid computing around the world since the inauguration of the Grid concept. A non-exhaustive list includes the GrADS project [9], the VGrADS project [55], the Globus project [39], the Condor project [109], the Enabling Grids for E-sciencE (EGEE) project [8], the GridBus project [83], the GridFlow project [16], the GridLab project [3], the TeraGrid project [107] and the Unicore project [93]. Here we summarize some of the more related and more influential ones on our research.

2.1.2 Grid Projects Related to Our Research

Globus

The Globus project, started in 1996, produces by far the mostly widely-used Grid middleware, the Globus toolkit [39]. It is an open source toolkit that provides fundamental technologies for people to share computing power, databases, and other tools securely online across corporate, institutional, and geographic boundaries without sacrificing local autonomy. The toolkit includes software services and libraries for remote procedure call, resource monitoring, discovery, and management, plus security and file management.

The latest release of Globus toolkit version 4(GT4) converged with web services standards on building Grid middleware and service-oriented-applications. A web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine processable format called Web Service Definition Language (WSDL). Other systems interact with the Web service in a manner prescribed by its description using Simple Object Access Protocol (SOAP) messages [90, 99]. In this way, a user can write his own client program that invokes services reside in the Grid server. Several standard services are

implemented in a Globus toolkit 4 container.

The three mostly commonly used services are the job management service,reliable file transfer service and the delegation service. The Grid Resource Allocation and Management (GRAM) service enables users to execute a program remotely and get a handle to manage the job. A client can then use this handle to query the jobs status, kill the job and obtain notifications if the job status changes. The Reliable File Transfer (RFT) service enables users to stage in all the necessary files before a remote job starts and stage out the results to the next computing job. The delegation service can delegate a user's credential through the web service security authentication system so that a remote job can run with the same permission as the user on that resource.

Both the GrADS and the VGrADS project use Globus as their infrastructure and add user level services on top of it. We use Globus in most of our work directly or indirectly (through VGrADS), as do many other Grid proejcts. However, GT4 services alone can not provide the type of automation and performance we are looking to achieve, because Globus is designed to provide the basic functionalities for distributed computing and is largely single job oriented, thus not directly applicable to a workflow application execution.

GrADs

Since 1999, the Grid Application Development (GrADS) Project has worked to attack the problems inherent in Grid computing [26]. The GrADS research has focused on five inter-institutional efforts: Program Execution System, Program Preparation System, Macro Testbed, MicroGrid, and Applications. Based on those inter-institutional projects, the GrADS project proposed two key concepts [58]. First, applications are encapsulated as *configurable object programs* (COPs) which include not only the code for the application but also a portable strategy for mapping the program onto the available distributed resources and a mechanism to evaluate how well that mapped

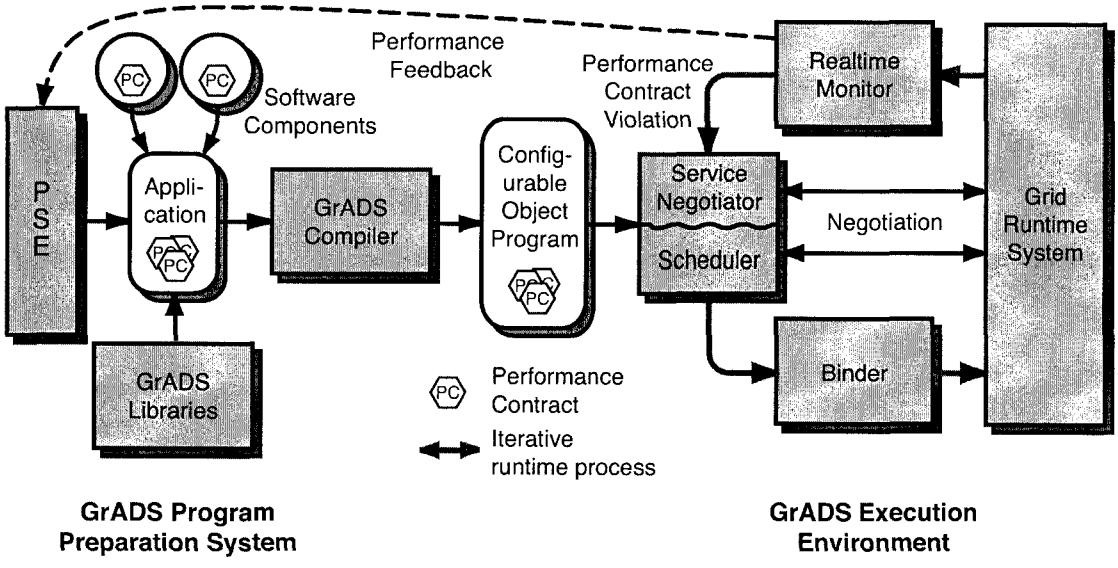


Figure 2.1 : GrADSoft Architecture

program will run. Second, the system relies upon performance contracts that specify the expected performance of modules as a function of available resources [115].

Figure 2.1 from Kennedy *et al.* [58] illustrates the overall architecture of the GrADS software.The left side of Figure 2.1 depicts tools used to construct COPs from either domain-specific ready-to-use components such as a MPI or multi-threaded program. The right side of Figure 2.1 depicts actions when a COP is delivered to the execution environment. The GrADS infrastructure first determines which resources are available and uses the mapper to map the application components onto an appropriate subset of these resources. Then the GrADS software invokes the binder to tailor the COP to the chosen resources and starts it on the Grid. Once launched, contract monitor[71] tracks its execution and detects anomalies. The rescheduler takes corrective action if necessary based on the monitor's feedback and the application's requirement.

The GrADS project demonstrated through many proof-of-concept experiments

that it is possible to construct reasonably efficient schemes, such as process swapping, checkpoint/restart and dynamic load balancing, for dynamic rescheduling of Grid applications onto different resources during execution. It implemented a migration framework that takes into account both the system load and application characteristics with the help of a contract monitor. It also showed that high performance can be achieved on the Grid for several different kinds of numerical applications with a low implementation and execution overhead. The GrADS framework can handle applications from varying disciplines with varying requirements, such as the biological sequence alignment application FASTA [121], the propositional satisfiability problem solver GridSAT [24] and the computationally demanding problem of determination of 3-D structure of large macromolecular complexes from electron cryomicroscopy [66]. GrADS project laid the foundation of the workflow application execution on a Grid and also led to the VGrADS project that this thesis is built on.

VGrADs

The Virtual Grid Application Development Software (VGrADS) project is based on the earlier GrADS project. It extends GrADS by introducing the concept of virtual grids (VGs), that is, sets of selected and bound resources [55]. The virtual grid execution system (vgES) provides an additional level of abstraction and implements a simple interface for resource specification, resource selection, and resource binding in a complex Grid environment. Figure 2.2 from Kee *et al.* [55] illustrates the overall architecture of the original vgES architecture. The system includes a novel resource description language (vgDL), a resource selection and binding component (vgFAB), a dynamic resource information retrieving component(vgAgent), a distributed monitoring component (vgMON) and an application launcher (vgLaunch). The system is built on top of the Globus middleware [4] which provides the standard Grid resource allocation and management (GRAM) service.

The concept of virtual grid allows separation of concerns between levels of the

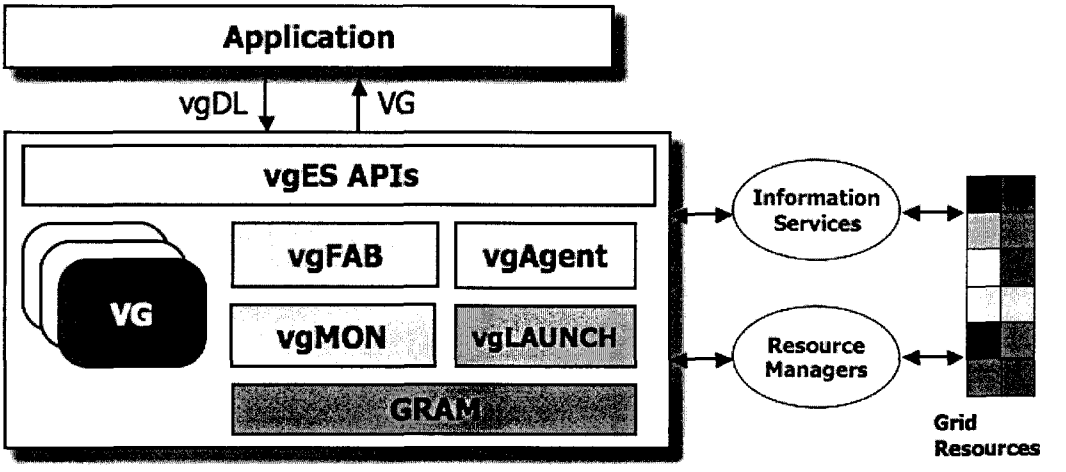


Figure 2.2 : Virtual Grid Execution System (vgES) Architecture

system and is the key to allow scalable scheduling. Results show that resource selection and binding for virtual grids of tens of thousands of resources can scale up to Grids with millions of resources while identifying good matches in less than one second [22]. Recently the virtual grid execution system extended support to include a "slot" resource abstraction, representing not only the quantity but also the duration when resources are available. Combined with advanced batch queue delay prediction and application performance prediction, the slot-based resource abstraction makes it possible for a Grid application to virtually reserve resources it needs on a busy batch queue controlled resource. All together, they provide the leverage for scheduling availability and allow more-advanced applications to run with VGrADS support.

My thesis work is part of the VGrADS project. The work in Chapter 4 was built on top of the virtual grid execution system and used vgDL to describe resource needs for a workflow application. The work in other chapters are also motivated by the collaboration work with other project members in various phases of the VGrADS project and could be incorporated into the virtual Grid abstraction.

2.1.3 Other Grid Projects

Condor

Started in 1988, the Condor project has been focusing on customers with large computing needs and environments with heterogeneous distributed resources [82]. Condor is a specialized workload management system that provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. However, its main goal is to achieve high throughput for a system, not a typical object for high performance computing where the main objective is to finish a large amount of computation as fast as possible. It is not the case for Condor where an application would wait for a resource to become available.

The first generation of the Condor system grouped agents, resources, and matchmakers together to form what they called a Condor pool. The user submits jobs to an agent. Then the agent advertises itself through the ClassAd mechanism to a matchmaker, which is responsible for searching potential matching agent and resource pairs. Once introduced, the agent will contact the resource. If the resource is available, a safe execution environment (sandbox) will be created for the job to protect the resource from any mischief [109]. Although this version of the Condor system enabled users to harness the computing power of many workstations with a single portal, the size of the Condor pool was limited by having only one matchmaker.

The second generation of the Condor system introduced the concept of *flocking* which allows resource sharing between different Condor installations. There are two flavors of flocking, gateway flocking and direct flocking. In a gateway flocking Condor system, the structure of two existing pools is preserved, while two gateway nodes pass information about participants between the two pools. In a direct flocking Condor system, an agent can advertise its ClassAd to multiple matchmakers in different pools.

With the development of the Globus toolkit described in Section 2.1.2, the Condor project developed the Condor-G system that allows a user to treat the Grid as an

entirely local resource. It comes with a personal desktop agent that allows the user to submit jobs, query a job's status, cancel the job and be informed of job termination or problems by invoking the GRAM services underneath. Condor-G empowers end-users to improve their productivity by providing a unified view of distributed resources [42]. However, the Condor-G system does not support workflow application execution directly. We will describe a workflow management system called DAGMAN in Section 2.2.3 that builds on top of Condor. All the resources in the Condor pool are accessible directly instead of having a local resource manager.

GridBus

The GridBus project is located at the Grid Computing and Distributed Systems (GRIDS) lab in the Department of Computer Science and Software Engineering at the University of Melbourne, Australia. The project name GRIDBUS is derived from its research theme: to create next-generation GRID computing and BUSiness technologies that power the emerging eScience and eBusiness applications [83].

The Gridbus project covers wide research topics on Grid economy, workflow application scheduling, service level based resource management, Grid environment simulation and other related areas. The Gridbus project has a market-based Grid resource broker and a budget aware just-in-time workflow scheduling system. They also have proposed several plan-ahead scheduling approaches such as greedily searching for the most cost-effective resources to meet the budget constraint [123], using genetic algorithm to achieve multiple goals [124] and adjusting the critical path of a workflow application dynamically to achieve better performance [67]. Many of these proposed approaches are evaluated on their GridSim toolkit [104] that models and simulates systems-users, applications, resources, and resource brokers (schedulers) in a parallel and distributed computing environment.

The GridBus project works on many areas that overlap with my research but my research addressed the scalability problem, put more emphasis on the plan-ahead

scheduling approach and focused on batch queue managed local resources and resource reliabilities.

GridLab

The GridLab project is one of the biggest European research undertakings in the development of application tools and middleware for Grid environments [3]. GridLab provides twelve application-oriented Grid services and toolkits providing capabilities such as dynamic resource brokering, monitoring, data management, security, information, adaptive services and more.

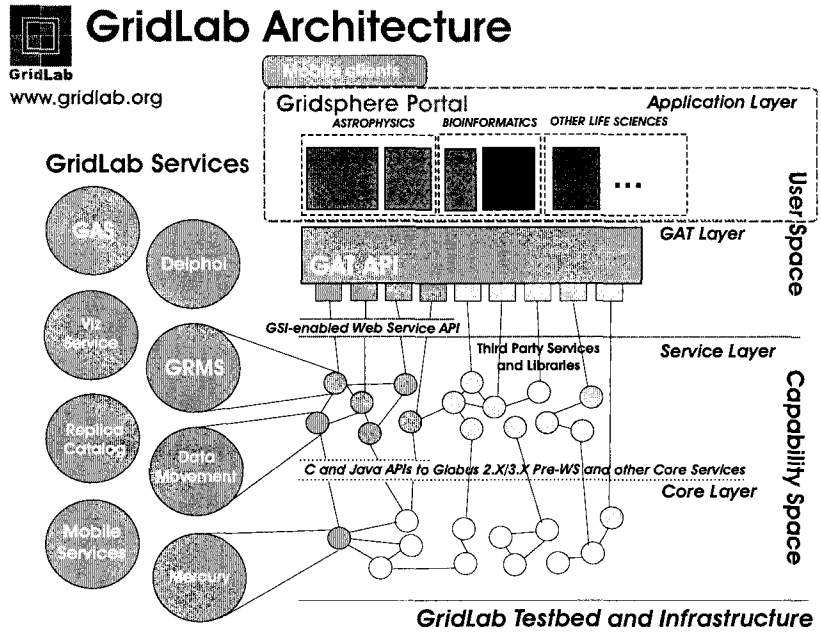


Figure 2.3 : GridLab Architecture

Figure 2.3 from Allen *et al.* [3] shows the GridLab project architecture. At the highest layer there is the application layer that is a web service application development portal called GridSphere. Below it is the Grid Application Toolkit (GAT) that is a set of coordinated, generic and flexible APIs for accessing Grid services. The

service layer covers the whole range of Grid capabilities such as Grid resource management and brokering, data access and management, Grid authorization, Grid network monitoring and performance prediction service, Grid monitoring infrastructure and Grid data and visualization services. Some of the GridLab services overlap with the Condor-G system. Gridlab uses GRAM to submit jobs to remote resources. The main object of the GridLab project is to provide end users a unified Grid platform to easily develop and test Grid-enabled application. However, it does not directly support workflow applications and does not provide resource co-allocation and application level scheduling.

2.2 Workflow Management Tools

2.2.1 Workflow application Overview and Notation

Workflow applications are widely used in scientific fields as diverse as astronomy [10], biology [60, 66] oceanography [47], and earthquake science [28]. It is the most important type of application that is suitable to run on a large scale distributed systems and especially Grids. In a workflow application, the overall application is composed of multiple (usually coarse-grain) tasks linked to each other by either data or logic dependences. This property makes workflow application an ideal form of application to run on a distributed system since the tasks in a workflow application can run on distributed resources in an asynchronized manner.

The *directed acyclic graph (DAG)* is an abstract description and is frequently used to represent a workflow application. We define an *abstract DAG* as a pair $G = (V, E)$, where V is a set of nodes, each representing an application task, and E is a set of edges, each representing a data dependence between tasks. We will later denote the source of the dependence as the *predecessor* tasks and the sink of the dependence as the *successor* tasks. Our complexity measures will often use v as the size of set V and e as the size of set E. We will later refer to an abstract DAG as the *DAG model.* We assume that an abstract DAG always has a single entry node and a unique exit node

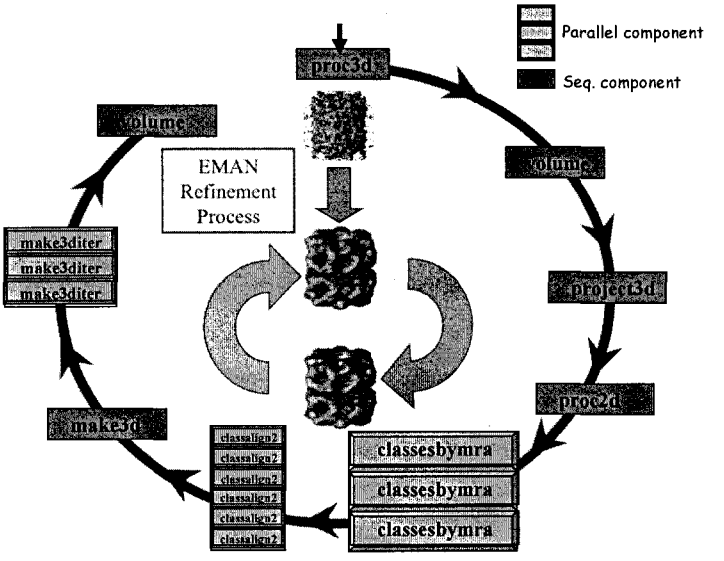


Figure 2.4 : EMAN Refinement Workflow

because we can insert dummy entry and exit tasks into the DAG that do not take any time to run and have no input or output files. We also quantify the needs of particular applications using a popular and simple metric: *Communication-Computation Ratio (CCR)*. Following Blythe *et al.* [11], we define the CCR of a DAG as

$$CCR = \frac{total\ communication\ cost}{number\ of\ tasks \times AvgCompCost}$$

2.2.2 Workflow applications

In this thesis, we use five workflow applications in various experiments to test our workflow scheduling algorithms. Here, we will describe their background and the characteristics of the DAGs that represent them.

EMAN

EMAN [Electron Micrograph Analysis] is a bio-imaging application developed at the Baylor College of Medicine [66]. It primarily deals with 3D reconstruction of single

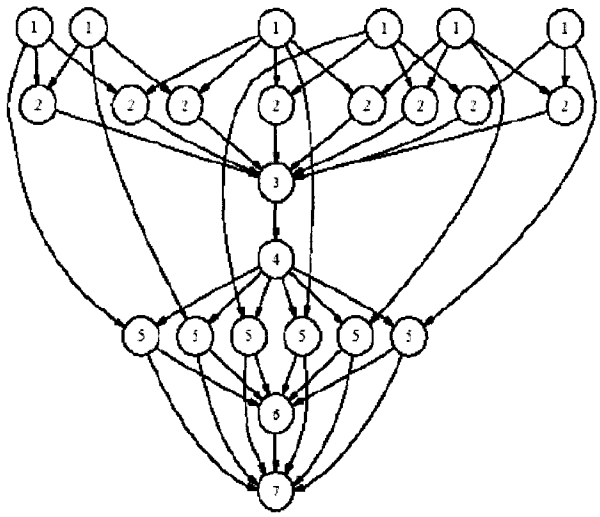


Figure 2.5 : A Small Montage Workflow

particles from electron micrographs. Human expertise is needed to construct a preliminary 3D model from the "noisy" electron micrographs. The refinement from a preliminary 3D model to the final 3D model is fully automated and is the most computationally intensive step that benefits from harnessing the power of the grid. The EMAN refinement can be represented by the workflow depicted in Figure 2.4. It is essentially a linear workflow with some sequential and parallel stages. The important and time-consuming steps are the large parameter sweep steps like "classesbymra".

Montage

Montage is a data-intensive astronomy application to create custom image mosaics of the sky on demand [10]. It consists of four steps: (i) Re-projection of input images to a common spatial scale; (ii) Modeling of background radiation in images to achieve common flux scales and background levels; (iii) Rectification of images to a common flux scale and background level; and (iv) Co-addition of re-projected, background-corrected images into a final mosaic. Figure 2.5 shows the structure of

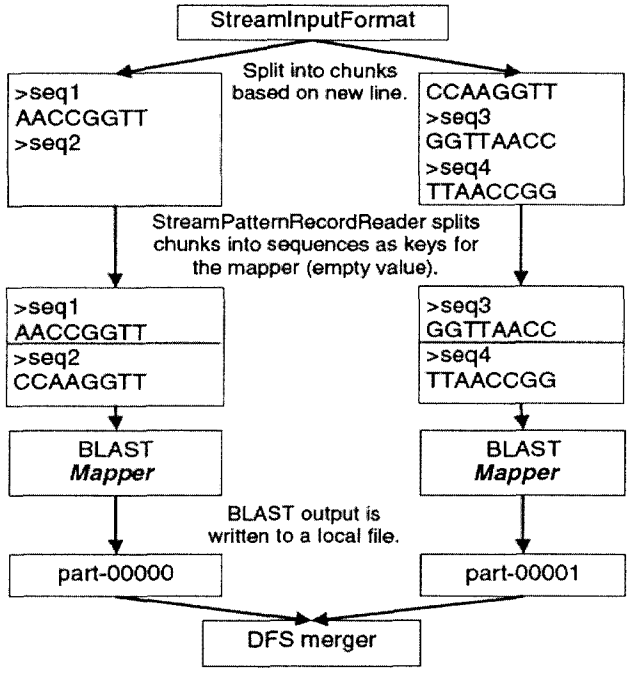


Figure 2.6 : BLAST Workflow

a small Montage workflow. The workflow consists of some highly parallel sections that can benefit from execution over multiple grid sites. Because this application is data-intensive, potentially large files are transferred on the edges of the workflow.

BLAST

Basic Local Alignment Search Tool (BLAST) [60] is a bioinformatic application that finds regions of local similarity between primary biological sequence information, such as DNA or protein sequences. Given a set of k sequences, the program compares each sequence to a database of n sequences and calculates the statistical significance of matches. The BLAST application uses a set of heuristic algorithms and is much faster than the traditional pattern matching dynamic programming. However, it just works to find the related sequences in a database search. Therefore, it cannot guar-

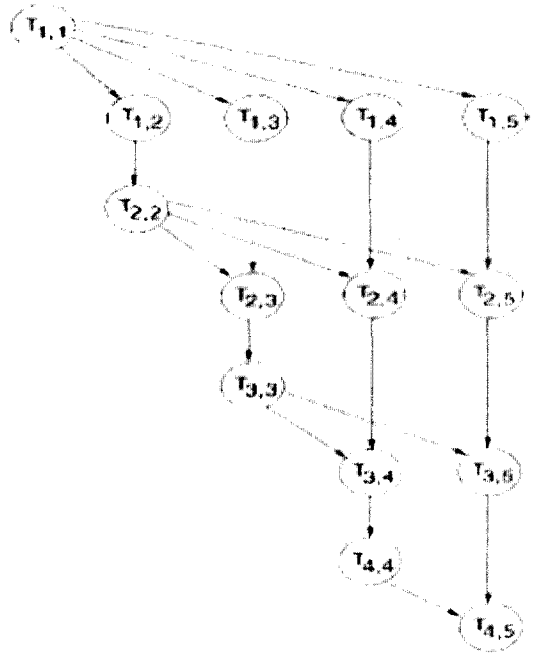


Figure 2.7 : Gaussian Elimination Workflow

antee the optimal alignments of the query and database sequences as in the dynamic programming. To further speed up the matching process, one often partitions the k input sequences and runs the BLAST matching algorithm in parallel. Figure 2.6 shows the structure of the BLAST application workflow that the pattern matching process for the k input sequences can be potentially run in parallel since there is no data race between them.

Gaussian Elimination

The Gaussian Elimination algorithm is widely used in computational science for the solution of a system of linear equations [?]. It systematically applies elementary row operations to a system of linear equations until it converts the system to upper triangular form. Once the coefficient matrix is in upper triangular form, one can use back substitution to find a solution. Figure 2.7 shows the structure of the Gaussian Elim-

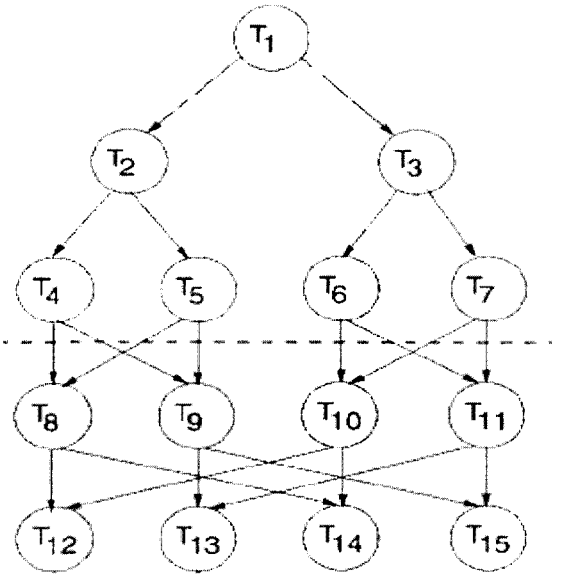


Figure 2.8 : Fast Fourier Transform Workflow

ination application workflow. We can see that it has a long critical path and there is less and less parallelism as the application executes. These two particular characteristics can affect the effectiveness of scheduling methods in a workflow execution system as we will see in later chapters.

Fast Fourier Transform

The fast Fourier transform (FFT) is a set of efficient algorithms to compute the discrete Fourier transform (DFT) and its inverse. A discrete Fourier transform (DFT) transforms one function, which is often a function in the time domain, into its frequency domain representation. By far the most common FFT algorithm is a divide and conquer algorithm that recursively breaks down a DFT of any composite size N = N1×N2 into many smaller DFTs of sizes N1 and N2, along with O(N) multiplications by complex roots of unity traditionally called twiddle factors. The most common use of this FFT algorithm is to divide the transform into two pieces of size N / 2 at

each step. Figure 2.8 shows the DAG structure of this commonly used fast Fourier transform algorithm.

2.2.3 Related Workflow Management Projects

Since it is an important type of application there are many research projects on workflow application management systems and many of them work closely with one or several Grid projects we described in section 2.1.2. Here we present several influential projects.

Dagman

DAGMan (Directed Acyclic Graph Manager) is a workflow meta-scheduler for Condor. It manages dependencies between jobs at a higher level than the Condor Scheduler which is solely match based [72].

Condor finds matching resources for the DAG tasks, but it does not schedule jobs based on dependencies. It is DAGMan's job to make sure those dependencies are honored. DAGMan reads a specific input file format that includes a list of the programs and the dependencies in the DAG and a Condor submit description file for each program in the DAG. It then submits jobs to Condor in an order that satisfies all the dependencies. DAGMan is also responsible for monitoring, recovery and reporting for the set of programs submitted to Condor. DagMan is one of the most widely used workflow execution system although it lacks sophisticated scheduling mechanisms.

Pegasus

The Pegasus project explores issues related to scientific workflow management. It works with domain scientists to support their distributed computations in a scalable and reliable way [46].

The Pegasus project has four sub-projects: the Pegasus Mapper, Pegasus-WMS, Ensemble Manager and MCS. The Pegasus Mapper first reads in an XML formated

input abstract workflow (DAX) and all the associated catalogs that help the planning. Then the mapper tries to reduce the workflow DAG mainly based on the data size and dependancies. Finally, the mapper schedules different parts of the application onto different distributed resources based on a relatively simple list scheduling algorithm or genetic search based algorithms [11]. The Pegasus WMS is an end-to-end workflow management system that builds on top of the Pegasus Mapper, Condor, and DAGMan. Specifically, Pegasus WMS uses DAGMan (and through it, Condor) to execute the workflow application. Ensemble Manager manages the mapping and executions multiple workflows. It is build on top of Pegasus WMS. MSC is a metadata catalog service for the Grid [89].

The Pegasus project addresses the problem of automating workflow application execution but it does not emphasize on the performance. It also assumes that the underlying resources are dedicated (i.e. not batch queue controlled) which is more restrictive than the widely available batch queue system.

Triana

Triana is a workflow-based problem solving environment [106]. It is designed as a series of pluggable execution components. Triana has a workflow management GUI where users can drag and drop different components and connect them to create a workflow, which can easily be integrated with other systems. For example, Triana works with Pegasus to generate the DagMan input files for the GriPhyN project. Triana is also part of the GridOneD [88] project for creating Java middleware for grid applications. Triana is ready to use GridLab's GAT too. Kepler [86] and Taverna [87] are two other similar workflow-based problem solving environments that provide a GUI and execution methods. However, none of them have a sophisticated scheduling scheme and do not deal with the heterogeneous, dynamic and distributed resources directly.

2.3 Scheduling Algorithms

Scheduling is one of the more important research topics in high performance computing. In this section, I will first introduce the problem definition and common notation. Then, I will summarize related work on workflow application scheduling on homogeneous, heterogeneous and Grid platforms respectively.

2.3.1 Problem Definition and Notation

The inputs to a scheduling algorithm are an abstract DAG, a set of resources P and two performance prediction matrices $M_p = V \times P$ and $M_n = P \times P$. Here, $M_p[i][j]$ represents the estimated computation cost of node n_i on processor p_j measured in seconds. $M_n[i][j]$ represents the estimated communication cost of transferring data from processor p_i to processor p_j measured in MB/s. The cost of an edge (i,j) will depend not only on the mapping of its endpoints, but also on the amount of data transferred. Our complexity measures will often use the term p for the size of P. We will later refer to P as the *resource model*, M_p as the *cost model* and M_n as the *network model.*

The output of a scheduling algorithm is a *concrete DAG* $G = (V, E, M)$, where V and E are the same as in an abstract DAG and M is a map from V to P such that $M[v_i]$ is a pair (r_i, t_i), where r_i is the resource on which the node will be executed and t_i the time it will start. In this thesis, the objective of the scheduling algorithms is to output a concrete DAG corresponding to an abstract DAG such that certain metrics, such as makespan, cost or success rate, are optimized.

The process of scheduling a parallel application on a distributed platform can be described as follows. Given an application that consists of m "tasks" (e.g., computations, I/O operations), and a platform that consists of n "resources" (e.g., CPUs, disks, networks), compute a mapping of tasks to time and to resources (i.e., task i starts executing at time t on resource j). We will later use schedule to denote this process throughout the thesis. Occasionally, we will use "mapping" to denote the

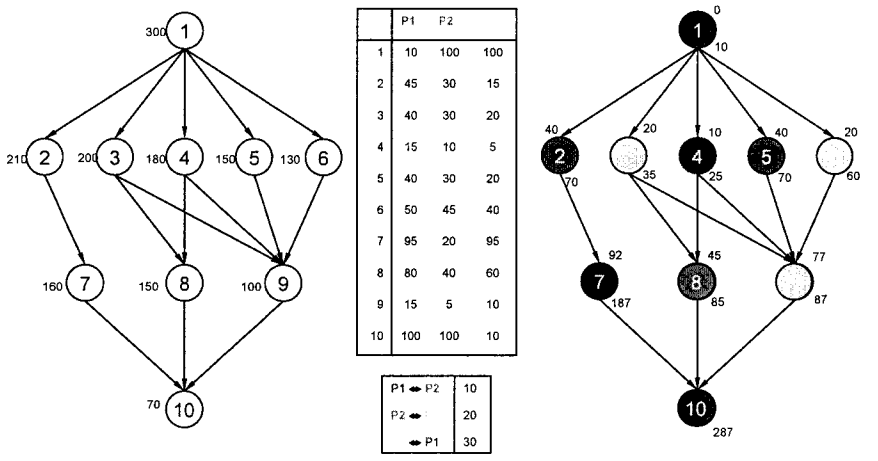


Figure 2.9 : A DAG schedule example

part of the schedule (or scheduling process) that determines only the task to resource mapping. That is, mapping is scheduling without the timing information, which is constructed by other means (e.g. by the dynamic scheduling mechanism). The scheduler usually queries the cost model to determine the running time of a particular task on a resource and checks with the network model to get the file transfer time in order for all the input files to be staged onto that resource. Figure 2.9 shows an example of a round-robin schedule applied on an abstract DAG (left) that produces a concrete DAG (right). The upper table in the middle of the Figure 2.9 shows the cost model and the lower table shows the network model. The number on the left side of each task in the abstract DAG denotes the size of the output from this task. After applying the round-robin schedule, we get a concrete DAG with a mapping (denoted by its color corresponds to the resource), a start time (upper right) and an end time(lower right) for each task. For each task, the start time is calculated by the finding the earliest finish time of each predecessor including the file transfer time. For example, the earliest finish time for task 3 is 35 sec to finish computation plus 10 sec to transfer 200MB output file of task 3 from $P3$ to $P2$ which is 45. Similarly, the earliest

finish time for task 4 is also 45, thus we get the start time of task 8 as 45. Here, we use round-robin just as an example to illustrate the basic concept of scheduling, we will describe in the following sections more sophisticated scheduling algorithms that provide better outcome.

Makespan and turn-around-time are two widely used metrics that measure the quality of the scheduling output. In this thesis, we will use makespan to denote the estimated running time or the "scheduled length" of a workflow application. For example, the makespan of the DAG in Figure 2.9 is 287. Turn-around-time is used to denote the actual measured time difference between the time an application is launched and is finished. It is usually different from makespan because it takes into account the external overheads (such as scheduling time) and because the performance models are usually not precise. The makespans may vary widely among DAGs, making it difficult to take meaningful averages or make cross-DAG comparisons. Following the methodology of other scheduling work [62, 111, 7], we use *Schedule Length Ratio* (SLR) as the main metric for the comparisons so that the results will not be sensitive to the size of the DAG. Conceptually, the SLR is a normalization of the makespan to an estimate of the best possible schedule length of a given DAG in a given environment. In a perfect world, we would use an optimal schedule for this estimate; however, since finding the optimal makespan is NP-complete, we instead use the estimated *critical path length.* The *critical path* through a DAG is the most costly path from the entry task to the exit task, while the *critical path length* is the total cost of tasks and edges along this path. Because the costs of tasks depend on where they are mapped, in this calculation we approximate the computation cost of a DAG task by its average cost over all possible processors. Similarly, we approximate the communication cost of a DAG edge by its average over all possible processor pairs. We compute the *Critical Path Including Communication (CPIC)* as the cost of the critical path using these estimates, and define

$$SLR = makespan/CPIC$$

Intuitively, a small SLR indicates a better schedule than a large SLR. An SLR of 1 occurs when all tasks and edges are mapped to average processors and network links, and no bottlenecks occur due to lack of resources. An SLR can be below 1 when some tasks are mapped to faster-than-average resources or when a schedule avoids much cross-processor communication, and above 1 when resources are limited or when the schedule uses slower-than-average resources. Our definition of SLR differs slightly from the usual definition of SLR in Kwok and Ahmad [62] that uses CPES (critical path excluding communication).We prefer our definition because it includes an approximation of communication cost, thus providing a more realistic standard of comparison.

2.3.2 Homogenous and Heterogenous DAG Schedulers

Many polynomial-time scheduling heuristics have been proposed although the scheduling problem in general is a NP-complete problem [43]. Scheduling happens at all levels of high performance computing ranging from machine instruction scheduling to workflow application scheduling. In the next section, we will present two categories of scheduling algorithms based on the resources it is used. Homogenous and heterogenous scheduling algorithms are mostly used in system level computing (i.e. instruction, thread, process scheduling) while Grid workflow application schedulers work at the application level. The difference between homogenous and heterogenous DAG scheduling algorithms is that the former assumes the underlying resources are identical while later assumes the resources have different capabilities.

McCreary et al. [70] compared five different heuristics for scheduling DAGs on multiprocessors and Kwok et al. [62] did an excellent survey on the large body of literature on scheduling a DAG onto a set of homogeneous processors. Despite the different assumptions of the underlying resources, we can grossly classify various scheduling algorithms into three categories: list scheduling, clustering scheduling, task duplication scheduling and level based scheduling. We describe them in the

following sections.

List scheduling heuristics

One of the first areas DAG scheduling comes into play is in compiler technology. Instruction scheduling is a critical component in the back end of every compiler. The most commonly used version of instruction scheduling moves instructions within a basic block. It first determines the dependencies between instructions and creates a DAG to represent the block. Each instruction is a node in the DAG and the scheduler gives higher priority to instructions on the critical path. It then schedules the tasks in descending order of their priorities. This type of DAG scheduling is simple to implement and can achieve near optimal performance in most cases. Therefore, a family of similar scheduling algorithms have been developed. We usually refer to those scheduler algorithms as list scheduling.

Numerous heuristics have been proposed for assigning the priorities and most of the time, the priority of a task is either a function of top-level (t-level), which is the length of the longest path from an entry node to the task itself; or bottom-level (b-level), which is the length of the longest path from the task itself to any exit node; or length of the critical path (CP) or some combination of these. This type of approach works well in a homogenous environment where all the resources are the same. However, it faces a chicken-and-egg dilemma in a heterogenous environment because all these attributes can not be calculated until the schedule is computed while computing the schedule relies on them. For example, a task's running time is unknown until a mapping has been made since the task takes different amount time to finish on each resource. In order to avoid this problem, approximations of computation cost and communication cost are used. The most commonly used are average values such as the average computing time on all the resources and average network bandwidth. The best, worst and median value have all been proposed but there is no decisive conclusion as to which one should be used [94].

Here are a few of the more influential list-scheduling heuristics. The Modified Critical Path (MCP) [118] heuristic uses the As Late As Possible (ALAP) time, which is defined as the length of critical-path less the b-level as the priority. If two tasks have the same priority, the tiebreaker is the maximum ALAP time of the their successors. MCP also tries to insert a task to a processor that has previously been mapped to allow the earliest start time. The Earliest Time First (ETF) [48] heuristic assigns priority to tasks with a higher static level. Static level is defined as the maximum sum of computation costs along a path from the task to an exit task It is equivalent to the b-level without communication cost. EFT then computes the earliest start times for all the ready tasks on all the processors and assign them to the processor with the earliest start time. The Dynamic Level Scheduling (DLS) [100] heuristic was among the first list scheduling algorithms applied on a heterogenous enviroment. It uses the Dynamic Level (DL) attribute, which is the difference between the static level of a task and its earliest start time on a processor. For all the ready tasks the value of DL is calculated for all the processors. The task-processor pair giving the highest value of DL is scheduled next.

Heterogeneous Earliest Finish Time(HEFT) [111] is a well-established list-based algorithm known to perform well on heterogeneous platforms [7, 111]. In the node prioritizing phase, HEFT uses an *upward rank* which is defined as

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} (\overline{c_{i,j}} + rank_u(n_j)),$$

where $succ(n_i)$ is the set of immediate successors of node n_j, $\overline{w_i}$ is the average (estimated) computation cost of node n_i and $\overline{c_{i,j}}$ is the average (estimated) communication cost of edge $E(i,j)$. Averages are computed over the set of all resources in M_p or the network model M_n, respectively. We assign $rank_u(n_{exit}) = 0$ and traverse the DAG edges backwards to compute the other upward ranks. In the processor selection phase, HEFT assigns each node, in order, to the processor that gives the earliest finish time, i.e. the minimal EFT. HEFT uses an insertion-based policy to find the earliest available time of a processor p_j. Instead of using the time p_j finishes its last assigned

node, HEFT tries to find an idle slot of p_j that is later than the available time of the node n_i (the earliest time that all n_i's predecessors finish) and long enough to finish the execution of n_i. Upon examination, we discovered that the upward rank used in HEFT is a heterogeneous adaptation of the definition of *b-level* commonly used in list-based scheduling algorithms. Thus, it can be considered as a heterogeneous version of MCP(Modified Critical Path) [118] algorithm. The computation complexity of this version of HEFT is $O(v^2 + vp)$. The HEFT algorithm is considered one of the best algorithms for scheduling tasks onto heterogeneous processors [7]. We also compare this algorithm with our approaches in the rest of this thesis.

Clustering scheduling heuristics

Sarkar [97] proposed a two-step method for instruction scheduling on multiprocessors with communication. The main goal is to reduce unnecessary communication costs among tasks.

- Aggregate tasks in the DAG together into clusters of tasks, with the intent that all tasks in a cluster to execute on the same processor.
- If the number of clusters is larger than the number of processors, then merge the clusters further to the number of physical processors, and also incorporate the network topology in the merging step.

The aggregation part depends on criteria that varies from one heuristic to another. One such clustering scheduling heuristic is called Edge Zeroing (EZ) proposed by Sarkar in [97]. It first sorts all the edges by weight and selects tasks on the highest weighted edges for aggregation. The aggregation process maps two tasks to a cluster and eliminates the communication cost if the merging does not increase the current parallel completion time. The parallel completion time is approximated by the maximum b-level over all the tasks. Another heuristic is called Linear Clustering (LC) [59]. It first zeros all the edges on the current critical-path. Then it removes the nodes

and edges on them from the graph and reduces the entire path to one node. This process is repeated for the unexamined portion of the graph until all paths have been examined. If there are still more clusters than the number of processors, it tries to merge all cluster pairs that are not overlapping and merge clusters that can shorten the overall execution time. The Dominant Sequence Clustering (DSC) [120] and its heterogenous version [20] considers the dominant sequence (DS) of the graph and is reported to have a good performance with low complexity of $O((v+e)logv)$ [70]. The dominant sequence is the length of the critical path in a partially-scheduled DAG. DSC assigns the priority of a free task as the sum of the t-level and b-level. The priorities of other tasks are just their b-levels. It picks the free or partially free task with the highest priority. This task is merged into the cluster of one of its predecessors if that reduces its t-level, otherwise, it starts a new cluster. The t-level of the successor tasks are updated and the algorithm iterates until all the tasks are examined.

We will propose a clustering scheduling heuristics in Chapter 7 to reduce the total batch queue wait time for a workflow application.

Task duplication scheduling heuristics

The basic idea behind task duplication based (TDB) scheduling algorithms is to minimize inter-processor communication delay or network overhead by executing copies of tasks on multiple resources. In this way, some tasks can start earlier because copies of their predecessors are running on the same resource and this eventually leads to earlier overall completion time of the entire program. Task duplication based scheduling algorithms are particularly useful for systems with high communication cost and data centric applications. Most task duplication based algorithms pay the most attention to the tasks on the critical path and join or fork tasks.

The first duplication based scheduler was proposed by Kruatrachue [61]. It combines some ideas used in list scheduling with duplication to reduce the makespan. The algorithm considers each task in descending order of priority which is the b-level

excluding the communication cost. Then it tries to pick a processor for each task. The algorithm first calculates the start time of the task on a processor without duplication of any predecessor. Then the algorithm attempts to duplicate the predecessors of the task into the same processor until either the processor is used up or the start time of the task does not improve further. This process is repeated for other processors and the task and its duplicated predecessor tasks are scheduled to the processor that gives the earliest start time.

Ahamd et al. [2] proposed a duplication based algorithm called Critical Path Fast Duplication (CPFD). The intuition behind it is to select the important tasks for duplication. They classified the task in a DAG into 3 categories in order of decreasing importance: Critical Path Nodes (CPN), In-Branch Nodes (IBN) and Out-Branch Nodes (OBN). CPNs are on a critical path. An IBN node is a task that is not a CPN and from which there is a branch reaching a CPN. An OBN is a node that is neither a CPN nor an IBN. The CPFD algorithm works like this. It first determines the critical path and creates the CPN-Dominant sequence which contains the topological order of all the CPNs and IBNs. Then for each task in the CPN-Dominant sequence, CPFD schedules it to the processor that gives the smallest value of earliest start time (EST) by recursively duplicating its important predecessors. The time complexity of CPFD is $O(ev^2)$ and

Task duplication scheduling not only can increase the performance but also provide fault tolerance since some tasks may still finish despite of some resource failures. We will propose a task duplication scheduling heuristic that focuses on providing the right amount of fault tolerance for a workflow application in Chapter 6.

Level based scheduling heuristics

Some argue that the main reason for heterogeneous algorithms to fail to provide good outcome is it becomes so difficult to estimate the b-level, t-level and the critical path without knowing the actual schedule. Iverson et al. [68] proposed a level based heuris-

tic scheduling (LHBS) that does not rely on those estimations. The characteristic of a level-based scheduling algorithm is that it proceeds by partitioning the DAG into levels of independent nodes. Within each level, a LHBS can apply various heuristics [13] to map the independent nodes to the processors. The simplest approach is to use a Greedy algorithm that maps the nodes to the fastest processors which has a computational complexity of $O(vp)$. Three heuristics, min-min, max-min and sufferage, are also widely used to compute the mappings for the independent tasks in a level. Details of these heuristics are presented in Braun *et al.* [13]. The computational complexity of the heuristic scheduling scheme is $O(v^2p)$, which is more expensive than the greedy heuristic.

Both the GrADS [11] and Pegasus [89] schedulers use a version of LHBS. Mandal et al. [69] and Sakellariou [94] also proposed similar level based heuristics that schedule independent sub-tasks in a workflow application level by level. This type of approach avoids the chicken-and-egg dilemma for those critical path based algorithms but this approach has a tendency to over parallelize one level which may leads to communication overhead [127].

2.3.3 Grid Schedulers

Dong et al. [31] and Yu et al. [122] both did an excellent summarization on the state-of-the-art scheduling technologies in a Grid environment. There are two major types of scheduling in a Grid environment namely resource scheduling and application scheduling. A Grid resource scheduler is used to manage distributed resources in a Grid. Its common goal is to increase the utilization or balance of the resources. Meanwhile a Grid application scheduler is usually used to increases the performance, reliability and cost effectiveness of Grid application. Here we will present both the resource and application scheduling techniques but we focus on application scheduler techniques in this thesis.

Resource Schedulers

Most Grids today consist of batch queue controlled clusters. In these Grid environments, the individual resources (clusters, computing farms, servers, supercomputers) are managed by their local resource management (LRM) systems , such as PBS [79], SGE [73], LSF [65] and Condor [82]. Many LRMs are already mature commercial products. There are also numerous researches on how to achieve good schedule for a set of independent jobs on a local resource [13].

A LRM only controls a single resource while a typical Grid environment consists of several distributed resources. In order for the user to use all the shared Grid resources, most Grid environments have a portal from which the users can submit jobs. Users can use a meta-scheduler that contacts all the distributed resources to secure the resources for the jobs and then schedule and launch them onto those resources. Many meta-schedulers have been developed such as GRMS [3], HPC Synergy [112], Moab [25], GridWay [84] and SPRUCE [81]. Among them, HPC Synergy and Moab are commercial products that target mostly on enterprise clusters. GRMS is part of the GridLab project [3]. GridWay works closely with Globus Project [4] and SPRUCE specializes in providing urgent advanced reservation on TeraGrid [107]. The services those meta schedulers provide include automatic resource selection [51], advance resource reservation [102], co-scheduling, on-demand resources [81], support for workflow applications and fault tolerance which are critical steps towards the automated workflow application execution vision we have. However, none of them provide all of these services and few provide good support for automatic workflow application execution.

Application Scheduler

There are several flavors of application schedulers. For example, a static scheduler determines the schedule before the application starts to run while a dynamic scheduler postpone the schedule decision until the application runtime. An independent

job scheduler schedules several independent applications at the same time while a workflow job scheduler schedules one application with internal dependences between tasks. Here we will mainly describe the workflow application scheduling techniques that are related to our thesis.

Current Grid workflow management systems use simple approaches such as first-come-first-served with matchmaking as in Condor DAGMan [82], the Data Grid resource broker [132] and the GridLab resource broker [3], or random allocations or round robin as in Pegasus [89]. Since a Grid environment is a special heterogeneous platform, most of the DAG scheduling algorithms that work in a heterogeneous environment can be applied to workflow DAGs executed on the Grid. However, since the Grid environment is dynamic and even non-deterministic, it is usually hard for an unmodified heterogenous scheduling algorithm to achieve good performance on it. Several approaches have been proposed to improve the schedules in a Grid environment. The most commonly-used approach is dynamic scheduling which does not make the schedule decision until a job is available to run. Several workflow management systems [82, 89] adopt this approach in that they use a match maker to dispatch the job to the resource that best matches the job's requirement when the job is ready to run. Dynamic scheduling has two major advantages

- The algorithm usually has low computing complexity and is simple to implement, thus a good fit for a runtime system.
- The algorithm can make decisions based on the current system configuration instead of relying on prediction or estimation

However, since dynamic scheduling does not have a plan phase, it assumes that by shortening the current job's execution time, it will help the workflow's performance which is not always the case.

Since the scheduling problem in general is a NP-complete problem and a Grid environment is particularly heterogenous, some has proposed search based heuristics

for the Grid environment. Blythe et al. [11] introduced a fuzzy attribute α during the level by level scheduling. For each level, the algorithm randomly chooses a schedule that produces the makespan between L_{min} and $L_{min} + \alpha * (L_{max} - L_{min})$ where L_{min} and L_{max} are the shortest and longest makespan of this level. The overall algorithm runs many times and records the best schedule. This approach can find a better schedule but takes longer to schedule. Another popular class of search based heuristics is the genetic algorithm. In a genetic algorithm, both the resource mapping and task execution order are usually represented as strings. After the initial population is generated, different algorithms apply different across and mutation rules on the population, represented by strings, in hope for breeding a better generation. Yu et al. [124] compared several search based algorithms in multi-objective workflow scheduling and showed that they are good candidates for optimizing multiple objectives such as the performance and budget.

Although search based algorithms have better theoretical performances, they generally take long time to run. Since a Grid environment is dynamic and has a large number of resources, the advance in the schedule quality of a search based algorithm may not be enough to overcome the schedule time it takes. Therefore, we will focus on more light weight scheduling algorithms such as the level based heuristic and other traditional heterogenous algorithms. In this thesis, we will present several new techniques that can improve the performances of the existing scheduling algorithms in a dynamic multi-cluster Grid.

Chapter 3

Performance of Scheduling Algorithms in a Multi-cluster Grid Environment

This chapter presents a comparison of the performance of scheduling algorithms in a multi-cluster Grid environment. The multi-cluster Grid environment is different from traditional heterogeneous environments because of the drastic cost differences between the inter-cluster and the intra-cluster data transfers. In this chapter, we analyze the performance of several scheduling algorithms that represent two classes of widely used scheduling algorithms for Grid computing. Based on our experiments, we introduce a new measurement called effective aggregated computing power (EACP) that dramatically improves the performance of some schedulers.

3.1 Introduction

Although Grid technologies enable the sharing and utilization of widespread resources, the performance of parallel applications on the Grid is sensitive to the effectiveness of the scheduling algorithms used. In this chapter we are going to study the performance of several traditional static scheduling algorithms in a simulated multi-cluster environment and we focus on scheduling the important class of *workflow applications*. As described in Section 2.2.1, a workflow application consists of multiple (usually coarse-grain) tasks linked to each other by data dependences, typically requiring file transfers.

Scheduling parallel and distributed applications is known to be NP-complete in general [43]. Numerous heuristics have been proposed for scheduling DAGs onto a heterogeneous or homogenous computing environment [6, 111, 92, 48]. Section 2.3.2

gives a survey of existing scheduling algorithms and showes that list-based scheduling heuristics are generally accepted as the best overall approach, exhibiting both low complexity and good results [62]. However, Iverson [68], Illvarasan [35] and Atakan [30] argue that the pre-computed order for list-based strategy cannot be used in heterogeneous environments and propose a new heuristic class that we call the *level-based strategy.*

A Grid environment usually consists of many clusters with special properties that poses even more challenges for scheduling applications because not only are the processors heterogeneous but also the inter-processor communication variance is larger. Looking over surveys of state-of-the-art Grid scheduling algorithms [122, 31], we can see that many Grid projects simply use dynamic dispatching mechanisms similar to Condor [82]. Besides that, to the best of our knowledge, the list-based and the level-based algorithms are the only two scheduling heuristics implemented by a Grid project. Blythe *et. al.* [11] reported that the level based strategy outperformed the random matching strategy by more than 50%. However, to the best of our knowledge, there has been no published research that directly compares the performance of list-based and level-based algorithms in a Grid environment.

In this chapter, we evaluate the schedules produced by several well-known list-based and level-based scheduling algorithms. Relying on tens of thousands of experimental runs, we show how the performance of these algorithms varies with differences in resource environments and application DAGs. We analyze these results to explain why some scheduling algorithms perform better in certain settings and less well in others. Based on these observations, we introduce a promising new scheduling concept, called *effective aggregated computing power (EACP)* and demonstrate how it can be used in scheduling algorithms.

The rest of the chapter is organized as follows. Section 3.2 briefly covers the basic characteristics of a Grid environment and introduces all the scheduling algorithms that we will evaluate in this chapter. Section 3.3 presents our applications, the

experimental environments we are using, and the Grid parameters we vary in the experiments. Section 3.4 presents our results; it also defines effective ACP and shows how it works in a scheduling algorithm. Section 3.5 concludes the chapter with a summary of contributions.

3.2 Background and Related Work

A typical Grid environment consists of many clusters, where the intra-cluster communication is fast (often as fast as 10 Gigabit/sec) but the inter-cluster communication can be 10 to 1000 times slower. Thus, the Grid is not just a heterogeneous resource pool, but also an unevenly distributed (but hierarchical) interconnection network. Furthermore, while many homogeneous processors reside in any one cluster, the processors in different clusters are often significantly different. As Section 3.4 shows, these features have a big impact on how scheduling algorithms originally designed for homogeneous or heterogeneous platforms perform in Grid environments.

As we mentioned in Section 3.1, the level-based and list-based algorithms are the most used ones in Grid environments and we want to compare their performance. For our experiments, we have chosen some representative and effective algorithms in both categories. This section gives a brief overview of each of those algorithms.

3.2.1 Static Scheduling Algorithms

Heterogeneous Earliest Finish Time(HEFT) [111] is a well-established list-based algorithm known to perform well on heterogeneous platforms [7, 111]. For more detail, refer to the Section 2.3.2 and Topcuoglu *et al.* [111]. Both Ma *et al.* [83] and Cao *et. al.* [16] use HEFT to help schedule application DAGs onto Grid resources. The computational complexity of this version of HEFT is $O(v^2 + vp)$. *Levelized Heuristic Based Scheduling(LHBS)* [69] is a level-based algorithm for Grid scheduling we described in Section 2.3.3. The complexity of the LHBS using only the greedy heuristic is $O(vp)$; we will refer to this as *Greedy LHBS*. The complexity of the LHBS using

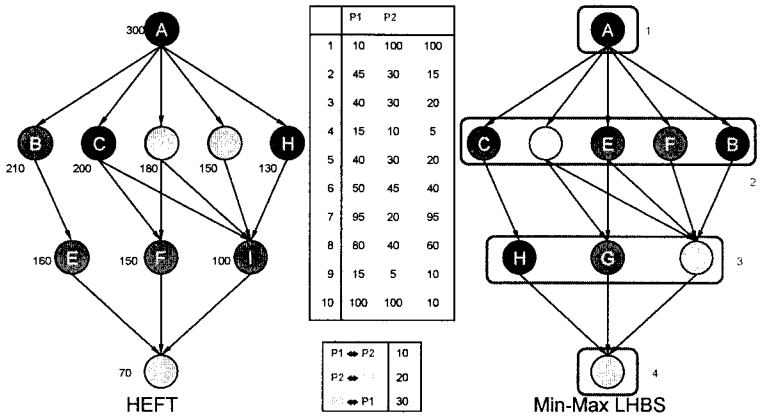


Figure 3.1 : HEFT and LHBS scheduling algorithms

the any combinations of the three heuristics is $O(v^2p)$; we will refer to this variant as *Heuristic LHBS.*

Figure 3.1 depicts an example that illustrates the difference between HEFT and LHBS. The color of the tasks in a DAG denotes the resources it is mapped to and the table in the middle shows the time it takes for each task to run on each cluster (the performance model) and the time it takes to transfer files between clusters (the network model). The letter on each node denotes when the scheduler computes the map for each node. We can see that LHBS always schedule all nodes one level before moving to the next level. In contrast, HEFT can schedule a node (i.e node E) before all the nodes (i.e. G,H) in the parent level get a mapping. This is the major difference between these two types of schedulers and they both have advantages and drawbacks. The major argument against HEFT is that the order of which the scheduler computes the mapping is not accurate in a heterogeneous environment since the critical path can not be determined before a schedule is done. On the other hand, although the order LHBS uses is not affected by the resources characteristics, it only tries to optimize the makespan for a single level thus may leads overall less performance.

Hybrid Heuristic Scheduling (HHS) [94] is a class of algorithms that use hybrid

versions of the list-based and level-based strategy.The version we study in this chapter first computes levels as in LHBS, then processes tasks in each level following the prioritized order used by HEFT. This version has the same complexity as HEFT: $O(v^2 + vp)$. Sakellariou [94] reports that it can achieve better performance than HEFT.

3.3 Experimental Methodology

In order to study how well these scheduling strategies perform in the Grid environment, we implemented the algorithms described in Section 3.2 and compared the schedules produced on a variety of DAGs and grids. To achieve a thorough comparison, we developed a simulation platform to create test cases. The platform consists of three key components: the DAG generator described in Subsection 3.3.1, the cost generator described in Subsection 3.3.2, and a Grid generator described in Subsection 3.3.3. As Subsection 3.3.4 discusses, our experiments combined these to schedule and evaluate over 10,000 combinations of DAGs and grids.

3.3.1 DAG Generator

We use DAGs from actual runs of the EMAN and Montage applications described in Section 2.2.2, with the total number of tasks, the communication patterns and output file sizes taken from those cases. Besides the DAGs from real applications, we also implemented a DAG generator that can generate various formats of weighted pseudo-application DAGs. The following input parameters were used to create a DAG.

- Type of DAG: Unlike other DAG generators [7, 111], our DAG generator can generate different formats of DAGs. Currently, we support *fully random, level,* and *choke* formats. In a random DAG, each task can be connected to any task on a higher level (to ensure that the graph is acyclic). In a level DAG, a task can only connect to tasks on the level immediately above. In a choke DAG,

there is one level (the choke point) that has only one task; it connects to all the tasks on the levels above and below it. Tasks in other levels are connected randomly and uniformly distributed as in the random graph.

- Total number of tasks in the DAG, λ.
- Shape parameter, α: α represents the ratio of the DAG height (i.e. number of levels) to the width (i.e. maximum number of tasks in a level). The height and the width of the DAG are generated using the method described by Topcuoglu, Hariri, and Wu [111], which takes α and λ as parameters.
- Out degree of a task, η : Each task's out degree is randomly generated from a uniform distribution with mean value η.

3.3.2 Cost Model

Given a DAG, whether from a real application or automatically generated, we generate base costs for the tasks and edges using three parameters.

- The lower and upper bound of the *data size*, ϵ, ϕ: The data size attached to each edge in a generated DAG is randomly generated from a uniform distribution between the lower and upper bound. In level graphs, all edges between two adjacent levels have identical data size; in random and choke graphs, we generate costs for every edge independently.
- *Communication-Computation Ratio (CCR).* We set this ratio defined in Section 2.2.1 as a parameter and combine it with the total data size and average bandwidth in the resource pool to compute the average computation cost for a task:

$$AvgCompCost = \frac{total\ file\ size/avg\ bandwidth}{number\ of\ tasks \times CCR}$$

- *Range*: The task computation costs for generated DAGs are independently randomly generated from a uniform distribution from $AvgCompCost \times (1 -$

range) to $AvgCompCost \times (1+range)$. For EMAN and Montage DAGs, we use uniform costs for each level, reflecting the behavior of the actual applications.

This gives us a base cost for every task, which will be modified by the Grid model.

3.3.3 Grid Model

Our resource model is based on a tool that generates populations of representative compute clusters, as described by Kee, Casanova, and Chien [54]. This tool uses empirical statistical models of cluster characteristics (e.g., number of processors, processor clock rate) obtained from a survey of 114 real-world clusters. Using this tool we generated a resource pool that contains over 18,000 processors grouped in 500 clusters, which we refer to as *the universal environment.* We also semi-manually generated two smaller resource sub-pools. They both have roughly 300 processors, but one groups them into 20 clusters while the other has only 4 clusters. We will later refer the resource pool with 20 clusters as *the many-cluster environment* and the other as *the big-cluster environment.* Given the resource model, we computed the computational cost matrix $M_p[i][j]$ by scaling the base cost for DAG task i by the clock rate of processor j.

Our network model is based on a tool that generates end-to-end latency matrices according to the actual latency data collected over the Internet [126]. Following the experimental results of Yang *et al.* [119] and Denis *et al.* [29] we assigned the bandwidth based on the latency. Low-latency links had high bandwidth, consistent with the data in Bo *et al.* [126]. Given the latency and bandwidth of each network link, it was a simple matter to compute the communication cost matrix M_n.

The costs we generated are static, although actual Grids can have dynamic costs due to variances in load. However, we claim that the static data helps us focus on performance of the algorithms and factors out the uncertainties of resource and network behavior. We will explore the effects of dynamic costs on the algorithms in Chapter 5.

3.3.4 Experimental Setup

We used our DAG generator to produce DAGs with the following parameters:

- Type = {random, level, choke}
- λ = {300, 1000, 3000}
- α = {0.5, 1.0, 5.0}
- η ={1.0, 2.0, 5.0}

We generated 5 random DAGs for each possible parameter combination. In addition, we used 30 EMAN DAGs and 30 Montage DAGs. For all of these DAGs, we applied our cost model with the following parameters:

- (ϵ, ϕ) = { (20,1000),(100,1000),(500,1000)}
- CCR = {0.1, 1.0, 10}
- Range ={0.15, 0.4, 0.85}

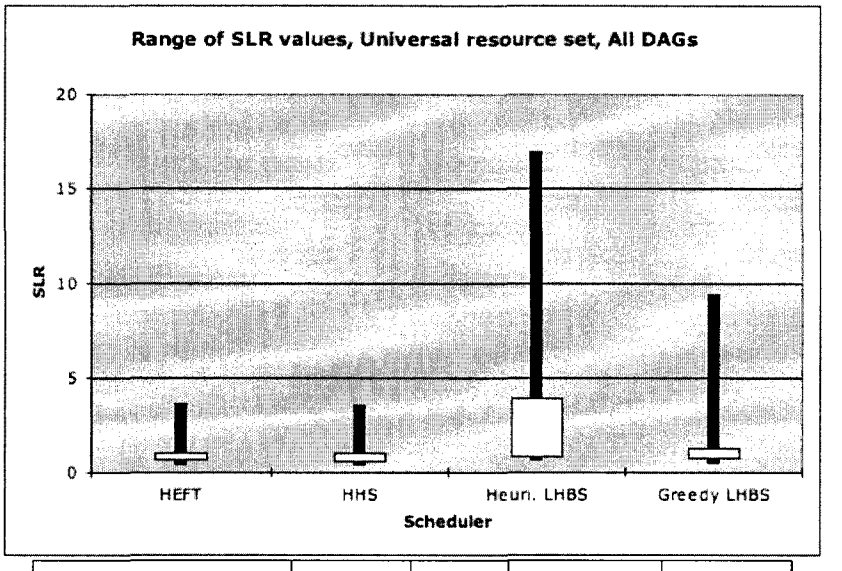
With three Grids and four scheduling algorithms, we collected about 120,000 schedules and their associated makespans.

3.4 Results

We will use SLR described in Section 2.3.1 to measure the schedule quality since the size of the DAGs in the experiment varies greatly. Over the entire set of DAGs, Grids and schedulers, SLRs range from 0.06 to 88. (The range of makespans is even greater.) Moreover, the algorithm that produces the best schedule (low SLR) for any individual DAG varies with no obvious pattern. Once the results are aggregated, however, a somewhat clearer picture emerges.

3.4.1 Results Analysis

Figure 3.2 shows the range of SLRs for each scheduling method on all DAGs for the universal resource set. The top and bottom of the white boxes are the 75th and 25th percentile SLRs for each scheduler, while the top and bottom of the black lines are the 90th and 10th percentile. It is clear that all the methods have many high-



	HEFT	HHS	Heuristic LHBS	Greedy LHBS
Mean SLR	1.99	1.97	5.58	2.90
Std. Dev. SLR	4.12	4.12	10.35	6.05
Median SLR	0.90	0.89	1.21	0.92
% of best cases	56%	60%	1%	29%
% of worst cases	0.5%	0.3%	80%	19%

Figure 3.2 : Aggregate behavior of scheduling methods

SLR outliers, but that the bulk of the results from the HEFT, HHS, and Greedy LHBS methods are comparable. The included table shows the average results for

each method. Despite the high variance of data, the differences between the means are statistically significant at levels far less than $p = 0.001$ (according to paired t-tests). Even the 1% difference between HEFT and HHS has a statistical significance of $p = 6 \times 10^{-6}$, although that difference may not be noticeable in practice. The last two lines of the table show how often each method returned the best and worst result for the same DAG among the four algorithms we tested. The percentages add up to more than 100% due to ties; HEFT and HHS often computed equivalent schedules, particularly for choke DAGs. This would lead us to believe that HEFT and HHS produce better schedules than level-based methods on average. However, we did not observe the clear advantages of HHS over HEFT reported by Sakellariou and Zhao [94].

The difference in behavior was not, however, consistent across types of DAGs, as shown by Figure 3.3. In particular, all of the methods produced good schedules for EMAN. Most of the differences are statistically significant (the exceptions are HEFT and HHS results for level and EMAN DAGs), but many are too small to be important in practice. Nor was the difference between methods true of all resource sets, as Figure 3.4 shows for random DAGs. We can clearly see that the LHBS algorithms perform much worse in the larger resource pool. The differences in the figures are all statistically significant except for the two LHBS algorithms in the big-cluster resource set. However, many are likely smaller than the uncertainties in our simulation.

After examining some of the schedules, we hypothesized that most of the differences were due to LHBS methods emphasizing parallelism over communication costs. One scenario is that LHBS might assign some DAG tasks to clusters that have a earlier start time in order to to achieve a shorter makespan in one level. If these tasks require input from two or more clusters, the estimated communication costs might be equivalent for that level. At the next level, however, having the tasks on different clusters might require additional inter-cluster communications. This sce-

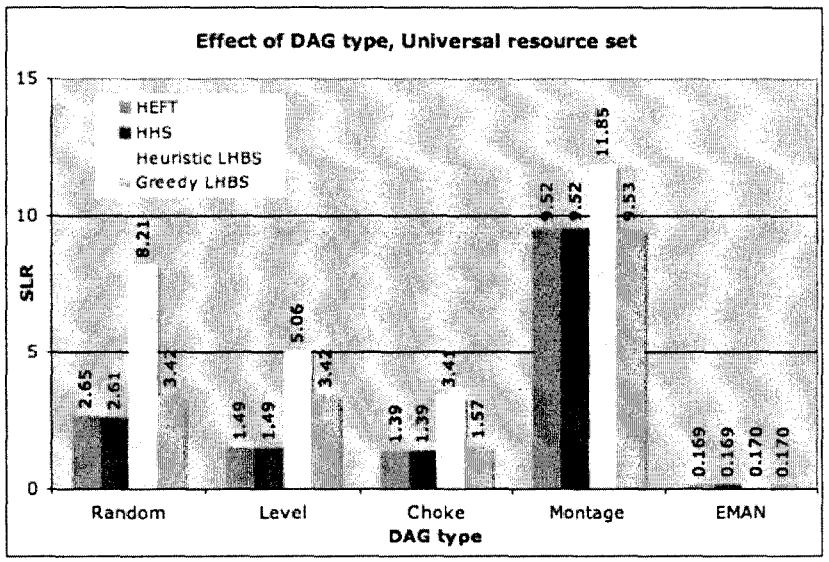


Figure 3.3 : Results for different DAG types

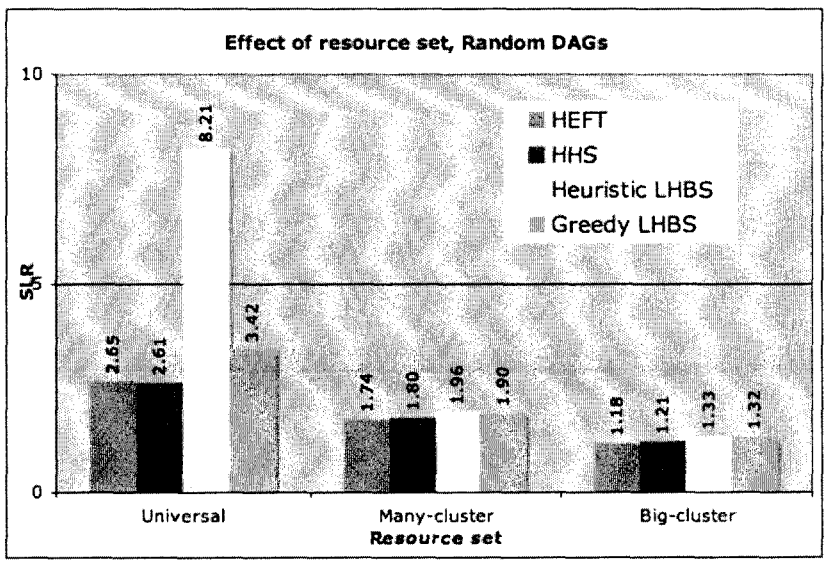


Figure 3.4 : Algorithms Performance on Different Resource Models

nario would obviously have more impact when a DAG required more point-to-point communication. (All-to-all communication, as in EMAN, does not necessarily suffer,

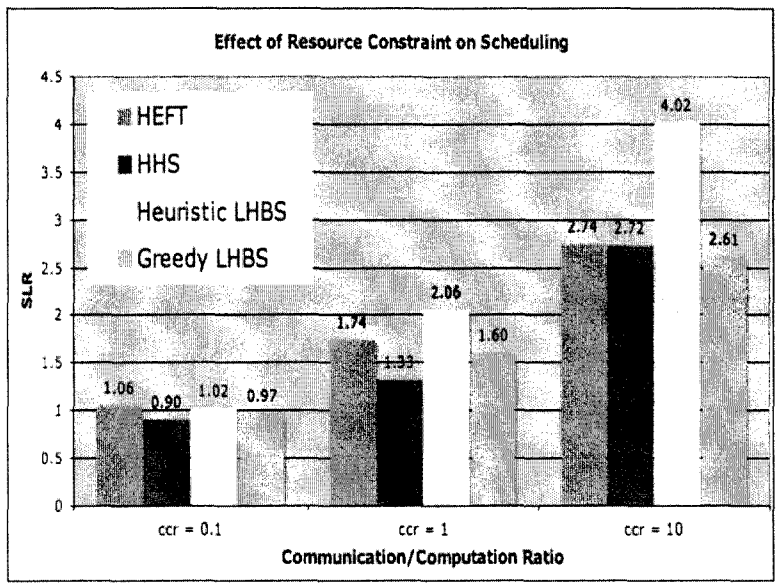


Figure 3.5 : Results for varying communication-computation ratios (CCR)

because the inter-cluster communication is almost always required.) This may have a smaller impact on HEFT and, to a lesser extent, HHS because tasks with high future communications requirements could be scheduled earlier with higher rank, when the resources nearby (i.e. processors within the same cluster) may have not yet been allocated.

To test this, we examined the sensitivity of the algorithms to various DAG attributes. Figure 3.5 shows the average SLR for low-communication (CCR=0.1) , medium-communication (CCR=1), and high-communication (CCR=10) DAGs. We can see that the performance difference among algorithms is very sensitive to CCR. We think it is because high communication costs affect the performance of LHBS the most as expected. Wide DAGs should also show the effect, since there are more opportunities for inappropriate parallel assignment. Figure 3.6 shows this for wide ($\alpha = 5$), square ($\alpha = 1$), and narrow ($\alpha = 0.5$) DAGs. Figures 3.5 and 3.6 consider only the random, level and choke graph types.

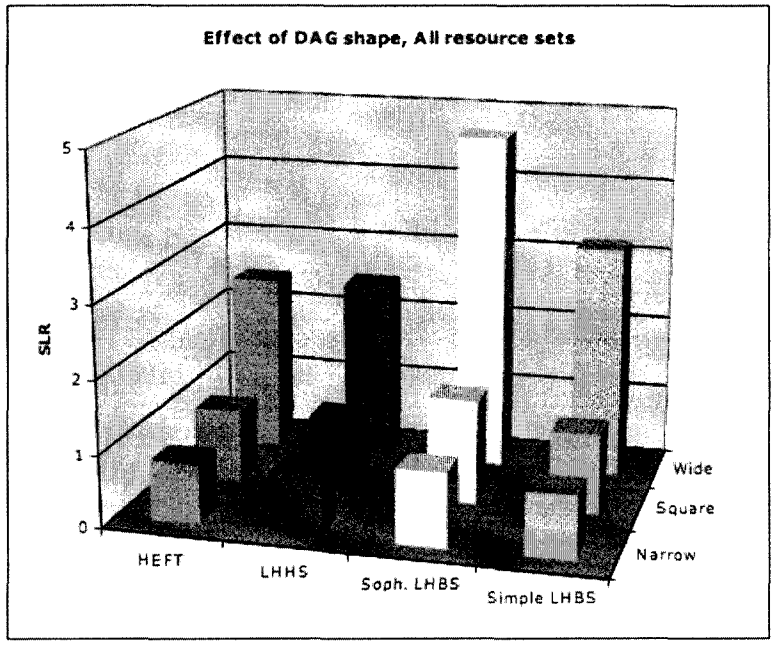


Figure 3.6 : Results for varying shapes (α)

It may be less apparent why our hypothesized parallelism/communication trade-off affects the large universal environment much more than the others. The connection is in the characteristics of the resource pools. As we will see in Chapter 4, our algorithms typically select processors from clusters with the fastest nodes. Table 3.1 lists the number of nodes and their speed in the four highest-GHz clusters in each of the three Grid environments. Clearly, the per-node speeds of these clusters in the universal resource environment are closer than in the other environments. At the same time, the top cluster in the universal environment is larger than in the others. Therefore, a relatively narrow DAG (e.g. width=40) can be run entirely on a single, fast cluster in the universal environment. Running the same DAG on the many-cluster or big-cluster environment must either use a slower cluster (e.g. the second cluster in the big-cluster environment) or multiple clusters (e.g. all four displayed clusters in the many-cluster environment). Figure 3.7 illustrates this effect. When

	Universal		Big-Cluster		Many-cluster	
	nodes	speed	nodes	speed	nodes	speed
First	78	4.2 Ghz	38	4.2 Ghz	13	4.2 Ghz
Second	6	4.2 Ghz	52	3.0 Ghz	18	3.8 Ghz
Third	103	4.1 Ghz	88	2.8 Ghz	17	3.7 Ghz
Fourth	118	4.1 Ghz	34	2.0 Ghz	6	3.6 Ghz

Table 3.1 : The Configuration of The Four Clusters with Fastest Processors in The Resource Pool

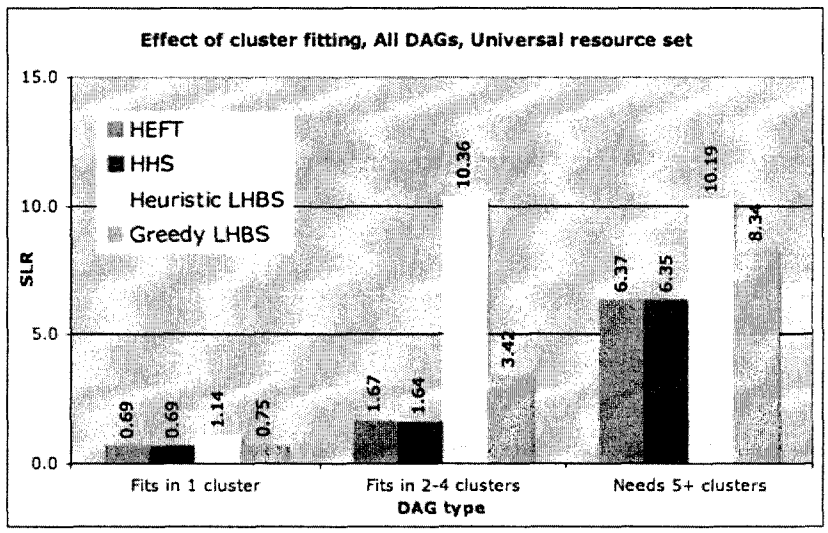


Figure 3.7 : DAG Performance in Universal Resource Environment with Different Widths

the DAG's width is less than the number of nodes of the fastest cluster or is larger than all the nodes in the fastest four clusters, the difference between algorithms are much smaller than when the DAG's width is in between. In other words, when the choices between clusters are obvious, all the algorithms perform relatively the same, while when the choices are tough, different algorithms can perform very differently. The above observations suggest that we could improve the quality of schedules for

Grid environments by choosing the clusters on which to run more intelligently.

3.4.2 Effective ACP

To investigate further, we introduce the notion of *effective aggregated computing power (EACP)* and apply it within the *two-level scheduling* approach in Chapter 4. In short, our two-level scheduler performs a very fast selection phase to select a suitable subset resource from the large resource base represented by the real Grid. It then performs a more complex scheduling step, such as LBHS, to map the application to nodes within the chosen subset of the total resources. Chapter 4 will describe our approach in detail.

We define Aggregated Computing Power (ACP) for a cluster A as

$$ACP(clusterA) = \sum_{B \in A} computing\ power\ of\ \ node\ B$$

We use the node's clock rate as an approximation of the computing power, although we could use more sophisticated performance models [105] as well. ACP represents the peak computing power of a cluster, but this may not all be usable on a particular DAG. For example, consider running 20 independent tasks on two clusters. Cluster A consists of 100 processors running at 1GHz, while cluster B consists of 30 processors running at 2 GHz. Our unit of comparison is one processor running at one GHz. Although A apparently has a higher ACP (100 units vs. 60 units), the DAG can utilize at most 20 processors in either cluster. Therefore, we introduce the notion of *effectiveness* which only aggregates the computing power up to the width of the DAG.

$$EACP(clusterA, jobJ) = ACP(subcluster_E)$$

where sub-cluster E has just enough nodes to run job J with the maximum parallelism possible. In our example, cluster B has 40 effective ACP units while cluster A has 20.

Within the two-level scheduling algorithm described above, the selection phase chooses nodes from clusters with the highest *effective ACP* for the given DAG. After

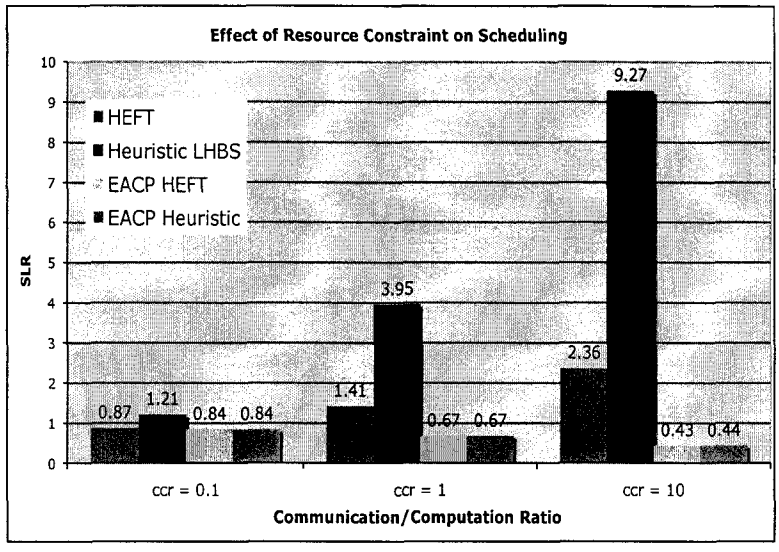


Figure 3.8 : Comparing EACP version with the standard version

this selection, we apply the HEFT, LHBS and HHS algorithms to the smaller universe of resources. Below we will refer to this as the *Effective ACP version* or simply the *EACP version* of each standard algorithm. Figure 3.8 and 3.9 show how the EACP versions of HEFT and Heuristic LHBS compared to the corresponding standard algorithms under the universal resource environment the three generated classes of DAGs. The EACP versions of the other algorithms exhibited very similar results. The leftmost set of bars of Figure 3.8 represents DAGs that have low communication cost (CCR =0.1). In this case, the EACP version algorithms do not have a large advantage over the standard HEFT or the heuristic LBHS scheduling algorithms. The middle set represents DAGs that have medium communication cost (CCR =1.0) and the rightmost set represents the most communication intensive DAGs (CCR =10). We thought that the standard methods would be more likely to make bad trade-offs between parallelism and communication in these cases. The results confirm our beliefs. The EACP versions of HEFT and Heuristic LHBS outperformed their standard versions by factors of 2 to 20 in aggregate. Both EACP algorithms performed better

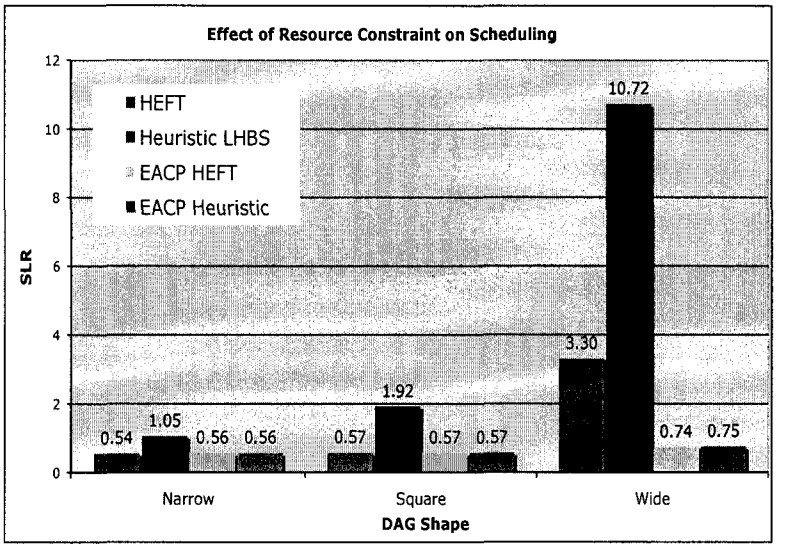


Figure 3.9 : Comparing EACP version with the standard version

than any standard algorithm.

Similarly, Figure 3.9 shows that the EACP version algorithms have much better performance than the standard algorithms when the DAG is wide ($\alpha = 5.0$) and is similar to standard versions for other cases. Taken together, Figures 3.8 and 3.9 show that 2-level selection based on effective ACP can vastly reduce the inter-cluster communication cost when communication is significant. In addition, the EACP version algorithms are more scalable in very large Grid environments since the complex scheduling algorithms are only applied to a subset of the universal resources. Chapter 4 will quantify the scalability achieved by this two level decoupled approach that separates the resource selection and scheduling.

However, the results may vary depending on the Grid used. For example, Figure 3.10 shows the results of similar experiments using the big cluster environment shows that the EACP version of HEFT can perform 10 to 20% worse than the standard HEFT algorithm . We can explain this from the entries of Table 3.1. In the big-cluster grid, the highest ACP cluster (the third) has relatively slow processors,

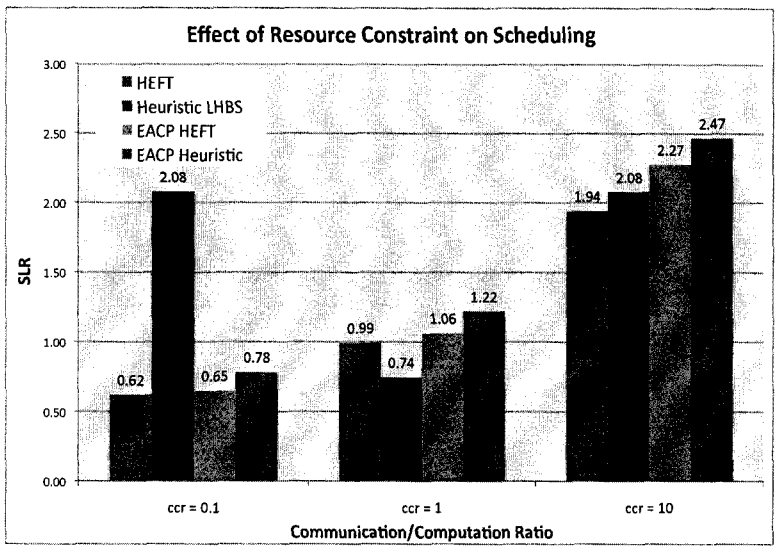


Figure 3.10 : Comparing EACP version with the standard version

so the fastest two clusters are likely to have the highest EACP for all but the largest DAGs. However, it happens that the network connection between these two clusters is slow in our experimental setting. Thus, selection based on EACP actually increases communication costs because it puts data movement on a slow link. More work is clearly needed to take effects like this into account.

3.5 Conclusions

In this chapter, we compared the performance of several algorithms that represent alternative major approaches to scheduling on three different Grid environments. Our experiments show that the list-based, and hybrid, scheduling algorithms are effective in a Grid environment, outperforming level-based scheduling methods on many combinations of environments and DAGs. The experiments also show how different factors in a Grid computing environment affect the performance of the scheduling algorithms. The most critical question for scheduling in the Grid environment is whether to assign a task to a cluster different from its parents: performance of the

algorithms is highly sensitive to this question. Finally, the experiments demonstrate that using effective aggregate computing power (EACP) in the selection phase of a two-level algorithm, then scheduling to the resulting virtual grid with a standard algorithm, can produce significantly improved schedules over the standard version of the same algorithm.

Chapter 4

Decoupled Resource Selection and Scheduling

This chapter presents our work on producing good schedules in a scalable manner in a Grid environment with hundreds of thousands of computing nodes. In this chapter, we are going to focus on the scheduler's speed which is the time to compute the schedule instead of the quality of the schedule which is the time for the application that follows the schedule to finish. The key idea is to decouple resource selection and scheduling so that we only schedule the workflow application on a subset of the total available resources. Furthermore, our results show that it is possible to achieve similar or even better performance than the traditional approach that combines selection and scheduling by selecting the resource subsets judiciously.

4.1 Introduction

In this chapter, we focus on one potential problem that may keep us from achieving good performance for a Grid application: the scalability of application scheduling. One distinguishing feature of grid platforms is the large number of individual resources, with the largest systems containing tens or even hundreds of thousands of resources [18]. This volume of resources raises scalability issues, especially in resource discovery and resource monitoring. In this chapter we specifically address the scalability of the scheduling algorithm itself: how can one compute an efficient application schedule in a short amount of time while considering a large number of potential resources?

We observe that although the resource environment may contain large numbers of resources, all of which are mostly likely taken into consideration when computing

a schedule, typically only a small subset of these resources is used for running the application. In essence, most scheduling heuristics perform *implicit resource selection*: the set of resources used by the application emerges from the computation of the schedule. In this work, we improve the scalability of the scheduling process by performing *explicit resource selection.* In contrast to the traditional *one-step* approach, which considers all available resources when scheduling, we use a *decoupled* approach, which selects the resources for consideration first and then schedules the application on these resources.

We use the *Virtual Grid* (VG) abstraction introduced in Section 2.1.2 and by Kee *et al.* [55]. A VG provides a high-level, hierarchical abstraction of the resource collection that is needed and used by an application. A user creates a VG specification, written in the Virtual Grid Description Language (vgDL), and passes it to the Virtual Grid Execution System (vgES). The vgES performs fast resource selection in grid environments with hundreds of thousands of resources, returning a set of selected physical resources on which one can schedule the application. The set of selected resources is typically many orders of magnitude smaller than the whole universe of resources, and the running time of a scheduling algorithm over this smaller subset of resources is also orders of magnitude shorter.

While decoupling resource selection from scheduling in large-scale systems as described above clearly improves scalability of the scheduler itself, a key question is: what is the impact of decoupled resource selection and scheduling on the quality of the resulting schedule? In this chapter we study decoupled resource selection and scheduling in the context of workflow applications in large-scale highly heterogeneous grid environments and make three contributions:

1. We demonstrate how the VG abstraction can be leveraged to decouple resource selection and application scheduling in a generic way (i.e., our approach is in principle applicable to any scheduling algorithm and any grid application).

2. One key issue in our decoupled approach is that of choosing an appropriate re-

source selection methodology. We discuss and provide a quantitative evaluation of several factors that affect the construction of an appropriate VG specification.

3. Using simulations of representative workflow applications on representative grid environments, we quantify the trade-off between scalability and schedule quality for our decoupled approach, demonstrating that it achieves schedule quality comparable to that achieved by one step approaches, at dramatically higher scalability.

This chapter is organized as follows. Section 4.2 presents our decoupled resource selection and scheduling idea. It also discusses our resource selection strategy in detail, by introducing the Virtual Grid concept, the specific scheduling and selection methodologies used and the application context in which we evaluate it. Experimental evaluation and results are presented in Section 4.3. Section 4.4 discusses related work. Section 4.5 concludes the chapter with a summary of contributions and perspectives on future work.

4.2 Decoupled Application Scheduling in Grid Environments

4.2.1 Virtual Grid and Resource Selection

As we described in the introduction, our proposed solution to address the scheduler's scalability problem is to decouple resource selection from application scheduling. In the first phase, we perform *explicit resource selection.* In the second phase, we perform scheduling within the selected resources rather than on the whole resource universe. The key point here is that a decoupled approach makes it possible to compute schedules faster, by several orders of magnitude, making application scheduling scalable to large-scale platforms. In fact, this decoupling may make it possible to run expensive scheduling algorithms on the explicitly selected resources.

We claim that using a system such as vgES (see Section 4.2.1) to perform explicit resource selection makes it possible to achieve schedules that are comparable in

quality to the ones obtained when letting the scheduling algorithms perform implicit resource selection over the whole resource universe, at dramatically higher scalability. Although our decoupled approach is generic, in this chapter we discuss and evaluate it in the context of workflow applications, as seen in Section 4.2.4.

A fundamental challenge for grid applications is to describe and obtain appropriate resources to enable efficient, high performance execution. This is challenging from many standpoints, including the definition of an appropriate abstraction to describe resource needs, the difficulty of finding appropriate resources quickly in an environment with many thousands of resources, and interacting with diverse, autonomous resource managers that implement their own resource management and access policies. As noted in the introduction, the VGrADS project [55] approaches this by allowing the user to specify its resource needs using a high-level language, vgDL [22], which our execution system, vgES [55], uses to find and allocate appropriate resources for the application, returning a VG abstraction, which is really an active entity (i.e., runtime object). By contrast with traditional low-level resource description and selection systems [9, 5] that focus on individual, quantitative resource characteristics, the VG provides a high-level, hierarchical abstraction of the resource collection that is needed by an application. The application can then use the VG to find specific information about the allocated physical resources, to deploy application components, and to modify or evolve the resource collection.

We refer the reader to previous research by Kee *et al.* [55, 22] for details regarding the vgES system and we only describe here features of vgDL that are relevant for this work. The vgDL language uses high-level resource abstractions that correspond to what grid application programmers typically use to organize their applications portably across many different resource environments. VgDL was designed based on a detailed study of half a dozen real-world applications. This showed that in order to design for performance (and to manage complexity) portably, application developers typically use three simple resource abstractions to aggregate individual resources.

Consequently, vgDL contains three resource aggregates, distinguished based on homogeneity and network connectivity: (i) LooseBag — a collection of heterogeneous resources with no guarantee of good connectivity; (ii) TightBag — a collection of heterogeneous resources with good connectivity; (iii) Cluster — a well-connected set of homogeneous resources. Each aggregate specifies a range for its size (i.e., number of resources). The user can specify constraints on attributes of individual resources within the aggregate (e.g., clock rate, processor architecture, memory, etc.), or constraints on aggregate attributes (e.g., total aggregate memory, total aggregate disk space). Aggregates can be nested (e.g., a LooseBag of Clusters) to arbitrary depth. With these resource aggregate abstractions, an application can structure the specification of its resource environment in a top-down fashion and decorate components with constraints when needed or desired. In addition to constraints, applications can also express resource preference by using a scalar rank function: a user-defined expression of basic arithmetic operators, resource attribute and resource aggregate attribute values that define a scalar value that represents the quality of that resource set for the application's request.

The Virtual Grid Execution System (vgES) uses efficient search techniques based on resource classification in a relational database. Table indices and other sophisticated database optimization techniques make the search highly scalable in environments with large number of resources. For instance, Figure 4.1 shows that it takes no more than 5 seconds for the vgES system to process one million resources for various queries on a Pentium4 3.2 Ghz processor. The different lines in the figure represent different types of query with L, T, C meaning LooseBag, TightBag and Cluster respectively and the number denoting the size of the requested VG. We will see in Section 4.3 that the ability to perform such resource selection in a few seconds is key for improving the scalability of application scheduling on large-scale platforms.

Given that vgDL makes it possible to specify high-level, qualitative resource requirements and that vgES can perform fast resource selection in large-scale resource

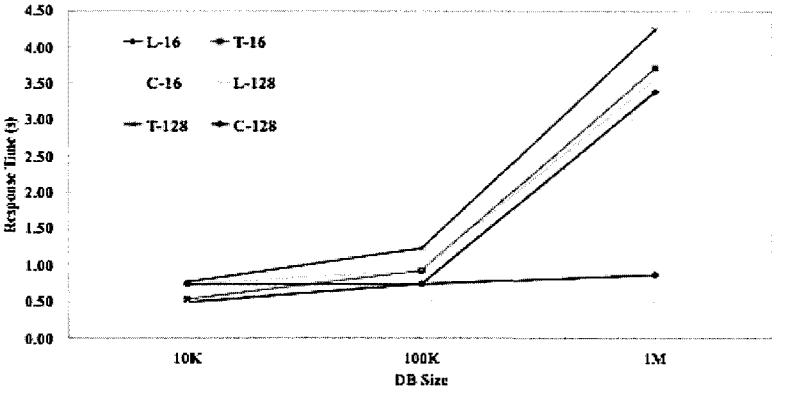


Figure 4.1 : Time to complete vgDL queries with vgES.

environments, the VGrADS project provides an ideal foundation for decoupling resource selection from application scheduling.

4.2.2 Scheduling Algorithms

While our decoupling approach is applicable to any scheduling algorithm, we chose to apply it to a specific workflow-scheduling algorithm to evaluate our approach. We use a *greedy* level-based (LHBS) workflow scheduling scheme as described in Section 2.3.3. The computational complexity of our greedy scheduling scheme is $O(vp)$ in which v denotes the number of jobs and p denotes the number of resources. There are two reasons why we use the greedy LHBS in our experiments. First, it has the best scalability among all the heuristics we tested in Chapter 3. As we will see in Section 4.3, even the greedy LHBS takes several hours to finish on the largest experiment setting, a heuristic LHBS will take an order of magnitude more time. Secondly, although the greedy LHBS does not produce the best schedule, its schedule quality is very similar to the list based heuristic HEFT and performs better than the more expensive heuristic LHBS in our previous experiments. Note that while Chapter 3 tests various scheduling heuristics' performance (schedule quality),

this chapter focuses on how to make the scheduler scalable (time to compute the schedule), which is mostly orthogonal to the choice of the scheduling algorithm itself since our approach can be applied generically to any scheduling heuristics.

4.2.3 Selection Methodology

Now that we have picked a scheduling algorithm, we must decide on a resource selection strategy. Resource selection must be done according to the application's needs and we consider three classes for three different types of such needs:

1. **Class 1**: A set of resources that have high computing power but not necessarily good network connection between them, as needed by a computationally intensive application.

2. **Class 2**: A set of resources that are connected with high bandwidth and low latency but do not necessarily have high computing power, as needed by a communication intensive application.

3. **Class 3**: A set of resources that have relatively balanced computing power and connectivity, as needed by a balanced application that is neither compute- nor communication-intensive.

It is relatively straightforward to generate selection criteria for class 1: simply select the resources with the fastest processors. However, we need the help of the vgDL specifications and of the vgES system to select the resources that meet the requirement of class 2 and 3. The key concept here is the TightBag. Recall from Section 4.2.1. that a TightBag is a collection of heterogeneous nodes with good connectivity. It matches the requirement of class 2 perfectly. For class 3 we will use vgDL to specify a hierarchy of aggregates. The idea is to aggregate several TightBags into a single LooseBag so that we can get both high computation power and high connectivity.

The above classes provide bases for performing resource selection following three broad characterizations of an application's resource needs. For each such application we perform resource selection according to the three above classes. We expect that class 1 will be best for applications with low CCRs (described in Section 2.2.1), and that class 2 will be best for applications with high CCRs. We will verify that the CCR value of the application provides good guidance for selecting the appropriate resource selection method.

The final key element for resource selection is the specification of a bound on the number of required resources. One could ask for as many (potential) resources as there are resources in the whole universe of resources. This will not lead to any scalability improvement over a traditional application scheduling approach that performs implicit resource selection. Instead, as a simple heuristic, we request as many resources as the maximum width of the DAG representing the application's workflow. The intuition behind this choice is that this is the maximum number of resources that can be used by the application at a given time. Any additional resource would stay idle for the entire application execution.

4.2.4 Case-Study: Workflow Applications

We explore our approach of decoupled resource selection and scheduling in the context of two real workflow applications, EMAN [66] and Montage [10]. These applications fall into the general class of workflow applications. We described our two target applications in Section 2.2.2. We use different versions (with different numbers of tasks) of the EMAN refinement workflow DAG in our experiments. The largest EMAN DAG has a maximum parallelism of over 300. Similar to the EMAN workflow, we use different versions of Montage for our experiments and with the largest DAG has a maximum parallelism of over 300.

4.3 Experimental Evaluation

4.3.1 Methodology

Simulation Environment

In order to perform repeatable experiments on a large-scale resource environment we resort to simulation. Our simulated environment consists of three key components: the resource model, the network model, and the application model.

We use a similar resource model to the one in Chapter 3 and we generate a resource pool that contains over 36,000 hosts, which we call the *resource universe.* Our network model is also similar to the one we used in Chapter 3. We generated end-to-end latencies between compute clusters according to a truncated normal distribution. We set the mean of this distribution to 100ms, conforming to the results in Morris *et al.* [74], and we bounded the latencies from 1 to 200ms. For the network bandwidths, we set the connection within a cluster as 1000Mb/s and the interconnection between clusters range from 10Mb to 100Mb/s. These numbers are primarily based on results by Yang and Denis and their collaborators [119, 29]. Furthermore, we ensured that the higher the latency the lower the bandwidth.

Our application model comes directly from the real-world applications described in Section 4.2.4. For each application we generate DAGs that follow the same structure as those of the applications, but we vary their CCR and their widths. When simulating application execution, the execution times of the tasks on resources come from the DAG task weights and the performance models described in Mandal *et al.* [69], and the data transfer times come from the DAG edge weights and the latencies and bandwidth in our network model.

Since this is a simulated environment, we must make some assumptions that may not hold for real resources. We assume that we have an accurate performance model for tasks for both scheduling and computing the simulated makespan. (In fact, we have such models for EMAN and Montage.) We argue that since both the one-step

and the decoupled scheduler use the same performance model, this does not bias the comparison. We assume that the network performance is stable and predictable. This assumption eliminates the random error that may be introduced by the network fluctuation. We have found it to be the case for our experiments with EMAN, although other applications may see more variation. We assume that the resources are available immediately, and will remain available for the duration of the application. We assume that we already obtained all the resource information before the start of the experiment. We have these assumptions so that we can compare the performance of these scheduling algorithms on a level playground. Once again, we believe that these assumptions do not bias our comparison between the two scheduling approaches. We will address the scheduling issues caused by the dynamic and unreliable nature of a multi-cluster Grid in Chapter 5 and Chapter 6.

Experimental Setup

We first generate forty EMAN and Montage DAGs with five different CCRs and eight different widths. We use the greedy scheduling algorithm described in Section 4.2.2 to schedule these DAGs on the simulated resources. For each DAG, we first run the scheduling algorithm on the whole resource universe, which we refer to as the *one-step approach*, and record the running time of the scheduler. We then run the scheduling algorithm on smaller subsets of resources explicitly selected using the methodologies in Section 4.2.3. The running time for this *decoupled approach* is measured as the sum of the time for selection and time to compute the schedule. In both cases we record the (simulated) makespan of the application. To run our experiments, we used the Rice Terascale Cluster which is composed of Intel 900 MHz Itanium2 machines [113].

In order to determine how resource selection affects scheduler performance, we selected 10%, 7%, 3%, 1% and 0.3% of the "best" resources, corresponding to the resource selection methods for class 1 in Section 4.2.3. We will later refer this as the *simple selection approach.* We also performed selections based on vgDL specifications.

```
VG = TightBagOf(node)[1:n] {
    node = [(Processor == OPTERON) || (Processor== ITANIUM )]
        [Rank=Clock]
}
```

Figure 4.2 : vgDL for class 2 type of resource abstraction

To satisfy the requirements of class 2, we generated the vgDL description shown in Figure 4.2, requesting one TightBag of OPTERON and ITANIUM nodes. The "[1:n]" means there are at most "n" nodes in the TightBag; we set n as the maximum DAG width. We will later refer to this selection methodology as the *one TightBag approach.* Similarly, we generated the vgDL description shown in Figure 4.3 for class 3. Since we want to group as many nodes into a TightBag as possible, we set the size of the TightBag as 500 which is large enough to run the biggest DAG in our experiment. In our experiments, we set "m" as 3 and 5 and we will later refer to them as the *Three TightBag approach* and *Five TightBag approach* respectively. Later we will also refer to the three TightBag approach as the *LooseBag approach* since we use this for most of the DAGs belonging to class 3.

Finally we refer to the implicit resource selection approach used by the one-step approach as the *Universe approach.*

4.3.2 Results

Figure 4.4 shows that the one-step scheduler's total running time which includes the resource selection time and the scheduling time is linear in the number of resources considered. Figures 4.5 and 4.6 further breaks down the number and shows the average scheduler running time of the one-step and decoupled approaches for EMAN and Montage. We can see that the time used in the decoupled approaches is only a small fraction of the time used in the one-step approach, since the number of selected

```
VG = LooseBagOf(tb)[1:n] {
    tb = TightBagOf(node)[1:500] [Rank= Nodes] {
      node = [
                (Processor == OPTERON) || (Processor== ITANIUM )
            ]
            [
                    Rank=Clock
            ]
    }
}
```

Figure 4.3 : vgDL for class 3 type of resource abstraction

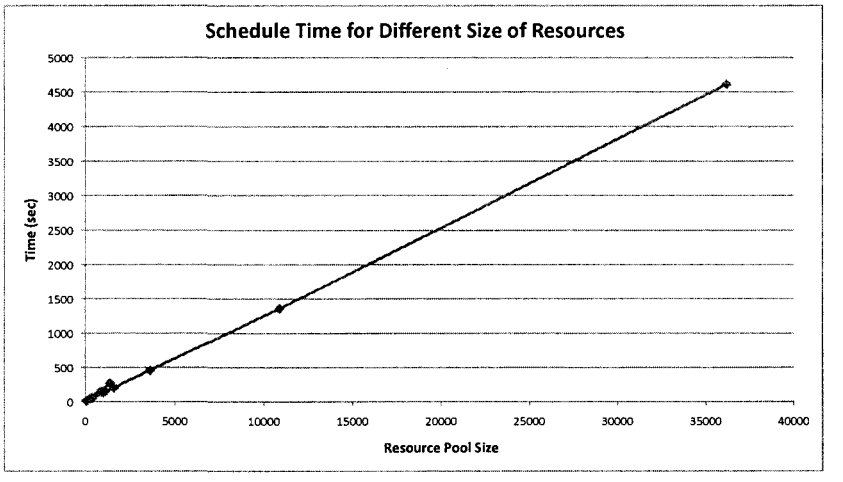


Figure 4.4 : Average Scheduling+Selection Time for Different Sizes of Resources

resources is much less than the full grid. This confirms our hypothesis of better scalability of the decoupled approach.

Figures 4.7, 4.8, and 4.9 show the combined makespan (yellow) and scheduling (blue) time for a range of simulations. In all charts, the total *turnaround* time for the application is the overall height of the bar. For the "Simple Selection", "One Tightbag", and "LooseBag" bars, we used the scheduling time for the case in Figures 4.5 and 4.6 that selects the least resources more than the maximum width of

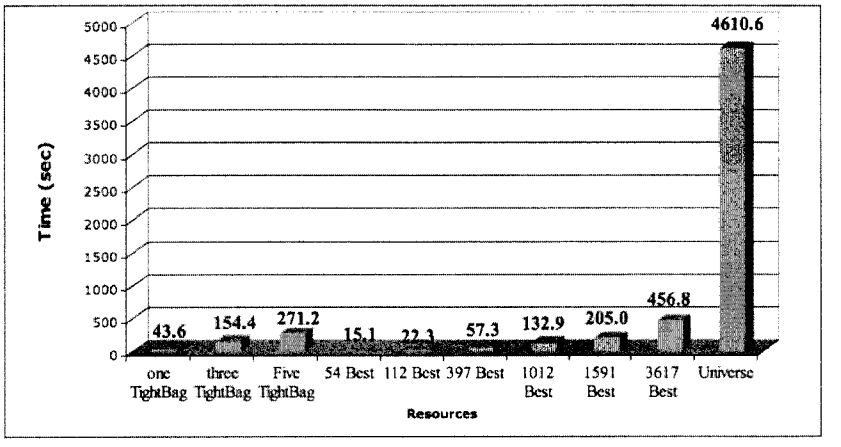


Figure 4.5 : Average Scheduling+Selection Time for EMAN DAGs

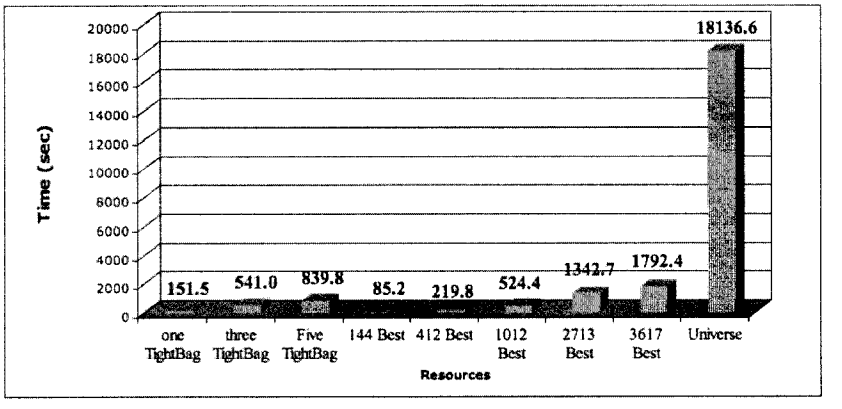


Figure 4.6 : Average Scheduling+Selection Time for Montage DAGs

the DAG. For example, for a test DAG of width 518, the Simple bar uses the "1012 Best" scheduling time, the TightBag bar uses "One TightBag", and the LooseBag bar uses "Three TightBags" since three TightBags most likely to hold just enough resources for the DAG (as compared to Five TightBags). All results are averages over a collection of EMAN and Montage DAGs.

Figure 4.7 shows results for computation-intensive DAGs belonging to class 1. We observe that all the decoupled approaches have much better turnaround time

compared to the one-step approach. Among decoupled approaches, the one TightBag approach performs the worst since it does not provide enough computing power. The simple selection approach performs the best with makespan only 2% worse than the one-step approach. This confirms our hypothesis that simple selection is very suitable for these applications.

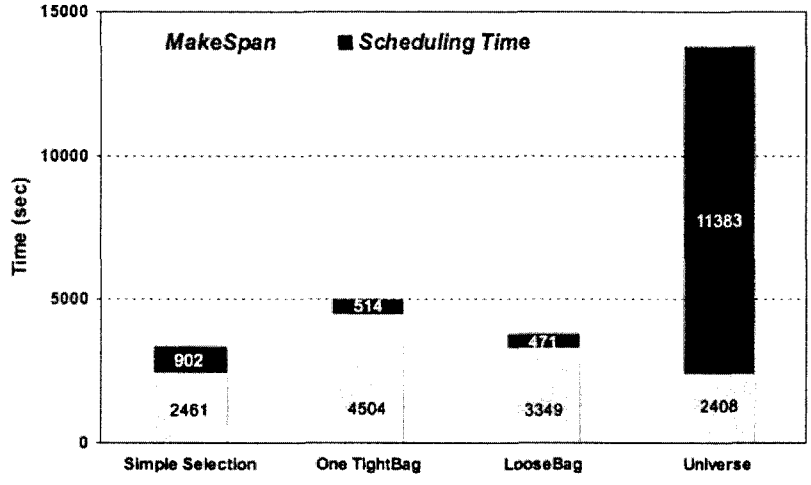


Figure 4.7 : Average MakeSpan and Scheduling Time for DAGs with CCR=0.1

Figure 4.8 shows results for communication-intensive DAGs belonging to class 2. We observe that all decoupled approaches have lower turnaround time than the one-step approach. The one TightBag approach has the best performance and outperforms the one-step approach by almost 66%. The main reason for this result is that all selected resources are closely connected, which avoids greedily choosing nodes with poor connectivity. A better scheduling heuristic for the Universe case might reduce its makespan, but at the cost of even higher scheduling time. This confirms our hypothesis that pre-selecting a TightBag is appropriate and efficient for scheduling this class of applications.

Figure 4.9 shows results for DAGs with relatively balanced communication and computation requirements, such as those in class 3. In all cases, the decoupled ap-

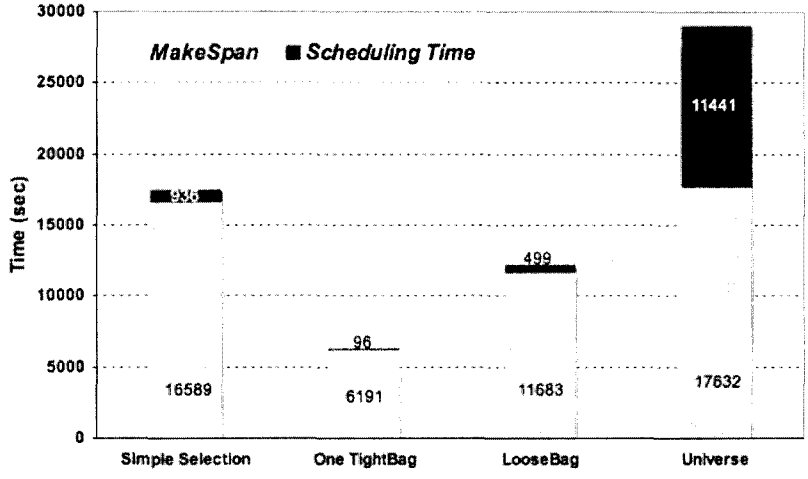


Figure 4.8 : Average MakeSpan and Scheduling Time for DAGs with CCR=10

proaches have lower turnaround times than the one-step approach due to their lower scheduling times, with gains of up to 50%. Also as we expected, the Simple approach

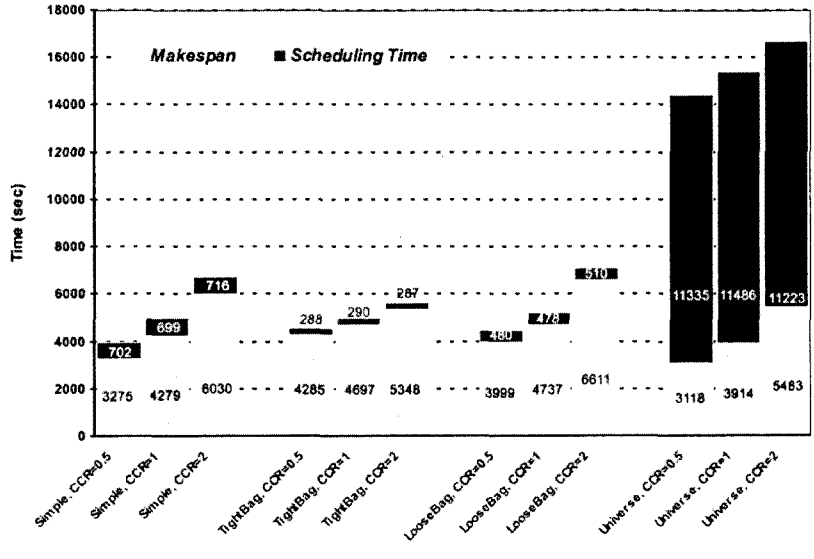


Figure 4.9 : Average MakeSpan and Scheduling Time for DAGs with CCR=0.5,1,2

performs relatively better as the CCR gets below one (i.e. more computation-intensive code) and the TightBag approach performs relatively better as the CCR gets above one. Unexpectedly however, the LooseBag approach does not show a clear advantage. Here are two possible reasons

1. The simple selection may implicitly select nodes that are close since fast nodes are more likely found in a few clusters than scattered around the grid.

2. The bandwidth between the TightBags within the LooseBag we choose may happen to be very low.

If reason 1 is true, we can further simplify our VGDL requests, while if reason 2 is the case, we may have to devise more complex queries.

In summary, our experiments confirm our hypothesized advantages for decoupled scheduling over the one-step approach. They also confirm our hypotheses of best scheduling methods for computation and communication-intensive applications. However, they do not match our expectations for balanced applications.

4.4 Related Work

Current grid workflow management systems use simple approaches as we described in Section 2.3.3. However, even those simple scheduling approaches (other than the random or round-robin approach) has the same time complexity as the greedy LHBS algorithm we used in our experiment. Therefore, they may also face the same scheduler scalability issues. Mandal *et al.*[69] and Blythe *et al.*[11] have developed level-based scheduling algorithms to schedule workflow applications onto a multi-cluster Grid as we described in Section 2.3.3. A key limitation of their approach however is that it is not scalable to large numbers of resources as it takes into account all the resources during scheduling.

4.5 Conclusions

In this chapter, we presented a decoupled mechanism that leverages the concept of a Virtual Grid to schedule workflow applications onto large-scale grid environments. Our approach improves scalability when compared to traditional scheduling approaches as schedules can be computed dramatically faster. Furthermore, our experimental results show that even when the decoupled approach increases the makespan slightly, the difference is more than made up by the reduced scheduling time. Therefore, our proposed approach can dramatically decrease workflow applications' turn-around-time.

Chapter 5

Hybrid Scheduling Mechanisms

In this chapter, we present our work on a hybrid scheduling mechanism that dynamically executes a top-down static scheduling algorithm using real-time feedback from the execution monitor. The motivation behind this mechanism is that, although static algorithms can achieve good schedule performance when the resources are static, Grid resources are dynamic. Changes to Grid resources can dramatically affect the application's performance in ways that a static schedule cannot account for. Our experimental results show that our hybrid rescheduling approach achieves the best performance among all the scheduling approaches we implemented on both exclusive resources and those with dynamic external loads.

5.1 Introduction

In Chapter 3 and Chapter 4 we examined the quality and performance of several scheduling heuristics for workflow applications in a multi-cluster Grid environment. Like most of the previous studies, we assumed that the task execution time and data transfer time were known beforehand. However, a real-world multi-cluster Grid environment is usually dynamic and unpredictable at least in three aspects: batch queue wait time, performance of individual processors (particularly the shared disk read/write speed) and network bandwidth. Therefore, it is difficult in general to accurately estimate the execution time for each task of the DAG and the communication time between them. To avoid this problem, many Grid projects use either dynamic dispatching mechanisms based on matchmaking [3, 72, 87] or application dependent scheduling [89].

We argue that we can still harness the better theoretical performance of static scheduling by integrating the scheduler with the application execution system. We propose a hybrid approach in which we first statically select the appropriate resources based on each resource's effective aggregated computing power as we proposed in Chapter 3, then dynamically schedule each task onto the selected resources based on the task's updated performance model and the execution monitor readings. Each task is launched with an execution monitor that can adjust the performance model parameters of the resources where the task is mapped. If the environment changes, the execution system and scheduler may re-select appropriate resources and re-map subsequent tasks according to the feedbacks from the monitors. In order to make our runtime decision efficient and scalable, our scheduler maps each task to a cluster, leaving the individual compute node assignment to the local resource manager. This is much more efficient because the number of clusters is usually at least an order of magnitude smaller than the total number of processors in those clusters. Furthermore, since a correct decision on when to trigger a reschedule is not always easy to make, we propose a two phase rescheduling mechanism that can mitigate the effect of a bad decision. The two phase decoupled approach further improves the scalability of the scheduler as shown in Chapter 4. The objective of our scheduling algorithm is to output a schedule for the workflow application such that the application's *turn-around time*, is minimized.

The rest of the chapter is organized as follows.Section 5.2 describe in detail our hybrid scheduling mechanism. Section 5.3 presents our Grid test-bed environment, the application DAGs we use and our experimental design. Section 5.4 presents our results and compares them with other approaches. Section 5.5 discusses the related works that address the dynamic nature of the Grid. Section 5.6 concludes the chapter with a summary of contributions.

5.2 Cluster based Hybrid Scheduling

The dynamic nature of a real-world Grid environment requires support in the scheduling system for detecting and responding to changing resources. In short, some form of dynamic rescheduling is needed to ensure the performance of applications. We have developed the framework shown in Figure 5.1 to provide this support, and to enable comparisons of our work to previous dynamic and static scheduling methods.

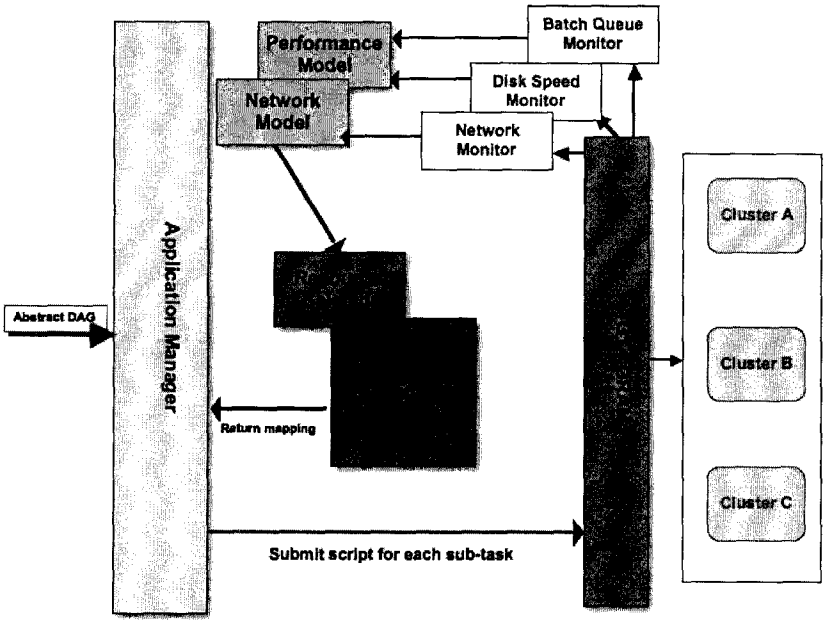


Figure 5.1 : The system design

As the figure shows, our framework consists of three major components: the monitor, the scheduler, and the application manager. The scheduler is responsible for resource selection and mapping the DAG to the resources. The monitor is responsible for monitoring the status messages generated by the Grid-run time middleware. The application manager is responsible for working with the scheduler and monitor to execute a workflow application on a multi-cluster Grid. In the following sections, we further describes how these three components work to implement our new hybrid

scheduling method. We will also note how previous static and dynamic schedulers can be implemented in this framework.

5.2.1 Scheduler

Section 2.3 describes the basic concepts of workflow application scheduling. We present the techniques we choose or propose in our cluster based hybrid re-scheduling mechanism.

Static Schedule Method:

We use a list-based scheduling algorithm since Chapter 3 showed that they generally perform well in a static Grid environment. Our algorithm is a modification of the popular HEFT [111] algorithm. Instead of the $rank_u$ used in HEFT, we simply use the earliest finish time (EFT) as the rank which means we favor the task that can finish first.

We assume that all clusters are controlled by local batch queues (as is the case of our test Grid). In the resource mapping phase, we choose the batch queue resource that can finish the task the earliest. Although we only select the batch queue instead of the individual processor, we have a low-cost way to keep track of the earliest start time for the queue by maintaining all the compute nodes' earliest start time in a heap data structure.

Resource Selection

We presented in the effective aggregate computing power (EACP) concept in Chapter 3 and used it to improve the makespan of the scheduling. The effective aggregate computing power is used to estimate the computer power of a cluster for an individual DAG. In this chapter, we extend this approach by taking into account the network model and make a finer estimation. The pseudo-code for our resource selection procedure is given in Figure 5.2 and Figure 5.3.

```
Algorithm: estimateTime(Resource cluster)
  double time[2]
  double dtime, ctime, ltime,fileSize
  for each level L in the dag
      ltime ← the longest task running time of jobs in L
      fileSize ← the total file output size of all the jobs in L
      if (L.width() > cluster.size)
         time[0] ← time[0] + ltime * L.width / cluster.size;
         time[1] ← time[1] + ltime + fileSize / cluster.bandwidth()
      else
         time[0] ← time[0] + ltime
         time[1] ← time[1] + ltime
  return time
```

Figure 5.2 : The DAG ACP estimation procedure

Figure 5.2 shows our algorithm to estimate the computing power of a batch queue resource. The resource computing power consists of two running times. *time*[0] estimates the execution time, including the queue wait time and the running time, if the DAG is mapped to only this resource. *time*[1] estimates time when other resources come into play. We denote the running time for the first case as the **exclusive time** and the second as the **collaborative time** and we define the computing power of a resource as the lesser of the two. We estimate the running time for the DAG level by level for each resource. For all the tasks in one level, we find the longest task running time and denote it as its *level time*. We also sum the output file sizes from this level's tasks. If the queue has enough processors to execute all the tasks in this level simultaneously, we add the level time to both the exclusive time and the collaborative time. Otherwise, we compute the additional computing time by

```
Algorithm: ResourceSelect ( DAG dag, Queue[] res )
  Map <Queue, double[]> queueTimePair_list
  for each queue in res
      queueTimePair_list.add( dag.estimateTime(queue))
  sort queueTimePair_list
  Queue best ← queueTimePair_list.first()
  List<Queue> selectedResource
  selectedResource.add(best)
  count ← best.size()
  est_acp ← best.getClusterTime ()
  while (count < dag.width() )
      resourceTimePair_list.removeFirst()
      sec_best ← resourceTimePair_list.first()
      new_acp ← min(selectedResource.getEACP(), sec_best.getCollaborateTime())
      if ( new_acp < est_acp)
        selectedResource.add(sec_best)
        est_acp ← new_acp
      else
        break
      count += selectedResource.size()
  return selectedResource
```

Figure 5.3 : The selection procedure

assuming the additional jobs would wait until the first batch of jobs finishes and add that to the level time in the exclusive time. We compute the communication time by dividing the task's total output file size by the queue's average bandwidth connecting

to its neighbors and add that to the level time in the collaborative time.

Figure 5.3 illustrates our resource selection algorithm based on the estimated exclusive and collaborative time for each cluster. We first sort the resources by their computing power and then apply a greedy algorithm. We pick the resource with the most computing power (shortest time) and then try to put more resources into the pool until the number of processors in the pool is more than the DAG's width. For each new resource in the pool, we estimate the new aggregate computing power as the lesser of the new queue's collaborative time and the existing pool's computing power. We stop the procedure if adding a new resource actually decreases the pool's aggregate computing power (increases the execution time).

Rescheduling

After we statically select the batch queue resources, we apply the scheduling algorithm dynamically, meaning we compute the mapping for a task at run-time when its predecessors have finished. There are two reasons we choose a dynamic over a static mapping. First, only the dynamic mapping can take advantage of updates to the performance model. The scheduler consults the performance model constantly during dynamic scheduling but only when rescheduling is necessary during static scheduling. Second, the dynamic mapping incurs less overhead if rescheduling is needed. Because static scheduling maps all the unexecuted tasks to a resource, many task mappings are no longer useful if rescheduling happens. Furthermore, in many cases, a task stages out the files to the resource where its successors run but if a reschedule happens, the file transfer might be wasted.

In our hybrid scheduling mechanism, rescheduling happens in the form of resource re-selection. When the application manager decides it is necessary to do a reschedule, the scheduler does another resource selection based on the current performance model. We will present the formula on which a reschedule decision is based in Section 5.2.3. Furthermore, instead of just using the newly selected resources, we combine these

two resource pools to be our new resource pool. The rule to combine the selected resources with the existing ones can be expressed in the following formula

$$Res_{new} = \begin{cases} Res_{sel}, & Res_{sel} \subset Res_{old} \\ Res_{sel} \cup Res_{old}, & Otherwise \end{cases} \tag{5.1}$$

where Res_{new} is the new resource pool while the Res_{sel} is the selected resource pool and Res_{old} is the existing resource pool. The intuition is that in the first case, the additional old resources would decrease the computing power of the resource pool based on the algorithm in Figure 5.3; otherwise they would have been selected. In the second case, we leave the decision of migrating the rest of the DAG to the scheduling algorithm instead of forcing the migration. Therefore, when a reschedule is triggered, the rest of the DAG would either be confined to a subset of the existing resources or gradually migrate to the new resources depending on the scheduler's decision on the trade off between better computing performance and more communication time. This two phase approach avoids the potential penalty of extra communication cost since the scheduler would avoid the new resources if the extra communication cost is too high. Furthermore, we can see that our two phase rescheduling approach also avoids the potential performance penalty when resource selections become a cycle. In this case, the application would be migrated back and forth between several resources if we used the traditional one phase rescheduling. It would be a much smoother transition in our two phase approach because our intermediate resource pool always includes the previous resources.

5.2.2 Monitor Component

The monitor component adjusts the performance model constantly so that it can reflect the observed performance of the underlying resources. There are three sub-components in the monitor component which monitor the batch queue wait time, the network bandwidth and the disk write speed, respectively. All three monitors use the

status notification (callback) system of the Globus middleware layer [4]. The batch queue wait time monitor records the time when a task enters the batch queue and the time when the task gets to run. It then calculates the new batch queue wait time and updates the batch queue wait time linearly

$$Wait_{new} = Wait_{old} \times 0.7 + Wait_{observed} \times 0.3$$

The reason we choose a fading memory model is that we want to smooth out possible system performance fluctuations. The coefficients are heuristic and can be tuned, but our experience shows that incorporating roughly one third of the new time into the overall wait time does a good job tracking the actual performance. We acknowledge that a more sophisticated model could do a better job but this is not the main focus of our research. The network bandwidth monitor records the observed transfer time and then computes the new transfer time according to the same linear model. It then divides the transfered file size by the transfer time to get the new network bandwidth. Similarly, the disk write speed monitor records the computing time of a task and calculates the new disk writing time. Although different resources may have different performance models our monitor will update the coefficient in them according to the latest time calculated. We will discuss the performance models we used in our experiments in detail in Section 5.3.2.

5.2.3 Application Manager

The application manager makes sure the workflow application is executed on the Grid resources correctly and on time. Figure 5.4 shows how it works. When a DAG that represents a workflow application arrives, the application manager first invokes the scheduler to select the resources and then collects the tasks that are ready to run. Once a task finishes, the application manager checks to see if this task is the exit task (each DAG has a unique dummy exit task) and all the tasks in the DAG are finished. Otherwise, it checks if a reschedule should be triggered. If so, it invokes the resource

selection again. Finally, it schedules and submits all the successors that are available to run.

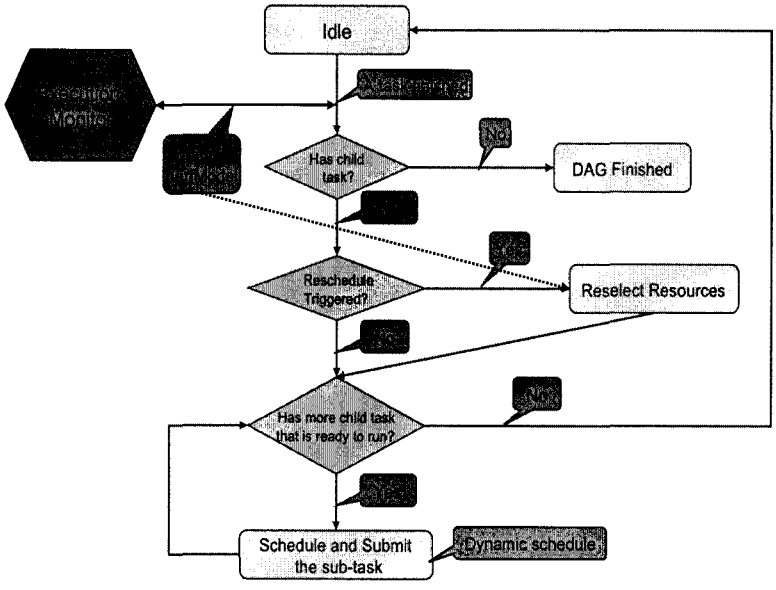


Figure 5.4 : The application manager

The reschedule trigger takes two parameters from the user, the tolerance level **T** and the monitor window size **WS**. The reschedule trigger calculates the actual to estimated performance ratio and signals a reschedule when the average ratio in the most recent WS task exceeds the tolerance level

$$\sum_{i=n-WS+1}^{n} \frac{act(i)}{est(i)} > WS \times (1+T)$$

where the current task is the nth task that finishes. Note that we do not trigger a reschedule if the actual performance is better than we estimated. It is generally very difficult to find an optimal parameter pair to always make the right decision. Section 5.4 will demonstrate that the two-phase approach we described in Section 5.2.1 is fairly robust even if a poor decision is made

5.3 Experimental Methodology

In order to study how well our rescheduling strategies perform in a Grid environment, we implemented the schedule algorithm and two-step approach described in Section 5.2. We will refer to this as the **hybrid rescheduling approach**. Besides that, we implemented three other strategies. The **static approach** uses the usual static mechanism with a resource selection phase but never triggers a reschedule. The **static rescheduling approach** computes a new static schedule with the updated performance models when rescheduling is triggered. This approach does not perform resource selection and is similar to the techniques used in Yu *et al.* [125]. The **dynamic approach** simply dispatches a task to the resource that has the earliest estimated finish time without taking into account the file staging time or the updated performance model. This approach is very similar to Condor's approach [72]. The rest of this section will further introduce our experimental environment.

5.3.1 Workflow Applications

We use DAGs taken from two real Grid applications, EMAN [66] and Montage [10] described in Section 2.2.2. In the Montage DAGs, the tasks in the same level have different execution times while the execution time is the same for EMAN DAGs. In addition, we used two well-known parallel algorithms that have been widely used in workflow scheduling research: Fast Fourier Transform (FFT) and Gaussian elimination; both are also described in Section 2.2.2. All paths in the FFT DAG are theoretical critical paths since all tasks in the same level have the same performance model and all dependencies are from one level to the next. However the real critical path depends on the application mapping. In contrast, there is a unique critical path in the Gaussian DAG that goes through the pivot steps.

5.3.2 Performance Model

Since we are only interested in the running time of the application instead of the real output, we chose to represent all four workflow applications using DAGs that consist of the same configurable tasks. We pre-installed the task executable on all the resources in our Grid test-bed. Each executable takes three parameters: the number of iterations *it*, the output file size *size* and the output file location. Since different resources have different capacities, the real time it takes the same configured sub-tsk to run on different resources are different. However, we use a linear performance model for all the resources with different parameters. Our performance model is

$$T = C + Co_{it} \times it + Co_{size} \times size$$

where C is a constant representing the execute overhead (such as cache build-up) and Co_{it} is the coefficient related to the computing iterations (the major computation of each task is a loop) and Co_{size} is the coefficient related to the disk read and write amount.

The run time of a task also depends on the batch queue wait time and the network bandwidth. As we stated earlier, these two coefficients along with the Co_{size} are dynamic and hard to predict. Therefore, we only set the initial values for each resource while the monitor system adjusts them during the DAG's execution. We obtain those initial values by running different configurations of the task executable on each resources many times. We list their values for each resources in the Figure 5.5.

5.3.3 Grid Model

Our multi-cluster Grid environment has four clusters: the Ada and RTC clusters at Rice university, the Eldorado cluster at University of Houston and the Lonestar cluster at the University of Texas at Austin. Since Ada, RTC and Eldorado were heavily used, in order to finish our DAGs within reasonable time we reserved the batch queues on them. The majority of our DAGs used the three clusters that we

reserved. Figure 5.5 shows the configuration of our Grid test-bed and the coefficients

	Ada	RTC	Eldorado	Lonestar
Nodes	8	16	16	64
CPU Type	Opteron	Itanium	Itanium	Xeon
CPU Speed	2.2 GHz	0.9 GHz	0.9 GHz	2.6 GHz
C	4.5	6.5	31.5	4.5
Co_{it}	37.8	52.1	50.72	26.8
Co_{size}	0.13	0.25	0.43	0.31
Wait Time	30	60	60	1800

Figure 5.5 : The Cluster Configuration and Performance Model

of our performance model. The unit for wait time is seconds.

5.3.4 Experimental Setup

We generated four cases for each type of DAG and applied four scheduling mechanisms on each case. In addition, we have four batches of experiments run with different execution environments. Two of them have a Grid environment with no loads on the reserved batch queues. The difference between them is the reschedule trigger parameters T and WS. The first batch of experiments uses $T = 0.2$ and $WS = 3$ while the second batch uses $T = 0.3$ and $WS = 5$. Later we will refer to the first batch of experiments as the *aggressive batch* since they are more likely to reschedule and the second batch as the *conservative batch.* The third batch of experiments use the aggressive batch's reschedule trigger parameters but we submitted a periodic load of 8 jobs onto the Ada batch queue. We will later refer to this batch as the *queue loaded batch.* The last batch of experiments uses the same reschedule parameters but we introduced an artificial disk write load on the Ada cluster. We will later refer

to this batch as the *disk loaded batch.* We ran the four scheduling mechanisms for one DAG consecutively to minimize the impact of the dynamic environment on the results.

Since the turn-around time varies widely among DAGs, we use *Schedule Length Ratio* (SLR) that we defined in Chapter 2. Intuitively, a small SLR is indicative of a better schedule than a large SLR and a SLR greater than 1 means the real turn-around time is longer than the static estimated finish time. With four batches of experiments, sixteen DAGs and four scheduling mechanisms, we have a total of 256 DAG executions. It took us about three months to collect all the schedules and their turn-around time. We used more than 6000 cpu hours since half of our runs did not finish because of various hardware and software failures.

5.4 Results

We now present our experimental results. Over the entire set of SLR numbers, the hybrid rescheduling approach outperformed the other three by as much as 45 percent. Figure 5.6 shows the overall results for each approach with batch queue reservations. The height of the bar indicates the mean SLR while the line segment shows one standard deviation. We can see that all approaches have a mean SLR value over 1 which means the average turn-around time is more than the makespan computed by the static scheduler. This confirms our hypothesis that predicting execution time accurately is difficult. For our experiment, the main reason why most SLR numbers are greater than 1 is that our initial performance model does not take into account the disk and network contention. Thus, the network bandwidth and disk write speed are lower when multiple tasks are mapped to the same batch queue resource. We will further analyze the results from different experiment settings in the rest of this section.

Figure 5.7 presents the results of the aggressive batch of experiment runs. Figure 5.8 presents the results of the conservative batch of experiments run. The differ-

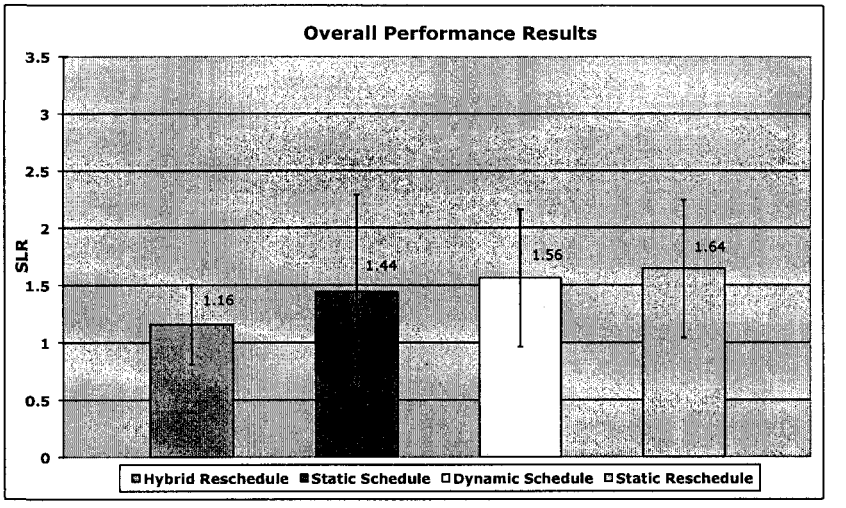


Figure 5.6 : Aggregate Results

ence between these two batches are the rescheduling trigger parameters. In both cases, the difference of means between the hybrid rescheduling and the dynamic scheduling, and between the hybrid rescheduling and the static rescheduling are statistically significant with over 95% confidence level and alpha set as 0.05. The hybrid rescheduling also outperforms the static approach on average, but the difference is not statistically significant. These results echo our findings in Chapter 3 that the static approach works well in a more stable environment.

In Figures 5.7 and 5.8 we can also see that the performance of static rescheduling is most sensitive to the rescheduling policies. While it is obvious that the static and the dynamic approaches are little affected by the rescheduling since they don't do rescheduling at all, it is interesting to see the hybrid rescheduling approach is virtually not affected either. The reason is that the reschedule is done in two phases as we described in section 5.2.1. The resource selector first selects a suitable resource and then the scheduler can gradually move some tasks to the new resources or leave them where they were depends on the estimated communication costs. Thus, the reschedule decision plays a less important role in our execution time since the real

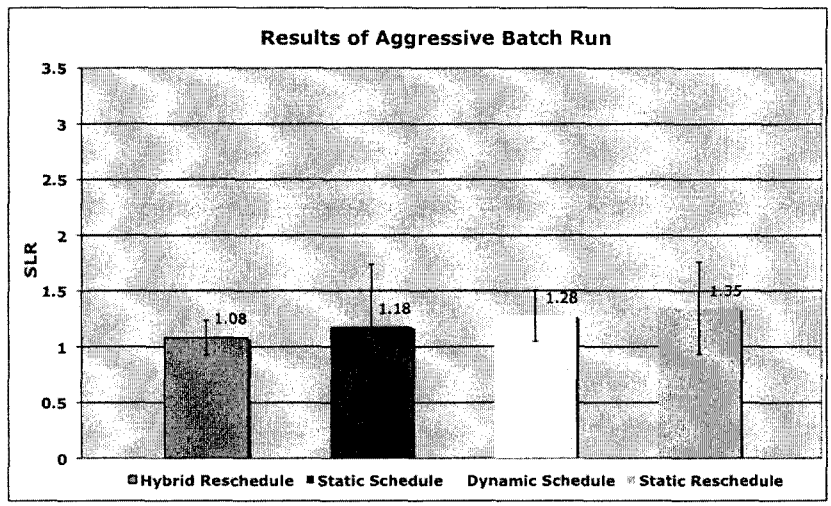


Figure 5.7 : Results of Aggressive Rescheduling Batch

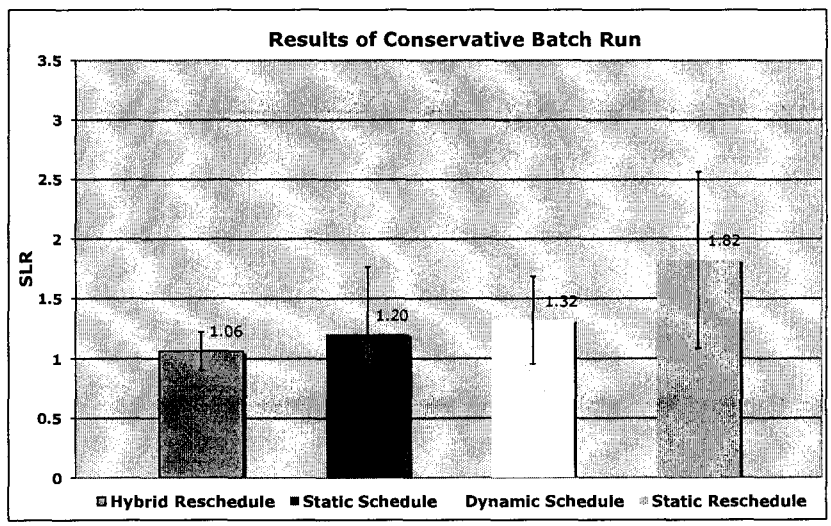


Figure 5.8 : Results of Conservative Rescheduling Batch

migration decision is made by the scheduler. In addition, the resource selector in our hybrid rescheduling mechanisms provides a global view of the DAG and could correct the reschedule trigger mistakes by selecting the correct resources. However, this is not the case in the static rescheduling approach. We know that it is usually

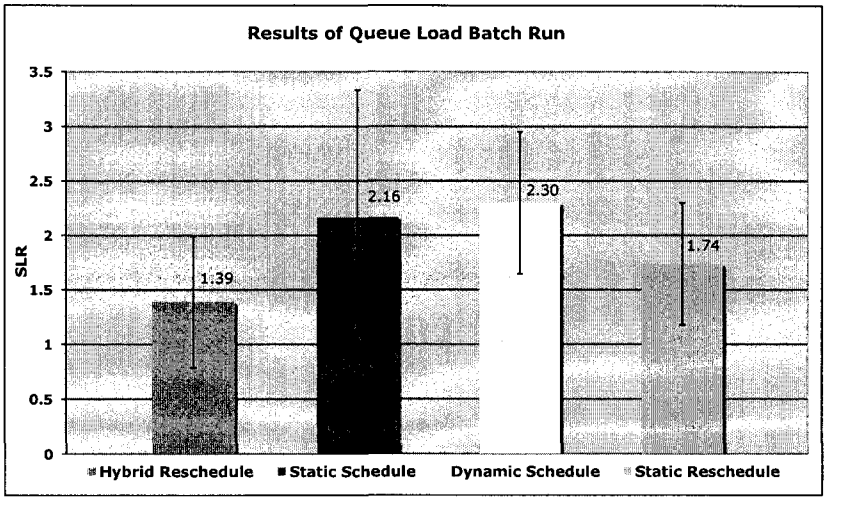


Figure 5.9 : Results of Artificial Batch Queue Loads Batch

difficult to find a good set of rescheduling trigger parameters, threshold and window size, that works well for all system performance changes. In our case, it appears that the reschedule trigger in the aggressive batch is more effective than the trigger in the conservative batch for the experimental system since the static reschedule performs 30% slower in the conservative batch. However, the aggressive batch's trigger may not work well on other resources. Without a resource selection phase, the entire rest of the DAG will be remapped whenever a reschedule is triggered. Therefore, a bad reschedule decision can negatively affect the overall performance of a workflow application significantly.

Figure 5.9 presents the results of the queue loaded batch. In this experiments, we submit a periodic load of queue jobs onto the Ada batch queue which is the fastest among the clusters we use. We can see that the external batch queue loads affect the static and dynamic approaches the most while the hybrid rescheduling approach again outperforms the others. The paired t-test shows that all the differences of mean value between the hybrid rescheduling and other approaches are statistically significant. This shows that our hybrid rescheduling approach can maintain a certain level of

performance when the underlying resources performance unexpectedly deteriorate.

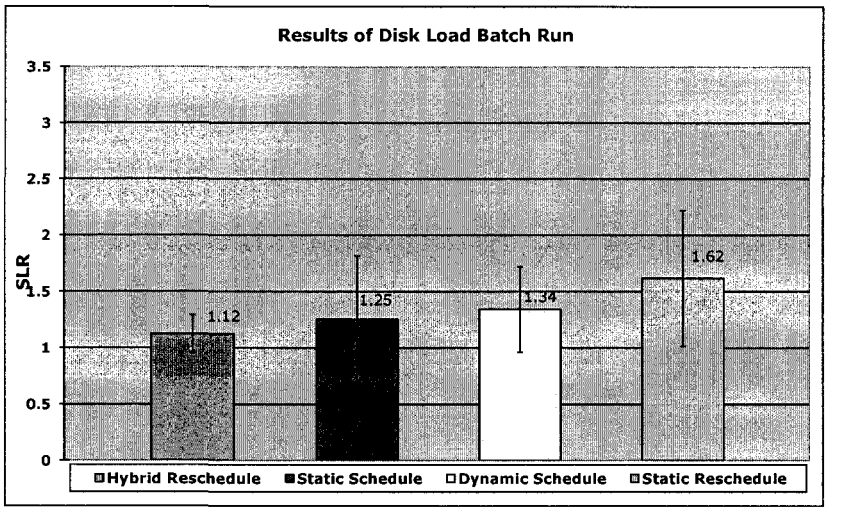


Figure 5.10 : Results of Artificial Disk Write Loads Batch

Figure 5.10 shows the results of the disk loaded batch. In this case, the extra disk write loads' effect not as dramatic because the tasks are not I/O bound. However, the static rescheduling approach suffers most because it assigns tasks to multiple clusters without the resource selection phase. This caused a lot of communication overhead when it tried to migrate the DAG when it detected the performance of Ada deteriorating. This experimental result also confirms that our hybrid rescheduling approach is effective in an unpredictable environment.

5.5 Related Work

As we mentioned in section 7.1, the traditional scheduling does not take into account that a real Grid environment is dynamic. Some efforts have been made to address these new complexities.

Condor [72] and DAGMan [82] provide checkpoint and migration when a resource is no longer available. However, Condor does not consult a performance model or the

network model. The mapping decision is solely based on finding the resource that matches the individual task's needs.

Sakellariou and Zhao [95] propose a low-cost rescheduling scheme which only starts rescheduling when the delay in a task would create a longer critical path (delay>slack of a successor). Their work shows improvements over the original static scheduling algorithms in a simulated environment. However, there is a chicken and egg dilemma in their approach that the slack of each sub-task can not be calculated precisely if the prediction of the job compute time and communication cost are not accurate.

Rahman *et. al.* [67] propose a scheduling algorithm that would compute the critical path dynamically and schedule the task on the critical path first. Their work shows that the dynamic critical path algorithm can generate a better schedule by up to 20% in simulated environments. However, similar to Sakellariou's approach, the absolute latest start time (ALST) of a sub-task can not be calculated precisely without the assumption of known computation and communication cost.

Yu *et. al.* [125] propose an adaptive rescheduling schema that takes advantage of additional resources during execution. Their work also shows improvement over static algorithms of up to 20% when new resources become available during the execution. However, this has limited usefulness in our multi-cluster Grid environment since the total number of resources is usually far greater than a single DAG's needs. This work is also done in a simulated environment.

Contract-based rescheduling was implemented in the GrADS project [26] on a real-world testbed. Vadhiyar *et. al.* [114] proposed a performance oriented migration framework that takes into account both the load changes and the remaining execution times of the subtasks. However, this framework is only designed for a single iterative MPI job.

Our work is closest to the GrADS project [26] but we extend it to handle workflow applications which, we argue, are significantly more complex and difficult to scale. In addition, most scheduling algorithms mentioned above assign each task to an

individual processor while we assign a task to a cluster or batch queue. Furthermore, we propose a new rescheduling mechanism and test our approaches on a real-world multi-cluster Grid.

5.6 Conclusions

The major contributions of this chapter are: (1) we propose a light-weight hybrid scheduling mechanism that works with local batch queue resource managers; (2) we propose a two-step rescheduling decision approach that mitigates the effect of a bad rescheduling decision; and (3) evaluate the performance of our approach in a real-world multi-cluster Grid and confirm that it performs well. Our experiments show that the static scheduling approach works reasonably well in a relatively predicable environment but the performance predictions are usually over-optimistic.

Furthermore, our hybrid scheduling mechanism performs better by correcting the prediction based on the runtime feedbacks. Our experiments also show that the runtime rescheduling policy is critical to the performance of the rescheduling approaches. However, if the application does not know the reason of the performance deterioration, it is very difficult to general policy to work well on all resources. We apply a two step approach to this problem so that we can take a relatively aggressive rescheduling policy while leaving the real migration decision to the scheduler. Our experiments show that this approach works better than the single step approach, such as the static rescheduling mechanism we implemented, in most cases.

Chapter 6

Fault Tolerance and Recovery for Workflow Applications

In this chapter, we present our work that combines fault tolerance mechanisms such as over-provisioning and checkpoint-recovery approaches with existing workflow application scheduling algorithms. We analyze our approach's impact on the workflow application's performance, reliability and resource usage under different reliability models, failure prediction accuracies and application types.

6.1 Introduction

Recent developments in grid infrastructure technologies make it possible to execute large and distributed applications [10, 38, 17] on it. Many of these applications fall in the category of workflow applications we described in Section 2.2. At the same time, the recent growth in size and complexity of the grid infrastructure makes it susceptible to failures at all system levels - power supply, computing hardware, network, operating system, grid middleware, etc. For example, the study in Iosup *et al.*[50] shows that the mean time between failures (MTBF) on Grid5000 [44] is only around 12 minutes. Hence, not only is managing and scheduling workflow applications a hard problem studied in detail [69, 127, 122], challenges in providing reliability to workflow executions also arise because of the unreliable nature of the underlying hardware and software.

To address the reliability challenges, existing grid systems resort to fault tolerance and recovery mechanisms [80] such as checkpoint-recovery and over-provisioning. Checkpoint-recovery techniques make it possible for the workflow to resume execution

from the last checkpoint instead of restarting from the beginning, should a failure occur. Over-provisioning [53] techniques replicate a task on more than one resource to increase the probability of successful execution. Although these techniques address the reliability challenges to some extent, to the best of our knowledge, no large-scale study has been done on how effective they are when coupled with workflow management and scheduling. In this chapter, we study the performance, cost and effectiveness of different fault tolerance mechanisms when combined with different scheduling techniques.

The main contributions of this chapter are:

- We propose and implement several scheduling and fault tolerance mechanism combinations.
- We evaluate the reliability, performance and cost of different mechanisms with a large scale reliability and resource model and provide a quantified model for the three-way trade-offs.
- We evaluate the effect of resource reliability and the accuracy of the failure prediction on the reliability, performance and cost of each mechanism

The rest of the chapter is organized as follows. Section 6.2 presents the details of the combined fault tolerance and scheduling techniques that we proposed and implemented. Section 6.3 describes our experimental design. Section 6.4 presents our results and evaluation. Section 6.5 presents related work and section 6.6 concludes the chapter with a summary of our contributions.

6.2 Scheduling with Fault Tolerance

Fault tolerance and recovery techniques used to mitigate the effects of workflow failures in grid systems fall in two major categories: (a) checkpoint-recovery and (b) over-provisioning/replication. In this section, we describe how we integrated these

fault tolerance and recovery techniques during scheduling and execution phases. First, we briefly describe the two scheduling algorithms and the traditional fault tolerance techniques used in this study.

6.2.1 Scheduling and Fault Tolerance Techniques

We will use two scheduling heuristics in our study. The first one is a list based algorithm called HEFT [111]. Our work in Chapter 3 has shown that it performs well in a multi-cluster grid environment. The second is a duplication based algorithm called DSH that was first proposed by Kruatrachue *et al.* [61]. Please refer to Section 2.3.2 for more details of both algorithms.

Checkpoint-recovery techniques are widely used for applications that run for a long time. The basic idea is that a usually combined effort of the application and the support system stores the intermediate state of the application periodically on a reliable storage system. The stored state of an application is usually called a "checkpoint". If the application or the resource crashes during the execution of the application, one can restart the application from the latest checkpoint instead of wasting the entire work. Checkpoint-recovery is very effective for recovering from application failures but it can not prevent the failures.

Over-provisioning is a more proactive techniques that duplicates an application onto multiple resources. In case one or more copies of the application or the resources fail, one can still get the result as long as at least one copy finishes. Even though checkpoint-recovery and over-provisioning are two different approaches, they can also complement each other because checkpoint-recovery techniques are applied mostly during workflow execution, while over-provisioning is applied mostly during the scheduling/planning phase. However, it is not trivial to combine them with the traditional workflow scheduling algorithms.

6.2.2 Scheduling Algorithms with Over-provisioning

Since the HEFT and DSH scheduling algorithms themselves do not take into account any fault tolerance, we integrated the over-provisioning technique with the vanilla HEFT and DSH scheduling algorithms to develop a fault-tolerant scheduling scheme.

Fault Tolerance Using Over-provisioning

We use over-provisioining/replication as the primary mechanism for fault tolerance when scheduling workflow tasks onto resources. In the most general case, each workflow task has performance constraints (expressed through performance models and deadlines) and reliability constraints (expressed through an user designated success probability). We will show the exact constraints we used in our experiment in the next section. Our goal is to find the smallest set of resources to replicate the given workflow task to satisfy these constraints.

Kandaswamy *et al.* [53] described an effective algorithm to find the smallest subset of resources that satisfies these constraints for an individual task. In the cases when it is not possible to satisfy both the success probability and deadline constraints, the task over-provisioning algorithm returns all possible resource combinations tagged with the success probabilities for each resource set solution, so that a best-effort replicated set of resources can be chosen.

The task over-provisioning algorithm that determines the set of resource the tasks should be replicated on uses (a) performance models for the estimation of computation time on a resource for the workflow task, (b) network latency, bandwidth and intermediate data sizes for the estimation of data transfer times and (c) reliability models (based on Weibull distribution) of resources for the estimation of resource failure probabilities. Let the application deadline be d, the required success probability be x and $[1 \ldots M]$ be the set of available resources. The algorithm defines h_{r_i} to be the expected completion time for a task on resource r_i, which is obtained by aggregating performance models and data-transfer time estimates. The algorithm also

defines m_{r_i} to be the probability that the task fails on resource r_i, which is obtained using the reliability model as described in section 6.3.1.

The problem is to find a subset, $P = \{r_1, r_2, \ldots r_m\}$ of $[1 \ldots M]$ such that the following holds true:

- $1 - m_{r_1} \times m_{r_2} \cdot \cdot \times m_{r_m} \geq x$
- $|P|$ is minimum
- $max(h_{r_1}, h_{r_2}, \cdot \cdot h_{r_m}) \leq d$

The algorithm finds the degree and resources for over-provisioning by carefully enumerating a selected portion of the subsets of [1..M] and returning the smallest subset of resources that satisfies all the conditions [53].

Integrating Fault Tolerance with HEFT/DSH

The over-provisioning we described above works well for an individual task but it is not designed for workflow applications. Figure 6.1 describes the algorithm that we use to integrate the task over-provisioning algorithm with HEFT. First, we sort the tasks in the DAG by upward ranks. Then, we assign each task to the resource that has the earliest finish time. After a task is assigned, we check its predecessor task. If all successor tasks of the predecessor have been assigned to a resource, we invoke the task over-provisioning algorithm(TOP) to find a set of resources on which the predecessor should be replicated so that its deadline and success probability constraints are satisfied. We only invoke TOP to duplicate a task after all its successor tasks have been assigned because it takes into consideration the communication time when it computes the degree of duplication. We set the deadline as 30% more than the task's finish time without duplication and the probability constraint as 0.95. We assign the predecessor task to the resources in the set. If there is no resource set that can statistically guarantee a success probability of 0.95, the over-provisioning algorithm returns a set of resource sets and the corresponding success probabilities.

```
HEFT-Dup(DAG dag, Resource res, PerfModel pM)
  Task[] tasks = dag.sortTask(res)
  for each task t in the tasks
    t.mapResource(res, pM)
    t.assigned = TRUE
    for each pTask in t's parent Task
      Vector dupRes;
      if ( pTask.allChildrenAssigned() == True )
        dupRes = TOP.getDupRes(pTask, res, pM)
        if (dupRes.isEmpty() )
          dupRes = TOP.getAllDup(pTask, res, pM)
          dupRes = dupRes.selectMostReliable()
        for each resource r in dupRes
          if ( pTask is not assigned to r already )
            assign pTask to r
            if (pTask replicated to LIMIT resources)
              break
        end
    end
  end
```

Figure 6.1 : HEFT with Over-provisioning

In this case, we pick the resource set with the highest reliability. We assign the predecessor to those resources in addition to the original one if it is not included in the replication resource set. We also limit the total number of resources on which one task can be replicated. In case the total number of resources is more than the

limit, we keep the resource that HEFT assigns this task to and select the subset of resources TOP returns that has the best reliabilities. We will discuss in section 6.4 how this limit affects the outcome.

Similarly, we integrate the DSH scheduling algorithm with over-provisioning to get the DSH with over-provisioning algorithm. The only difference is that we duplicate the predecessor for the performance purpose first and then invoke the TOP for better reliability.

6.2.3 Scheduling Algorithms with Checkpoint-recovery

To mitigate the effect of failures during execution time, we use checkpoint and resubmission of workflow steps. We chose to implement a light-weight checkpoint strategy that saves only the current location of the intermediate data as opposed to a heavy-weight checkpoint strategy that saves the data on a separate system [80]. We made this choice because the performance of the heavy-weight checkpoint mechanism relies heavily on the reliability and performance of the backup system. This could lead to a chicken and egg problem since the backup system's reliability also relies on over-provisioning. Hence, we decided to focus on light-weight checkpointing.

Since a workflow application consists of multiple tasks, it is natural to do a light-weight checkpoint when each task finishes. If a task fails to finish due to resource unavailability, we restart it on the most reliable resource that is available based on the reliability prediction. However, since we only implement light-weight checkpointing, it is possible that some of the resources on which the predecessor tasks were running are also not available. In this case, we restart those predecessors on new resources. However, this approach has a potential to cause infinite loops if a task fails repeatedly while its predecessors always finish after restart. Therefore, we put a limit on the number of times a task can restart. In our study, the limit is set to three, since our initial experiments showed that a higher limit did not provide better reliability and used more resources.

Since checkpoint and recovery happen during execution time, we can apply them to the over-provisioning version of HEFT and DSH directly. So, we have the following combined versions - (a) HEFT with over-provisioning and checkpoint-recovery and (b) DSH with over-provisioning and checkpoint-recovery. We will refer to the combined versions as over-provisioning with checkpoint version of HEFT and DSH.

6.2.4 Whole DAG Over-provisioning and Migration

The fault tolerance strategy described in section 6.2.2 is task based, which means that it only guarantees the statistical success probability of an individual task, not the entire DAG. For a workflow application with N tasks, with each task having a success probability of s_i, the success probability of the entire workflow is

$$SuccProb_{overall} = \prod_{i=1}^{N} s_i$$

The DAG success probability can be very low when N is large. For example, the success probability of a 100-task workflow application where each task has a 99% success probability is only 36.6%. Therefore, in addition to the task based fault tolerance strategy, we also propose a whole DAG over-provisioning (WDO) mechanism that replicates the whole DAG onto multiple resources.

Figure 6.2 describes the algorithm that we use for whole DAG over-provisioning. We first estimate the makespan of the entire DAG for each resource using the exclusive time estimation method developed in Section 5.2.1. We then compute the failure probability of the entire DAG according to each resource's reliability model. After we sort the resources by their failure rate in descending order, we apply a greedy approach that assigns a DAG to the resources with the highest reliabilities until the aggregated success probability is over 0.95. The aggregate success probability is

$$SuccProb_{overall} = 1 - \prod_{i=1}^{M} f_i$$

where f_i is the failure probability of resource i and M is the number of resources.

```
DAG-Dup(DAG dag, Cluster res, PerfModel pM)
   TreeSet<Entry<Cluster, Double>> relSort
   float time, failProb
   for each resource r in res
       t = dag.getEstimateTime(r, pM)
       failProb = r.reliabilityModel.getFailProb(t)
       reliabilitySort.put(r,failProb)
   end
   Sort resources in relSort by the reliability
   for each resource r in relSort
       calculate the aggregate success prob.
       if overall success prob ≤ 0.95
          assign dag to r
          if ( dag replicated to LIMIT resources)
             break
       else
         break
   end
```

Figure 6.2 : Whole DAG Over-provisioning

We can also combine the whole DAG over-provisioning algorithm with the checkpoint-recovery mechanism, which happens during the execution time. Therefore, we have 10 different scheduling and fault tolerance mechanism combinations in total. We will apply these in our workflow management system and analyze their reliability, performance and resources usages.

6.3 Experimental Methodology

To study how these fault tolerance and scheduling strategies perform in a multi-cluster grid environment, we implemented a prototype workflow management system that schedules and executes a workflow application on a simulated multi-cluster grid based on our previous work described in Chapter 3 and Chapter 4. We use this system to schedule and execute workflow applications with different fault tolerance and scheduling techniques. In this section, we will first discuss the resource reliability models we use. Then, we present our experimental design which includes (a) the chosen resource configurations, (b) the workflow applications, (c) performance models used and (d) the number of experiments.

6.3.1 Resource Reliability Model

Recent studies [91, 50, 98, 75] show that the mean time between failures (MTBF) on modern high performance clusters is best modeled by a Weibull distribution [117]. However, the shape and scale parameters are different for each study. Nurmi *et al.* [75] and Schroeder *et al.* [98] report that the shape parameter is less than 1, which means that the hazard rates (the frequency a system or component fails) decrease with time. In contrast, Iosup *et al.* [50] report that the shape parameter is greater than 1, which indicates an increasing hazard rate over time. Hence, we wanted to explore both regions for the shape parameter in our study and created two sets of reliability configurations - one set with shape parameter ranging between 0.5 and 0.9 according to Schroeder *et al.* [98] and the other set with shape parameter ranging between 10 and 13 according to Iosup *et al.* [50]. For a given range of shape parameter, we generated three reliability models based on three mean values of the scale parameter - three days (for shaky), one week (for normal) and three weeks (for stable). Note that the expected value (the MTBF in our case) of a random variable that follows a Weibull distribution with a shape parameter k and scale parameter λ is $\lambda\Gamma(1 + \frac{1}{k})$. So, with two ranges of shape parameters, we explore 6 different reliability models

in this study. The resource failures are randomly generated following the Weibull distributions in the reliability model. Figure 6.3 presents the six different Weibull parameters in our experimental setup.

WeiBull distribution		Shaky	Normal	Stable
[Iosup]	shape (k)	11.3~12.8	11.3~12.8	11.3~12.8
	scale (λ)	28~64 hour	84~192 hour	250~650 hour
[Nurmi] [Schroeder]	shape (k)	0.61~0.90	0.61~0.90	0.61~0.90
	scale (λ)	28~64 hour	84~192 hour	250~650 hour

Figure 6.3 : Weibull Parameters in Our Experiment

6.3.2 Experimental Setup

We use a multi-cluster simulated grid environment with nine clusters that have the same processor configuration as nine sites in the TeraGrid [107]. Correspondingly, there is a Weibull distribution for each cluster with a pair of shape and scale parameter. Although the scale and shape parameters are different for each cluster, they are within the range of one of the six models listed in Figure 6.3. For example, in a shaky model with large shape parameters, the nine clusters have their own distinguished shape parameters between 11.3 and 12.8 and scale parameters between 28 and 64 hours.

Similar to our approaches in Chapter 3 and Chapter 4, we generated three types of DAGs corresponding to three different parallel applications and algorithms - Montage, Fast Fourier Transform and Gaussian elimination that we described in Section 2.2.1. We also generated two types of DAGs that represent common parallel programming models - fork-join and level. For each type of DAG, we generated over 100 DAGs

with configurations differing in the total number of tasks, the average size of the task, and the computation to communication ratio. We use historical performance models generated from the performance data we collected in Chapter 5. The estimated running time of those DAGs on the clusters we use, based on the performance data and without duplication, range from a few hours to a month. The success probabilities of those DAGs range from almost zero to almost one based on the estimation we used in whole DAG over-provisioning (WDO) algorithm.

In total, we used 635 different DAGs and 6 different reliability models. Since the failures are randomly generated, we ran each DAG and reliability model combination 10 times. For each run, we used all 10 different scheduling and fault tolerance mechanism combinations so that each approach sees the same resource failures. In addition, we ran 3 batches of experiments allowing the number of times a task was duplicated or restarted from checkpoints to vary as we described in section 6.2. Therefore, we collected over 1 million different executions' results for each batch of experiments so we have 4 million data points in total. We will discuss the experiments and results in the following section.

6.4 Results

We present our experimental results for the algorithms we described in section 6.2. We have two basic scheduling algorithms, HEFT and DSH. We denote the over-provisioning versions of them with an "_O" at the end and the checkpoint-restart version with a "_C" at the end. An "_OC" in the end means both fault tolerance mechanisms are applied. In addition, we have the duplication based whole DAG over-provisioning (WDO) algorithm and its checkpoint-restart version "W_C". Figure 6.4 shows the overall percentage of workflow applications that successfully finished after using one of the ten scheduling and fault tolerance technique combinations. From the graph, we observe that the over-provisioning mechanism can increase the DAG success probability by around 25% while light-weight checkpoint-restart can increase

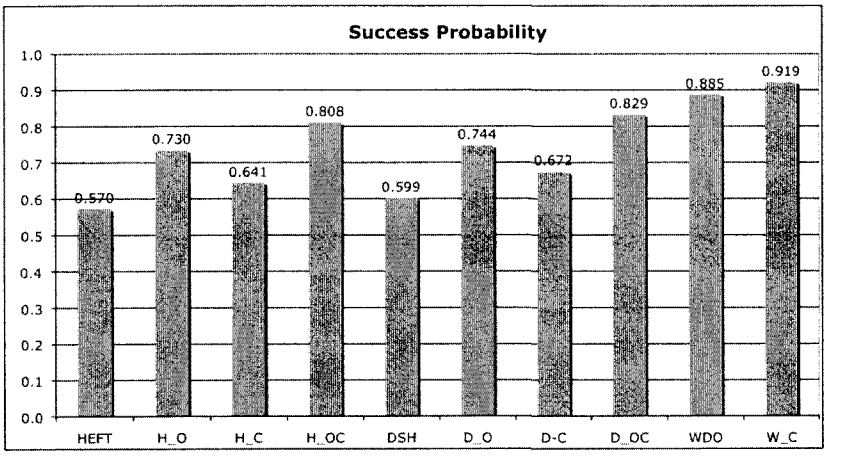


Figure 6.4 : Overall Success Probability

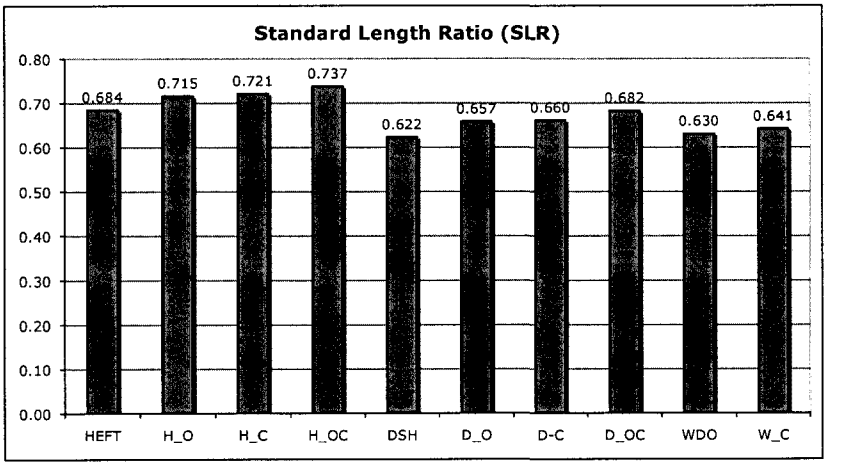


Figure 6.5 : Overall Standard Length Ratio (SLR)

the success probability by around 12%.

Figure 6.5 shows the workflow performance for each approach. We compare the Schedule Length Ratios (SLRs), described in Chapter 2, of the methods rather than raw makespan to ensure a level playing field. In general, both high-reliability methods and low-reliability methods will complete small DAGs, which can run before

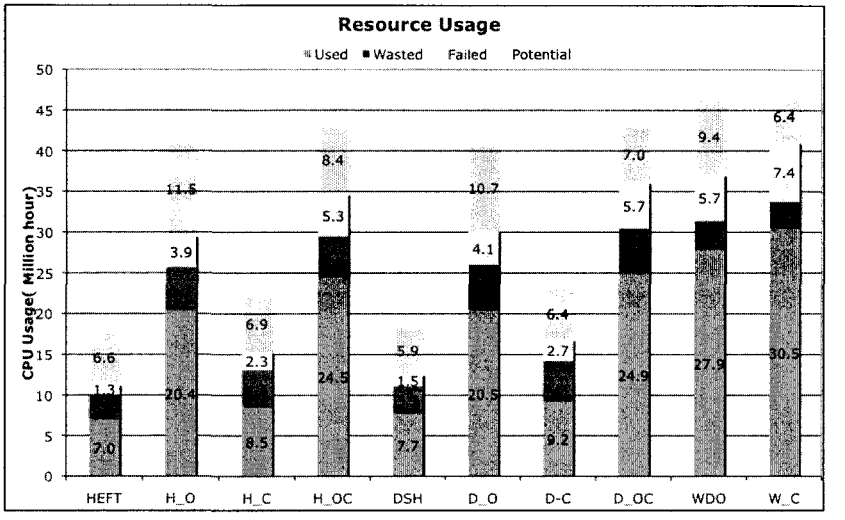


Figure 6.6 : Overall Cpu Time Usage

the MTBF becomes a significant factor. However, more reliable methods complete longer-running DAGs while other methods do not. Therefore, averaging run times of completed executions would clearly favor less-reliable methods, without giving insight into any overhead of either method. Normalizing the results using SLR avoids this bias. Lower SLR indicates better performance. We can see that over-provisioning only increases the SLR by at most 5% while checkpoint-restart increases SLR by at most 6%.

The relatively small performance penalty is because the makespan for the schedule with over-provisioning is the same as that of the original schedule unless the original resource assigned by the scheduler is down. In that case, the penalty is just the completion time difference between the next fastest resource and the fastest one. Also, we see that the duplication based scheduling (DSH) has a 10% advantage over HEFT and produces the best schedules (i.e. lowest SLRs) among all algorithms. Thus, some over-provisioning decision could also make the DAG run faster in the same fashion. Whole DAG over-provisioning also performs better than HEFT. This is because WDO almost eliminates all the communication time unless there are task

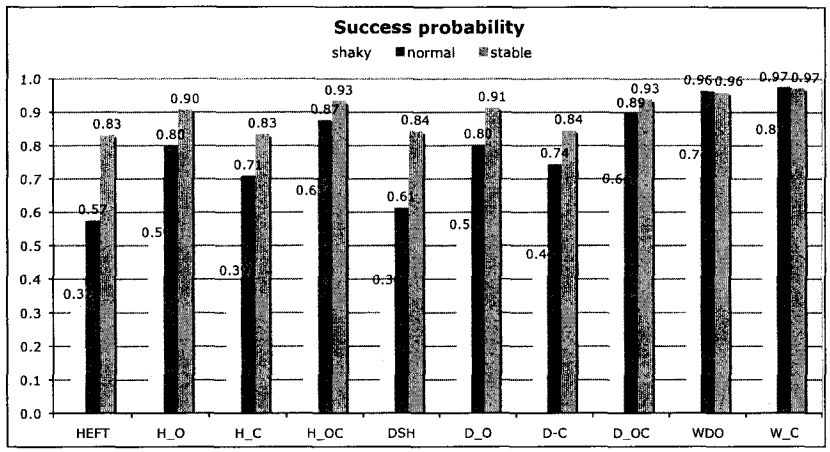


Figure 6.7 : Success Probability with Different Reliability Models

failures since each task's parent is already assigned to the same resource.

Figure 6.6 shows the resource usage for each approach in terms of total CPU hours used. However, since each approach finishes a different number of DAGs, we divided the total usage into three parts in order to further analyze the result. "Used" resource time denotes the total CPU hours consumed by the completed tasks in the DAGs that successfully finished. The "wasted" resource time is the total CPU hours consumed by the completed tasks in the DAGs that failed to finish. The "failed" resource time is the total CPU hours consumed by the failed tasks no matter whether the DAG finished or not. The solid stacked bar in figure 6.6 thus shows the aggregated CPU hours that all workflows used including all three usage types. We can see that over-provisioning uses around 2.5 times more resources than HEFT while checkpoint restart uses 1.5 times more resources than HEFT (and similarly for DSH). Besides that, we also calculated the total CPU hour that failed DAGs would need to complete successfully, if there were no more resource failures. We call this the "potential" resource usage and plot it as a transparent bar on top of the solid bar. We can see that since HEFT and DSH have a lower completion rate, the "potential" resource usage is higher relative to the amount used. Whole DAG over-provisioning (WDO) uses 20% more resources

than HEFT with over-provisioning but since it completes more DAGs, it would use just 10% more resources than HEFT with over-provisioning (or about 5% more than DSH with over-provisioning) taking into account the "potential" resources to finish all the DAGs.

Figure 6.7 illustrates how resource reliability affects the overall success probability of workflow applications with our approaches. Scale parameters affect the DAGs' success probability more than the shape parameters, we categorize our reliability models into three groups corresponding to the reliability characteristics of the resource. The results show that, using HEFT only, the average success probability of workflows is 32% when they are executed on the most unreliable resources, referred to as the shaky resources. The average success probability of workflows using HEFT only is 83% when they are executed on the most reliable resources, referred to as the stable resources (similarly for DSH). We refer to the third group as the normal resources, where reliability is somewhere between the two. We can see that the more fault tolerant techniques we use, the less is the dependence of the workflow application's success probability on the underlying resource reliability. The success probability of HEFT on the stable resources is over 150% more than that on the shaky resources. The algorithms with over-provisioning alone has a better success probability on stable than on shaky by about 80%. Meanwhile, the whole DAG over-provisioning with checkpoint-restart has only about 20% difference in success rate. Also, the less reliable the resource is, the more impact the fault tolerance techniques have on the success probability. The scheduling algorithms with over-provisioning improves the reliability average by over 50% than their base algorithms on shaky resources, while only by 10% on stable resources. Similarly, the scheduling algorithms with checkpoint-recovery improves their performance over the base algorithms by 17% on shaky resources but have almost the same success probabilities on stable resources. Finally, we notice that the whole DAG over-provisioning (WDO) and whole DAG over-provisioning with checkpoint-recovery (W_C) provides the best success probabil-

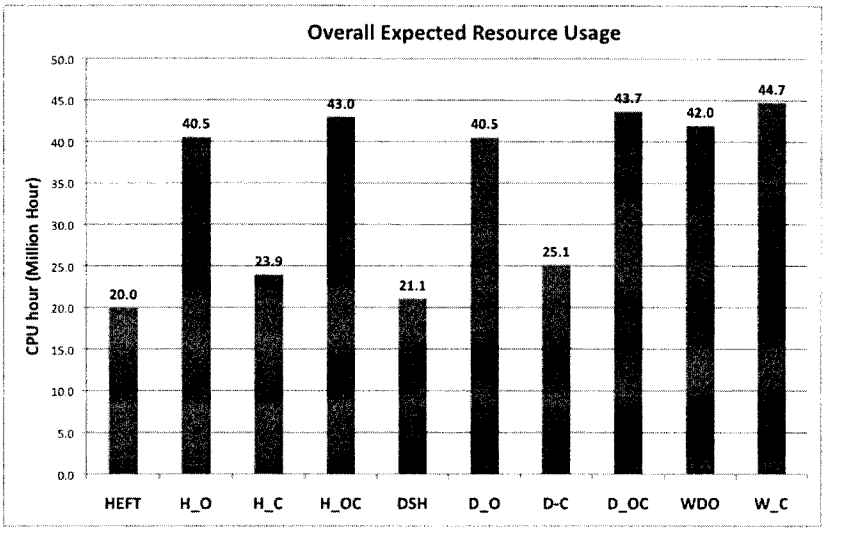


Figure 6.8 : Expected Resource Usage

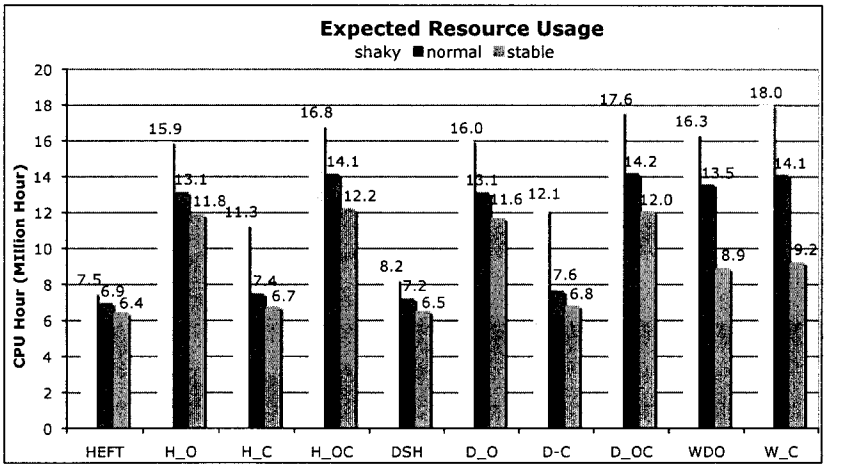


Figure 6.9 : Expected Resource Usage with Different Reliability Models

ity on all resource types.

Figures 6.8 and 6.9 show the expected resource usage for each approach overall and under different reliability models. We use the expected value instead of the actual value because algorithms using no or fewer fault tolerance techniques complete

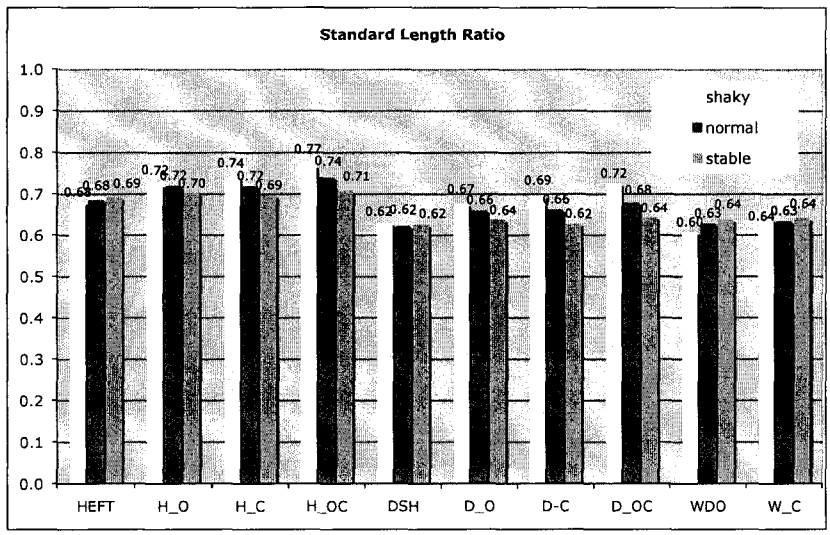


Figure 6.10 : Performance with Different Reliability Models

a smaller number of DAGs. Figure 6.8 illustrates the idea. To better compare their total resource usage, we incorporate the success rate of the approach to normalize the resource usage. We view the repeated run of a DAG as a Bernoulli process in a sense that it resembles a scientist tries to complete her DAGs on unreliable resources using any of our schedulers. If one run fails, she simply retries it until it succeeds. This is a Bernoulli process if the trials are independent. We know that the expected number of trials before one sees a success is $1/p$ where p is the success probability for an individual trial. We then calculate the expected resource usage for algorithm *algo* as the $cpu\_hour_{algo} \times 1/p$. That is the expected resource one approach uses to get a successful execution. From Figure 6.8 we can see that the over-provisioning versions of the scheduling algorithms use about twice the expected resources of their base algorithms while the checkpoint-recovery versions use around 20% more. Figure 6.9 shows that the over-provisioning versions of the algorithms have about 100% more expected resource usage than the vanilla scheduling algorithm on shaky resources, while they use 50% more on stable resources. We notice that WDO uses less to-

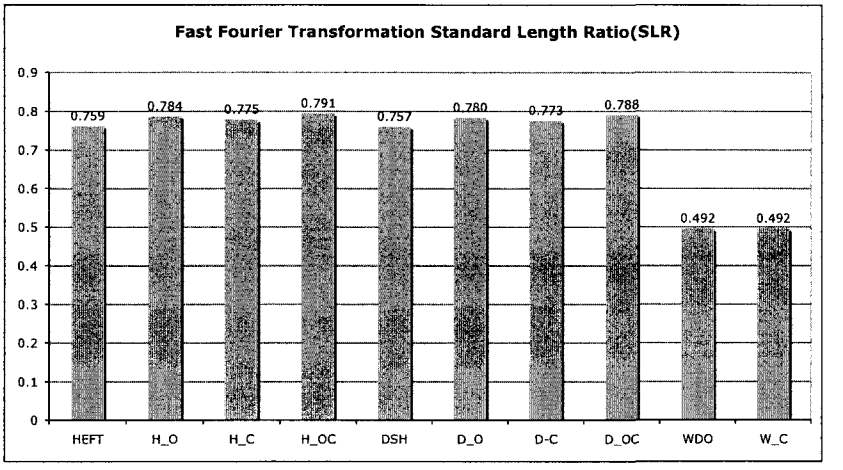


Figure 6.11 : Fast Fourier Transform Performance

tal expected resources than all over-provisioning and checkpoint-recovery combined approaches while providing higher reliability. We also measured the performance in terms of SLR under different reliability models, shown in Figure 6.10. The reliability does not affect the performance much. Each approach's SLR differences on different reliability models are within 10% and the difference between different approaches is similar to the data in figure 6.5.

We grouped data for DAGs that represent different types of applications. Most of the data representing a single type of application are similar to the overall graphs. However, certain applications showed some distinct features . For example, Figure 6.11 shows the Fast Fourier Transform(FFT) application's performance. The WDO technique has an almost 50% improvement over the other methods. We believe that is because it eliminates all the potential communication costs that occur in the expensive message exchange phase of FFT while DSH would not duplicate the whole DAG because it would delay the start time of those tasks.

Since failure probability prediction is hard, we tested the robustness of our fault tolerance approaches with respect to failure prediction accuracies. Figure 6.12 shows

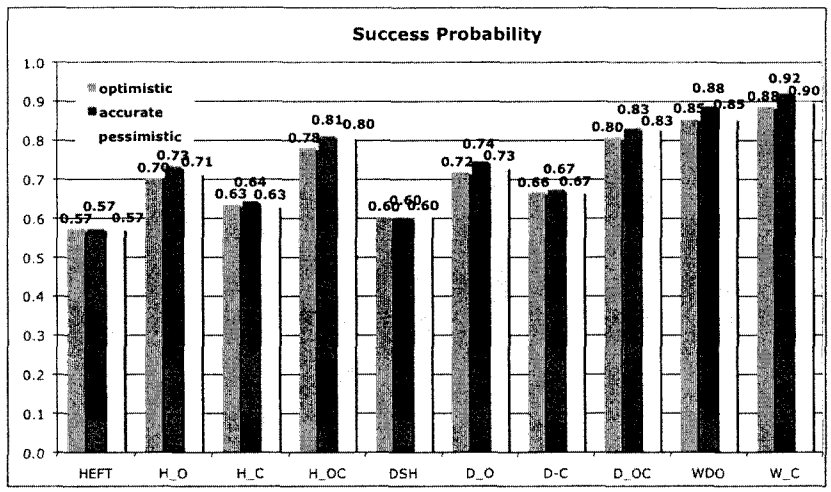


Figure 6.12 : Success Probability with Different Failure Prediction Accuracies

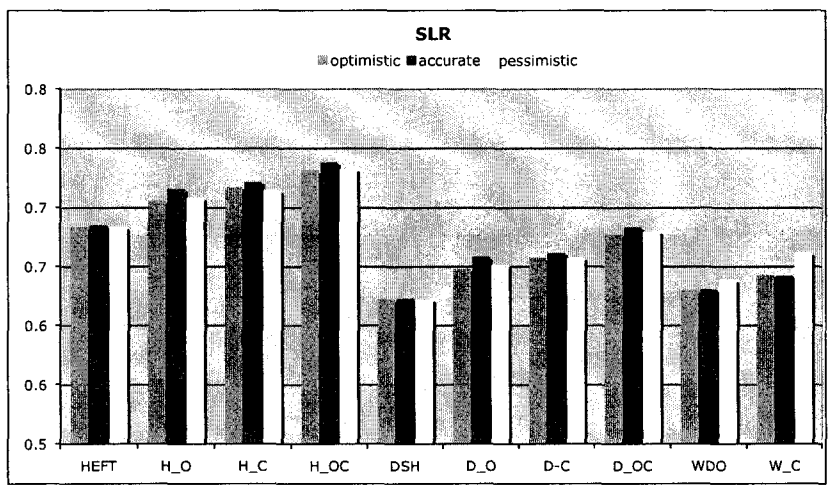


Figure 6.13 : Performance with Different Failure Prediction Accuracies

the workflow application execution success probabilities with different failure prediction accuracies. The accurate prediction is the failure probability that we get from the Weibull distribution's cumulative distribution function. For the optimistic prediction, we multiply the accurate failure probability by a random number evenly distributed between 0 and 1. Therefore, the expected failure probability is half of the accurate

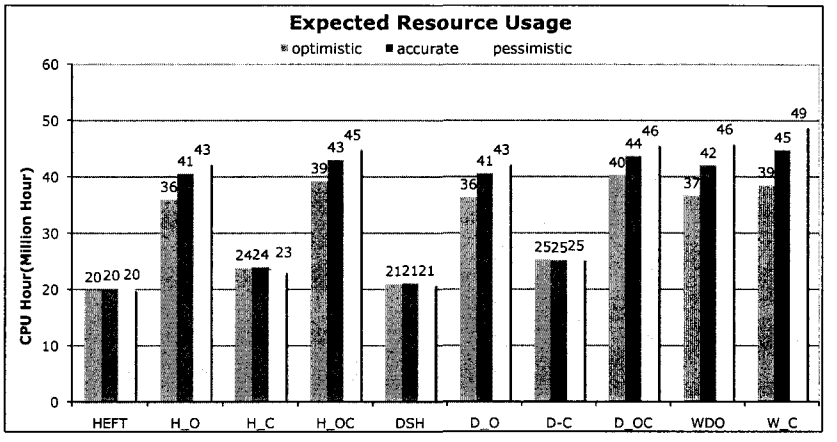


Figure 6.14 : Resource Usage with Different Failure Prediction Accuracies

one. For the pessimistic prediction, we divide the accurate failure probability by a random number evenly distributed between 0 and 1 thus make it higher and we cap the pessimistic prediction under 1. From Figure 6.12 we can see that although the accurate prediction always leads to best success probabilities, all the fault tolerance mechanisms are pretty robust under inaccurate failure predictions. Figure 6.13 shows that the approaches' performance is minimally affected by the failure probability prediction.

Figure 6.14 shows the expected resource usages under accurate, optimistic and pessimistic failure probability predictions, analogous to Figure 6.8. We see that the pessimistic prediction can lead to increases of as much as 20% in resource usages over optimistic prediction. It is because the over-provisioning algorithms over-replicate applications onto more resources than necessary under pessimistic prediction. Since optimistic prediction does not lead to significantly lower success probability, but does cause less duplication, its expected resource usage is even better than with accurate prediction. This suggests that it is better to err on the side of optimism in estimating failure probabilities, at least from the resource usage point of view.

We believe that one reason why our over-provisioning mechanisms does not use

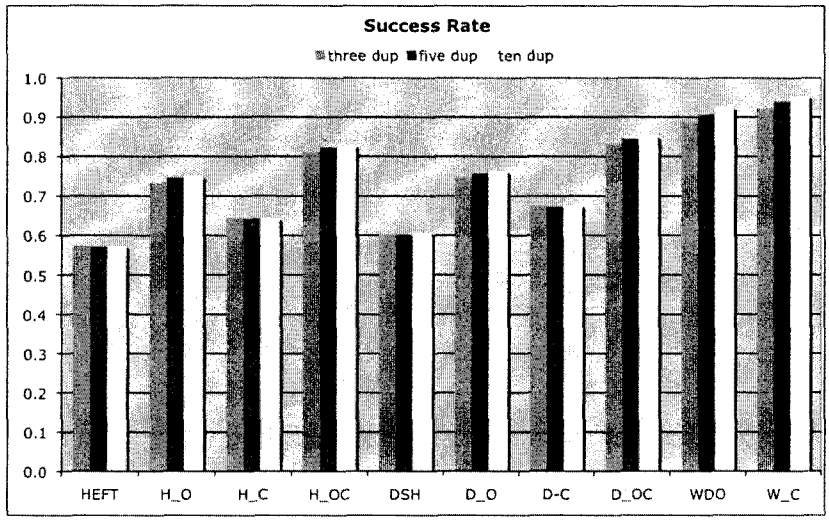


Figure 6.15 : Success Probability with Different Replication Limits

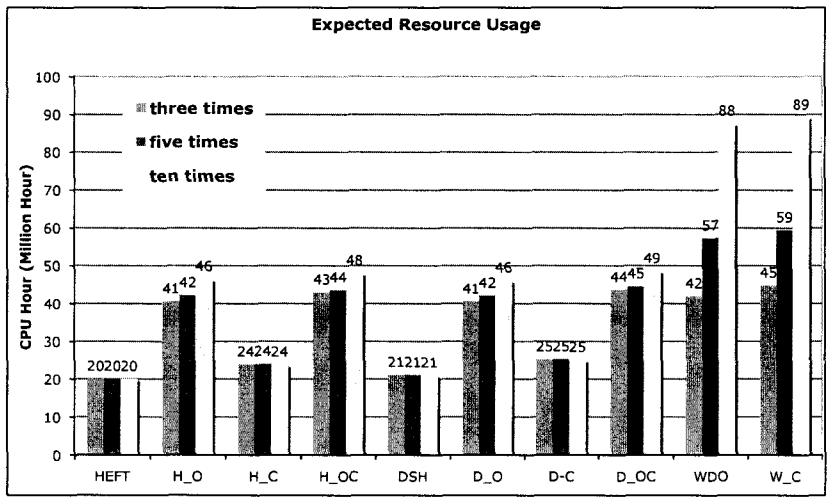


Figure 6.16 : Resource Usage with Different Replication Limits

too many resources under pessimistic failure predictions is that we have a limit set for how many resources each task can be over-provisioned. The default value is set to 3. Thus, even if the failure prediction is too pessimistic, we will not over-provision too much. To show the effect of the resource limit, we also used 5 and 10 as the

limit. Figure 6.15 shows that it does not affect the application success probability does not affect more than 5%. The performance differences between different levels of replication are also within 5% of each other. However, Figure 6.16 shows that it does affect the expected resource usage. We can see that the replication limit 10 batch uses almost 100% more resources than the replication limit 3 batch while providing only less than 3% more success probability.

Overall, our experiments evaluate the three way trade-off between reliability, performance and resources usage in a large-scale simulated environment. We believe this can be a useful reference for future workflow application developers to balance between these three aspects.

6.5 Related Work

Workflow application scheduling on grids [69, 122, 127] is an active area of research. Workflow scheduling has largely focused on heuristic techniques using performance models to qualitatively select resources and map tasks to the resources that have good performance [57]. Few scheduling algorithms take into account reliability of the grid resources.

One of the most widely implemented fault-tolerance techniques on computational Grid is simple retry [80] which means the application is resubmitted on a resource in case of a failure. In many workflow management frameworks [72, 46], the remaining portion of the workflow is resubmitted in case of a failure.

Hwang *et al.* [49] present a failure detection service (based on notifications) and a flexible framework for handling Grid failures. Therefore, we assume that a failure could be detected relatively shortly after it occurs in our work. Budati*et al.* [15] present a reliability-aware system which uses a resource's prior performance and behavior to get better performance and reliability on large-scale donation-based distributed infrastructures. However, their system mainly targets P2P solving systems. Limaye *et al.* [63] have developed a checkpoint/restart mechanism that places check-

points based on system reliability. We could incorporate their work to increase our checkpoint-recovery's statistic success rate. Dongarra *et al.* [32] use the product of failure rate and unitary instruction execution time to guide the scheduling of independent tasks onto heterogeneous clusters. Their work could also be used to increase the base line algorithm's reliability in our system.

6.6 Conclusions

This chapter presents workflow scheduling and execution mechanisms that incorporate a balanced approach toward reliability and performance which is not very sensitive to the underlying resource reliability prediction. It also presents a new algorithm that replicates the whole DAG (WDO) onto several clusters which provides the best reliability. From the experiments, we observe that the fault tolerance techniques are effective. They can increase the reliability of workflow executions by as much as 200% and do not affect performance by more than 10%. Also, we presented a quantitative model for the three-way trade offs between the reliability, performance and resource usage. We believe it could be valuable to system architectures who want to design a fault tolerance / high availability system.

Chapter 7

Batch Queue Resource Scheduling for Workflow Applications

In this chapter, we present our work on reducing the resource provisioning overhead for the workflow application running on batch queue controlled resources. Our approach groups a workflow application into several aggregations and uses the batch queue to acquire resources for each aggregation, overlapping queue wait time of one with the execution of others. We implemented a prototype of this technique and the experimental results show that our approach can eliminate as much as 70% of the wait time over more traditional techniques that request resources for individual workflow tasks or that acquire all the resources for the whole workflow at once.

7.1 Introduction

In this Chapter, we will focus on a special but very important case in the second step of our decoupled scheduling approach that schedules a workflow onto an individual cluster to achieve the best turn-around-time. Clusters (parallel computers with high-speed interconnects and shared file systems) have become the most common high-performance computing platform. With the emergence of super clusters that often have more than 100k cores in one cluster [18], a single cluster now can usually provide enough resources for a workflow application to run in maximum parallelism. Therefore, it is more and more likely that the resource selector in the decoupled scheduling approach we used in Chapters 3, 4 and 5 will select a single cluster as a TightBag that provides both the high computation power and tight connections. The whole DAG over-provision (WDO) mechanism we proposed in Chapter 6 also dupli-

cates an entire workflow application onto a single cluster. We therefore consider this special case (i.e. executing a workflow on a single cluster), and attempt to minimize the turnaround time in that environment.

Workflow execution systems can get access to a cluster either locally, through collaborative Grid organizations such as TeraGrid [107], or through national supercomputing centers like TACC [108]. In any case, these clusters are shared and usually managed by a local resource management system that has its own resource sharing methodology and policy. Among them, commercial or open source batch queue scheduling software [79, 25, 65] is the most popular resource management system. Section 7.2 gives more details on the background of both workflow applications and batch schedulers.

The main goals of a site using batch queues are usually to achieve high throughput and maximize the system utilization. Consequently, many production resources have long queue wait times due to the high utilization levels. In addition, although it is not unusual for a single cluster to have several thousand processors, a single user usually can only obtain a small portion of the total available resources (without special arrangements). This creates performance problems for large scale workflow applications because each sub-task in the workflow could experience long delays in the job queue before it runs. The queue wait time overhead is sometimes much more than the workflow applications runtime [101]. Alternately, one could submit an entire workflow as a single batch queue job. However, this might cause an even longer wait for a larger resource pool to become available at once.

Our work seeks to reduce workflow turnaround time by intelligently using batch queues. We accomplish this by aggregating workflow tasks together and submitting them as a single job into the queue. Section 7.3 describes our method in greater detail. This approach can greatly reduce the number of jobs a workflow execution system submits to the batch queue. By overlapping some tasks' wait times with others executions, we further shorten the batch queue wait times for the workflow

applications. As we will see in Section 7.4, our scheduling reduces the queue wait time overhead without requiring special system privileges and using only user-level mechanisms. We conclude our presentation with a discussion of related work in Section 7.5 and our conclusions and future work in Section 7.6.

7.2 Background

7.2.1 Batch Queues

Batch queues have become the most popular resource management method on computational clusters. A batch queue system is normally a combination of a parallel-aware resource management system (which determines "where" a job runs) and a policy based job scheduling engine (which determines "when" a job runs). We are mostly interested in the job scheduler component, treating the individual processors as homogeneous. To illustrate how this scheduler works, we describe the widely-used open-source Maui batch queue scheduler [51, 12]. The experiments in Section 7.4 are based on simulations of this scheduler.

The Maui scheduler, like many batch queue schedulers, is essentially a policy based reservation system. The key idea is to calculate a priority for each job in the queue based on aspects of the job and the policy of the queue system. The priority of each batch queue job is determined by job properties, such as the requested resource requirements (number of processors and total time), its owner's credentials, and the time it has waited in the queue. These properties are combined in a formula with weights configured by the system administrator. For example, to favor large jobs, a site would choose a high (and positive) weight for the resource requirements.

When a batch queue event happens, i.e a job finishes, a new job is submitted, etc, the Maui scheduler calculates all jobs' priorities and starts all the highest-priority jobs that it can run immediately. It then makes a reservation in the future for the next highest priority job according to the already running jobs' requested finish time to ensure it will start to run as soon as possible. Given that reservation, a backfill

mechanism attempts to find jobs that can start immediately *and* finish before the reservation time. Once a job begins execution, it runs to completion or until it exhausts its requested resources.

Maui, like some other schedulers [73, 79, 25], can provide advance reservation services at a user level. This allows the user to request a specific number of resources for a given period of time, effectively gaining a set of dedicated resources and eliminating the queue wait time. However, advance reservation is not available at all sites, usually involves system administrator assistance, and always requires notice beforehand. Furthermore, Snell et al. [102] showed that advance reservation can decrease the system utilization and has the potential to introduce deadlocks. We therefore assume advance reservation is not available in this chapter.

One advanced feature of Maui that we do use is the start time estimation functionality. A user can invoke the *showstart* command to get the estimated start time of a job in the queue or a new job (specified with number of processors and duration) to be submitted. This can be done by computing the job's priority, building (or querying) the queue's future schedule, and determining when the job would run. Note that, because new high-priority jobs could be submitted before the queried job runs, the estimate may not be exact. However, it is a useful piece of information to use in scheduling.

7.2.2 Workflow Application Execution

Executing a workflow is conceptually simple. Whenever a task is ready to execute (i.e. all its predecessors have completed), it can be scheduled for execution. However, doing this naively in a batch queue environment could potentially create long waits for every task to begin. Nevertheless, this is common practice. There are two general ways [52] (other than advanced reservations) to reduce this batch queue overhead. One way is to aggregate the workflow tasks into larger groups [101]. This would reduce the total number of job submissions needed to complete the workflow thus may leads to

less overall batch queue wait time. However, larger jobs may need to wait for longer in the batch queue for all the resources it requested to be freed. Therefore, it is essential to have an intelligent way to aggregate the workflow so that the total batch queue wait time can be reduced. The other method is to use virtual reservation technology [56, 78, 42, 116]. This provisioning technique enables users to create a personal dedicated resource pool in a space-shared computing environment. Although there are various implementations, the key idea is to submit a big placeholder job into the space shared resource site. When the placeholder job gets to run, it usually installs and runs a user-level resource manager on its assigned computing nodes. The user-level resource manager (in our case, the workflow execution system) then can schedule jobs onto the those computing nodes without going through the site's resource manager again. Our work draws inspiration from the virtual reservation implementation, but attempts to choose a more propitious size for the placeholder job.

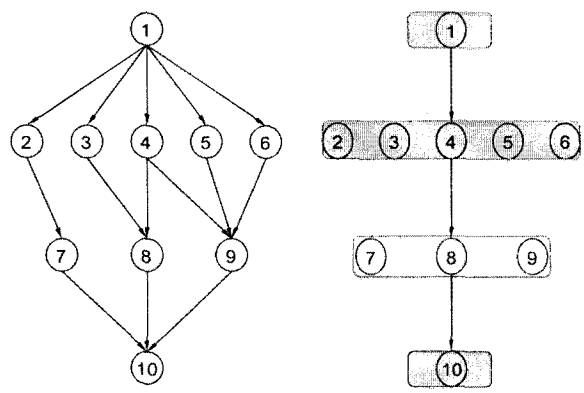


Figure 7.1 : Workflow Application Aggregation

7.3 Workflow Application Aggregating

Our workflow aggregating technique groups the workflow tasks into larger units. Figure 7.1 shows an example. The left side of the figure is the original DAG that represents a workflow application. The right side of the figure is an aggregated version of the same DAG in which we group all the tasks in the same level into one aggregation. Our goal is to choose an aggregation that will reduce the total batch queue wait time. The main idea behind our approach is that we can aggregate the workflow by level and submit a placeholder job for the later levels before their predecessors finish. In this way, we can overlap the running time of the predecessor level with the wait time of the successor levels.

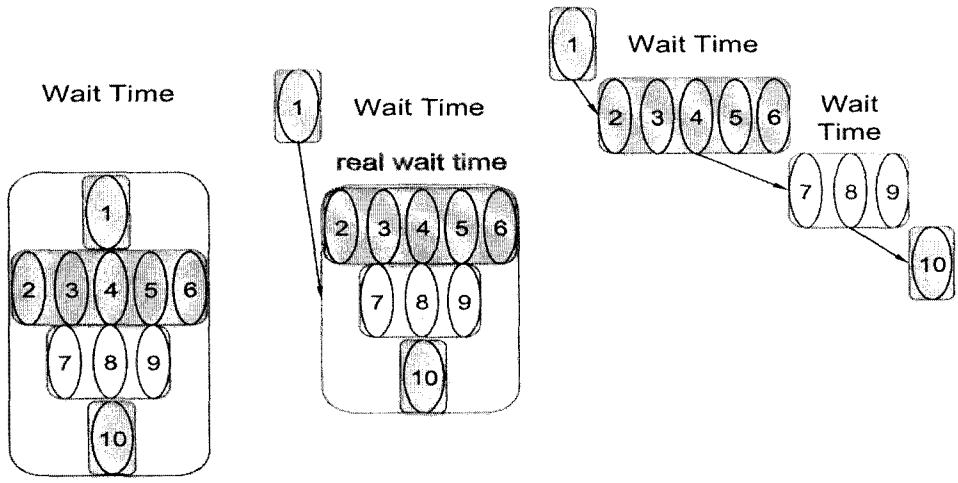


Figure 7.2 : Workflow Application Cluster by Level

Figure 7.2 illustrates this idea. Placeholder jobs are represented by rectangles that contain one or more levels of tasks. The yellow rectangles represent the wait time of the placeholder job in the queue. The left portion of the figure shows a grouping of the workflow DAG in Figure 7.1 into a single placeholder job. We can see that it may need a long wait time before it can start. The middle of Figure 7.2 shows a grouping of the same workflow DAG into two aggregations and submitting them

in turn. A placeholder job is submitted into the queue as soon as its predecessor placeholder job starts. It asks for enough resources for the tasks it holds to run in full parallelism. The wait time seen by the users for the later task is the dark yellow area marked "real wait time". We can see that it is less than the queue wait time for the second aggregation because of the overlap with task 1's execution. Ideally, if the first placeholder job gets to run immediately and the later jobs' wait times do not exceed their predecessor's run times, the queue wait time for the entire workflow application is eliminated, as shown on right side of Figure 7.2. However, this perfect overlap cannot be guaranteed. Furthermore, if the wait time for a placeholder job is less than its predecessor's run time, it must pad its requested time to honor its dependences. In turn, this will affect the wait time of the placeholder job. Balancing these effects requires heuristic scheduling.

Our algorithm consists of two interrelated parts: an application manager shown in Figure 7.3, and a "peeling" procedure shown in Figure 7.4. The application manager is responsible for launching the workflow application and monitoring its progress. In general, it chooses partial DAGs and submits placeholder jobs to the batch queue system. Individual workflow tasks execute in the placeholder jobs when those jobs come to the front of the queue, with the application manager enforcing their dependences. The peeling procedure selects the partial DAGs to minimize the exposed waiting time. We now consider the parts in turn.

Figure 7.3 shows the application manager. After selecting and submitting the initial partial DAG (lines 1-5), the manager becomes an event-driven system. The primary events that it responds to are:

- A placeholder job starts to run (lines 8-16). The manger first starts all the workflow tasks associated with the job whose predecessor tasks have finished. Then it invokes the peeling procedure to form the next placeholder job and submit it to the queue.

- A placeholder job finishes running (lines 17-25). Normally, no processing is

needed. However, if the placeholder is terminated before all its tasks complete (e.g. because some predecessors were delayed in the batch queue), the manager must clean up. It cancels any placeholders that have not started, since some of their predecessors may be delayed. It also calls the peeling procedure to reschedule the unfinished DAG tasks (both interrupted tasks and those not yet run) and submits the new placeholder job into the queue.

- A DAG task finishes (lines 26-32). The manger starts all the successor tasks whose placeholder job is already running. One subtlety in the application manager is that the successors of a DAG task may be in the same placeholder or

```
Algorithm:runDAG (DAG dag, int sub_time)
1 task[] partial_dag ← levelize(dag);
2 int count ← 0;
3 Placeholder job ← peelLevel(partial_dag, sub_time, 0);
4 job. name ← count;
5 submit job;
6 while ( dag is not finished)
7 listen to batch queue and task events;
8     if (placeholder job_n starts to run at time t)
9         for all (task in job_n.getTasks())
10            if (all task predecessors have finished)
11                start task;
12        ear_finTime ← job_n.runTime;
13        partial_dag ← levelize(dag.unmappedTasks());
14        job ← peelLevel(partial_dag, t, ear_finTime );
15        job. name ← ++count;
16        submit job;
```

```
Algorithm:runDAG (DAG dag, int sub_time)
17      else if ( placeholder job_n finishes running at time t)
18              if ( job_n has unfinished tasks)
19                  partial_dag ← levelize(job_n.unfinishedTasks());
20                  for all ( pending placeholder job job_m )
21                      cancel job_m;
22                      add job_m.tasks() to partial_dag ;
23                  Placeholder jobResub ← peelLevel(partial_dag, t, 0);
24                  map all tasks in the partial_dag to jobResub;
25                  submit jobResub;
26      else if ( task dagTask finishes running at time t)
27              delete the dagTask from its placeholder job
28              for all (dagTask's successor task chd_task)
29                  if (chd_task's associated placeholder job is running)
30                      start chd_task;
31              if (dagTask's placehold Job has no more tasks to run)
32                  stop dagTask's placehold Job
```

Figure 7.3 : The DAG Application Manager

a different one. In the latter case, the manager must handle the possibility that a placeholder starts without any runnable tasks (lines 28-30). If all of a placeholder's tasks are finished, the manger finishes the job to free the batch queue resource.

We choose to submit a new placeholder job only after its predecessor begins running. There are several reasons for this design. In our experience with real queues, we discovered that multiple outstanding jobs in the queue interfered with each other. In turn, this often caused the wait time for already-submitted jobs to lengthen, which

```
Algorithm: peelLevel(levelized DAG, int sub_time, int ear_time)
int runTime_all, waitTime_all;
int peel_runTime[2], peel_waitTIme[2];
runTime_all ← est_runTime(DAG);
waitTime_all ← est_waitTime(runTime_all, DAG.width,sub_time);
peel_runTime[0] ← runTime_all;
peel_waitTIme[0] ← waitTime_all;
int level = groupLevel(DAG,sub_time, ear_time,
                        peel_runTime, peel_waitTIme);
if ( level == DAG.height)
   if (runTime_all * 2 < waitTime_all)
      return the whole remaining DAG in a batch queue job
   else
      return submit the remaining DAG in individual mode
else
   group levels to a partial_dag;
   map each dag job to the batch queue job;
   return the partial_dag in a placeholder job;
```

Figure 7.4 : The DAG Peeling Procedure

both added overhead and invalidated our existing schedules. In addition, it is possible that the execution order of the two placeholder jobs gets reversed which leads to even greater schedule overhead. Therefore, we did not have a good estimate of the later placeholder's start time. Although our current design misses the potential of overlapping two placeholder jobs wait times with each other or with running jobs, we can calculate the earliest start time of all the remaining tasks. This is one key to the aggregate decision procedure described in Figure 7.5.

Figures 7.4 shows the peeling procedure used by the application manager. We refer to this process as "peeling" because it successively peels levels of the DAG off of the unfinished work list. First (lines 1-6), the main peelLevel function estimates the wait time to submit the entire DAG as a single placeholder job. It then invokes the groupLevel function (lines 7-8 and Figure 7.5) to search for a better alternative. If groupLevel does not improve the wait time (lines 10-13), the peeling procedure chooses to submit the DAG either as a single placeholder job or as one job per task. The decision depends on whether the total wait time as a single job is twice the total run time of the DAG. The intuition for this is that individual submission can take advantage of the free resources or the backfill window. When the one giant placeholder job's wait time is twice as long as the run time, the individual submission has a better chance to finish earlier. This is a heuristic parameter chosen empirically. Otherwise, we use the partial DAG returned by groupLevel. The expected job start estimation we used is a best effort approach like the *showstart* command in Maui. However, our experience shows it is a reliable indicator of the wait time with one experiment showing the average estimated wait time is within 5% of the average actual wait time although the variance is high.

Figure 7.5 shows the key groupLevel procedure. Although the logic is somewhat complex, in essence we perform a greedy search for an aggregation of DAG that has enough granularity to hide later wait times and is wait-effective. We define the wait effectiveness of a job as the ratio between its wait time and its running time; a smaller ratio is better. The intuition behind this is that we want a job to either wait less or finish more tasks. However, we do not search for the globally best wait-effectiveness. This is because, once we group several layers of the DAG into a wait-effective aggregation, any later jobs' wait time can be overlapped with run time of this aggregation. Continually adding levels onto the current aggregation forfeits this benefit for the following levels.

Here is some more detailed explanation of our algorithm. After some initialization

in lines 1-6, the main loop in lines 8-37 repeatedly moves one DAG level from the remaining work to the next placeholder job until the aggregation is less wait-effective than the previous round. For each candidate job, lines 9-18 adjust the placeholder's

```
Algorithm: groupLevel (levelized DAG,int sub_time, int ear_sTime,
                                    int peel_runTime[2], int peel_waitTIme[2] )
1  int real_runTime[2];
2  int runTime_all, waitTime_all,leeway;
3  runTime_all ← peel_runTime[0];
4  waitTime_all ← peel_waitTIme[0];
5  real_runTime[0] ← peel_runTime[0];
6  partial_dag ← level one of DAG;
7  boolean giant ← true;
8  while partial_dag ! = DAG
9      peel_runTime[1] ← est_runTime(partial_dag)
10     real_runTime[1] ← peel_runTime[1];
11     do
12         peel_runTime[1] ← peel_runTime[1]+ leeway/2;
13         peel_waitTime[1] ←
14                             est_waitTime(peel_runTime[1], DAG.width,sub_time);
15         leeway← ear _sTime + real_runTime[1] - peel_waitTIme[1];
16     while leeway > 10 mins
17     if (leeway > 0)
18         peel_runTime[1] ← peel_runTime[1] + leeway;
19     int real_WaitTime ← peel_waitTIme[1] - ear_sTime;
20     if ( real_WaitTime < 0)
21         real_WaitTime ← peel_waitTime[1];
```

```
Algorithm: groupLevel (levelized DAG, int sub_time, int ear_sTime,
22      if (giant)
23         if (real_WaitTime > real_runTime[1])
24            add one level to partial_dag;
25            continue
26      giant ← false;
27      if (peel_waitTime[1] - ear_sTime > 0 )
28         if ( peel_waitTime[1] / real_runTime[1]
29                        > peel_waitTime[0] / real_runTime[0] )
30            break;
31         if ( peel_waitTime[1] / real_runTime[1]
32                        > waitTime_all /runTime_all )
33            break;
34      peel_waitTime[0] ← peel_waitTime[1]
35      peel_runTime[0] ← peel_runTime[1]
36      real_runTime[0] ← real_runTime[1]
37      add one level to partial_dag;
38 if (giant)
39    return DAG.height;
40 else
41    return partial_dag.height-1;
```

Figure 7.5 : The Peel Level decision Procedure

requested time to allow the workflow tasks to complete. As the left side of Figure 7.6 shows, this is sometimes necessary because the (estimated) queue wait time is less than the time to complete the current job, creating what we term the leeway. A simple iteration adds the leeway to the job request until it is insignificant. (Of course, if the wait time is more than the time to execute predecessors, then no adjustment

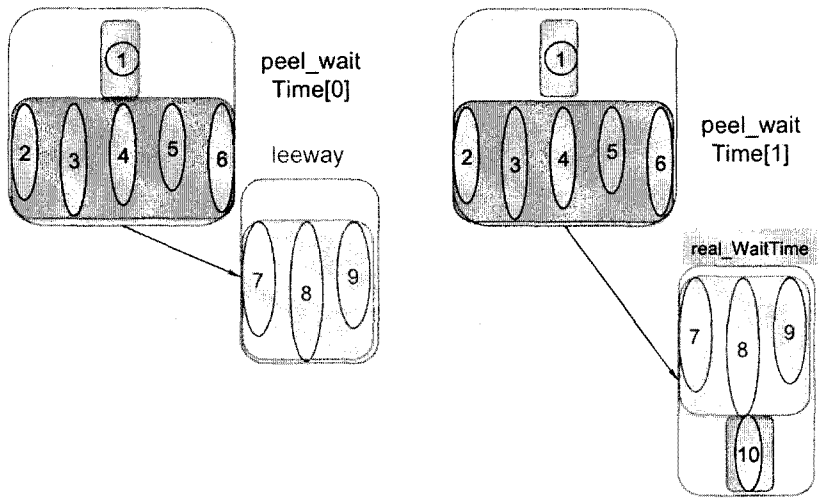


Figure 7.6 : Workflow Application Level Decision

is needed, as in the right side of Figure 7.6.) The loop then operates in one of two modes based on whether a good aggregation has been identified. If no aggregation has been selected, more levels are added until the real run time is significant enough to create overlap for the next aggregation(lines 19-25). Once this happens, the current candidate is marked as a viable aggregation. From then on, levels are added only while the wait-effectiveness of the aggregation continues to improve (lines 27-37).

7.4 Experiments

7.4.1 Experimental Methodology

To test the performance of our algorithm, we developed a prototype batch queue system simulator that implements the core algorithms of the Maui batch queue scheduler described in [51]. The input of the system is a batch queue log obtained from a production high performance computing cluster and a batch queue policy configuration file. The log contains the pseudo user id, submission time, start time, finish time, requested number of resources and time, and finish time for each job. I use

the job's submission time and the request information to compute the priority. Our system simulates the batch queue execution step by step based on the input. We also implemented the job start time estimation function (the *showstart* command). The estimation is based on the batch queue policy and all the existing queued and running jobs' maximum requested time. It does not forecast any future job submissions. Therefore, it is a best effort estimation within the knowledge of a batch queue scheduler.

We implemented the methods of Section 7.3 to submit placeholder jobs to this simulator. We also implemented the runtime algorithm depicted in Figure 7.3, using events generated by our simulator to drive the workflow management. We also implemented two other ways to execute a workflow application on a batch queue based resources. The first is a straightforward way to submit each individual task to the batch queue when it is available to run, which we will refer to as the *individual* submission method. The second is to submit a giant placeholder job that requests enough resources for the entire DAG to finish, which we will refer to as the *giant* submission method. We compare our algorithm, which we will refer to as the *hybrid* submission method, to the individual and giant method by simulating a DAG submission into the queue using different methods with exactly the same experimental configuration.

7.4.2 Experimental Setting

We generated DAG configurations for five workflow applications - EMAN, Montage, BLAST, FFT, and Gaussian Elimination described in Section 2.2.2. For each application, we used a similar approach as in our previous Chapters to generate 25 configurations for different data sizes. The total number of tasks in a workflow ranges from dozens to thousands, maximum parallelism ranges from 5 to 256, and total running time ranges from several hours to a week.

We gathered batch queue logs from four production high performance computing sites with different capacities and batch queue management systems. Figure 7.7 lists

Cluster	Institution	Batch	Length
Lonestar	Texas Adv. Computing Center	LSF	12 Mon.
Ada	Rice University	Maui	12 Mon.
LeMieux	Pittsburgh SuperComp. Center	Custom	12 Mon.
RTC	Rice University	Maui	12 Mon.
Star	University of Arkansas	Moab	10 Mon.

Figure 7.7 : The Clusters

the five clusters we studied at those sites. From each log, we collected all the jobs that terminated (either finished or reached the requested time limit) and their requested number of processors, requested running time, submission time and user id (used only for the user fair share computation). We also obtained the start time and finish time of each job to compute the actual job run time. Since most sites don't publish the details of their queuing policy and it can change from day to day, we generated three policies that favored large jobs (*FL*), small jobs (*FS*) or jobs that stay in the queue the longest (*FCFS*). These policies are derived from real site policies which all have a cap value on the resource component of the priority. For example, the FL policy does not assign a higher priory for a large job beyond a certain size. Each policy has a queue wait time component which does not have a cap value to avoid starvation. The FCFS policy has a particularly large weight on the wait time component.

Figure 7.8 shows our experimental settings. Since the batch queue loads and number of jobs in the queue fluctuate widely, the results of our algorithms depend highly on the time we simulate the submission. Therefore, we run each experimental configuration combination starting at 100 random times during the batch queue log's available time and report the mean results. In total, we ran over 700,000 experiments.

- Algorithms = {individual, giant, hybrid}
- Workflow Application = {EMAN, Montage, BLAST, FFT, Gaussian}
- DAG = { 25 for each workflow application}
- Batch Queue Logs = {Lonestar, Ada, LeMieux, RTC, Star}
- Batch Queue Policies = {FL, FS, FCFS}

Figure 7.8 : The Experiment Settings

7.4.3 Result Analysis

Figure 7.9 shows the average wait time of all workflow applications on five clusters. The wait time is defined as the

$$Time_{wait} = Time_{turn\text{-}around\text{-}time} - Time_{DAG}$$

where $Time_{DAG}$ is the time to run the DAG on the cluster with exclusive access. All but one of the differences between averages are statistically significant on a two-tailed paired t-test with p-value set at 0.05. We can see that our hybrid scheduling and submission method consistently has the least average wait time among the three execution methods. The single exception is on cluster Ada with queuing policy that favors large jobs, and that is the only statistical tie. In addition, our results indicate that, although the batch queue policy determines each job's priority, it does not affect our qualitative results significantly. However, the average wait time from each cluster varies greatly. For example, the average application wait time on the Lonestar cluster is only a fraction of the other four clusters. Furthermore, while the individual submission method waits significantly more time on the Ada and LeMieux clusters than the giant method, it waits much less time than the giant method on the RTC and Star clusters.

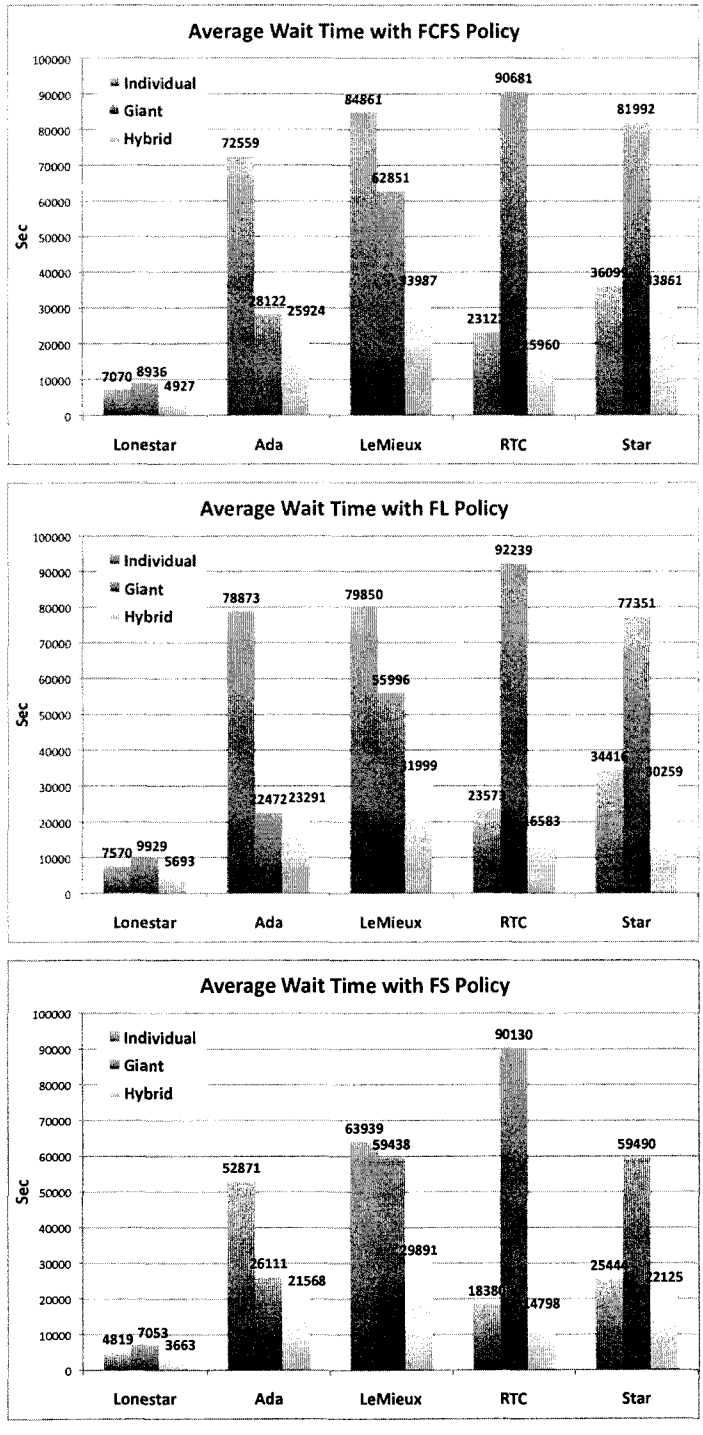


Figure 7.9 : Overall Average Wait time

Cluster	Cluster Size	Mean Jobs per Day	Mean Job Width	Mean Job Run Time	Mean Job Request Size	Actual Load	Request Load
Lonestar	5000 core	932	26.18 core	3.03 hour	274 hour	0.81	2.13
Ada	520 Core	1342	3.57 core	3.57 hour	25 hour	0.81	2.76
LeMieux	2048 Core	251	43.80 core	3.30 hour	329 hour	0.91	1.68
RTC	270 Core	108	2.43 core	13.69 hour	112 hour	0.57	1.87
Star	1200 Core	108	13.16 core	16.93 hour	1050 hour	0.83	3.94

Figure 7.10 : Cluster Configuration and Batch queue Job Characteristic

Since we ran the same set of experiments on each cluster, we hypothesized that the differences in the outcomes were the results of each cluster's unique combination of its configuration and usage pattern. Therefore, we further analyzed the characteristics of each cluster's batch queue jobs. We calculated averages for the number of jobs submitted each day, requested number of processors, actual time a job runs, requested CPU hours a job requests, the actual load and the requested load of the system over the duration of each log file. The actual load is calculated by dividing the total CPU hours used by the cluster's maximum capacity and the requested load is calculated by using the total CPU hours requested. Figure 7.10 presents each cluster's configuration and our calculations. The results clearly show each cluster has its own unique usage pattern, and we can use this to explain the variance in our experiment results. For example, Lonestar cluster has the largest computing capacity among the five clusters. This explains why the average wait time of workflow application on Lonestar is much less than on the other clusters since the it's much easier for Lonestar to fulfill the resource demand of the same workflow application than other clusters. The batch queue usage pattern can also affect the execution results in more subtle ways.

Figure 7.10 shows that the Ada cluster users tend to submit small jobs both in terms of processors and CPU hours. However, Ada's actual load is not particularly

light and it has a large number of jobs submission each day. This explains why the giant method is more effective on Ada than the individual method when the queue policy favors large jobs (see Figure 7.9). It is because the giant placeholder job would usually be the job with the highest priority in the queue and thus could start early. On the other hand, the individual job submission is less effective not only because the queue policy favors large jobs but also, since most jobs in the queue are small jobs, there are fewer opportunities to schedule an individual job by backfilling. However, Figure 7.9(c) does not show a very clear picture of why the giant method still performs relatively well when the policy favors small jobs (although the difference is much less). Figure 7.11 depicts more clearly the effect of the queue policy on the outcome for each method. We calculated the average of the *relative* wait time In Figure 7.11 by dividing each application's wait time by its running time before we computed the mean. In this way, we give each workflow's wait time an equal weight in the final result. Now, we can see that giant method actually performs worse when the queue policy favors small jobs in terms of relative wait time. Nevertheless, our hybrid method performs the best in terms of relative wait time under all three queue policies since it uses feedback from the batch queue scheduler. Combined with the results in Figure 7.9(c), we see that the giant method works relatively better for the bigger DAGs while the individual method works relatively better for the smaller DAGs.

We can also deduce from Figure 7.10 that the users of the Star cluster request long run times but not as many processors. In addition, we notice that the average requested load on Star is almost five times more than the actual load, the highest among all clusters we tested. This means the Star users tend to request many more CPU hours than they actually use. This can partially explain why the individual submission method works well on Star since the system reserves resources for the next highest priority job based on the running jobs' *requested* time. When a job finishes early, it creates a backfill window, so Star would have many backfill opportunities based on its usage pattern. Small jobs, as generated by the individual method, are

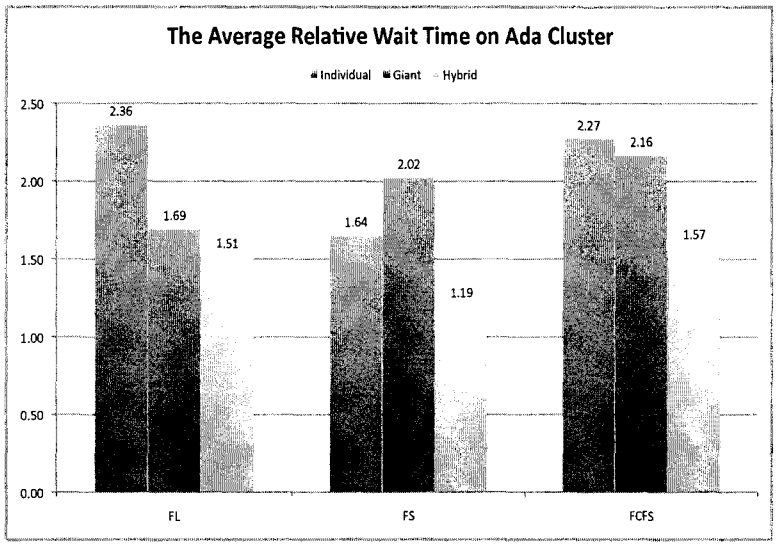


Figure 7.11 : The Effect of Queue Policy on Ada

more likely to be able to use these backfill slots. However, this does not explain why the giant method works better under a queue policy that favors small jobs on Star cluster.

We computed the average resource usage for our workflow applications on the clusters with FS queue policy. The resource usage for a workflow application is the sum of the actual running times for all placeholder jobs submitted into the queue. We gather this information by record the actual start, finish time and the number of processors each placeholder job requested. The wait time is not included. Figure 7.12 shows that the giant submission method uses almost three times more resources than the individual method while our hybrid submission method uses 10-20% less than the giant method. In both the hybrid and giant method, the additional CPU usage is mainly due to resources allocated to the placeholder according to the level with the maximum parallelism but not used on the other levels. On the Star cluster, we can see that the average giant placeholder job uses less than 600 CPU hours while Figure 7.10 shows the average job on Star requests over 1000 CPU hours. This means the giant

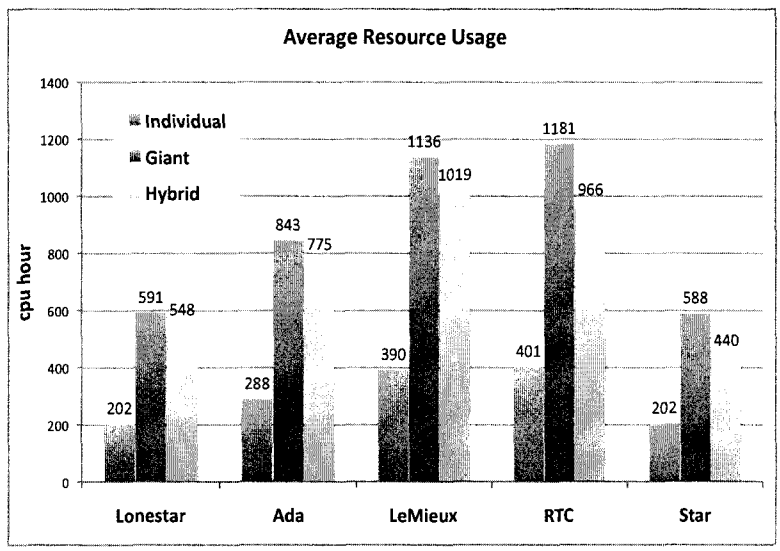


Figure 7.12 : The CPU Hour Usage

jobs are actually small compared to other jobs' requests (although, again referring to Figure 7.10, not their actual run time). This explains why all the execution methods work better under the queue policy that favors small jobs on the Star cluster. At same time, we can see that the idle processor overhead for both the giant and hybrid methods can be substantial. Despite the large job size and inaccurate job request on the Star cluster, our hybrid method again has the lowest mean wait time.

Figure 7.10 also explains the giant method's ineffectiveness on the small RTC cluster. When virtual reservations in the giant method request more than 128 processors (which about 30% of the total workflows do), it takes more than half the cluster. Even when the queue policy favors large jobs, such a job cannot run until almost all of the already running jobs on RTC finish. Figure 7.13 presents the average wait time of the workflows that require less (small DAG) or more (large DAG) than 128 processors on the RTC cluster. It shows the giant method indeed suffers the most when a single workflow application requires too much of the entire cluster. The same would be true for placeholders generated by the hybrid method, but the estimated

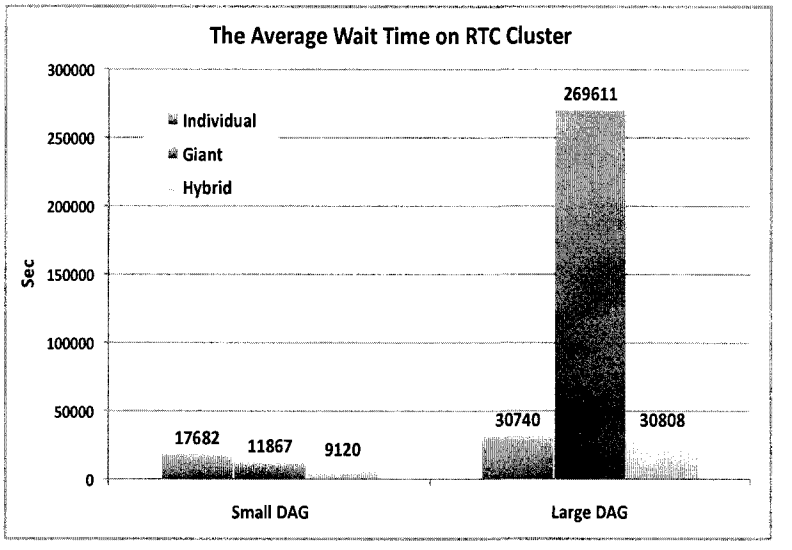


Figure 7.13 : The Average Wait Time of Small DAGs on RTC Cluster

wait times prevent our scheduler from generating such pathologies. As a result, the hybrid method submits nearly all the big DAGs in individual mode on Ada, performing competitively with the individual method and handily beating the giant method. On small DAGs, the hybrid method finds appropriate-sized placeholder jobs, and is able to outperform both other policies.

Figure 7.10 shows why the hybrid method performs the best on the LeMieux cluster. We can see that the LeMieux cluster's ratio of requested load to actual load is the lowest, which means that users do a good job in estimating their jobs' running time. That greatly improves the accuracy of the batch queue start time estimation and in turn reduces the opportunities for individual jobs to be backfilled. In short, the individual method has no leverage to schedule its small tasks. On the other end of the spectrum, the accurate wait time estimation helps the hybrid method avoid submitting large requests that would endure long waits, as the giant method is prone to do. As a result, we see a better advantage for the the hybrid method on LeMieux than any other cluster. Furthermore, LeMieux also has the highest actual load and

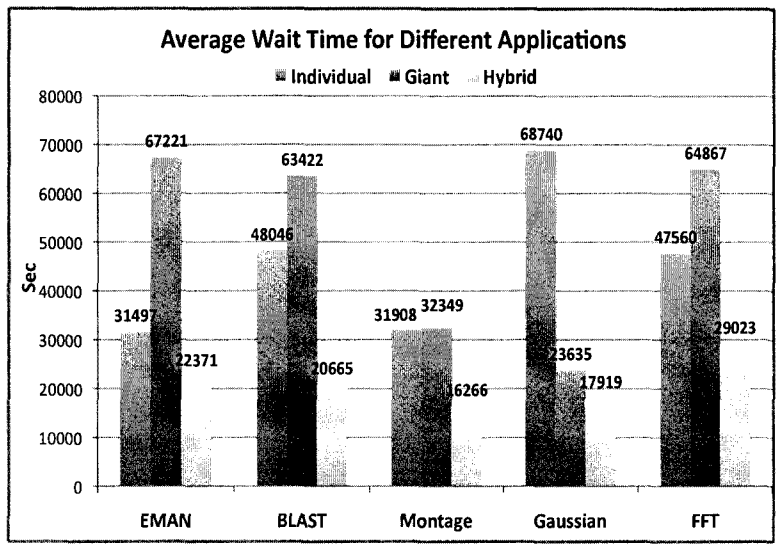


Figure 7.14 : Results on All Clusters With FL Policy

its jobs request the most processors which makes it hard for the giant method to get the highest priority and get to run quickly. Our hybrid execution method is more effective since we can choose the best granularity of the cluster so that they can get to run early and overlap the wait time with the previous cluster.

The type of workflow application can also affect the performance of the execution methods. Figure 7.14 shows the average wait time of the five workflows we tested averaged across all the clusters under the FL policy. While the giant method is best for Gaussian elimination, it is worst for the other four applications. The difference lies in the application configuration as shown in Section 2.2.2. The Gaussian elimination workflow has the most levels relative to the number of tasks among our test cases. EMAN and Montage both have a constant number of levels, and FFT grows logarithmically to a total of level 20 in our test while the longest Gaussian DAG has over 100 levels. Since the tasks in the individual submission method have to wait for the previous level to finish before they can be submitted into the queue, there are more stalls for the Gaussian workflow than other applications. Another reason is the

maximum parallelism for a Gaussian placeholder is 55 while other applications have up to 256 in our experimental settings. As we saw in Figure 7.13, the giant method performs better than the individual method when a DAG's maximum parallelism is small relative to the cluster. The giant method results on RTC cluster alone increase the average wait time for all the applications but the Gaussian workflow. Again, we see that our hybrid algorithm consistently has the least wait time for any workflow applications we tested.

7.5 Related Work

Brevik et al. [14] provided upper bound prediction of the queue wait time for an individual job. They used a binomial model and historical traces of job wait times in the queue to produce a prediction for a user specified quantile at a given confidence level without knowing the exact queuing policy of the resource. We use the estimate provided by the system itself, but in principle we could use any predictor.

There are several techniques for a user to reserve resources in a batch queue system without using the system's advanced reservation function. Condor glide-in [42] is used to create condor [82] pools in a remote resource. Nurmi et al. [78] implemented probabilistic based reservations for batch-scheduled resources. The basic idea is to use their wait time prediction [14] to choose when to submit a job so that it runs at a given time. Walker et al. [116] developed an infrastructure that submits and manages job proxies across several clusters. A user can create a virtual login session that would in turn submit the user's jobs through a proxy manager to a remote computing cluster. Kee et al. [56] developed a virtual grid system that allows a user to specify a number of resource reservations. Our work is inspired by these techniques to get a personal cluster from a batch queue controlled resource for each aggregation of tasks in the workflow application.

Limited research has been done on scheduling a workflow application on a batch queue controlled resources. Nurmi et al. [76] took into account the queue wait time

when each individual task in a workflow application is scheduled. Singh et al. [101] demonstrated the effectiveness of aggregating a workflow application using the Montage [10] application. Our approach builds on top of their ideas by dynamically choosing the aggregation for the workflow, whereas they use static mappings.

7.6 Conclusions and Future Work

In this chapter, we presented an algorithm that creates aggregations from a workflow application and submits them when the previous aggregation begins to run in the batch queue. The aggregation's granularity is computed to minimize the total wait time experienced by the workflow by overlapping most of the wait time and running time between the aggregations. By using system-provided estimates of the current queue wait time, we were able to substantially improve turn-around-time over the standard strategies of submitting many small jobs or a single large job. The results that we collected from running over half a million experiments using logs from five production HPC resources showed that our hybrid execution method consistently results in less overall wait time in the batch queue. We were able to accomplish this without any inside knowledge of the site policies, software/hardware configurations or usage patterns.

Not every batch queue resource management softwares provides the earliest job start time estimation, so in the future we would like to integrate this feature into open source systems. Moreover, we believe that providing support for workflow DAGs directly in the batch queue software would be a valuable service to users, particularly when coupled with intelligent scheduling techniques such as those we have presented.

Chapter 8

Conclusion

The objective of this dissertation is to develop new techniques to automate the process of running workflow applications on heterogeneous, distributed Grid systems and achieve good performance and reliability. To achieve this objective, the research leading to this dissertation resulted in designing and implementing novel approaches to schedule workflow applications.

8.1 Contributions

The primary contributions of this dissertation include:

- In Chapter 3, we investigated the performance of the scheduling algorithms in multi-cluster Grid environments. We are the first to compare the performance of two major classes of scheduling algorithms and to investigate the reason why some do not perform as expected. We also introduced the effective aggregated computing power (EACP) concept and showed it could drastically enhance scheduling algorithms' performance for applications that involve heavy communications.

- In Chapter 4, we studied the scalability of the scheduling algorithms and found that scalability of traditional scheduling algorithms could be a problem in a large Grid environment. We developed a generic approach to address this problem. We further verified that it improved the scalability of scheduling algorithms while achieving comparable performance.

- In Chapter 5, we measured how the resource performance unpredictability of a

Grid environment affects the scheduling algorithms. Based on our observations, we developed an application execution framework with performance feedbacks to address this issue and showed that combining the dynamic and static scheduling techniques can lead to good workflow application performance in a dynamic and unpredictable computing environment

- In Chapter 6, we modeled the reliability of large multi-cluster Grid systems and also the success probability of a workflow application running on such resources. We incorporated traditional fault tolerance techniques into workflow application scheduling heuristics and demonstrated how effective they are. We also estimated the additional resource usage.

- In Chapter 7, we proposed a novel DAG aggregation algorithm that can reduce the resource provisioning overhead for a workflow application on a batch queue controlled resource. The algorithm reduces the overall wait time by aggregating a DAG into several components and submitting each component into the batch queue so that its wait time is overlapped with its parent component's run time.

8.2 Future Work

The strategies and techniques developed in this dissertation are not only steps toward making Grid programming easier and efficient, they also can lead to future researches on various related platforms. Here are three closely related areas where we see good potential for us to explore many interesting ideas.

Workflow application manger. Although our dissertation developed new and effective techniques to help run a workflow application on a large distributed Grid environment, it would be great to put all the work together and build an open source workflow application manager. The application manager would incorporate all the techniques we developed in this dissertation into some existing workflow execution engines, such as Pegasus [89], and provide the performance and reliability we demon-

strated in our prototype implementations. The application manager would also be a valuable infrastructure on which we conduct future research experiments. It would be very helpful if the high performance computing (HPC) community would embrace our work and use the application manager to execute large scale workflows on production resources. This would help us identify the weakness of our work in a real production environment and not only motivate our future researches but also verify our results.

Parallel computing. Most scheduling algorithms for workflow applications are derived from traditional instruction or thread/process scheduling algorithms. The latest trend towards multi-core in the commodity hardware domain makes it possible to apply our findings to the future many-core based parallel computing infrastructures. It is because the model of multi-threaded (multi-process) computation in a run-time system can be modeled as a directed acyclic graph(DAG), they are essentially workflow applications at a lower level (finer grained).

Furthermore, as a single processor gets more and more cores, it can no longer keep a flat connection between cores. In addition, with the development of GPGPU (General Purpose computation on Graphics Processing Units) and Cell processors, the cores are no long homogenous in a single chip. These features make a multi-core processor machine resemble a multi-cluster Grid environment because not only the processing speeds of the heterogenous cores are different but the communication times between the cores are also not the same. Therefore, workflow scheduling is essential to enhance the performance of shared memory programs, such as those written in openMP [27], on a multi-core processor. Scheduling is also very important for parallel programs written in partitioned global address space (PGAS) such as Chapel [19], X10 [21],and Co-Array Fortran [33]. This is because in a partitioned global address space the program usually makes asynchronized calls, either implicitly or explicitly, that require communication between a pair of processors. Different communication patterns can greatly affect the program performance thus a good scheduling algorithm is essential to minimize unnecessary communications between processors in different

partitions which would be more costly than within a partition. For the similar reason, languages that target both shared memory and distributed memory architectures like High Performance Fortran [64] and Habanero [85] could also use workflow scheduling to enhance the program performance.

Compared with Grid workflow application scheduling, the multi-thread/process scheduling environment usually has less and fewer resource performance fluctuations which makes performance prediction very difficult. Therefore, to some extent, multi-thread/process scheduling is actually less challenging than Grid workflow application scheduling and it could determine the performance of the application more. We believe our dissertation experience can be applied to the parallel computing domain and improve the performance of future multi-core based parallel computing.

Cloud computing Another major development in the high performance domain recently is the emergence of cloud computing. Although there is no exact definition of cloud computing, most cloud computing infrastructure, with the help of virtualization techniques, enable users to create customized computing environments on demand. In general, there are three types of cloud computing. The first one is public cloud computing. A public cloud is maintained by an off-site third-party provider. It provides some sort of computing power (either hardware or software) in a flexible lease term and bills the user based on resource usage. Amazon EC2 [36] and Google App Engine [37] are two of the more influential cloud computing services with the former providing cloud infrastructure (hardware) and the later providing platform (software). The second type of cloud computing is private cloud computing. It essentially emulates public cloud computing on private networks since many companies hesitate to move their critical services off-site. The third type of cloud computing is a hybrid cloud that is a mix of a public cloud and a private cloud.

Although the virtualization techniques make the resources appear homogenous, the underlying resources are still distributed. Thus, we think cloud computing is another platform on which we can adept our techniques to run workflow applications.

For example, a public cloud computing site usually charges users by usage, we can extend our scheduling algorithms to provide a balanced approach towards performance and cost for the user. However, there are some differences between a traditional Grid and a cloud that affect the effectiveness of our approaches on a cloud. For example, a public cloud usually abstracts away the network topology or configuration underneath it. This makes the fine grained style of scheduling in our dissertation work not applicable directly although we can measure the or monitor the bandwidths and latencies between each node pairs. However, most of our work can be applied directly on private clouds.

8.3 Conclusions

In this thesis, we investigated an important class of workflow application scheduling problem and explored the potential of Grid computing. Although Grid computing, like many other once promising disruptive technologies, does not live up to the high expectation we originally projected, the research work that we did nevertheless help advance the state-of-the-art of high performance computing in general. These efforts not only help us gain in-depth knowledge of the large scale distributed computing but also teach us lessons that we could use to avoid set-backs in our future endeavor. In the end, just like Albert Einstein said, "if we knew what it was we were doing, it would not be called research, would it?"

Bibliography

[1] D. Abramson, A. Lynch, H. Takemiya, Y. Tanimura, S. Date, H. Nakamura, Karpjoo Jeong, Suntae Hwang, Ji Zhu, Zhong hua Lu, C. Amoreira, K. Baldridge, Hurng-Chun Lee, Chi-Wei Wang, Horng-Liang Shih, T. Molina, Wilfred W. Li, and P.W. Arzberger. Deploying scientific applications to the pragma grid testbed: Strategies and lessons. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 241–248, May 2006.

[2] Ishfaq Ahmad and Yu-Kwong Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):872–892, 1998.

[3] G. Allen, D. Angulo, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzysky, J. Pukacki, M. Russell, T. Radke, E. Seidel, J. Shalf, and I. Taylor. GridLab: Enabling applications on the Grid. In *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, pages 39–45. Springer-Verlag, 2002.

[4] Globus Alliance. `http://www.globus.org/`.

[5] G. Avellino, S. Barale, S. Beco, B. Cantalupo, D. Colling, F. Giacomini, A. Gianelle, A. Guarise, A. Krenek, D. Kouril, and A. Maraschini et al. The EU DataGrid Workload Management System: towards the second major release, 2003.

[6] Rashmi Bajaj and Dharma P. Agrawal. Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed*

Systems, 15(2):107–118, 2004.

[7] Sanjeev Baskiyar and Christopher Dickinson. Scheduling directed acyclic task graphs on a bounded set of heterogeneous processors using task duplication. *Journal of Parallel and Distributed Computing*, 65(8):911–921, 2005.

[8] Rüdiger Berlich, Marcus Hardt, Marcel Kunze, Malcolm Atkinson, and David Fergusson. Egee: building a pan-european grid training organisation. In *ACSW Frontiers '06: Proceedings of the 2006 Australasian workshops on Grid computing and e-research*, pages 105–111. Australian Computer Society, Inc., 2006.

[9] Francine Berman, Andrew Chien, Keith Cooper, Jack Dongarra, Ian Foster, Dennis Gannon, Lennart Johnsson, Ken Kennedy, Carl Kesselman, John Mellor-Crumme, Dan Reed, Linda Torczon, and Rich Wolski. The grads project: Software support for high-level grid application development. *International Journal of High Performance Computing Applications*, 15(4):327–344, 2001.

[10] G. B. Berriman, J. C. Good, A. C. Laity, A. Bergou, J. Jacob, D. S. Katz, E. Deelman, C. Kesselman, G. Singh, M H. Su, and R. Williams. Montage: A grid enabled image mosaic service for the national virtual observatory. In *In Proc. of Astronomical Data Analysis Software and Systems (ADASS) Conference*, page 2003, 2003.

[11] Jim Blythe, Sonal Jain, Ewa Deelman, Yolanda Gil, Karan Vahi, Anirban Mandal, and Ken Kennedy. Task scheduling strategies for workflow-based applications in grids. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005)*. IEEE Press, 2005.

[12] Brett Bode, David M. Halstead, Ricky Kendall, Zhou Lei, and David Jackson. The portable batch scheduler and the maui scheduler on linux clusters. In *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, pages 27–27. USENIX Association, 2000.

[13] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Lasislau L. Bolonii, Muthucumara Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen, and Richard 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*, 61:810–837, 2001.

[14] John Brevik, Daniel Nurmi, and Rich Wolski. Predicting bounds on queuing delay for batch-scheduled parallel machines. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 110–118. ACM, 2006.

[15] Krishnaveni Budati, Jason Sonnek, Abhishek Chandra, and Jon Weissman. Ridge: combining reliability and performance in open grid platforms. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 55–64. ACM, 2007.

[16] Junwei Cao, Stephen A. Jarvis, Subhash Saini, and Graham R. Nudd. Gridflow: Workflow management for grid computing. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 198. IEEE Computer Society, 2003.

[17] Charlie Catlett and et al. `http://www.griphyn.org`, 2002.

[18] Texas Advanced Computing Center. `http://www.top500.org/`.

[19] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *International Journal High Perform. Computing Application*, 21(3):291–312, 2007.

[20] Wai-Yip Chan and Chi-Kwong Li. Heterogeneous dominant sequence cluster (hdsc): a low complexity heterogeneous scheduling algorithm. In *Communications, Computers and Signal Processing, 1997. '10 Years PACRIM 1987-1997 -*

Networking the Pacific Rim'. 1997 IEEE Pacific Rim Conference on, volume 2, pages 956–959 vol.2, Aug 1997.

[21] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538. ACM, 2005.

[22] A. Chien, H. Casanova, Y.-S. Kee, and R. Huang. The Virtual Grid Description Language: vgDL. . Technical Report CS2005-0817, University of California, San Diego, Department of Computer Computer Science and Engineering, Aug 2005.

[23] L. Choy, S. Petiton, and M. Sato. Resolution of large symmetric eigenproblems on a world wide grid. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 301–308. IEEE Computer Society, 2007.

[24] W. Chrabakh and R. Wolski. Gradsat: A parallel sat solver for the grid. In *W. Chrabakh and R. Wolski. GrADSAT: A Parallel SAT Solver for the Grid. In Proceedings of IEEE SC03, November 2003.*, 2003.

[25] Inc. Cluster Resources. `http://clusterresources.com/`.

[26] K. Cooper, A. Dasgupta, K. Kennedy, C. Koelbel, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, F. Berman, H. Casanova, A. Chien, H. Dail, X. Liu, A. Olugbile, O. Sievert, H. Xia, L. Johnsson, B. Liu, M. Patel, D. Reed, W. Deng, C. Mendes, Z. Shi, A. YarKhan, and J. Dongarra. New grid scheduling and rescheduling methods in the grads project. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 199–, April 2004.

[27] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[28] Ewa Deelman, Scott Callaghan, Edward Field, Hunter Francoeur, Robert Graves, Nitin Gupta, Vipin Gupta, Thomas H. Jordan, Carl Kesselman, Philip Maechling, John Mehringer, Gaurang Mehta, David Okaya, Karan Vahi, and Li Zhao. Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In *E-SCIENCE '06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 14. IEEE Computer Society, 2006.

[29] A. Denis and etc O. Aumage. Wide-area communication for grids: An integrated solution to connectivity, performance and security problems. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, pages 97–106. IEEE Computer Society, 2004.

[30] Atakan Dogan and Füsun Özgüner. Ldbs: A duplication based scheduling algorithm for heterogeneous computing systems. In *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing*, page 352. IEEE Computer Society, 2002.

[31] Fangpeng Dong and Selim G. Akl. Scheduling algorithms for Grid computing: State of the art and open problems. Technical Report TR06-504, Queen's University, 2006.

[32] Jack J. Dongarra, Emmanuel Jeannot, Erik Saule, and Zhiao Shi. Bi-objective scheduling algorithms for optimizing makespan and reliability on heterogeneous systems. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 280–288. ACM, 2007.

[33] Yuri Dotsenko, Cristian Coarfa, and John Mellor-Crummey. A multi-platform co-array fortran compiler. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 29–40. IEEE Computer Society, 2004.

[34] M. Dutra, P. Rodrigues, G. Giraldi, and B. Schulze. Uima grid: Distributed large-scale text analysis. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 317–326. IEEE Computer Society, 2007.

[35] P. Thambidurai E. Illvarasan. Levelized scheduling of directed acyclic precedence constrained task graphs onto heterogeneous computing system. In *First International Conference on Distributed Frameworks for Multimedia Applications (DFMA'05)*, pages 262–269. IEEE Computer Society, 2005.

[36] Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2`.

[37] Google App Engine. `http://code.google.com/appengine/`.

[38] Linked Environments for Atmospheric Discovery (LEAD) Portal. `https://portal.leadproject.org`.

[39] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[40] Ian. Foster and Carl. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kauffmann Publishers, Inc., 1999.

[41] Ian. Foster and Carl. Kesselman. *The Grid 2.* Morgan Kauffmann Publishers, Inc., 2003.

[42] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[43] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[44] Tristan Glatard, Johan Montagnat, and Is Cnrs. An experimental comparison of grid5000 clusters and the egee grid. In *In Workshop on Grid*, 2006.

[45] European GRID. `http://www.eurogrid.org`.

[46] Ewa Deelman Gurmeet Singh, Carl Kesselman. Optimizing grid-based workflow execution. *Journal of Grid Computing*, 3(3):201–219, 2005.

[47] Bill Howe, Peter Lawson, Renee Bellinger, Erik Anderson, Emanuele Santos, Juliana Freire, Carlos Scheidegger, António Baptista, and Cláudio Silva. End-to-end escience: Integrating workflow, query, visualization, and provenance at an ocean observatory. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 127–134. IEEE Computer Society, 2008.

[48] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[49] Soonwook Hwang and Carl Kesselman. Gridworkflow: A flexible failure handling framework for the grid. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, page 126. IEEE Computer Society, 2003.

[50] A. Iosup, M. Jan, O. Sonmez, and D.H.J. Epema. On the dynamic resource availability in grids. *Grid Computing, 2007 8th IEEE/ACM International Conference on*, pages 26–33, Sept. 2007.

[51] David B. Jackson, Quinn Snell, and Mark J. Clement. Core algorithms of the maui scheduler. In *JSSPP '01: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer-Verlag, 2001.

[52] Gideon Juve and Ewa Deelman. Resource provisioning options for large-scale scientific workflows. *eScience, IEEE International Conference E-Science*, 0:608–613, 2008.

[53] Gopi Kandaswamy, Anirban Mandal, and Daniel A. Reed. Fault tolerance and recovery of scientific workflows on computational grids. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 777–782. IEEE Computer Society, 2008.

[54] Y.-S. Kee, H. Casanova, and A. A. Chien. Realistic modeling and svnthesis of resources for computational grids. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 54. IEEE Computer Society, 2004.

[55] Y.-S. Kee, D. Logothetis, R. Huang, H. Casanova, and A. A. Chien. Efficient resource description and high quality selection for virtual grids. In *Proceedings of the 5th IEEE Symposium on Cluster Computing and the Grid (CCGrid'05), Cardiff, U.K.*, May 2005.

[56] Yang-Suk Kee, C. Kesselman, D. Nurmi, and R. Wolski. Enabling personal clusters on demand for batch resources using commodity software. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–7, April 2008.

[57] K. Kennedy, K. Cooper, F. Berman, A. Chien, I. Foster, C. Kesselman, D. Reed, J. Dongarra, and R. Wolski et al. Toward a framework for preparing and executing adaptive Grid programs. In *Proceedings of NSF Next Generation Systems Program Workshop (International Parallel and Distributed Processing Symposium 2002), Fort Lauderdale, FL, April 2002*, 2002.

[58] Ken Kennedy, Mark Mazina, John M. Mellor-Crummey, Keith D. Cooper, Linda Torczon, Francine Berman, Andrew A. Chien, Holly Dail, Otto Sievert, Dave Angulo, Ian T. Foster, Ruth A. Aydt, Daniel A. Reed, Dennis Gannon, S. Lennart Johnsson, Carl Kesselman, Jack Dongarra, Sathish S. Vadhiyar, and Richard Wolski. Toward a framework for preparing and executing adaptive grid programs. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 322. IEEE Computer Society, 2002.

[59] S.J. Kim and J.C. Browne. A general approach to mapping of parallel computations upon multiprocessor architectures. In *Proceedings of International Conference of Parallel Processing*, volume 2, pages 1–8, 1988.

[60] Arun Krishnan. Gridblast: a globus-based high-throughput implementation of blast in a grid computing framework: Research articles. *Concurrency and Computation: Practice & Experience*, 17(13):1607–1623, 2005.

[61] Boontee Kruatrachue and Ted Lewis. Grain size determination for parallel processing. *IEEE Softw.*, 5(1):23–32, 1988.

[62] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *J. Parallel Distrib. Comput.*, 59(3):381–422, 1999.

[63] K. Limaye, B. Leangsuksun, Yudan Liu, Z. Greenwood, S. L. Scott, R. Libby, and K. Chanchio. Reliability-aware resource management for computational grid/cluster environments. In *GRID '05: Proceedings of the 6th IEEE/ACM*

International Workshop on Grid Computing, pages 211–218. IEEE Computer Society, 2005.

[64] David B. Loveman. High performance fortran. *IEEE Parallel & Distributed Technology: Systems & Technology*, 1(1):25–42, 1993.

[65] Load Sharing Facility (LSF). `http://www.platform.com/`.

[66] S. Ludtke, P. Baldwin, and W. Chiu. EMAN: Semiautomated software for high resolution single-particle reconstructions. *Journal Structure Biology*, 128(1):82–97, 1999.

[67] S. Venugopal M. Rahman and R. Buyya. A dynamic critical path algorithm for scheduling scientific workflow applications on global grids. In *Proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing.* IEEE CS Press, Dec 2007.

[68] J Follen MA Iverson, O Gregory. Parallelizing existing applications in a distributed heterogeneous environment. In *4th Heterogeneous Computing Workshop (HCW '95*, pages 93–100, 1995.

[69] Anirban Mandal, K. Kennedy, C. Koelbel, G. Marin, J. Mellor-Crummey, B. Liu, and L. Johnsson. Scheduling strategies for mapping application workflows onto the grid. In *High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE International Symposium on*, pages 125–134, July 2005.

[70] Carolyn McCreary, A. A. Khan, J. J. Thompson, and M. E. McArdle. A comparison of heuristics for scheduling dags on multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 446–451. IEEE Computer Society, 1994.

[71] Celso L. Mendes and Daniel A. Reed. Monitoring large systems via statistical sampling. *International Journal of High Performance Computing Application*, 18(2):267–277, 2004.

[72] Dagman MetaScheduler. `http://www.cs.wisc.edu/condor/dagman`.

[73] W. Gentzsch (Sun Microsystems). Sun grid engine: Towards creating a compute power grid. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 35. IEEE Computer Society, 2001.

[74] Robert Morris and et al. `http://pdos.csail.mit.edu/p2psim/kingdata`, 2004.

[75] Daniel Nurmi, John Brevik, and Rich Wolski. Modeling machine availability in enterprise and wide-area distributed computing environments. pages 432–441, 2005.

[76] Daniel Nurmi, Anirban Mandal, John Brevik, Chuck Koelbel, Rich Wolski, and Ken Kennedy. Evaluation of a workflow scheduler using integrated performance modelling and batch queue wait time prediction. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 119. ACM, 2006.

[77] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 9th IEEE Symposium on Cluster Computing and the Grid (CCGrid'09)*, May 2009.

[78] Daniel Charles Nurmi, Rich Wolski, and John Brevik. Varq: virtual advance reservations for queues. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 75–86. ACM, 2008.

[79] Open PBS. `http://www.openpbs.org/`.

[80] Kassian Plankensteiner, Radu Prodan, Thomas Fahringer, Attila Kertesz, and Peter Kacsuk. Fault-tolerant behavior in state-of-the-art grid worklow management systems. Technical Report TR-0091, Institute on Grid Information, Resource and Worklow Monitoring Services, CoreGRID - Network of Excellence, October 2007.

[81] Special Priority and Urgent Computing Environment. `http://spruce.teragrid.org/`.

[82] Condor Research Project. `http://www.cs.wisc.edu/condor`.

[83] GridBus Research Project. `http://www.gridbus.org`.

[84] Gridway Project. `http://www.gridway.org`.

[85] Habanero Multicore Software Research Project. `http://habanero.rice.edu/Habanero_Home.html/`.

[86] Kepler Project. `http://kepler-project.org/`.

[87] Taverna Project. `http://taverna.sourceforge.net/`.

[88] The GridOneD Project. `http://www.gridoned.org/`.

[89] The Pegasus Project. `http://pegasus.isi.edu/index.php`.

[90] Simple Object Access Protocol(SOAP). `http://www.w3.org/TR/soap/`.

[91] Narasimha Raju, Yudan Liu, Chokchai Box Leangsuksun, Raja Nassar, and Stephen Scott. Reliability analysis in hpc clusters. In *Proceedings of the High Availability and Performance Computing Workshop*, 2006.

[92] S. Ranaweera and D.P. Agrawal. A task duplication based scheduling algorithm for heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 445–450, 2000.

[93] Mathilde Romberg. The unicore architecture: Seamless access to distributed resources. In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, page 44. IEEE Computer Society, 1999.

[94] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 111–, April 2004.

[95] Rizos Sakellariou and Henan Zhao. A low-cost rescheduling policy for efficient mapping of workflows on grid systems. *Scientific Programming*, 12(4):253–262, 2004.

[96] M. Salahuddin, T. Hung, H. Soh, E. Sulaiman, O. Soon, L. Sung, and R Yunxia. Grid-based pse for engineering of materials (gpem). In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, volume 00, pages 309–316. IEEE Computer Society, 2007.

[97] Vivek Sarkar. *Partitioning and scheduling parallel programs for execution on multiprocessors.* PhD thesis, Stanford University, Stanford, CA, USA, 1987.

[98] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 249–258. IEEE Computer Society, 2006.

[99] Web Services. `http://www.w3.org/2002/ws/`.

[100] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, 1993.

[101] Gurmeet Singh, Mei-Hui Su, Karan Vahi, Ewa Deelman, Bruce Berriman, John Good, Daniel S. Katz, and Gaurang Mehta. Workflow task clustering for best effort systems with pegasus. In *MG '08: Proceedings of the 15th ACM Mardi Gras conference*, pages 1–8. ACM, 2008.

[102] Quinn Snell, Mark J. Clement, David B. Jackson, and Chad Gregory. The performance impact of advance reservation meta-scheduling. In *IPDPS '00/JSSPP '00: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 137–153. Springer-Verlag, 2000.

[103] R. Souto, R. Avila, P. Navaux, M. Py, T. Diverio, H. Velho, S. Stephany, A. J. Preto, J. Panetta, E. Rodrigues, E. Almeida, P. Dias, and A. Gandu. Processing mesoscale climatology in a grid environment. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 363–370. IEEE Computer Society, 2007.

[104] A. Sulistio and R. Buyya. A grid simulation infrastructure supporting advance reservation, 2004.

[105] Y. Sun, X.-H. Chen and M. Wu. Scalability of heterogeneous computing. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing (ICPP'05)*, pages 557–564. IEEE Computer Society, 2005.

[106] Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. The Triana Workflow Environment: Architecture and Applications. In Ian Taylor, Ewa Deelman, Dennis Gannon, and Matthew Shields, editors, *Workflows for e-Science*, pages 320–339. Springer, New York, 2007.

[107] TeraGrid. `http://www.teragrid.org/about`.

[108] Texas Advanced Supercomputing Center. `http://www.tacc.utexas.edu/`.

[109] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience: Research articles. *Concurrency and Computation: Practice & Experience*, 17(2-4):323–356, 2005.

[110] J. Tian and K. Ma. Super-resolution imaging using grid computing. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 293–300. IEEE Computer Society, 2007.

[111] Haluk Topcuouglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 2(13):260–274, 2002.

[112] Univa UD. `http://www.univaud.com/`.

[113] Rice University. `http://cohesion.rice.edu/centersandinst/citi/research.cfm?doc_id=5949`.

[114] Sathish S. Vadhiyar and Jack J. Dongarra. A performance oriented migration framework for the grid. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 130. IEEE Computer Society, 2003.

[115] Fredrik Vraalsen, Ruth A. Aydt, Celso L. Mendes, and Daniel A. Reed. Performance contracts: Predicting and monitoring grid application behavior. In *GRID '01: Proceedings of the Second International Workshop on Grid Computing*, pages 154–165. Springer-Verlag, 2001.

[116] E. Walker, J.P. Gardner, V. Litvin, and E.L. Turner. Creating personal adaptive clusters for managing scientific jobs in a distributed computing environment. In *Challenges of Large Applications in Distributed Environments, 2006 IEEE*, pages 95–103, 2006.

[117] Weibull and Waloddi. A statistical distribution function of wide applicability. In *Journal of Applied Mechanics*, 1951.

[118] M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, 1990.

[119] L. Yang, J. M. Schopf, and I.Foster. Improving parallel data transfer times using predicted variances in shared networks. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005)*. IEEE Press, 2005.

[120] Tao. Yang and Apostolos Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.

[121] Asim YarKhan and Jack J. Dongarra. Biological sequence alignment on the computational grid using the grads framework. *Future Generation Computer Systems*, 21(6):980–986, 2005.

[122] Jia Yu and Rajkumar Buyya. A taxonomy of scientific workflow systems for Grid computing. *SIGMOD Rec.*, 34(3):44–49, 2005.

[123] Jia Yu, Rajkumar Buyya, and Chen Khong Tham. Cost-based scheduling of scientific workflow application on utility grids. In *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*, pages 140–147. IEEE Computer Society, 2005.

[124] Jia Yu, M. Kirley, and R. Buyya. Multi-objective planning for workflow execution on grids. *Grid Computing, 2007 8th IEEE/ACM International Conference on*, pages 10–17, 19-21 Sept. 2007.

[125] W. Yu, Z. Shi. An adaptive rescheduling strategy for grid workflow applications. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007*, pages

1–8. IEEE International, 2007.

[126] Bo Zhang, T. S. Eugene Ng, Animesh Nandi, Rudolf Riedi, Peter Druschel, and Guohui Wang. Measurement based analysis, modeling, and synthesis of the internet delay space. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 85–98. ACM, 2006.

[127] Y. Zhang, C. Koelbel, and K. Kennedy. Relative performance of scheduling algorithms in grid environments. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 521–528. IEEE Computer Society, 2007.

[128] Yang Zhang, Charles Koelbel, and Keith Cooper. Batch queue resource scheduling for workflow applications. *Cluster Computing, 2009 IEEE International Conference on*, 0, 2009.

[129] Yang Zhang, Charles Koelbel, and Keith Cooper. Hybrid re-scheduling mechanisms for workflow applications on multi-cluster grid. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:116–123, 2009.

[130] Yang Zhang, Anirban Mandal, Henri Casanova, Andrew A. Chien, Yang-Suk Kee, Ken Kennedy, and Charles Koelbel. Scalable grid application scheduling via decoupled resource selection and scheduling. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 568–575. IEEE Computer Society, 2006.

[131] Yang Zhang, Anirban Mandal, Charles Koelbel, and Keith Cooper. Combined fault tolerance and scheduling techniques for workflow applications on computational grids. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:244–251, 2009.

[132] Y. Zhao, M. Wilde, I. Foster, J. Voeckler, T. Jordan, E. Quigg, and J. Dobson.

Grid middleware services for virtual data discovery, composition, and integration. In *Proceedings of the 2nd workshop on Middleware for grid computing*, pages 57–62. ACM Press, 2004.