# SCHEDULING SCIENTIFIC WORKFLOW APPLICATIONS IN COMPUTATIONAL GRIDS

---

A Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

---

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

---

By

Yonghong Yan

May 2007

# SCHEDULING SCIENTIFIC WORKFLOW APPLICATIONS IN COMPUTATIONAL GRIDS

---

Yonghong Yan

APPROVED:

---

Dr. Barbara Chapman, Chairman
Department of Computer Science

---

Dr. Jung-chang Huang
Department of Computer Science

---

Dr. Christoph Eick
Department of Computer Science

---

Dr. Rong Zheng
Department of Computer Science

---

Dr. Min Ru
Department of Mathematics

---

---

Dean, College of Natural Sciences and Mathematics

# Acknowledgements

An undertaking such as a dissertation is not completed without the support of many people. My first debt of gratitude must go to my advisor, Dr. Barbara Chapman. She patiently provided the vision, encouragement and advise necessary for me to proceed through the doctorial program and complete my dissertation. Special thanks go to my committee, Dr. Jung-chang Huang, Dr. Christoph Eick, Dr. Rong Zheng and Dr. Min Ru for their helpful suggestions and time.

I am very grateful to Dr. Daewon Byun and his research group from the Geosciences Department who helped me understand their workflow application and provided valuable comments along my work. Also I appreciate the HPCTools group members: Dr. Babu Sundaram, Dr. Lei Huang, Chunhua Liao and Laksono Adhianto for the valuable discussions and friendships. It has been a great and enjoyable time to work with them. Special thanks to the faculty and staff of the Computer Science Department for their support and influence along on my career path. It has been a great experience to meet so many talented people during my study.

I want to acknowledge my wife, Fuhua Zhou, my son, Jonathan Yan, my parents and my parents-in-law for their love, support, encouragement and understanding in dealing with all the challenges I have faced. Nothing in a simple paragraph can express the love I have for you.

# SCHEDULING SCIENTIFIC WORKFLOW APPLICATIONS IN COMPUTATIONAL GRIDS

---

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

---

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

---

By

Yonghong Yan

May 2007

# Abstract

A computational grid, built using grid computing technology, is a network of computing resources that work together as a single, uniform operating environment. It can be viewed as a virtual supercomputer designed for large-scale applications. One important characteristic of these applications is that they are no longer being developed as monolithic and single-executable codes, but incorporate multiple dependent computational modules. The execution of these applications involves the concurrent and sequential execution of multiple modules in a predefined order, and the automatic and timely data transfer between modules. These applications are often referred to as scientific workflow applications.

A very important issue in executing a scientific workflow application in computational grids is how to map and schedule workflow modules onto multiple distributed resources, and handle module dependencies in a timely manner to deliver users' expected performance. The goal of this research is to develop a workflow system to address the issue of workflow scheduling in computational grid environments. In our work, we have developed a grid workflow description language that addresses the limitation of lacking support for resource request specification in current related efforts. An integrated workflow scheduling architecture has been defined that provides the capabilities of workflow execution planning, resource allocation and execution coordination. Our workflow scheduler applies advanced scheduling techniques, such as planning, resource reservation and performance predictions in the resource allocation process. The simulation results show that our workflow scheduler reduces the workflow execution time by about 20% on average under moderate to high resource load, compared to the scheduling policies used in most of current workflow systems.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

A computational grid [1], built using grid computing technology, is a network of computing resources that work together as a single, uniform operating environment. It provides pervasive access to resources with advanced computational and storage capabilities, and can be viewed as a virtual supercomputer designed for large-scale applications. These applications are massive computational problems from scientific and engineering domains, such as earthquake simulation [2], and climate/weather modeling [3, 4, 5] that require huge computation power and storage resources for their operations. Computational grids are one of the most cost-effective strategies for meeting the resource needs of these applications.

One important characteristic of these applications is that they are no longer being developed as monolithic and single-executable codes, but incorporate multiple dependent computational modules, and entail transfer and storage of a large amount of data. The execution of these applications involves the concurrent and sequential

execution of multiple modules, and the automatic and timely data transfer between modules. These applications are often referred to as scientific workflow applications, and a module is often called a task in workflow terms. A very important issue in executing a scientific workflow application in computational grids is how to map and schedule workflow modules onto multiple distributed resources, and handle module dependencies in a timely manner to deliver users' expected performance. The goal of this research is to develop a workflow system to address this issue in computational grid environments.

## 1.1 Workflow Applications in Computational Grid Environments

There are mainly two topics in the effort of deploying workflow applications in computational grid environments: the workflow description means and the workflow scheduler. Workflow description is about how to model workflow structure, i.e., describing the workflow tasks and their dependencies. Most workflow applications are modeled as graphs; the graph vertices denote workflow tasks and the graph edges denote task dependencies. Workflow scheduling is about executing workflow tasks in the right order, on the right resources and at the right time. It can be as simple as performing a topological sort [6], and then launching the workflow tasks according to the topological order. In grid environments, these two issues become much more complex than they appear to be. For example, we must coordinately allocate multiple resources to the workflow tasks before their execution, and reduce the

queue waiting time for workflow tasks that are submitted to resources for execution. In describing a workflow, resource request information for workflow tasks must be specified to support for resource allocation decision making.

There have been many efforts to develop a workflow system and workflow scheduling algorithms for grid environments, see a list of these efforts in [7]. In most of these efforts, such as DAGMan [8], Taverna [9], Karajan [10] and Triana [11], the workflow description methods focus on the expressive capability of describing workflow structures, and they do not support for specifying scheduling related information. The workflow scheduling strategy is basically an extension of the workflow enact engine with grid job launching and file transfer services. They lack the capability of allocating resources for workflow tasks, thus requiring users to manually specify resource allocation details. This is a very inflexible and impractical approach in dynamic and virtual grid environments. In efforts to develop workflow scheduling algorithms, such as ASKALON [12], Pegasus [13], and Gridbus[14], grid resource management issues are not well addressed. For example, the grid scheduling hierarchy, which may introduce significant performance overhead during workflow execution in high-load environments, is not taken into account in the scheduling process. Because of that, some assumptions made when evaluating those algorithms are unrealistic in computational grid environments.

Advanced scheduling techniques, such as resource co-allocation, resource negotiation and advanced reservation, and performance prediction, have been studied in the past several decades. Some of them have been implemented in commercial schedulers for computing clusters or supercomputers. Recently, they have been active and

open research topics in the scope of grid computing. Yet we found very few efforts to study and use those techniques in grid workflow systems. We believe (and have proved by this work) that those advanced scheduling techniques can greatly improve the overall workflow execution performance.

## 1.2 Research Goals and Contributions

The goal of this work is to develop a grid workflow system with advanced scheduling techniques, and to study the performance impacts of these advanced scheduling techniques on the overall workflow performance in computational grid environments. In the workflow system, the description method should support for specifying information required for workflow scheduling. The workflow scheduler should have the capability of co-allocating resources for workflow tasks, and should be able to apply different advanced scheduling techniques in the scheduling process to deliver users' expected quality of services. We summarize the contributions of our work presented in this dissertation as follows:

1. The definition of a workflow system architecture that integrates a workflow-orchestrated execution planner and resource allocator, a workflow enact engine and a runtime system.
2. A workflow description language that addresses the issue of lacking scheduling support we mentioned herein.
3. A workflow scheduling (resource allocation and planning) algorithm that applies those advanced scheduling techniques we mentioned herein.
4. A simulation environment that closely models a real computational grid in those aspects relevant to workflow scheduling and the performance analysis of our scheduling algorithm under the simulated grid.

Along this work, a paper in the Journal of Grid Computing, and a few conference papers and book chapters have been published or in progress. A website, with contents being updated, has been set up to provide the latest technical documentation and software update, and the URL is "http://www.cs.uh.edu/~gracce".

## 1.3 Dissertation Organization

The detailed description of this work is organized in the rest of the dissertation. In the following chapter, grid computing model, grid architecture and grid resource management issues are introduced and discussed. Then in Chapter 3, we introduce grid scientific workflow applications and workflow systems. In Chapter 4, related efforts of grid workflow systems are studied and based on the studies, we motivate our work. Chapter 5 presents the features and technical details of the workflow description languages designed in this work. Following it in Chapter 6, we present our workflow scheduling architecture and algorithms for workflow resource allocation and execution planning. Chapter 7 shows our simulation results and analyzes the performance improvements by using our scheduler on workflow executions. We finally conclude this dissertation and discuss the future works in Chapter 8.

# Chapter 2

# Grid Computing

In this chapter, we introduce the grid computing model, grid architecture and discuss the resource management issues in grid environments.

## 2.1 Background

A new computing model is driven by the emerging applications, and is evolved or revolved from available computing models that are incapable to meet the requirements of these applications. The grid computing model follows this pattern: the foundation computing models are distributed computing and high performance computing; and the applications are "Big Science" applications [15].

### 2.1.1 Distributed Computing and High Performance Computing

Distributed computing represents a decentralized computing model using two or more computers communicating over a network to accomplish a common objective. The types of hardware, programming languages, operating systems and other resources of these computers may vary drastically. Distributing computing model ranges from loosely-coupled computing networks to tightly-integrated computing element pools for domain-specific applications. Distributed computing technology, such as client/server architecture [16] and CORBA [17], are widely used in enterprise applications, for example, web servers. One fundamental issue in distributed computing that is familiar to most of the computer users is to provide secure and transparent accesses to remote resources. The Transport Layer Security (TLS) and Secure Sockets Layer (SSL) [18], and the secure shell (SSH) [19], which provide secure communication protocol and local execution environment of remote machines, are the widely used and standard solution to this issue.

High performance computing (HPC) refers to the use of (parallel) supercomputers or computer clusters, computing systems linked together with commercially available interconnects, to solve large-scale science and engineering problems. Most of these applications are developed with Message Passing Interface (MPI) [20] or OpenMP [21] to exploit the parallel processing power of such systems. An HPC system normally has a batch scheduler, such as SGE [22], LSF [23] or PBS [24], installed to schedule user submitted jobs and to manage the parallel computing resources. The

500 most powerful publicly-known supercomputers in the world are listed in the TOP500 project [25], with update every 6 months.

### 2.1.2 Big Science and Global Science

After World War II, the style of scientific research developed defines the organization and character of much research in physics and astronomy and later in the biological sciences, and they are referred to as "Big Science" [26]. Big Science is characterized by the use of large-scale instruments and supercomputing facilities, and by involving scientists from multiple organizations. Some of the best-known Big Science projects include the high-energy physics facility CERN [27], the Hubble Space Telescope [28], and the Apollo program [29]. Big Science requires big computers, or supercomputers that are fundamentally different from personal computers in their ability to model enormous systems. Most of the supercomputers listed in TOP500 are built to solve such problems.

The popularity of the Internet and the World Wide Web [30] enables the direct remote resource access and the instantaneous information sharing around the globe. In a science research project, individuals across the world share information through WWW and emails in a similar way to organization-level collaborations. Researchers access remote computing resources and launch computational jobs through a secure and remote shell [19]. The science problems are globalized: individuals and resources from geographically distributed organizations can collaboratively work together without knowing each other.

## 2.2 Grid Problems and Grid Computing

The level of information sharing via Internet/WWW in the collaboration to solve the Big Science problems is mainly by file exchanging using FTP, HTTP, or email; and the mostly used approach to access computing resource is by individual's remote secure shell. Such sharing and resource access pattern cannot fully achieve the close collaboration at the organization level of real world. Consider this situation, excerpted from an article – "Big Computer for Big Science" [31]: A visiting neutron scattering scientist at ORNL sends data from her experiment to a supercomputer at SDSC for analysis. The calculation results are sent to Argonne National Laboratory, where they are turned into "pictures". These visualizations are sent to a collaborating scientist's workstation at North Carolina State University, one of the core universities of UT-Battelle, which manages ORNL for DOE.

To make their discoveries in this situation, scientists must interact with supercomputers to generate, examine, and archive huge datasets. To turn data into insight, this interaction must occur on human time scales, i.e., over minutes. But using the conventional resource sharing approach, multiple asynchronous mail exchange, manual or script-based file transfer and remote resource access are involved, and they occur in days or even weeks. Such a situation is very common in the globalized Big Science research and the Internet age; and the real and specific problem that underlies the situation is

> "the coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations. The sharing is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving and resource brokering strategies emerging in industry, science, and engineering. This sharing is, necessarily, highly controlled, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs." [32]

This is the well-known "Grid Problem" and grid computing is the emerging computing model to solve this problem. Grid computing enables the sharing, selection, and aggregation of a wide variety of resources including supercomputers, storage systems, data sources, and specialized devices that are geographically distributed and owned by different organizations for solving large-scale computational and data intensive problems in science, engineering, and commerce. A grid can be viewed as a seamless, integrated computational and collaborative environment, which is often referred to as a "grid cyberinfrastructure" [33].

By modeling a virtual computer architecture from many distributed computers that is able to distribute process execution across these computers, grids provide the ability to perform computations on large data sets, by breaking them down into many smaller ones, or provide the ability to perform many more computations at once that would be possible on a single computer. So a grid can be viewed as a distributed supercomputer system; the grid resources are geographically distributed and connected via Internet, and it is presented to users as a single high-performance

(virtual) computer system. A user will have access to the virtual computer that is reliable and adaptable to her needs, and the individual resources will not be visible to her.

Grid computing has been defined by different people in different literatures. The widely accepted two definitions, by the pioneers of grid computing, Ian Foster (also known as the father of grid computing), Carl Kesselman and Steve Tuecke are as follows:

> A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities. [1]

> Grid computing is concerned with "coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations." The key concept is the ability to negotiate resource-sharing arrangements among a set of participating parties (providers and consumers) and then to use the resulting resource pool for some purpose. [32]

### 2.2.1 Grid Categories

The concept of grid computing started as a project to link geographically dispersed supercomputers [34], but now it has grown far beyond its original intent. The grid infrastructure can benefit many applications, including collaborative engineering, data exploration, high-throughput computing, and distributed supercomputing. In

[35, 36], some major grid applications, deployed grid environments and research/development projects are collected. According to the applications a grid is targeted, current grid infrastructures are classified into different categories and in [1], those categories are discussed in details. In the following, the two most common categories are introduced:

**The computational grid** category denotes systems that integrate distributed high performance computing systems to provide higher computational capacity than the capacity of any constituent machine in the system. A grid in this category provides the aggregated resources to tackle problems that cannot be solved on a single system. Typically, applications that require distributed supercomputing are grand challenge problems such as weather forecasting or physics experiments.

**The data grid** category is for systems that provide an infrastructure for data and information processing and mining from data repositories that are distributed in a wide area network. It is specialized in providing an infrastructure of storage management and data access, replication, meta-data management and domain information retrieval. Target applications have datasets that approach or exceed a petabyte in size and involve data producing and consuming elements that are not just computing systems and human users, for example, a physics experiment collider.

## 2.3 Grid Architecture and Middlewares

The vision of grid computing is to create virtual dynamic organizations through secure, coordinated resource-sharing among individuals, institutions, and resources.

The realization of this vision requires the (re-)definition of and the solutions to the issues in distributed and high performance computing fields. A grid architecture is also required to identify fundamental system components, to specify the purpose and function of these components, and to indicate how these components interact with one another [32].

### 2.3.1 Issues in Grid Computing

Grid computing is a highly collaborative distributed computing model; solutions to traditional distributed computing issues, such as security and resource management, do not scale well in grid computing. Furthermore, grid computing introduces other issues, such as information services and data management. We summarize those grid issues in the following and refer readers to [37] or related literature for more detailed discussions.

**Security:** Most distributed computing systems use identity-based authentication and authorization control. As the typical case, a user is given a username and password for accessing a computing system; when she is ready to launch her applications, she logs into the system and submits the application jobs. In a grid environment, users or their agents simultaneously need accesses to multiple resources from different administrative domains that have different security mechanisms. This requirement creates several security issues [38]. The two typical ones are:

Single sign-on: A user should be able to authenticate once (e.g., when starting a computation) and initiate computations that acquire resources, use resources, release

resources, and communicate internally, without further authentication of the user.

Interoperability with local security solutions: While the grid security solutions may provide inter-domain access mechanisms, an access to a resource will typically be determined by a local security policy that is enforced by local security mechanisms. It is impractical to modify every local resource to accommodate inter-domain accesses.

**Resource Management:** Grid resources are from different administrative domains that have their own local resource managers and a grid does not have full control of these resources. When managing these resources, a grid resource management system should respect the usage policies enforced by local resource managers, and meanwhile, deliver user required quality of services and improve global resource usage. This dilemma, i.e., managing a resource without ownership, is referred to as "site autonomy" and "heterogeneous substrate" issues [39]. Another requirement for grid resource management comes from the fact that some grid applications, such as workflow, require resources to be allocated based on the application execution patterns and coordinated allocations of multiple resources simultaneously are necessary in order to deliver application-level quality of services. Also, resource management should be able to adapt application requirements to resource availability, particularly, when the requirements and resource characteristics change during execution. These issues are referred to as resource co-allocation and online-control [40].

**Information Services:** Information services play an important role in grids [41]. They indicate the status and availability of grid entities, i.e., compute resources, software libraries, networks, etc., without which there would be little coordination in such a dynamic environment as a grid. A grid information system should provide two

types of services, the accounting service and the auditing service. Grid accounting maintains historical information of resource status and job resource consumption for the purpose of performance prediction, resource allotment, charging and application performance tuning. Grid auditing provides runtime information of resource load status and application resource consumption for the purpose of resource allocation and resource usage control.

**Data Management:** Data-intensive, high-performance computing applications require the efficient management and transfer of terabytes or petabytes of information in wide-area, distributed computing environments. Data management is concerned with how to provide secure, efficient and transparent access to distributed, heterogeneous pools of data on wide-area grid resources [42]. In providing such services, grids should harness data, storage, and network resources located in distinct administrative domains, respect local and global policies governing how data can be used, schedule resources efficiently (again subject to local and global constraints), and provide high speed and reliable accesses to data.

**Standardization:** Grid computing is a highly integrated system and a grid is built from multi-purpose protocols and interfaces that address those fundamental issues described above. The grid vision requires protocols (and interfaces and policies) that are not only open and general-purpose but also standard. It is standards that allow to establish resource sharing arrangements dynamically with any interested party and thus to create something more than a plethora of balkanized, incompatible, non-interoperable distributed systems [43].

### 2.3.2 Grid Layered Architecture and Middlewares

> "*Any software problem can be solved by adding another layer of indirection.*"
>
> — Steven M. Bellovin

The general solutions to most of the grid issues are to aggregate the diversity and heterogeneousness of grid resources to create a uniform interface. The approach to such solutions falls into the "indirection" strategy above, that is, a layer-up architecture to hide and redirect those issues described before by developing a standard interface. We present the grid layered architecture in Figure 2.1, which is an extension of the architecture described in the grid anatomy paper [32], and it is similar to the one presented in [44]. This architecture is not implemented as a fully-integrated giant software; instead it integrates software utilities and tools that are already deployed or are being developed to solve specific issues. These utilities and tools are often referred to as "grid middleware". Most current efforts in grid research and development provide middleware solutions to the issues included in this architecture.

In Figure 2.1, the bottom "Fabric" layer represents different distributed resources from different administrative domains, such as supercomputers or parallel computing clusters, storage systems, scientific instruments and data resources. Those resources are managed by domain-specific resource managers and users access them via the non-standard interfaces of the resource managers or directly via the Operating System API. On a computing resource, the local resource manager, known as the local scheduler or batch scheduler, is responsible for allocating computing elements to

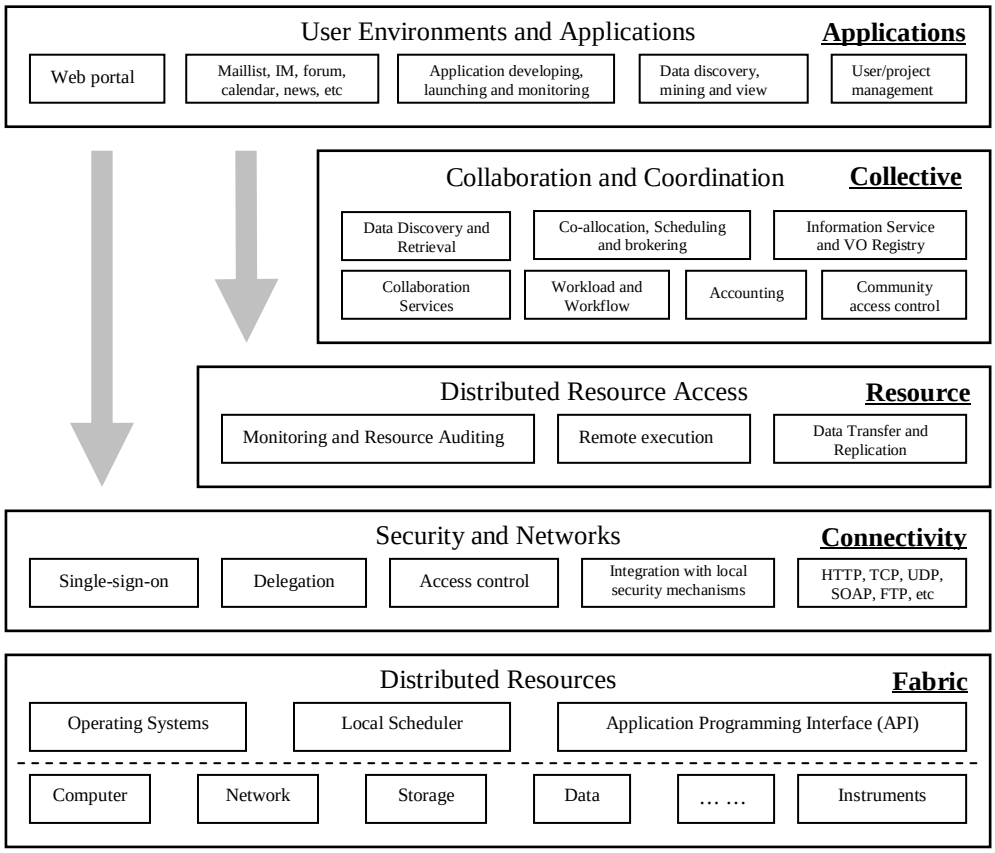



Figure 2.1: Grid Layered Architecture

users' jobs, launching them and monitoring their executions. Some well-known local schedulers include Sun Grid Engine (SGE) [22], Platform Load Sharing Facility (LSF) [23], Portable Batch System (PBS)[24] and IBM Load Leveler [45]. System administrators of these systems may also deploy a resource monitoring system, such as Ganglia [46], to report resource load/failure status.

From bottom-up, the second layer, the "Connectivity" layer provides the *core capabilities* of the grid architecture for sharing individual resources, namely, the security infrastructure and the networking protocol. The security infrastructure is

the middleware solutions to the grid security issues we mentioned before, and it is the core of the grid architecture. Currently, the standard solution is the Grid Security Infrastructure (GSI) [38], which uses public key cryptography as the basis for its functionality. The networking capabilities make use of the widely deployed standard protocols, such as HTTP, TCP/IP, etc., and recently, web service communication protocols, such as SOAP [47], also become part of this layer as the grid architecture moves to service oriented architectures [48].

The third layer, the "Resource" layer, provides interfaces for *single* resource sharing, which include remote computational resource access, data sharing and replication and resource monitoring and auditing. From this layer, users are able to access individual resources using standard grid interfaces. Middleware solutions in the "Resource" layer and the "Connectivity" layer address those grid issues we mentioned in the last section and provide a standard and uniform interface to build higher level services to access multiple resources collaboratively. The middlewares in these two layers serve as the software glue between the grid resources and the applications. The open source Globus Toolkit [49] is the *de facto* standard for providing these middlewares. We have a dedicated section next to introduce it.

While the "Resource" layer is focused on interactions with a single resource, the next layer in the architecture contains protocols and services that are not associated with any one specific resource but rather capture interactions across collections of resources [32]. For this reason, the next layer of the architecture is referred to as the "Collective" layer, the fourth layer in Figure 2.1. Typical services include resource

co-allocation, application scheduling on multiple resources, application workflow execution and monitoring, data discovery and retrievals. The work described in this dissertation belongs to this layer. Software developed in this layer varies according to the application needs of the upper "Applications" layer and it is hard to provide a general solution that fits the various types of applications.

Softwares in the "Applications" layer provide an end-user environment to use a grid for domain applications, for example, a portal interface to submit computation job or transfer files from web browser; an interface or tool to develop grid applications. The middleware functionalities in this layer also depend on the requirements of applications and users, and they are developed together with services in the "Collective" layer.

To build a grid, the development and deployment of a number of middlewares are required, including those for the standard services, such as security, information, data and resource allocation services, and those services for application development, execution management, resource aggregation, and scheduling. Again, choosing which middleware to develop and/or to deploy depends on the requirements of both the applications and users, and there is no standard configuration for setting up a general-purpose grid.

### 2.3.3 The Globus Toolkit

The Globus Toolkit provides a fundamental enabling technology for the grid, letting people share computing power, databases, and other tools securely across corporate,

institutional, and geographic boundaries without sacrificing local autonomy. It forms the basis of many on-going efforts to provide a computational grid, and is considered to be the standard infrastructure for grid computing. The Globus provides software services and libraries as a toolkit of functions for resource management, security, data management and information services. These functions may be used individually or together and we summarize them as follows:

- The Grid Security Infrastructure (GSI) [38] provides functions of single/mutual authentication, encrypted communication and credential delegation in grid interactions. These functions are essential for security as well as for accounting that a job is associated clearly with the user who submitted it. User access rights and permissions will be used to determine whether and with what priority a job is executed on the machine selected by the user or by a scheduler.

- Grid Resource Allocation and Management (GRAM) [39] provides resource management interfaces for a distributed resource. It accepts a job submitted to the grid via a script in Globus's own scripting language, and passes it on to the local scheduler on the selected target for actual queuing and execution, thereby acting as an interface to various kinds of heterogeneous resources.

- Data Management Services provides mainly two functions: data movement and data replication. For data movement, there are two services, GridFTP [50] and Reliable File Transfer (RFT) Service. GridFTP provides for the secure, robust, fast and efficient transfer of (especially bulk) data; and RFT service is a wrapper of file transfer utilities to allow byte streams to be transferred in a reliable

manner, for example, resume dropped connections. Globus data replication service [51] provides for the registration and lookup of replica information of grid data.

- The Monitoring and Discovery System (MDS) [41] is the grid information service. Using MDS, users are able to discover what resources are available and to monitor those resources. MDS also provides subscription interfaces to allow subscribers to be notified when the resource status is changed to a certain state.

Earlier versions of Globus (1.x, 2.x) adopted a toolkit-based approach to realizing a grid infrastructure. The latest Globus Toolkit 4 (GT4) is a major revision of this software that aims to provide Globus functionalities in the form of web services. It uses technologies such as the Simple Object Access Protocol (SOAP) [47] and Web Services Description Language (WSDL) [52] to enable the creation, management and discovery of grid resources as web service instances.

## 2.4 Resource Management in Grids

Grid systems are interconnected collections of heterogeneous and geographically distributed resources harnessed together to satisfy the different needs of grid users. One of the biggest issues in grid systems is how to schedule users' jobs or tasks onto multiple grid resources to achieve some performance goal(s), such as minimizing execution time, minimizing communication delays, maximizing resource utilization and/or load balancing [53]. From a system's point of view, this distribution choice becomes

a resource management problem, which is the process of identifying requirements, matching resources to applications, allocating those resources, and scheduling and monitoring grid resources over time in order to run grid applications as efficiently as possible [54].

In a grid environment, an end user submits to the grid resource management system (RMS), a grid scheduler in another term, a job to be executed along with some constraints like job execution deadline, or the maximum cost of execution. The function of the RMS is to take the job specification and from it estimate the resource requirements like the number of processors required, the execution time, and memory required. After estimating the resource requirements, RMS is responsible for discovering available resources and selecting appropriate resources for job execution, and finally schedules the job on these resources by interacting with the local resource management systems. This process makes decisions about two things: what resource(s) should be allocated for the job; and when the job should be launched on the allocated resource(s). Different scheduling policies and algorithms are applied during these two stages depending on the characteristics of the job, for example, a job with higher priority is considered earlier than lower-priority jobs.

In traditional computing systems, resource management is a well-studied problem. Resource managers such as batch schedulers, workflow engines, and operating systems exist for many computing environments. These resource management systems are designed and operate under the assumption that they have complete control of a resource and thus can implement the mechanisms and policies needed for effective use of that resource in isolation [54]. Unfortunately, this assumption does not

apply to the grid and there are mainly two issues that complicate the development of a grid scheduling system.

## Multiple layers of schedulers

A grid scheduler works on top of multiple local schedulers and the scheduling process is much more complex than in a local scheduler as grid scheduling makes resource allocation decisions involving resources over multiple administrative domains. We refer to this as the grid scheduling hierarchy (see Figure 2.2) and a grid scheduler as a grid *metascheduler* [55]. A grid scheduler discovers, evaluates and co-allocates resources for grid jobs, and coordinates activities between multiple heterogeneous schedulers that operate at local or cluster level. According to this definition, a grid scheduler has two main capabilities: the *scheduling* capability that allows it to co-allocate resources for applications requiring collaboration between multiple sites, and the *meta* capability to negotiate with local schedulers to satisfy global grid requests.

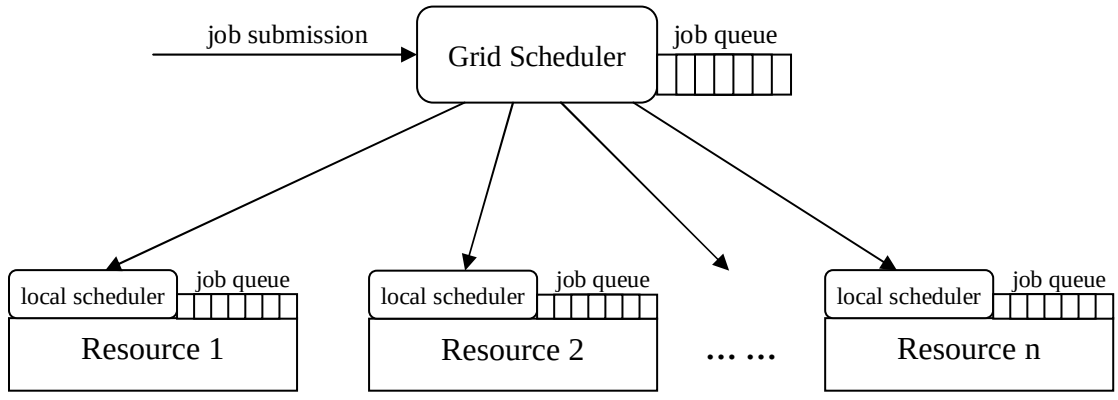



Figure 2.2: Grid Scheduling Hierarchy

## Resource Management without Ownership

One of the primary differences between a grid scheduler and a local scheduler is that the grid scheduler does not own the resources and therefore does not have control over them. The grid scheduler must negotiate with the local schedulers to request the resources that become available locally. Furthermore, the grid scheduler does not have control over the set of jobs submitted to it, or even know about the jobs being sent to the resources it is considering to use, so decisions that trade-off one jobs access for anothers cannot be made in the global sense. This lack of ownership and control causes the grid RMS not able to fully apply its resource management authority. Instead, it acts mainly as a *coordinator* between multiple local schedulers to make sure that the overall system performances are maintained at a level that is satisfied by users, while respecting the local policies of the resources.

# Chapter 3

# Scientific Workflow Applications

In this chapter, we introduce scientific workflow applications and how this type of application is specified and scheduled in computational grid environments.

## 3.1 Grid Scientific Workflow Applications

Grid scientific modeling and simulation, such as environment and earth science research [5, 56, 57], automobile collision simulation [58], space and astronomy exploratory [59], and physics collision experimentation [60, 4], are applications that make intensive use of numerical tools, require high compute power for the simulation and visualization, and entail transfer, storage and analysis of a huge amount of data. A simulation of these problems incorporates multiple dependent modules to be executed in predefined order on multiple computational resources. An uninterrupted simulation requires transfer and storage of the module data between the execution

resources in a timely manner. These applications are often referred to as scientific workflow applications [61].

A workflow can be defined as a collection of processing steps (also termed as tasks or modules), and the orders of task invocation or conditions(s) under which task must be invoked and/or the data-flow between these tasks. The execution order of two tasks forms a dependency relationship between them, and the two tasks are referred to as the parent task and the child task. A parent task must be completed before its child tasks start and the child task is executed using or based on the output of its parent tasks. Tasks without dependency relationships are referred to as sibling tasks. The relationships of the inter-dependent tasks are either data-dependencies, as of files between dependent tasks, or control flow, such as loops or conditional branches.

Figure 3.1 shows the graph representations of two workflows: a rectangle represents a workflow task or module, and an arrow represents the dependency relationship between the connected tasks. The workflow of Figure 3.1:A has only data dependencies. Each of the tasks, *T1* to *T7*, processes and produces data and the output of a task are used by its child tasks. The data dependencies are pre-defined in advance, thus determining the execution order of these tasks. We often refer to this type of workflow as static workflow, or data flow.

A workflow with control dependencies are referred to as dynamic workflow, or control flow. In this type of workflow, some dependency relationships are only established during workflow execution. For example, in the workflow of Figure 3.1:B, the module *md2* generates one of the three files, *F1*, *F2* or *F3*, in different loops. The

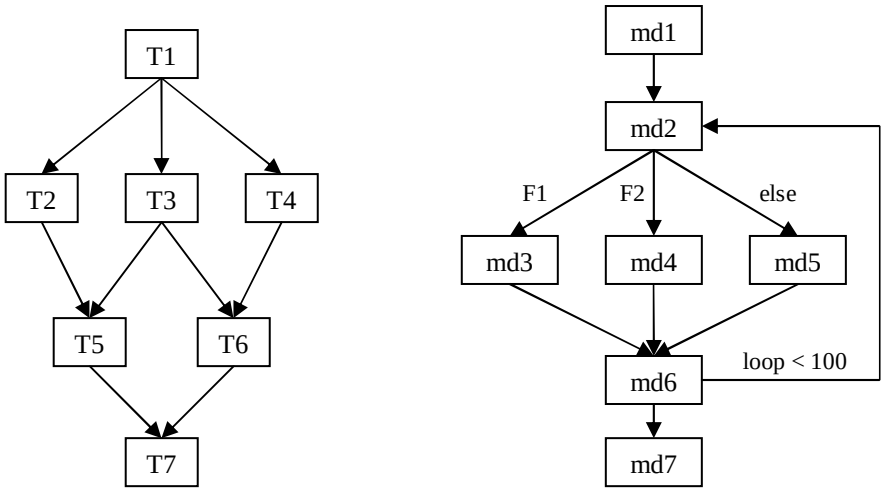



Figure 3.1: Workflow Examples

three files are processed by module *md3*, *md4* or *md5*, respectively. The loop count is 100. The dependency relationship between task *md2* and *md3* is established only when *md2* generates file *F1*. Scheduling dynamic workflows is much more complex than scheduling a static workflow because of the complexity in allocating resources for uncertain task relationships in advance. Also based on the fact that most of the scientific applications are for data processing and visualization that can be specified as static workflows, we focus our research on static workflows at this stage.

To execute a workflow, a user provides a workflow specification describing the required tasks and their dependency details, and submits this specification to a workflow scheduler for execution. The scheduler allocates resources for each of the workflow tasks and launches the tasks in the workflow-defined order. During the workflow execution, the scheduler is responsible for handling task dependencies, such as transferring the dependent files, and for launching tasks when their dependencies

are resolved. The system for specifying and scheduling workflows is often called a workflow management system.

## 3.2 Workflow Management Systems

The Workflow Management Coalition (WfMC) is an international organization to promote and develop the use of workflow through the establishment of standards for software terminology, interoperability and connectivity between workflow products [62]. The WfMC defines a workflow management system as "a system that completely defines, manages, and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic" [63]. With this definition, WfMC also defines a reference model for the architecture of a workflow system, as shown in Figure 3.2. The reference model describes the major components and interfaces within a workflow architecture. The two core components of any workflow system are the workflow enactment service, also termed as workflow engine, and the workflow process definition tools, often referred to as workflow description or modeling tools in literatures of grid computing. We refer readers to [63] for the details of this model and the discussion of the two core components in the context of grid computing are as follows.

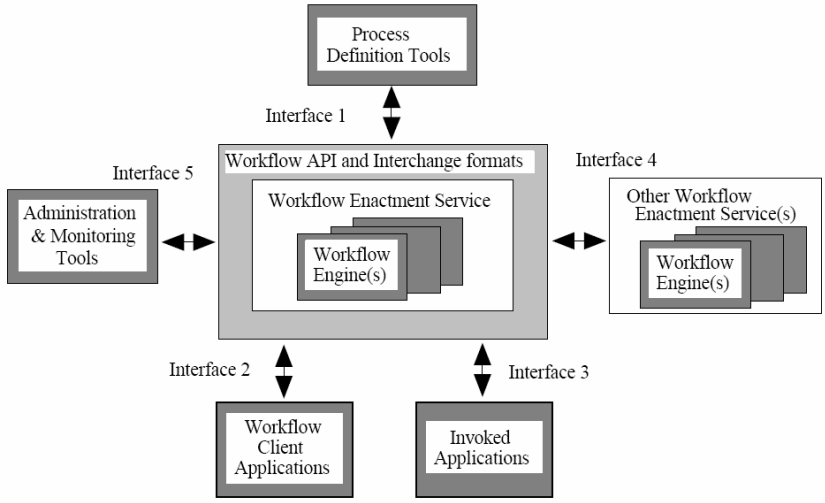



Figure 3.2: The Workflow Reference Model

### 3.2.1 Workflow Specification

Workflow description is the basis of the establishment and execution of grid workflows and provides a meta-model to describe physical processes. The widely-used methods for workflow description are the workflow description language and the GUI graphical specification. A workflow description language defines standard syntax and semantics for specifying workflow tasks and their relationships; thus it provides an abstract and formalized representation of the complex workflow structures in text format. In GUI graphical specification method, the definition or inclusion of tasks is by mouse click or drag-and-drop; and the definition of control flow and data flow between tasks is accomplished by connecting task icons with specialized arrows specifying the control dependencies or data dependencies. Although intuitive and easy-to-use for end users, the drag-drop graphical interface method still relies on a description language to represent the workflow internally. So it is considered as a user interface

of a workflow language, and in most cases, it is less powerful than the language itself. In developing a workflow system, a graphical interface is developed after finalizing the definition of the workflow language itself.

There are mainly two styles to define a workflow language, control-flow style and data-flow style. In control-flow style, the execution order of workflow tasks are specified explicitly. It is either in the order of specification statements or specified using imperative-programming syntax, such as, "if", "while", "sequential", "parallel", "fork" and "join". For example, in the control-flow description of the workflow of Figure 3.3, the task execution order is {A, parallel{B, C, D}, parallel{E, F}, G}; and the parallel{B, C, D} means that task B, C, and D can be executed in parallel, but they are all after task A. The formalized model for control-flow style is the Petri Net [64], and in [65, 66], the authors show how Petri Nets are used for workflow description.

Using data-flow style in a workflow description language, the workflow consists of a set of tasks, one or more start tasks, and a set of dependency relationships. The dependency relationships determine the execution order of the workflow tasks. Data-flow description is the Directed Acyclic Graph (DAG) representation of the workflow and it is able to specify most of the current scientific workflow applications. Compared to the control-flow style, only dependency relationships between relevant tasks need to be specified, as shown in Figure 3.3. For big workflows, the control-flow style makes it very complex and error-prone for users to reason about the sequential or parallel execution orders of the tasks. But using data-flow style, the task execution order can be easily determined by the workflow scheduler based on the

task dependency relationships. So the DAG-based data-flow style is a much easier method for users to describe static workflow than the Petri Net-based control-flow style.

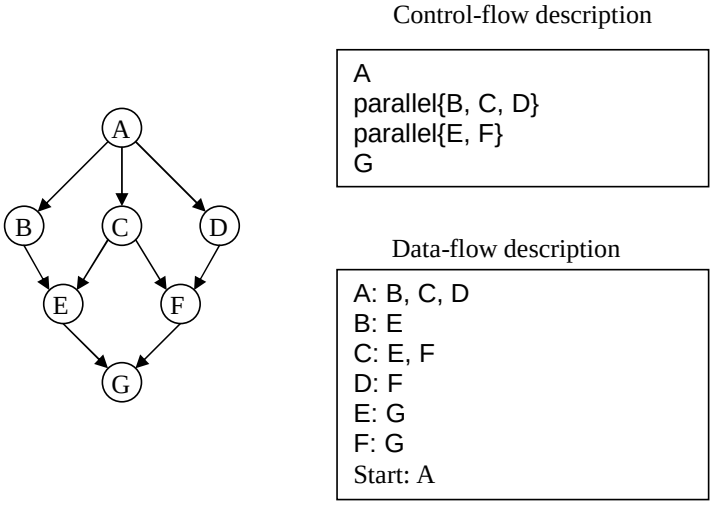



Figure 3.3: An Example Workflow Specification

A workflow language that allows users to specify the control flow and data flow among tasks by itself is not enough. In grid environments, there is a need for a language to provide information to the workflow scheduler to help the resource allocation decision-making process. The workflow task specification should provide resource request information of the task execution, and/or scheduling hints such as past execution information, as well as the details of how to launch the task. The workflow scheduler uses such information to discover and select suitable resources for the workflow task.

### 3.2.2 Workflow Scheduling

The workflow scheduler, also termed as workflow enact engine, provides a run-time environment that executes the workflow tasks and coordinates the task execution and dependency handling to make sure they are processed in the right order. It can be as simple as performing a topological sort [6], and then launches the workflow tasks according to the topological order. In computational grid environments, this issue becomes much more complex than it appears to be. For example, we must allocate multiple resources to workflow tasks according to the task dependency relationships and reduce the queue waiting time for the tasks that are submitted to resources for execution. So a grid workflow scheduler should have at least two capabilities: first, resource allocation, which distributes tasks onto multiple resources, and second, task execution and coordination, which submits tasks to the resource's local schedulers in the right order, and handles task dependencies. In a DAG workflow, the task dependencies determine the order of task submission, which is the topological order of the workflow DAG. In this order, the earliest start-time of each task can be calculated easily, as long as we know when the workflow itself should be started. Since the launching time of each task is determined by the workflow dependency relationships and the workflow launching time, workflow scheduling is mainly concerned with how to allocate resources for each task according to the task execution order. A resource should be allocated to a task right after its dependencies are resolved and no delay should be incurred because of the unavailability of resources.

#### 3.2.2.1 Resource Allocation Strategies

There are two strategies for allocating resources for workflow tasks in a workflow scheduler: just-in-time allocation and look-ahead allocation. In the former, the scheduler discovers resources and makes allocation decisions for a task when its dependencies are resolved. But it may face situations, especially in high-load grid environments, when no resource is available for a ready task. Workflow execution has to be put on hold, while the scheduler continues to search for a resource.

In look-ahead allocation, the scheduler plans the execution of all or a subset of tasks and makes allocation decisions for them in advance. When a resource is allocated for a task, the scheduler is confident or can guarantee that the resource will be available. So there are no, or only short, delays waiting for resources to be available when a task is ready, even in high-load environments. The planning process requires the prediction of both workflow execution and resource availabilities. It involves resource discovery, negotiation and reservation for the workflow tasks.

#### 3.2.2.2 Workflow-Orchestrated Co-Allocations

A workflow execution requires multiple resources to be simultaneously available in order to deliver the best performance. For example, tasks without dependencies should be scheduled on different resources for concurrent execution; tasks with data dependency should be allocated on the same resource to save data transfer cost. A scheduler should be able to co-allocate these resources based on the workflow dependency relationships. Grid resource co-allocation involves resources across different

administrative domains that have different allocation policies. As we mentioned before, a grid scheduler does not have control over the resources and requests resources via the local schedulers. A workflow scheduler in computational grids should have the ability of coordinating resource requests from different local schedulers with regards to the workflow task relationships and an allocation decision made should minimize the unnecessary delays in execution due to the unavailability of resources for ready tasks.

#### 3.2.2.3 Network and Data-Aware in Scheduling

In a single computing system, such as a cluster, which normally has shared file systems and high-speed interconnection between nodes, resources being managed are mainly CPUs and memory, and the local scheduler does not need to consider the impacts of network bandwidth and data size on the application performance. A resource allocation decision is purely about how many CPUs, and on which nodes, should be allocated for a particular job. In a grid environment, a workflow task requires input/output data to be staged-in/out from/to different locations connected by the Internet that does not guarantee network bandwidth. The time spent on data transfer may become a significant part of the task round time if a huge amount of data is transferred on a slow network. When choosing computational resources for the workflow tasks, a grid scheduler should mediate the time for data transfer compared with the task execution time; and should also check whether the candidate resources have enough storage space for the input and output data. So compared with a local scheduler, a grid scheduler must be data and network aware and takes

the network bandwidth and workflow data size into consideration in making resource allocation decisions.

# Chapter 4

# Related Work and Motivation

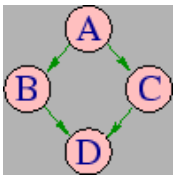
In this chapter, we review some related efforts in the research and development of workflow description languages and workflow schedulers in the area of grid and distributed computing. Based on our study, we motivate our work to develop an advanced scheduling system for grid scientific workflow applications.

## 4.1 Related Work: Grid Workflow Description Languages

There have been a large number of efforts to develop a workflow description language to support different types of workflow in different application area. In this section, we review those that are relevant to grid computing and web service architecture.

### 4.1.1 Condor DAGMan

Condor [67] is a resource management system for distributed computing resources, such as clusters of PCs, and the Directed Acyclic Graph Manager (DAGMan) [8] is the workflow scheduler in Condor. DAGMan uses DAG as the data structure to represent job dependencies, and a data flow is specified by a parent-child relationship. The following code fragment is the Condor DAG specification of a diamond workflow. Condor does not support for specifying intermediate files between tasks, and users have to specify data transfer through the preprocessing and postprocessing script associated with each job. DAGMan supports partial workflow execution in the workflow description files. If a job is marked as DONE, then this job is not scheduled.




```
# Filename: diamond.dag
#
# Job spec in the format of ''Job <jobName> <JobCondorScript>''
Job  A  A.condor
Job  B  B.condor
Job  C  C.condor
Job  D  D.condor
# the preprocessing and postprocessing scripts for jobs
Script PRE  A top_pre.csh
Script PRE  B mid_pre.perl  $JOB
Script POST B mid_post.perl $JOB $RETURN
Script PRE  C mid_pre.perl  $JOB
Script POST C mid_post.perl $JOB $RETURN
Script PRE  D bot_pre.csh
# dependency relationships
PARENT A CHILD B C
PARENT B C CHILD D
# Number of retries if a job fails
Retry  C 3
```

Condor DAG Example

### 4.1.2 The Chimera Virtual Data Language

The Chimera Virtual Data System (VDS) [68] is a set of tools for data-processing workflow management, including expressing, executing, and tracking the results of workflows. The workflow description language of Chimera is called Virtual Data Language (VDL), and it is a data-flow style language. In VDL, a set of application programs are described as *transformations* (TR) and the executions of transformations are described as *derivations* (DV). Derivations produce or consume data files, which are described as data objects. The following code fragment is an example of a VDL transformation, one of its derivation and a data processing job. We refer interested readers to [69] for the syntax details. In Chimera, VDL definitions are stored in a catalog that provides for the tracking of the provenance of all files derived by an application. Chimera VDS contains the recipe to produce a given logical file, and the dependencies between derivations in terms of these data files constitutes the application abstract workflow in the form of a DAG of program execution steps.

```
TR t1( output a2, input a1,
       none env="100000",
       none pa="500" ) {
   app vanilla = "/usr/bin/app3";
   arg parg = "-p "${none:pa};
   arg farg = "-f "${input:a1};
   arg xarg = "-x -y ";
   arg stdout = ${output:a2};
   profile env.MAXMEM = ${none:env};
}
```

```
DV t1(
   a2=@{output:run1.exp15.T1932.summary},
   a1=@{input:run1.exp15.T1932.raw},
   env="20000", pa="600" );

export MAXMEM=20000
/usr/bin/app3 -p 600 \
   -f run1.exp15.T1932.raw -x -y \
   > run1.exp15.T1932.summary
```

Chimera VDL Example

### 4.1.3 Taverna XScufl

Taverna [9] is a workflow environment for grid life-science applications. Taverna uses an extended Scufl [70], XScufl, as the workflow description language. The Scufl language is essentially a data-flow centric language. In Scufl, a logical service, an individual step within a workflow, is called *processor*, which can be regarded as a function of some set of input data to a set of output data. A set of *data links* connect data source processors to data destination processors. Taverna developed a set of processor plug-ins that handle the data flow on data links, for example, A WSDL Scufl processor implemented by a single Web Service operation described in a WSDL. The fields of the Web Service operation request message correspond to the input ports and the fields of the return message to the output ports.

### 4.1.4 ASKALON and Karajan

ASKALON and Karajan workflow systems both define a control-flow style language to describe application logics [71, 10]. In this style, the task execution order, which is implied by task dependency relationships, is explicitly specified using *sequential* and *parallel* syntax. For example, for the diamond workflow we used before, the specification could be "{*sequential A,* {*parallel B, C*}*, D*}". It is read: task $A$, task group {*parallel B, C*} and task $D$ must be executed in sequential order, where tasks $B$ and $C$ can be executed concurrently. In this way, the dependency relationships between tasks are implicitly constrained by the task execution order that is specified by using the two syntactics. In addition to these two, ASKALON and Karajan

introduce other imperative programming structures, such as *for*, *if*, *switch*, etc., to specify complex control-flow logics. Karajan also allows the definition and use of variables and functions in the specification.

### 4.1.5 Triana

Triana [11] provides a graphical environment to enable the composition of workflow applications through mouse input. In the Triana workflow language, a component, the unit of execution, is a Java class with an identifying name, input and output "ports", a number of optional name/value parameters and a single process method. Triana uses both data-flow and control-flow in workflow description. In the case of data-flow, data arriving on the input "port" of the component triggers execution, and in the case of control-flow, a control command triggers the execution of the component. The execution of workflow within Triana is decentralized; data or control flow "messages" are sent along communication "pipes" from sender to receiver.

```
<tool>
  <name>Tangent</name>
  <description>Tangent of the input data</description>
  <inportnum>1</inportnum>
  <outportnum>1</outportnum>
  <input>
    <type>triana.types.GraphType</type>
    <type>triana.types.Const</type>
  </input>
  <output>...</output>
  <parameters>
    <param name="normPhaseReal" value="0.0" type="userAccessible"/>
    <param name="normPhaseImag" value="0.0" type="userAccessible"/>
    <param name="toolVersion" value="3" type="internal"/>
  </parameters>
</tool>
```

A Triana Component Definition

### 4.1.6 Others

YAWL [72] is a workflow language built upon two main concepts: workflow patterns and Petri Nets [64]. It was developed by taking Petri Nets as a starting point and adding mechanisms to allow for more direct and intuitive support of different workflow patterns. Similar to the Karajan and ASKALON approach, it is a control-flow style language.

Business Process Execution Language (BPEL) [73] is an XML-based workflow definition language to describe enterprise business processes in web services. In BPEL, a workflow step is described using WSDL. For scientific applications, either extensions to the language or the wrapping of the applications is needed to use BPEL.

Semantics web [74] standards, Resource Description Framework (RDF) [75] and Web Ontology Language (OWL) [76], aim to provide another structuring and description framework that allows data to be integrated in a much larger-scale than what current HTML-framework provides. The general-purpose semantic web standards are very abstract and additional vocabularies need to be defined for a specific field.

### 4.1.7 Discussion

Most existing workflow description languages focus on being expressive enough to describe the data flow and control flow of workflow structures. They lack the features for users to provide resource request information to support resource allocations

for workflow tasks by the scheduler. For example, the resource request information for multiple dependent tasks could be specified to support workflow-orchestrated resource co-allocation and execution planning. One workaround is to make the resource multi-request using another method, such as RSL [77] that is independent of the workflow description. However, users have to derive the resource multi-request from the workflow task relationships, and the derivation is a complex reasoning process. The workflow scheduler has to refer to two specifications, one for resource allocation and one for workflow scheduling, which complicates its decision making process during resource allocation and scheduling.

Aside from scheduling support, a workflow language should provide ease of use and should support workflows with a plethora of different requirements. Some typical features include helping handle errors produced before, during and after task executions and dependency handling, support for partial workflow specification, and support for application-specific utilities for task launching, termination and restarting. We have found that these features are not, or are only partially supported in the workflow language development efforts.

## 4.2 Related Work: Workflow and Grid Application Scheduling

In most workflow systems, such as DAGMan [8], Taverna [9], Karajan [10], Kepler [78] and Triana [11], the fundamental mechanism of the workflow scheduling is to

perform breadth-first traversal of the workflow structure and then launch workflow tasks based on the traversal order. They do not have resource allocation capability and resource information for workflow tasks are specified in the workflow description. Thus, they are not the focus of our study. In the rest of this section, we review those efforts that have advanced workflow scheduling and resource allocation capabilities.

### 4.2.1 ASKALON Workflow Scheduler

The ASKALON workflow scheduler [71, 79] provides three algorithms for workflow application scheduling: Heterogeneous Earliest Finish Time (HEFT), a genetic algorithm, and a just-in-time algorithm acting like a resource broker. In the HEFT algorithm, the scheduler considers allocating a resource to a workflow task that can complete the task the fastest. The genetic algorithm is a look-ahead scheduling one that applies techniques, such as performance prediction, workflow partition and resource reservation, in the workflow resource allocation and execution planning process. When dealing with control flow in which some dependencies can only be determined at run time, the algorithm makes assumptions about whether to handle the dependency or not in scheduling. Incorrect assumptions are resolved by appropriate run-time adjustments such as undoing existing optimizations and rescheduling. But in the performance evaluation of these algorithms, ASKALON made several assumptions that are unrealistic in computational grid environments. For example, it assumes a resource is allocated upon request, thus the grid scheduling hierarchy issue is skipped.

### 4.2.2 Gridbus Workflow Scheduler

Gridbus workflow scheduler [14, 80] applies a look-ahead and budget/deadline-driven workflow scheduling algorithm. A workflow submitted by a user for execution has costs associated with each task and the user also specifies a budget and a deadline that must be met. The algorithm partitions the workflow into subworkflows, each of which has a budget and a deadline. As along as the budgets and deadlines of all the subworkflows are met, the budget and the deadline of the workflow are met. But this algorithm targets a utility grid that assumes a service level agreement between service provider and service consumer, which implies that there is no queue waiting in the workflow execution, neither the impact of resource load on the workflow performance. For computational grid environments with a grid scheduling hierarchy, these implications are not applicable.

### 4.2.3 Pegasus and GridFlow

Pegasus [13] is the Chimera workflow manger that takes the abstract workflow and constructs a job execution DAG with scheduling information from the application DAG logic. This process includes querying Globus MDS to find resources for computation and data movement, and querying a Globus replica location service to locate data replicas. It finally produces a concrete workflow conforming Condor DAG specification, and submits the specification to DAGMan for execution. Other than suffering the same limitation of Condor DAG, it is also not allowed to specify resource request information to aid the Pegasus resource allocation process.

In GridFlow [81], a workflow is executed according to a simulated schedule. If large delays occur in sub-workflows, the rest or all of the workflow may be sent back to the simulation engine and rescheduled. The concept of a simulated schedule is similar to the execution plan or schedule. But GridFlow does not address resource co-allocation and reservation issues in the simulated schedule.

### 4.2.4 Grid Scheduling Related Work

There are also many systems that address specific issues of grid co-scheduling. Globus GRAM [39] and RSL [77] are the early, de-facto standards for providing solutions for secure job execution in metacomputing environments. DUROC [82] is an early effort to address the issues of resource co-allocation in the context of Globus and RSL. Globus GARA [40], Maui Silver [83] and the architecture defined in [84] introduce advanced reservation into the GRAM co-allocation architecture [85]. SNAP [86], which extends Globus' GRAM and GARA, proposes a service negotiation protocol for grid scheduling.

The K-Grid scheduler [87] is a performance-oriented resource allocation service for knowledge discovery and data mining applications. It predicts the computational and I/O cost for each allocation and makes the best-possible decisions based on this estimation. But the K-Grid scheduler does not reserve resources for applications and relies on the grid resource discovery services to find the best available resources.

The Community Scheduler Framework (CSF) [88] implements a set of grid services which provide basic capabilities for grid job submission and resource reservation.

These services, developed as wrappers for some local scheduler utilities, provide a good starting point to develop a brokerage system. But CSF services only cater for single executable jobs and lack functionalities for grid co-scheduling.

Maui Silver [83] is an advanced reservation-based grid scheduler which allows a single job to be scheduled across distributed clusters. Silver relies on the local scheduler to specify and coordinate the job workflow, which limits it usage to simple workflow applications.

Nimrod/G [89] is a resource management system with a focus on computational economy and schedules tasks based on their deadlines and budgets. Nimrod/G also addresses issues of scheduling single jobs, and does not address the requirements of workflow applications.

MARS [90] proposes an on-demand scheduler which discovers and schedules the required resources for a critical-priority task to start immediately. MARS uses a forecasting strategy to predict runtime resource parameters, such as queue lengths, utilization, etc.

### 4.2.5 Discussion

Based on our study, we found there are mainly two types of schedulers, application-level schedulers and system-level schedulers. An application-level scheduler [91] manages the execution of a single application and improves the performance of the application by utilizing the available resources. A system-level scheduler, often termed as a resource manager, manages all the applications and resources of a system with goals

to improve the overall system utilization and load balancing. An application-level scheduler works on top of one or multiple system-level schedulers, and has to coordinate with the system-level schedulers to utilize the resources. So the development of an application-level scheduler must take into account the existence of system-level schedulers, and the scheduling policies and issues in the underlying system-level schedulers. In our context, the workflow scheduler is an application-scheduler and the local scheduler is a system-level scheduler.

However, in reviewing those related efforts, we have found that the system-level scheduling and grid resource management issues have not been taken into enough account in the workflow scheduler development. As a result, when evaluating workflow scheduling algorithms, the assumptions made are unrealistic for real computational grid environments. For example, they assume that a resource is allocated upon request or a reservation is always granted upon request. Those assumptions are not unreasonable for a dedicated grid environment with low resource load and that guarantees resource availability and high network bandwidth. But in a grid that includes resources linked by the unreliable Internet and used for a variety of grid applications, the data transfer, resource load and queue waiting contribute significantly and negatively to the workflow execution performance. So those assumptions do not apply in those grid environments. As a result, the scheduler developed may not adapt to real and dynamic computational grid environments.

On the other hand, we have noticed that the efforts to develop a general purpose grid scheduler provide solutions to the grid resource management issues, such as resource co-allocation, performance prediction and resource reservation. But most of

those solutions are developed and used in a system-level scheduler and very few efforts study and use those techniques in grid workflow systems. We believe, if applied in a workflow scheduler, those advanced scheduling techniques can greatly improve the overall workflow execution performance.

To apply those techniques into the workflow scheduling process, an extensible workflow system architecture is required to integrate those techniques into the workflow scheduler. Most of current workflow schedulers are developed as integrated softwares to provide the functionalities of resource allocations, scheduling, dependency handling and run time coordinations. As a result, the additions and/or plug-ins of new features to the workflow system require either the redesign or extensively patch work. This approach results in an inextensible and unmodularized system software that is hard for further integrations.

## 4.3 Motivation: The GRACCE Framework

Based on our discussion of the related works to develop workflow description languages and workflow schedulers, we summarize several issues in the following aspects that are not well addressed in current efforts in order to provide support for automatic execution of workflow applications in computational grid environments:

- **Scientific workflow scheduling:** The development and evaluation of a workflow scheduler in computational grid environments should take into account the existence of the grid resource management issues that introduce performance

overhead of wokflow execution. The design of scheduling algorithms should apply those techniques for grid system-level scheduling in order to improve the workflow execution performance.

- **Workflow description languages:** Current workflow description languages have enough expressiveness capabilities for various types of application, but lack support for workflow resource allocations and scheduling in dynamic grid environments. To provide that, a workflow description language must be extended to allow task resource request to be specified in workflow description.

- **Integration support:** Current workflow scheduler and grid resource management system are designed and developed independently. To support the integration of new features with the workflow system, the workflow system architecture should be design to be extensible and modularized. For example, the change of resource allocation and scheduling algorithms should not introduce the redesign of the workflow run time system. The architecture should provide an extensible framework that can accommodate the diverse range of requirements imposed both by the applications and by the underlying grid systems.

Driven by the needs of workflow application support in grid environments, the GRACCE (Grid Application Coordination, Collaboration and Execution) project [92] was proposed to address the requirements above. The vision of GRACCE is to provide domain scientists with an integration framework for building a customizable grid application environment, from the management of a workflow application and

its dataset, to the automatic execution and viewing of results. In the GRACCE framework, end users are only required to provide descriptions of their workflow applications. GRACCE is responsible for allocating grid resources to workflow tasks, placing tasks on resources for execution, monitoring them, and returning the results back to users as desired. More specifically, the solutions provided by the GRACCE framework to address those requirements are as follows:

- **Application modeling and description:** GRACCE provides a modeling language, GAMDL, to address the limitation of lacking resource allocation support in current efforts. GAMDL also has several advanced features that are not available in other efforts, and is the basis for the integration of a grid scheduler and a workflow system in GRACCE.

- **The GRACCE workflow scheduler:** The GRACCE scheduler targets on scientific workflow applications in computational grid environments. It addresses the issues of grid resource management and workflow scheduling in the workflow scheduling and planning algorithms. It also applies advanced scheduling techniques, such as resource co-allocation, performance prediction and execution planning in the algorithms.

- **A modular and integration framework:** GRACCE is developed as an integration framework that includes modularized workflow system, runtime systems and a scheduler. It allows each subsystem to be designed and developed independently and allows new techniques or algorithms to be plugged in or turned on/off during the workflow scheduling process on the run-time.

One workflow application that we have been working on, the Air Quality Forecasting (AQF) [93], is the original motivation of our GRACCE framework. AQF application is an integrated computational model for regional and local air quality forecasts, and is composed of three subsystems: the PSU/NCAR MM5 weather forecast model, the SMOKE emission system, and EPA's CMAQ chemical transport model. An AQF execution is a computational sequence of the three subsystems with increasing resolution and decreasing geographical boundaries. Figure 4.1 illustrates the workflow of a nested 2-day forecasting operation over a single region of interest by a three-domain computation. The 36km domain computation provides coarse forecast data over the continental USA, the 12km domain provides data across the south central USA, and the 4km domain forecasts air quality across a smaller geographic region.

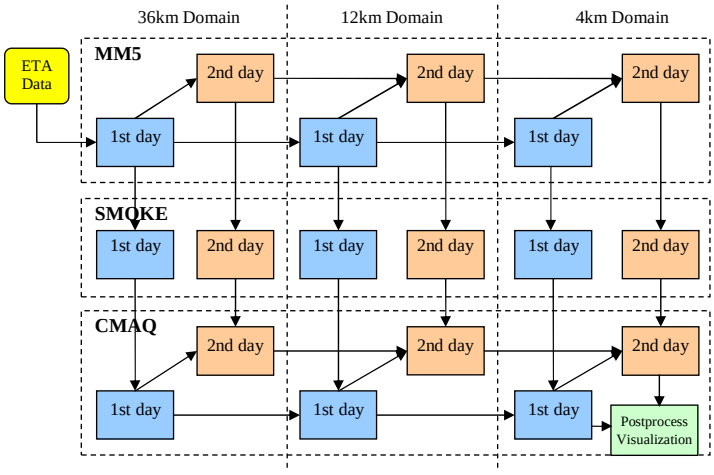



Figure 4.1: AQF Application Workflow

AQF application represents most of current grid workflow applications and has the same requirements for grid middlewares if deployed in grid environments. The

original design of the GRACCE framework is the result of the collaboration work between AQF users and us during the process of enabling AQF on the UH campus grid [94, 95, 93, 96]. So although targeting AQF application at the beginning, the GRACCE framework provides a set of middlewares for the deployment, integration and execution of grid scientific workflow applications.

# Chapter 5

# Grid Application Modeling and Description Language

In this chapter, we present the high-level abstract language we developed for grid workflow application description, the Grid Application Modeling and Description Language (GAMDL).

## 5.1 GAMDL Capabilities and Features

GAMDL was designed to address the major limitation of current workflow description languages in support for resource allocation for workflow tasks. It is feature-rich, and is more powerful and flexible than those related efforts we have studied in the last chapter. We summarize the capabilities and features of GAMDL as follows:

- GAMDL supports the description of both data-flow and control-flow logic (loops and conditional branches) at a high level of abstraction.

- By separating the description of application logic and execution workflow, GAMDL supports the specification of partial workflows and reoccurring workflows without introducing additional complexities.

- GAMDL allows multiple jobs to be associated with a workflow module. The scheduler may choose the most suitable one according to the hardware and software environment of the allocated resource. For example, a workflow module has two binary codes, one for Intel X86 architecture and one for PowerPC architecture. GAMDL allows them to be specified as two jobs and lets the scheduler choose the right one based on the architecture of the resource allocated for this module.

- GAMDL supports the definition of nested or hierarchical workflows, i.e., a workflow contains another workflow.

- To support resource co-allocation of workflow dependent modules, GAMDL associates the specification of resource requests and execution schedules with the module specifications to. When allocating resources for a module, the scheduler evaluates resources based on not only the module itself, but also on its parent, child and sibling modules. For example, given a parent module and a child module, the scheduler considers allocating the same resource for them so that there is no need to transfer the intermediate files on the network.

- GAMDL allows for the specification of application-specific scripts for various

purposes, such as preprocessing, postprocessing, checking the state of job execution, cleaning temporary file, killing jobs, etc.

- To improve the usability and GAMDL documents's readability, GAMDL introduces several new features. For example, it allows similar modules to be easily described using multi-value properties, and the description document is structured by using entity uid and uid references.

## Implementation

GAMDL is XML language-based and the GAMDL syntax is developed as a set of XML-Schema [97]. XML is the most widely used modeling language for workflow description in grid computing and has a very rich set of development tools. XML-Schema is used to define a set of rules to which an XML document must conform in order to be considered "valid". As a W3C standard, it provides a rich data model that allows us to express sophisticated structures and constraints used in GAMDL. The use of XML-Schema for GAMDL helps us easily develop a GAMDL parser using the open source XML development library, Apache XMLBeans [98]. XMLBeans binds XML data with Java objects through the schema of the data expressed in XML-Schema. In our example, after we have designed the GAMDL XML-Schema, the XMLBeans compiler takes the GAMDL schema and generates Java codes that access a GAMDL document. All the data types, XML documents and elements in GAMDL are mapped to Java classes. Using these automatically generated codes, we can easily develop a GAMDL parser in pure Java language.

A simple GUI interface, GRACCE Appdesc that can display a GAMDL workflow as graph, is developed. In Figure 5.1, we show a snapshot of its interface. The Appdesc is also the interface of our workflow system. From the Appdesc, users can load a GAMDL application and workflows from XML documents, schedule them and monitor their executions using the workflow graph. We refer interested readers to the GRACCE website [92] for more information.

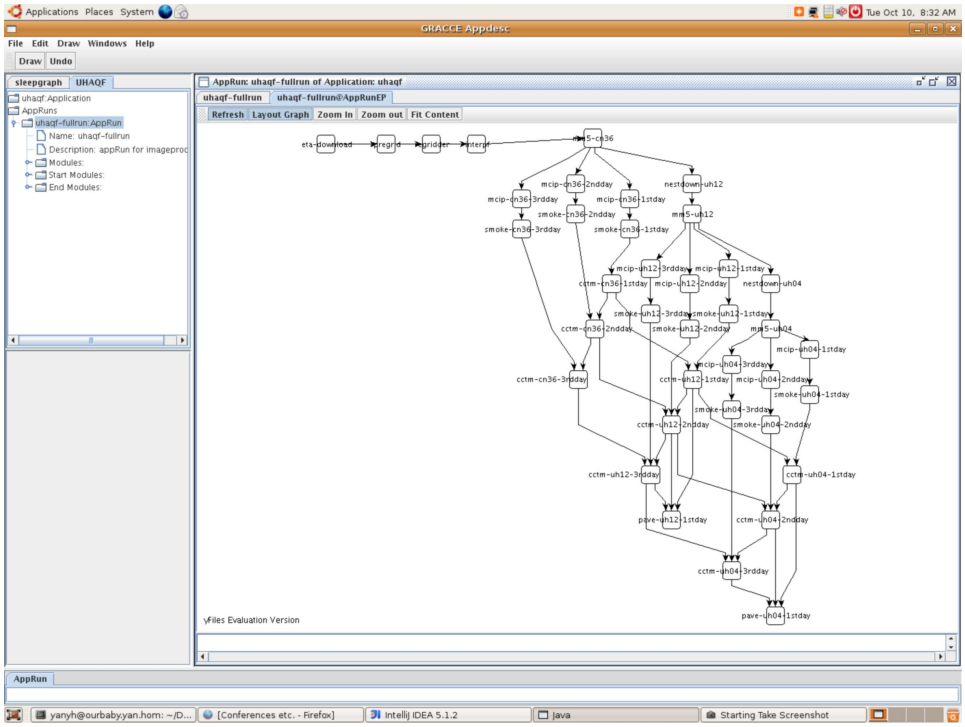



Figure 5.1: GRACCE GUI Interface – Appdesc

## 5.2 GAMDL Entities and Core Concepts

GAMDL is a data-flow style language, and the description of a workflow application includes the specification of the application entities and the specification of the relationships and dependencies between entities. In this section, we discuss the main entities used in GAMDL.

### 5.2.1 Execution Specification

In distributed and heterogeneous environments, a computational job is often specified in an abstract, platform-independent format by users and runtime systems translate the specification to a platform-dependent launching script that is used to create the execution instance. The main issue in developing a specification language for different level of users and runtime is how to flexibly define the properties of an execution to such an extent that different levels of users can specify them in the abstraction level only they need. To support such flexibility, GAMDL splits an execution specification into two parts, the execution schedule and the execution configuration. An execution schedule includes information about *when and where* the executable is launched, such as its start time and host name. An execution configuration includes information about *how* the executable is launched, such as its launcher (e.g., a bash shell), directory, arguments and environment variables. In this separation, the platform-independent information in the execution configuration is supplied by users. When the scheduler allocates a resource for the execution, it fills in the execution schedule. Based on the architecture of the allocated resource, the runtime

system fills in the platform-dependent part of the execution configuration. Another field that is part of an execution specification is the resource request. The resource request lists the resource details required for this execution, such as the number of CPU and the memory size. Based on this information, the scheduler makes resource allocation decisions and generates an execution schedule. An execution is of ExeType type in GAMDL schema, as shown in the following code fragment.

```
<xsd:complexType name="ExeType">
  <xsd:sequence>
    <xsd:element name="executable" type="xsd:string"/>
    <xsd:element name="location" type="xsd:anyURI"/>
    <xsd:element name="version" type="gamdl:VersionType"/>
    <xsd:element name="exeType" type="gamdl:ExecutableTypeEnumeration"/>

    <xsd:element name="exeSchedule" type="gamdl:ExeScheduleType"/>
    <xsd:element name="exeConfig" type="gamdl:ExeConfigType"/>
    <xsd:element name="resrcReq" type="gamdl:ResourceReqType"/>

    <xsd:element name="supportArch" type="gamdl:SysArchType"/>
    <xsd:element name="requiredLibrary" type="gamdl:LibraryType"/>
    <xsd:element name="docPage" type="xsd:anyURI"/>
  </xsd:sequence>
  <xsd:attribute name="uid" type="gamdl:UidType"/>
</xsd:complexType>
```

ExeType Schema

In the ExeType schema, most of the fields (elements or attributes in XML terms), such as executable and location elements, are self-explanatory. The execution schedule is specified in the exeSchedule field of ExeScheduleType type, the execution configuration in the exeConfig field of ExeConfigType type and the resource request in the resrcReq field of ResourceReqType type.

```
<xsd:complexType name="ExeScheduleType">
  <xsd:attribute name="startTime" type="xsd:dateTime"/>
  <xsd:attribute name="host" type="xsd:string"/>
  <xsd:attribute name="hostCPUList" type="xsd:string"/>
  <xsd:attribute name="numCPU" type="xsd:integer"/>
  <xsd:attribute name="memSize" type="xsd:nonNegativeInteger"/>
```

```
  <xsd:attribute name="retry" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="ResourceReqType">
  <xsd:attribute name="startTime" type="xsd:dateTime"/>
  <xsd:attribute name="endTime" type="xsd:dateTime"/>
  <xsd:attribute name="maxNumCPU" type="xsd:integer"/>
  <xsd:attribute name="minNumCPU" type="xsd:integer"/>
  <xsd:attribute name="maxWTime" type="xsd:long"/>
  <xsd:attribute name="maxCPUTime" type="xsd:long"/>
  <xsd:attribute name="maxMem" type="xsd:nonNegativeInteger"/>
  <xsd:attribute name="minMem" type="xsd:nonNegativeInteger"/>
</xsd:complexType>
```

ExeScheduleType and ResourceReqType Schemas

The code fragment above shows the schemas of ExeScheduleType and ResourceReqType. The retry field in ExeScheduleType is used to specify how the system should rerun this schedule if the last execution fails. The retry string is defined in format of "[Integer]:[Integer]:[Integer][+|x|e]". The first integer is the maximum number of retries; the second one is the first interval (in second) and the base to calculate the interval between each retry; The last [Integer][+|x|e] is used to define how the interval is calculated in each retry, e.g., 4+ means the next interval is the current one plus 4; 2x means the next interval is 2 times the current one; 2e means that next interval is the current one powered by 2. For example, "retry="5:2:2x"" means that the scheduler should retry this execution up to 5 times if the first one fails, and the intervals between each try, in seconds, are 2, 4, 8, 16, 32.

```
<xsd:complexType name="ExeConfigType">
  <xsd:sequence>
    <xsd:element name="launcher" type="xsd:string"/>
    <xsd:element name="launcherArgu" type="xsd:string"/>
    <xsd:element name="directory" type="xsd:string"/>
    <xsd:element name="arguments" type="xsd:string"/>
    <xsd:element name="env" type="gamdl:EnvironmentType"/>
    <xsd:element name="stdin" type="xsd:string"/>
    <xsd:element name="stdout" type="xsd:string"/>
    <xsd:element name="stderr" type="xsd:string"/>

    <xsd:element name="validator" type="gamdl:ScriptType"/>
    <xsd:element name="preprocesser" type="gamdl:ScriptType"/>
```

```
      <xsd:element name="postprocesser" type="gamdl:ScriptType"/>
      <xsd:element name="cleaner" type="gamdl:ScriptType"/>
      <xsd:element name="killer" type="gamdl:ScriptType"/>
      <xsd:element name="doctor" type="gamdl:ScriptType"/>
    </xsd:sequence>
  </xsd:complexType>
```

ExeConfigType Schema

In the ExeConfigType type, the six fields of ScriptType type are used to specify the application-specific scripts for different purposes. The validator script is used to check whether a completed execution generates the expected results or not; for example, whether the data in the output files are correct or not. While the execution's exit code from the operating system tells whether it has completed or failed, the use of the validator script allows users to validate the execution using application-specific methods or algorithms. The preprocesser script and the postprocesser script are launched before the executable is launched and after the execution is complete. The cleaner script is used to clean out the temporary files after execution. The killer is the script used to kill the execution process and its child processes if it has any. This is very useful for terminating all the processes of an MPI program. The doctor script is used to check the health of the execution and it is often called by the run time system.

Using these six scripts and the retry feature mentioned before, users can design robust and automated failure detection and restart functions for an execution. For example, if a parallel MPI application runs much longer than its past executions, it is very likely that the application's parallel processes have lost the state of internal communications and it hangs forever. The runtime system detects this using the provided doctor script and kills the hanging processes using the killer script. It then

validates whether the expected results are generated using the validator script because the application may have lost communication state at the end of the execution, e.g., when closing the communication sockets after all the application data are generated. If it is a failed execution, the runtime system invokes the cleaner script to clean the temporary and incomplete output. According to the retry pattern configured in the retry string, the runtime system restarts the execution. These features are requested by a group of application users [93] and are very useful for applications that run frequently, e.g., daily. Users do not need to intervene very often to deal with those errors that can be recovered from by the system.

### 5.2.2 Module and Job Specification

The core entity in GAMDL for describing a workflow application is "module". A module is an application component to accomplish certain application goals, typically, processing input files and generating the required output files. A module is associated with one or more jobs and their executions are all able to accomplish the module's goals. A common case of having multiple jobs is when the module code has been compiled into several binaries for different platforms. Each of these binaries can be specified in one job. A module may have multiple input and output files. All the module jobs should consume these same input files and generate the same output files. The schema of the GAMDL module, the ModuleType, is shown in the following.

```
<xsd:complexType name="ModuleType">
  <xsd:sequence>
    <xsd:element name="inputFiles" type="gamdl:UidRefSetType"/>
    <xsd:element name="outputFiles" type="gamdl:UidRefSetType"/>
    <xsd:element name="jobSpec" type="gamdl:ModuleJobSpecType"/>
    <xsd:element name="dftExeConfig" type="gamdl:ExeConfigType" />
```

```
    <xsd:element name="dftExeSchedule" type="gamdl:ExeScheduleType"/>
    <xsd:element name="dftResrcReq" type="gamdl:ResourceReqType"/>
    <xsd:element name="metaJobSpec" type="gamdl:MetaJobSpecType"/>
  </xsd:sequence>
  <xsd:attribute name="uid" type="gamdl:UidType"/>
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="description" type="xsd:string"/>
  <xsd:attribute name="dftJobIndex" type="xsd:integer"/>
</xsd:complexType>
```

ModuleType Schema

The inputFiles and the outputFiles fields in the schema are self-explanatory. The jobSpec and the metaJobSpec fields are for specifying the two types of GAMDL module jobs, the regular job and the meta-job. A regular job is a single execution and it is of ModuleJobSpecType type. A meta-job, of MetaJobSpecType type, is a workflow. The use of a meta-job allows the construction of nested workflows where a workflow contains another workflow. In a MetaJobSpecType-typed meta-job, the workflow is specified by the appRun field of AppRunType type. We shall discuss this schema later on. The schemas for the two types of jobs are shown in the following.

```
<xsd:complexType name="ModuleJobSpecType">
  <xsd:sequence>
    <xsd:element name="exeSchedule" type="gamdl:ExeScheduleType"/>
    <xsd:element name="exeConfig" type="gamdl:ExeConfigType"/>
    <xsd:element name="resrcReq" type="gamdl:ResourceReqType"/>
  </xsd:sequence>
  <xsd:attribute name="index" type="xsd:integer" use="required"/>
  <xsd:attribute name="uid" type="gamdl:UidType" use="optional"/>
  <xsd:attribute name="name" type="xsd:string" use="optional"/>
  <xsd:attribute name="description" type="xsd:string" use="optional"/>
  <xsd:attribute name="exeSpecUid" type="gamdl:UidType"/>
</xsd:complexType>

<xsd:complexType name="MetaJobSpecType">
  <xsd:sequence>
    <xsd:element name="argu" type="gamdl:VariableType"/>
    <xsd:element name="appRun" type="gamdl:AppRunType" />
    <xsd:element name="appRunFile" type="xsd:anyURI" />
  </xsd:sequence>
  <xsd:attribute name="index" type="xsd:integer" use="required"/>
  <xsd:attribute name="uid" type="gamdl:UidType" use="optional"/>
  <xsd:attribute name="appRunUid" type="gamdl:UidType" use="required"/>
  <xsd:attribute name="name" type="xsd:string" use="optional"/>
  <xsd:attribute name="description" type="xsd:string" use="optional"/>
```

```
</xsd:complexType>
```
Schema for Module Job Specification

In the ModuleJobType schema, the exeSpecUid attribute references an ExeType-typed execution that is already defined. The three elements, the exeSchedule, the exeConfig and the resrcReq, are for specifying the job-specific execution details and resource request. Although the execution specification referenced by the exeSpecUid attribute also provides these, allowing them to be provided here gives users the option of supplying module-specific details for an execution, and allows them to customize an execution in one module without changing the execution itself. For the same reason, the dftExeConfig, dftExeSchedule and dftResrcReq fields in the ModuleType schema are for specifying the corresponding default for jobs. If a field is not given in the job specification, the corresponding default is used. Also, since a module may belong to one or more applications or workflows, these fields may be given in the application and workflow description, too, in order to provide custom execution details specifically for that application or the workflow.

Finally, we note that both regular and meta module jobs can be identified via an index, the index field of the above schemas. An index is a sequence number that orders the job specifications. The dftJobIndex field in the ModuleType schema tells the scheduler which job should be considered first when allocating resources.

### 5.2.3 Multi-Value Property and Entity Uid

In applications, such as AQF, multiple modules use the same executable with different, but similar execution details and input/output files. For example, there are 6 different CMAQ modules in the AQF workflow shown in Figure 4.1 for forecasting air quality in three domains and for two days. The differences in the use of the six modules are in the specification of the execution configuration and input/output file names. To describe them one by one is tedious work. Moreover, changes in the specification of the CMAQ execution may requires changes in that of all the six CMAQ modules. This has proved to be an error-prone editing process and the readability of the resulting description is poor. We have introduced additional language support for these to improve the usability and readability of workflow specifications. These new features in GAMDL are provided via the "multi-value property" and the "entity unique id".

A **Multi-value property** (mvproperty), as its name implies, is a property that may have multiple values. It is defined as $mvpName = \{v0, v1, ..., vn\}$, and is referenced by $\$mvpName$. $\#mvpName$ denotes the number of values defined. A reference to $mvpName$ replicates the referencing sentence $\#mvpName$ times; in each replica, the reference is replaced with a distinct one of its values. For example, if we define $dmsz = \{36k, 12k, 4k\}$, and $day = \{d1, d2\}$, the sentence $aqf\text{-}mm5\text{-}\${dmsz}\text{-}\${day}$ represents all six instances ($\#dmsz * \#day$) of the AQF MM5 modules in Figure 4.1. In an XML document, the replication of an mvproperty reference is on an element basis. When the GAMDL parser encounters a reference to an mvproperty, it replicates the nearest outer element that contains the reference. This element

is called the containing element of the mvproperty reference. The parser does not recursively process the same references in the child element of the containing element; instead, it instantiates all references to the mvproperty in a replicate element with the same value.

In the GAMDL description for the AQF application, the mvproperties are defined as follows:

```
md={mm5, smoke, cmaq}   # Three AQF subsystems
dmsz={36k, 12k, 4k}     # Three AQF domains
day={d1, d2}            # Two-day forecasting
vdmsz={12k, 4k}         # The visualized domain
```

The following code fragment describes the six CMAQ modules in Figure 4.1 using these mvproperties.

```
<module uid="cmaq-${dmsz}-${day}">
  <inputFiles>
    <ref uid="cmaq-${dmsz}-${day}-in1"/>
    <ref uid="cmaq-${dmsz}-${day}-in2"/></inputFiles>
  <outputFiles>
    <ref uid="cmaq-${dmsz}-${day}-out1"/>
    <ref uid="cmaq-${dmsz}-${day}-out2"/></outputFiles>
  <jobSpec name="cmaq-${dmsz}-${day} job spec">
    ... ...
  </jobSpec>
</module>
```

CMAQ Module Definition

The **entity unique id (uid)** is an attribute to uniquely identify an entity, for example a job or a module, as shown in their schemas presented above. It is provided by users when specifying an entity. To include a defined entity in another entity's specification, the user only needs to reference it by its uid. In the code fragment above for CMAQ module definition, the module uid is defined to be "cmaq-${dmsz}-${day}" which covers the six modules. The input/output files are specified by the uid references of the corresponding files. This is the typical usage of uids in GAMDL,

that is, the application entities are defined in several documents, one for each type of entity; and the documents for specifying entity relationships and workflows use these entities via their uid reference. In this way, changes in the entity specification do not necessitate changes in the documents for higher-level application specification. It enables the creation of well-organized document structures that match human approaches to organizing this information. The use of uids also enables the re-use defined entities in other applications; it can be used as the key or foreign key reference when storing an entity in an RDBMS or XML database.

## 5.3 GAMDL Application and Workflow

GAMDL models a domain problem as an application and an execution of the application as a workflow. It allows both data-flow and control-flow logics to be described using data-flow style syntax.

### 5.3.1 Application and Workflow Description

In GAMDL, application and workflow are two different concepts. An application is a high-level model of a domain problem from the viewpoint of an end user; a workflow is an execution instance of the application. An application definition includes the specifications of all the application entities and of the entity relationships and dependencies. A workflow definition specifies which application entities are included in the workflow execution and with which module(s) the execution starts; the runtime

workflow is then constructed based on the high-level application specification. One advantage of this separation is that it allows users to specify multiple and different workflows of an application based on their needs without defining a new application each time. The support for subworkflow and partial workflow is thus provided naturally from this separation. A GAMDL application is defined in an application document and a GAMDL workflow in an appRun document. The structure of the application document is illustrated in the following code fragment.

```
<application name="" uid="" description="">
  <version ... />
  <appExes>
    <ref uid="" />
      ... ...
  </appExes>
  <appDataFiles>
    <ref uid="" />
      ... ...
  </appDataFiles>
  <appModules>
    <ref uid="" />
      ... ...
  </appModules>
  <appMdRships>
    <pCnRshipSet>
      ... ...
    </pCnRshipSet>
  </appMdRships>
</application>
```

GAMDL Application Document Structure

As this code fragment demonstrates, the collection of the definition of all application entities is via uid references. These references are organized in three child elements, appExes, appDataFiles, and appModules, each for one of the three types of application entities respectively, i.e., execution, file, and module. The descriptions of module relationships and dependencies are in the appMdRships element: we discuss its details below. The next code fragment shows the *AppRunType* schema for the appRun document.

```
<xsd:complexType name="AppRunType">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string"/>
    <xsd:element name="dftMdJobIndex" type="xsd:integer"/>
    <xsd:element name="dftMdJobResrcReq" type="gamdl:ResourceReqType"/>
    <xsd:element name="dftMdJobExeConfig" type="gamdl:ExeConfigType"/>
    <xsd:element name="dftMdJobExeSchedule" type="gamdl:ExeScheduleType"/>

    <xsd:element name="mdRun" type="gamdl:MdInAppRunType"/>
    <xsd:element name="startMd" type="gamdl:UidRefSetType"/>
        ... ...
  </xsd:sequence>
  <xsd:attribute name="uid" type="gamdl:UidType"/>
  <xsd:attribute name="appUid" type="gamdl:UidType"/>
  <xsd:attribute name="startTime" type="xsd:string"/>
        ... ...
</xsd:complexType>

<xsd:complexType name="MdInAppRunType">
  <xsd:sequence>
    <xsd:element name="jobResrcReq" type="gamdl:ResourceReqType"/>
    <xsd:element name="jobExeConfig" type="gamdl:ExeConfigType"/>
    <xsd:element name="jobExeSchedule" type="gamdl:ExeScheduleType"/>
  </xsd:sequence>
  <xsd:attribute name="ref" type="gamdl:UidType"/>
  <xsd:attribute name="jobIndex" type="xsd:integer"/>
</xsd:complexType>
```

AppRunType

In the workflow schema, the application specification is referenced via the appUid attribute. The startTime string states when the workflow should be launched: it allows a sophisticated cron job format for reoccurring executions of the workflow. The mdRun element of MdInAppRunType type is for specifying the inclusion of application modules. It includes the module uid reference, the job index of the module, the execution details, i.e., the execution configuration and the execution schedule, and the resource request. This schema allows users to provide workflow-specific execution details and a module job index. As the ModuleType schema, the three elements, dftMdJobResrcReq, dftMdJobExeConfig and dftMdJobExeSchedule, are used to specify the default values of resource request and execution details of the workflow modules. These default values are used for a module whenever its mdRun element does not

provide them. Lastly, the starting module(s) of the workflow are specified in the startMd element as uid references to the workflow modules. The following code fragment is an example of an AQF workflow definition.

```
<appRun uid="uhaqfrun" appUid="uhaqf" startTime="2005-07-16T15:23:15">
  <mvproperty file="uhaqf.mvproperties"/>
  <modules>
    <ref uid="eta-download"/>
    <ref uid="mm5-${dmsz}-${day}"/>
    <ref uid="smoke-${dmsz}-${day}"/>
    <ref uid="cmaq-${dmsz}-${day}"/>
    <ref uid="postv-${vdmsz}-${day}"/></modules>
  <startMd><ref uid="eta-download"/></startMd>
</appRun>
```

An AQF Workflow Definition

In the workflow specification, users do not need to specify the module relationships and dependencies. They are all given in the appMdRships element of the application document. In the following two subsections, we explain how data-flow and control-flow logics are described in GAMDL.

### 5.3.2 Data-Flow Description

GAMDL models the application data-flow as a DAG, and captures both the dependency relationships between modules and the intermediate files associated with these relationships. A dependency relationship can be defined in either parent-children (PCn) pattern or child-parents (CPs) pattern. A PCn relationship, specified as a PCnRship element in GAMDL, has a parent module and one or more child modules, and a CPs relationship, as a GAMDL CPsRship element, has a child module and one or more parent modules. Intermediate files in a relationship are specified as pipes, one file per pipe. A **pipe** has a pipeIn element and a pipeOut element; the pipeIn

element represents the piped output file of the parent module, and the pipeOut element the piped input file of the child module. The next code fragment is part of the appMdRships element in the AQF application document. It describes the PCn relationships between the SMOKE and CMAQ modules, and between the CMAQ modules for the first day and the second day forecasting of the AQF application in Figure 4.1.

```
<appMdRships>
  <mvproperty file="uhaqf.mvproperties"/>
    ... ...
  <PCnRship parentMdUidRef="smoke-${dmsz}-${day}">
    <childMd uidRef="cmaq-${dmsz}-${day}">
      <viaPipe pipeIn="smoke-${dmsz}-${day}-out1"
        pipeOut="cmaq-${dmsz}-${day}-in1"/>
    </childMd>
  </PCnRship>

  <PCnRship parentMdUidRef="cmaq-${dmsz}-d1">
    <childMd uidRef="cmaq-${dmsz}-d2">
      <viaPipe pipeIn="cmaq-${dmsz}-d1-out1"
        pipeOut="cmaq-${dmsz}-d2-in1"/>
    </childMd>
  </PCnRship>
    ... ...
</appMdRships>
```

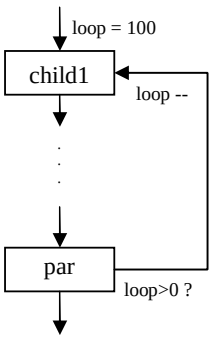
PCn Relationship Examples

### 5.3.3 Control-Flow Logic Description

GAMDL allows the specification of control-flow logics, such as loops or conditional branches, by introducing conditional pipes and variables. A **conditional pipe** associates a pipe with a Boolean *if* condition which will be evaluated after the module completes execution. If it evaluates to *true*, the pipe is processed; otherwise, it is not. If the conditions of all pipes in a relationship are evaluated to *false*, runtime dependency is not established and the child module will not be executed. A **GAMDL variable** is a <*name*, *value*> pair associated with an *if* condition. A new value can

only be assigned to the variable if the associated *if* condition evaluates to *true*; if no *if* condition is specified, an assignment is always made. If the *value* being assigned is in the form of *value1:value2*, *value1* is assigned if the *if* condition is *true* and *value2* is assigned otherwise.

In GAMDL, complex flow controls are achieved by the proper assignment of variable values and reasoning on the conditions associated with pipes and variables. The run time system can assign values to variables before a module's execution (in a preAssign element) and/or after a module's execution (in a postAssign element). The condition associated with a variable assignment or a pipe is permitted to reference system environment variables as well as GAMDL variables defined in other modules. In the following *for* loop example, the *child1* module postAssigns 100 to the loop index (*loop*) if the *loop* variable has not yet been defined (which means this iteration is entering the loop), or *${loop} - 1* in each subsequent iteration. In the *pCnRship* of *par* module and *child1* module, a null pipe (using */dev/null* file) is specified with an *if* condition as *${loop} > 0*. In each iteration, if the condition evaluates to "*true*", the pipe is established and control passes to the *child1* module.




```
<module uid="child1">
  ... ...
  <postAssign name="loop" value="${loop}-1:100"
    if="defined(loop)"/>
</module>

<PCnRship parentMdUidRef="par">
  <childMd uidRef="child1">
    <viaPipe inFileUidRef="/dev/null"
      outFileUidRef="/dev/null" if = "${loop} > 0"/>
  </childMd>
</PCnRship>
```

*For* Loop

In Section 5.5, we give a more detailed example showing how a workflow with

loops and conditional branches is specified using conditional pipes and variables. We want to note here that the control-flow specification introduces additional complexity in reasoning about the module execution order, and is best used for the specification of simple control-flow logics.

## 5.4 GAMDL's Support for Resource Co-Allocations

GAMDL introduces the specification of two types of information about a module to aid a workflow scheduler when it is making resource allocation decisions; the resource request information of a module or workflow job, and the historical and profiling information about a job's execution on a resource.

### 5.4.1 Resource Request Specification

As we mentioned before, the resource request of an execution is specified using the ResourceReqType type, shown below. It lists the resource details required by the execution. Two attributes we want to note here are the host and hostCPUList. These two fields are optional. If they are specified, it tells the scheduler that dedicated resources are requested; otherwise, the scheduler allocates resources for the execution and fills in these two fields. This allows users to manually allocate resources for some workflow tasks and specify them in the execution specification. Using those dedicated allocations as hints, the workflow scheduler can make much better decisions for tasks for which users do not specify dedicated resources.

```
<xsd:complexType name="ResourceReqType">
  <xsd:attribute name="startTime" type="xsd:dateTime"/>
  <xsd:attribute name="endTime" type="xsd:dateTime"/>
  <xsd:attribute name="host" type="xsd:string"/>
  <xsd:attribute name="hostCPUList" type="xsd:string"/>
  <xsd:attribute name="maxCPU" type="xsd:integer"/>
  <xsd:attribute name="minCPU" type="xsd:integer"/>
  <xsd:attribute name="maxWTime" type="xsd:long"/>
  <xsd:attribute name="maxCPUTime" type="xsd:long"/>
  <xsd:attribute name="maxMem" type="xsd:nonNegativeInteger"/>
  <xsd:attribute name="minMem" type="xsd:nonNegativeInteger"/>
</xsd:complexType>
```

ResourceReqType

GAMDL associates the resource specification with a workflow module, which provides a natural solution for specifying resource multi-requests according to the workflow. When allocating resources for module jobs, a scheduler makes resource allocation decisions based on the module dependency relationships, for example, sibling module jobs are allocated concurrent resources if possible. The scheduling process is thus orchestrated by the workflow. Under Globus RSL [77], users have to explicitly specify the resource multi-requests for the purpose of resource co-allocation [39, 82]. These resource multi-requests have to be constructed manually according to the dependency relationships of the workflow modules. While it would be possible to associate the RSL of module jobs with the workflow, the ability to specify resource multi-requests in RSL is then lost.

### 5.4.2 Job Execution Profile

The **Execution Profiles** of a module job in GAMDL are the historical and profiling information about the job executions on different grid resources. They provide the scheduler the historical information on observed executions in order to predict

its future execution. For applications like AQF that run every day under similar scenarios, it is very easy to predict the execution of a module job on the resource on which it has been executing. Based on such prediction, the scheduler can make much better resource co-allocation decisions for module jobs. Also statistical analysis, data normalization and scaling may be performed on the historical data for other purposes, such as improving resource usage.

The GAMDL schema for a job execution profile is ExeProfileType, shown in the following. GAMDL organizes the execution profiles of a module job based on the resources the job has run on, with one profile for each resource. A module job may have been executed several times on a resource; each execution is described as an execution scenario, consisting of a list of the resources consumed, of ExeScenarioType type. In each profile, scaling algorithms are defined to calculate the possible resource usage based on the available scenarios.

```
<xsd:complexType name="ExeProfileType">
  <xsd:sequence>
    <xsd:element name="name"/>
    <xsd:element name="resourceName"/>
    <xsd:element name="scenarios"
                 type="gamdl:ExeScenarioType"/>
    <xsd:element name="CPUTimeScalingAlgorithm"
                 type="gamdl:ScalingAlgorithmType"/>
    <xsd:element name="VMemScalingAlgorithm"
                 type="gamdl:ScalingAlgorithmType"/>
          ...
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ExeScenarioType">
 <xsd:sequence>
  <xsd:element name="name" type="xsd:string"/>
  <xsd:element name="numberOfCPU"/>
 <xsd:element name="hostCPUList"/>
  <xsd:element name="startTime"/>
  <xsd:element name="endTime"/>
  <xsd:element name="cpuTime"/>
  <xsd:element name="wallTime"/>
  <xsd:element name="memSize"/>
```

```
    <xsd:element name="swapSize"/>
    <xsd:element name="vmSize"/>
      ...
   </xsd:sequence>
  </xsd:complexType>

  <xsd:simpleType name="ScalingAlgoNameType">
    <xsd:restriction base="xsd:string">
     <xsd:enumeration value="LINEAR"/>
     <xsd:enumeration value="INVERSELINEAR"/>
     <xsd:enumeration value="SEQURE"/>
     <xsd:enumeration value="EXPONENTIAL"/>
    </xsd:restriction>
  </xsd:simpleType>
```

Job Execution Profile Specification

## 5.5 A GAMDL Example of a Workflow with Complex Control-Flow Logic

In this section, we use a workflow example in Figure 5.2 to show how complex control-flow logic, such as loops and conditional branches, is described in GAMDL. In the example workflow, the module *md2* generates different output files (*F1*, *F2* or others) in different loops, and these files are processed by module *md3*, *md4* or *md5*, respectively. The loop count is 100.

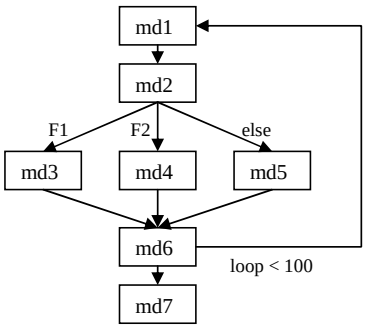



Figure 5.2: A Workflow with Loops and Conditional Branches

In the GAMDL description shown below, module *md1* postAssigns a *loop* variable, whose initial value is *100* and stride is *-1*. The module *md2* postAssigns two variables, *F1recent* and *F2recent*. *F1recent* is set to *true* if file *F1* is generated by *md2* in the last execution, otherwise *F1recent* is set to *false*; *F2recent* is handled similarly with respect to file *F2*. The pipe condition for *md3-md2* CPsRship is set to "*pipe(F1)* && ${*F1recent*}", which is evaluated to *true* if *F1* is generated in the last execution and is available for piping in. The *if* conditions for the *F2* pipe in *md4-md2* CPsRship and the else-pipe in *md5-md2* CPsRship are similar to the *F1* pipe. Loop control is specified in *md1-md6* CPsRship of *md1* and *md6* using a null pipe with condition "${*loop*}<*100* ".

In this example, GAMDL uses condition functions, such as *generated(F1)*, in a condition string. A **condition function** is a regular function (binary or script) that returns a Boolean value; it should not make any modification to its externals. In the following specification, the *pipe(fileName)* function checks whether a file can be piped in or not. The *generated(fileName)* function checks whether the module execution generates the specified file; the *defined(variableName)* function checks whether a variable is defined or not.

```
<application name="LoopCon Example" uid="loopcon" ...>
  <appModules>
    <module uid="md1">
      <postAssign name="loop" value="${loop}-1:100" if="defined(loop)"/>
    </module>
    <module uid="md2">
      <outputFiles>
        <ref uid="F1"/>
        <ref uid="F2"/>
        <ref uid="Fx"/></outputFiles>
      <postAssign name="F1recent" value="true:false" if="generated(F1)"/>
      <postAssign name="F2recent" value="true:false" if="generated(F2)"/>
    </module>
    <module uid="md3"><inputFiles><ref uid="F1"/></inputFiles></module>
    <module uid="md4"><inputFiles><ref uid="F2"/></inputFiles></module>
```

```
    ...
  </appModules>

  <appMdRships>
    <cPsRshipSet>
      <CPsRship childMdUidRef="md2">
        <parentMd uidRef="md1">
          <viaPipe> ... </viaPipe></parentMd></CPsRship>
      <CPsRship childMdUidRef="md3">            <!--md3-md2 CPsRship -->
        <parentMd uidRef="md2">
          <viaPipe if="pipe(F1) && ${F1recent}"
            inFileUidRef="F1" outFileUidRef="F1"/></parentMd></CPsRship>

      <CPsRship childMdUidRef="md4">            <!--md4-md2 CPsRship -->
        <parentMd uidRef="md2">
          <viaPipe if="pipe(F2) && ${F2recent}"
            inFileUidRef="F2" outFileUidRef="F2"/></parentMd></CPsRship>

      <CPsRship childMdUidRef="md5">            <!--md5-md2 CPsRship -->
        <parentMd uidRef="md2">
          <viaPipe if="!{F1recent} && !{F2recent}"
            inFileUidRef="Fx" outFileUidRef="Fx"/></parentMd></CPsRship>

      <mvproperty name="md345">
        <value>md3</value>
        <value>md4</value>
        <value>md5</value></mvproperty>

      <CPsRship childMdUidRef="md6">
        <parentMd uidRef="${md345}">
          <viaPipe if="" inFileUidRef="${md345}-out"
            outFileUidRef="${md345}-out"/></parentMd></CPsRship>

      <CPsRship childMdUidRef="md1">            <!--md1-md6 CPsRship -->
        <parentMd uidRef="md6">
          <viaPipe if=" ${loop} < 100" inFileUidRef="/dev/null"
            outFileUidRef="/dev/null"/></parentMd></CPsRship>

      <CPsRship childMdUidRef="md7">
        <parentMd uidRef="md6"><viaPipe ... /></parentMd></CPsRship>
    </cPsRshipSet>
  </appMdRships>
</application>
```

## 5.6 Summary

In this chapter, we presented the Grid Application Modeling and Description Language (GAMDL), a high level abstract language for domain users to describe a

workflow application. GAMDL, designed to address the limitations of current workflow languages with respect to support for resource co-allocations for workflow tasks, is feature rich. It associates resource request specification and execution profile with workflow module specification, so resource multi-requests can be easily constructed by software based on the workflow. The execution details of a workflow job can be specified in different contexts, thus providing a flexible means to organize and reuse the entities in different applications without having to redefine them. Using GAMDL, both data-flow and control-flow relationships can be described using DAG style structures. Users do not need to manually construct the application's control-flow if they have a data-flow application. Other features of GAMDL include the use of mvproperties to describe similar entities, and its support for nested and partial workflow.

# Chapter 6

# Resource Allocation and Scheduling of Workflow Applications

In this chapter, we present the grid workflow system architecture defined in the GRACCE framework. We first have a general overview of the architecture, then we discuss the three core components of the architecture. Finally, we discuss in more detail the algorithms used in workflow execution planning and resource allocations. To help present these algorithms, we simplify one term used in the last chapters. From now on, we use "task" to represent a module job; and a workflow consists of multiple tasks with dependencies.

## 6.1 The GRACCE Workflow System Architecture

GRACCE defines an architecture to implement a look-ahead scheduling and execution system for scientific workflow applications. The architecture addresses the issues of grid resource allocation, workflow execution and monitoring, and integrates their solutions into a middleware platform. From this platform, end users can build a custom grid environment to manage a grid application in its entire life cycle.

As shown in Figure 6.1, the GRACCE architecture has three subsystems, the Scheduler, the GridDAG workflow engine, and the EPExec runtime system. The **Execution Plan (EP)** in this architecture is a collection of the scheduling decisions for workflow tasks and the mechanisms for handling task dependencies. The EP is generated by the Scheduler in the scheduling process, and is used by the GridDAG to coordinate task dependencies. EPExec submits workflow tasks to their allocated resources and manages their execution according to the EP.

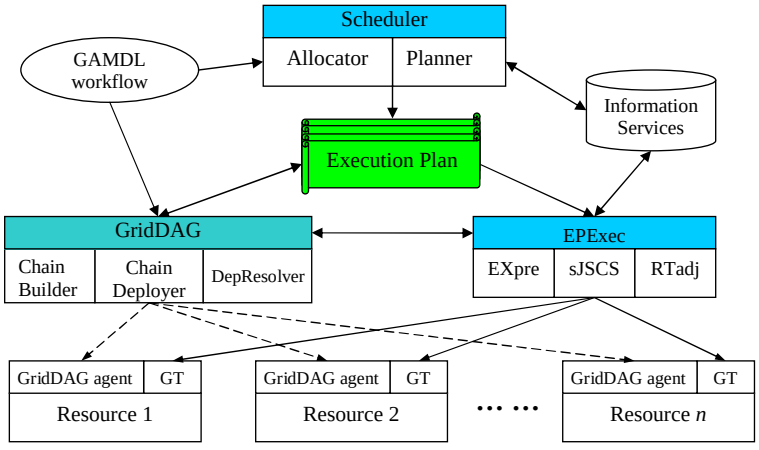



Figure 6.1: The GRACCE Scheduling Architecture

The **Scheduler**, with two components, the Allocator and the Planner, co-allocates

resources for workflow tasks and plans the workflow execution. The Allocator discovers suitable resources, negotiates the resource provision and makes reservations with resource providers. The Planner plans workflow execution and co-allocates resources for workflow tasks. It predicts the execution scenario for each task, which is about when and how the task should be launched. The scheduling process is based upon the application workflow; the decisions made are used to create the workflow EP.

The **GridDAG** is an event-driven workflow coordination system. At the scheduling stage, GridDAG decides how to handle dependencies and determines the event activities that are involved in the handling. These decisions are appended to the workflow EP. During workflow execution, GridDAG coordinates the execution of dependent tasks by handling and resolving task dependencies.

The **EPExec** is a runtime system for workflow executions. Given a workflow EP, EPExec submits tasks to the allocated resources, and monitors the execution of these tasks. EPExec sends events related to file availability or to the status change of task execution to GridDAG for the purpose of handling task dependencies. During execution, EPExec may adjust the EP according to the real execution scenario.

An application workflow has the following life-cycle in the GRACCE architecture:

1. Users request GRACCE to launch a workflow specified by a GAMDL document.
2. The GRACCE Scheduler plans the workflow execution and allocates resources for each task. It writes the decision details into the workflow EP.
3. The workflow EP is forwarded to GridDAG which will decide on and set up the

mechanisms of dependency handling; these details are added to the workflow EP.

4. According to the workflow EP, EPExec submits the first task of the workflow to its allocated resource and monitors it, thus beginning the execution cycle of the workflow.

5. During workflow execution, GridDAG handles task dependencies. When all dependencies of a task are resolved, GridDAG notifies EPExec to submit it to its allocated resource.

### 6.1.1 The GRACCE Scheduler

GRACCE's Scheduler has two components, the Planner and the Allocator. The Planner predicts and identifies the execution window for each task, and the Allocator searches a list of candidate resources, negotiates and makes the necessary agreement with resource providers. A task's execution window (EW) is a time frame during which a task is executed. EWstart denotes the EW start time, and EWlength denotes the EW length – EWlength is equal to the task wall-clock time plus a configurable buffer time. The EW of an ancestor task must be before the start-time of its dependent tasks, but the EW's of independent tasks can overlap. In this subsection, we only give a short overview of the functionalities of the Scheduler to help understanding the whole architecture. In Section 6.2, we discuss in thorough details the scheduling process and the algorithms used.

#### 6.1.1.1 Execution Planning: Identify Task Execution Window

Given a workflow, the Scheduler planning process identifies the EWs for each task using a breadth-first graph traversal algorithm. The algorithm starts with the allocation of resources for the first task of the workflow by the Allocator. When resources are allocated, the Allocator also identifies the task EW. Then, the Scheduler processes the children of the first task. First, Allocator discovers a list of candidate resources for each child task and calculates the cost of dependency handling between the resource(s) for the parent task and the candidate resources for child tasks. Secondly, the Planner predicts the task EW if it is run on the candidate resources. The EWstart is calculated by adding the EWstart and EWlength of the parent task as well as the time required for dependency handling. Thirdly, the task EW predicted for each candidate resource is processed again by the Allocator, which will allocate the best resource for the task and determine its EW. The Scheduler then moves on to process other tasks.

#### 6.1.1.2 Resource Co-Allocation, Negotiation and Reservation

The Allocator allocates computational resources for workflow tasks in a sequence of resource discovery, negotiation, and reservation. During resource discovery, the Allocator queries the Grid Information Services for resources that satisfy the task resource requirements and are available during its EW. Firstly, resources are selected by a simple match-making of each attribute of a task's specification with static resource information. The resources on which the task is able to run are further

evaluated according to their runtime information. Then, the selected resources are checked for their availability during the task EW, and the Allocator finally identifies a list of candidate resources. In the negotiation and reservation stage, the Allocator requests reservations for the candidate resources during a task's EW. If the local schedulers grant the requests, the Allocator chooses the one that can provide the earliest EW for the task. A reservation ID is returned that will be used to access the reservation later. If no reservation could be made on any of the candidates, grace periods are added to the EW and Allocator again requests reservations for other wall-clock periods within the EW until a reservation is made.

### 6.1.2 The GridDAG Workflow System

GridDAG is our event-driven workflow system; it is able to coordinate the scheduling and execution of the dependent tasks of a workflow job. Compared with other workflow enacting engines, GridDAG is a pure coordination system, without any execution or monitoring functionalities, which are provided by EPExec in GRACCE. This gives GridDAG the flexibility to integrate with various remote execution and monitoring utilities. Different coordination mechanisms can be developed in GridDAG without necessitating additional effort to integrate them with other GRACCE subsystems.

#### 6.1.2.1 The GridDAG Eventing Mechanisms

Events are notifications of a status change of task executions or file transfers, data availabilities, or other situations defined by users, such as for resource accounting

purposes. An event producer detects certain situations or a status change, generates the corresponding event messages and distributes them. An event consumer receives an event message and invokes the event handlers. The GridDAG event mechanism is based on the WS-Notification standard [99], so event messages are XML documents – which allows the implementations to be platform-neutral in distributed heterogeneous environments.

Four components in GridDAG support the eventing mechanisms: the event chain builder, chain deployer, GridDAG agent, and DepResolver. The chain builder reads the job EP forwarded from the Scheduler and generates the event chains according to the EP. An event chain is an ordered sequence of events from the participating producers to consumers. The chain deployer sends subscription requests to producers. A Subscription represents the relationship between a consumer, producer, and related event messages. These relationships constitute the runtime event chains of a workflow job. GridDAG agents coordinate the runtime event activities in each grid resource. Firstly, as a producer, GridDAG agents detect events occurring on the host resources and send out event messages. Secondly, as a consumer, GridDAG agents receive event messages from other agents or EPExec and take actions accordingly. DepResolver is the overall coordinator of dependency handling and resolving. DepResolver keeps track of the states of task dependencies and decides whether all of the dependencies are resolved.

#### 6.1.2.2 Data Dependency Handling

Using the GridDAG eventing mechanism to handle data dependency, file transfers can be in either destination-pull (D-P) or source-push (S-P) mode. The event sequences for these two modes are shown in Figure 6.2. In the D-P mode, when the GridDAG agent on the source resource detects that files are available (1), it sends a corresponding event to the destination GridDAG agents (2). The destination GridDAG agents fetch the files (3) and send events to DepResolver notifying it of file arrivals (4). For the S-P mode, when files are available (1), the source GridDAG agent transfers them to the destination resources (2) and sends an event to the destination GridDAG agents and to DepResolver indicating that the intermediate files have been transferred (3). We expect that the D-P mode works better when multiple destinations are waiting for the same set of data. The S-P mode is suitable for situations where data production and movement can be pipelined.

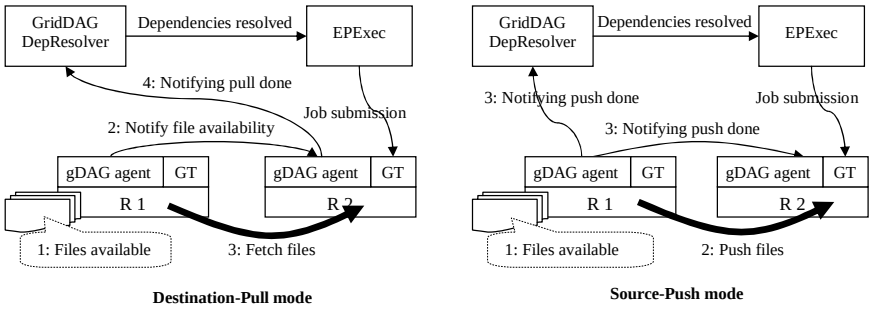



Figure 6.2: Event Sequence in File Transfer

### 6.1.3 The EPExec Runtime system

EPExec (EP Executer) is the runtime execution system for workflow jobs according to the job EP. EPExec has three components, EXEpre, sJSCS, and RTadj, to provide the functionalities of job submission, job monitoring, and the runtime adjustment of a job EP.

EPExec is implementation-independent of the Scheduler and GridDAG subsystems and communicates with them via platform-neutral event messages. This ensures it to be flexible enough to integrate with various middleware packages. Different EPExec's can be developed to support different methods of job submission and remote execution without requiring any changes to the Scheduler and the GridDAG.

#### 6.1.3.1 Execution Preparation

EPExec's execution preparation (EXEpre) adds the required information for job submission and workflow control to the job EP. The details depend on the grid middleware that it is developed on. Assuming that Globus GRAM is responsible for job submission and GridDAG for workflow coordination, its work can be summarized as follows:

- **Preparation for GRAM job submission:** EXEpre parses the tasks specification, generate a Globus RSL file, and specifies the locations and names of input and output files on the selected resources for the task.
- **Preparation for dependency handling:** EXEpre forwards the job EP to

GridDAG which sets up the event chains and configures the event consumers and producers, as discussed above.

#### 6.1.3.2 EP Execution and Monitoring

EPExec starts the workflow execution by submitting the job corresponding to the first task to its allocated resources according to the job EP. sJSCS (simple Job Submission and Control Service) is a utility to respond to such submission requests from EPExec. It calls the remote execution functions, such as Globus *globus-job-submit* to submit a single-executable job. The job is submitted using its resource reservation ID; this ensures the task is launched within its EW. A successful submission returns a global job ID, which EPExec uses for job monitoring and control.

EPExec monitors task executions in both passive-notification (P-N) mode and active-checking (A-C) mode. In the P-N mode, EPExec relies on the event messages about job status change to track the job. These messages are sent by the GridDAG agent on the resource where the job executes. In the A-C mode, EPExec calls sJSCS to query the current state of job execution. The P-N mode alleviates EPExec from the frequent calling of sJSCS; but EPExec may lose track of the job if the event mechanism fails. So normally, both the P-N and A-C modes are enabled in EPExec for a close monitoring of the job.

#### 6.1.3.3 Runtime Adjustment

EPExec coordinates task executions so that the executions follow the EP. But if a task completes after its EW, the RTadj (Runtime Adjuster) component of EPExec may take actions to adjust the EP or to make up the delay. In most situations, those tasks that depend on the late task can be started within their EW's and RTadj does not need to adjust them. But if the late completions cause the expiration of reservations of the dependent tasks and they cannot be started in their EW's, RTadj uses the following strategy to try to make up the delay:

First, EPExec submits these tasks to their allocated resources without using reservation. The jobs may be held in the resource local queues. RTadj then requests Scheduler to discover alternative resources for these tasks. If suitable resources are discovered and allocated, EPExec submits copies of these tasks to these resources. During execution, EPExec identifies the copy that it thinks will complete first and kills the others. Thus RTadj does its best to make up for the lost time in past job execution and to minimize the negative impacts on the execution of later tasks.

If it seems impossible to follow the initial job EP, RTadj will consider re-scheduling the rest of the tasks. In this case, RTadj forwards the job sub-workflow to Scheduler to reschedule. Re-scheduling may cause low resource usage or wastage because of the cancellation of prior reservations. The Scheduler tries to avoid this situation by scheduling other jobs onto these reservations if possible.

## 6.2 Resource Allocation and Workflow Execution Planning

In this section, we discuss the process and the algorithms of resource allocation and workflow execution planning, mainly the functionalities and internals of the Allocator and the Planner components of the Scheduler subsystem in the GRACCE architecture. The notations used in this section are listed in Table 6.1.

Table 6.1: Notation Used in the Algorithm Description

| Notation | Explanation |
|---|---|
| T$i$ | A workflow task $i$ |
| Child$_{Ti}$ | An array of the child tasks of task $Ti$ |
| Parent$_{Ti}$ | An array of the parent tasks of task $Ti$ |
| R$m$ | A resource $m$ |
| EW$_{Ti}$ | The Execution Window of Task $Ti$, which is a <EWStart, EWLength> pair. |
| ResReq$_{Ti}$ | The resource request specification of ask $Ti$ |
| AllocRes$_{Ti}$ | A resource allocation decision for Task $Ti$ |
| StartTime$_{Ti}$ | Task $Ti$'s start time |
| ExeTime$_{Ti}$ | Task $Ti$'s execution time |
| EndTime$_{Ti}$ | Task $Ti$'s end time, which equals StartTime$_{Ti}$ + ExeTime$_{Ti}$ |
| NumPE$_{Ti}$ | The number of Processing Elements (processors) required to run task $Ti$ |
| FileSize$_{Ti:Tj}$ | The size of intermediate files between parent task $Ti$ and child task $Tj$ |
| NumPERes$_{Rm}$ | The total number of processing elements of resource $Rm$ |
| Bandwidth$_{Rm:Rn}$ | The bandwidth between resource $Rm$ and $Rn$ |

### 6.2.1 Resource Allocation for a Workflow Task

As we mentioned before, the GRACCE Allocator is responsible for allocating resources for a workflow task based on its resource request. The resource request

specification of task $Ti$, denoted as $\text{ResReq}_{Ti}$, consists of the number of processing elements (PEs) and the minimum memory required, the start time, the optional execution time (one way to specify a deadline) and the requirements of hardware architecture and software configuration. An allocation decision, denoted as $\text{AllocRes}_{Ti}$, consists of the resource name, the total number of PEs, a list of these PEs, the available time, the estimated execution time if the task is executed on this resource, and a reservation ID (ResvID) if it has. These two objects can be represented as follows:

$\text{ResReq}_{Ti}$<NumPE, minMem, StartTime, ExeTime, ArchSW>

$\text{AllocRes}_{Ti}$<ResName, NumPEs, PEList, StartTime, ExeTime, ResvID>

The Allocator takes the resource request of the workflow task $Ti$, makes allocation decision and returns the decision(s) with one or more $\text{AllocRes}_{Ti}$ objects. The allocation process involves two steps, resource discovery and evaluation (Step 1), and resource negotiation and reservation (Step 2). Starting from all available resources, each step refines the results from the previous step. By the end of Step 1, one or more ResReq objects without valid ResvIDs, corresponding to one or more candidate resources, are returned. By the end of Step 2, one or more ResReq object(s) that have valid ResvID(s), corresponding to the reserved resource(s), are returned. We denote each of the two steps as $\text{Allocator}_{S1}$ and $\text{Allocator}_{S2}$ and the following formulas represent their operations:

$$\text{AllocRes}_{Ti}[\,] = \text{Allocator}_{S1}(\text{ResReq}_{Ti})$$

$$\text{AllocRes}_{Ti}[\,] = \text{Allocator}_{S2}(\text{AllocRes}_{Ti}[\,])$$

#### 6.2.1.1 Resource Discovery and Evaluation

Resource discovery and evaluation searches in grid information services for resources that match the resource specification of a workflow task. The resources discovered by this step are those that match the requirements of hardware architecture and software configuration of the workflow task, for example, the CPU architecture, the operating system and version, the total PEs and the libraries installed on the resource. This step does not consider what Quality of Service or performance a resource can provide for the task, and it only finds the resources that are able to "execute" the task.

The grid information services are provided by the Globus Toolkit's MDS [41]. The Globus MDS defines two key protocols: the GRid Registration Protocol (GRRP), which a resource uses to register with an aggregate directory, and the GRid Information Protocol (GRIP), which an aggregate directory or user uses to look up the status of a resource. When the Allocator starts up, it queries appropriate aggregate directory(s) to locate any potentially interesting resources. Having obtained the names and gross characteristics of these resources, the Allocator then uses GRIP to contact them directly and obtain detailed and up-to-date information.

For each of the discovered resources, the evaluation process is similar to Condor's "matchmaking" algorithm [67], which in its basic form takes two ClassAds and evaluates one with respect to the other. A ClassAd is a set of expressions that must all evaluate to true in order for a match to succeed. Expressions in one can be evaluated using property values from the other. A ClassAd can also include a rank expression that evaluates to a numeric value representing the quality of the match. An example

of a ClassAD for a grid job is shown below.

```
[
  Type="request";
  requirements = "NumPE == 32 & minMem == 16G &
       Arch == 'Ultra SPARC' & opSys=='Solaris 8.0'";
  Rank= cpuSpeedPE * memSizePE;
]
```

In our implementation of the matchmaking algorithm, we use the same format as ClassAd to represent task specifications and resource specifications, but the algorithm is developed by ourselves that is integrated with the workflow scheduler. In operation, the Allocator first translates the task resource request specification to ClassAd and invokes the matchmaking algorithm against the ClassAds representing available resources, and returns the computed rank of this match. The rank used in the ClassAd above, the product of the average CPU speed and memory size per PE, represents the computation power of the resource. The higher the rank, the faster the resource, hence the shorter time it takes for the resource to execute the task. The discovered candidate resources are ordered using this rank value.

#### 6.2.1.2 Resource Negotiation and Reservation

In the Step 1, the Allocator discovers and selects multiple candidate resources in ranked order for a workflow task; in this step, the Allocator communicates with the local schedulers of the candidate resources to finally make the resource allocation decisions for the task. As we mentioned before, the Allocator, which acts as a grid scheduler, operates on top of local schedulers that have ownership of resources. So

instead of being able to directly allocate a resource, the Allocator has to *request* the resource from the local scheduler and the local scheduler may or may not satisfy such request. If the local scheduler denies the request, the Allocator modifies the resource request specification and resubmits the request to the local scheduler for consideration. This process is called resource negotiation and it is the process of requesting "*advanced reservations*" of resources from the local schedulers.

"An advanced reservation is a possibly limited or restricted delegation of a particular resource capability over a defined time interval, obtained by the requester from the resource owner through a negotiation process" [100]. An advanced reservation ensures access to specific PEs during the specified time, and it is essentially a lock on a number of PEs. Each reservation, identified by a ResvID, consists of the number of hosts/PEs reserved, a start time, an end time, and an owner. During the time the reservation is active, only users or groups associated with the reservation have access to start new jobs on the reserved hosts. The reservation is active only within the time frame specified. When it becomes active, the associated users can submit jobs that reference the reservation by the ResvID and these jobs occupy the reserved resources when being executed. Jobs occupying the reservation may be killed by the local scheduler when the reservation expires. But most local schedulers allow the jobs to keep running when the reservation expires. Advanced reservation is supported by most current local schedulers, such as SGE, LSF, PBS Pro and Maui.

The resource negotiation and reservation process is thus a two-phase handshaking between the Allocator and the resource's local scheduler. The negotiation sequence and hand-shaking messages involved are shown in Table 6.2, and in Table 6.3, we

Table 6.2: Negotiation Sequence and Messages

| Sequence | Sender | Msg Type | Msg Format |
|---|---|---|---|
| 1 | Allocator | ResvRequest | <NumPEs, PEList, StartTime, EndTime> |
| 2 | LocalScheduler | ResvResponse | <YES/NO, ConfirmDeadline, ResvID> |
| 3 | Allocator | ResvAccept | <ResvID> |
| 4 | LocalScheduler | ResvConfirm | <ResvID, options> |

Table 6.3: An Example of a Negotiation Process

| Sequence | Msg | Explanation |
|---|---|---|
| 1 | ResvRequest<16, 12-28, 100A, 300A> | Request 16 PEs (12-28), from 1:00AM to 3:00AM |
| 2 | ResvResponse<Yes, 60, 232324> | Yes, confirms it within 60 seconds |
| 3 | ResvAccept<232324> | Ok, accept reservation 232324 |
| 4 | ResvConfirm<232324, options> | Confirmed |

show an example of the messages in a negotiation process. The process starts with the Allocator's submission of a reservation request in the form of a *ResvRequest* message to the local scheduler. The *ResvRequest* consists of the number of PEs and the time period requested. The local scheduler processes this request and responds with a *ResvResponse* message. If the local scheduler accepts it, the message contains a "*Yes*" answer, a deadline within which the Allocator must confirm to finally acquire the reservation, and the reservation ID. The Allocator next replies to the local scheduler with a *ResvAccept* message that encodes the reservation ID. Upon receiving this message, the local scheduler responds with a *ResvConfirm* message that includes all the details of the reservation. From this point, the Allocator can access this reservation using the reservation ID. This process can be represented by the following two formulas:

$$\text{ResvResponse} = \text{Allocator}(\text{ResName}, \text{ResvRequest})$$

$$\text{ResvConfirm} = \text{Allocator}(\text{ResName}, \text{ResvAccept})$$

### 6.2.2 Workflow Execution Planning

Workflow execution planning decides, for each workflow task, where (on which resource) and when it is launched; the Planner is responsible for performing this. To plan the workflow execution, two important parameters for each task are required, the (estimated) execution time of the task on the candidate resource and the network bandwidth between the target resources for its parent tasks and the candidate resource for the task. Performance prediction is one approach for generating these two parameters. We rely on widely used systems for this purpose, for example, application performance prediction using performance profiling [101, 102], and network bandwidth prediction using the Network Weather Service [103]. We use a Predictor to represent the prediction operations as follows:

$$\mathsf{ExeTime} = \mathsf{Predictor}(\mathsf{ResName}, \mathsf{TaskSpec}_{Ti}, \mathsf{ResReq}_{Ti})$$
$$\mathsf{NetworkBandwidth} = \mathsf{Predictor}(\mathsf{FromRes}, \mathsf{ToRes}, \mathsf{Time})$$

The Planner makes a planning decision for a workflow task based on the decisions made for its parent tasks. The planning process includes two steps: identifying the StartTime and ExeTime (the execution schedule) of the task on resources; and requesting resource advanced reservation on the resources. The planning process is shown in the following code fragment and the two steps are implemented in the two main "for" loops. For a task *Ti*, when the scheduling decisions for all its parent tasks have been made, the Planner starts processing it. The Planner first requests the Allocator to discover a list of candidate resources, and then evaluates each of the candidate resource by determining the StartTime and ExeTime of the task on

it. To determine the StartTime, the Planner sums two time values for each of its parent tasks: the EndTime and the time required to transfer the intermediate files between the resource allocated for the parent task and the candidate resource being evaluated. To calculate the time for file transfer, the Allocator calls the Predictor to acquires the network bandwidth information on the EndTime of the parent task. The greatest sum, which is the latest time when all the input data are available, is the earliest StartTime of the task. To determine the ExeTime of the task on a candidate resource, the Planner calls the Predictor to estimate the execution time of the task based on the performance prediction approach we mentioned before. After determining the StartTime and ExeTime of the task on a resource, the Planner sorts the candidate resources using the EndTime (StartTime + ExeTime) as the key.

In the second step, the Planner calls the Allocator to request reservations on the candidate resources in the sorted order. If a reservation is granted, The Planner then requests the Allocator to confirm the reservation, thus complete the planning process for the task.

```
/* sortedAllocRes is a AllocRes table sorted using the endTime (startTime + exeTime).
   The sortInsert function insert an AllocRes object in the endTime order. */
AllocRes_Ti[ ] = Allocator_S1(ResReq_Ti);

for (i=0; i<AllocRes_Ti[ ].size; i++) {
    startTime = -1;
    resName = AllocRes_Ti[i].ResName;
    for (j=0; j<Parent_Ti[ ].size; j++) {
        pTask = Parent_Ti[j];
        EndTime_pTask = StartTime_pTask + ExeTime_pTask;
        bd = Predictor(AllocRes_pTask.ResName, AllocRes_Ti[i].ResName, EndTime_pTask);
        eStartTime = EndTime_pTask + FileSize_pTask:Ti / bd; #estimated start time
        if (eStartTime > startTime) startTime = eStartTime;
    }

    exeTime = Predictor(resName, TaskSpec_Ti, ResReq_Ti);
```

```
    AllocRes_Ti[i].StartTime = startTime;
    AllocRes_Ti[i].ExeTime = exeTime;
    eEndTime = startTime + exeTime;
    sortInsert(sortedAllocRes, eEndTime, AllocRes_Ti[i]);
}

AllocRes_Ti = sortedAllocRes; # now it is sorted based on the QoS
for (i=0; i<AllocRes_Ti[ ].size; i++) {
    Msg(ResvRequest) = <AllocRes_Ti[i].NumPEs, AllocRes_cTask[i].StartTime,
                        AllocRes_cTask[i].StartTime + AllocRes_cTask[i].ExeTime>;
    resName = AllocRes_Ti[i].ResName;
    Msg(ResvResponse) = Allocator(resName, ResvRequest);

    if (ResvResponse.YES/NO == "YES") {
        Msg(ResvConfirm) = Allocator(ResvAccept<resName, ResvResponse.ResvID>);
        AllocRes_Ti[i].ResvID = ResvConfirm.ResvID;
        break;
    }
}
```

The algorithm above is the core of the workflow planning policy. For a whole workflow planning, the planning algorithm performs a breadth-first retrieval of the workflow DAG starting from the starting task(s), and processes the workflow tasks in the topologically sorted order. Depending on how deep the planning process retrieves the workflow DAG, the algorithm can be classified into three categories: full workflow planning, just-in-time planning(scheduling), and partial workflow planning.

#### 6.2.2.1 Full Workflow Planning

In full workflow planning, the algorithm processes the workflow tasks from the starting tasks till the end tasks, and makes scheduling decisions for all tasks. The advantage of this approach is that it allows the resource reservations to be requested much earlier than the StartTime, thus increasing the possibility of being granted,

even in heavily-load environments. But using this policy, the ExeTime of each workflow task must be provided or can be estimated (accurately). The accuracy of the task ExeTime greatly impacts the quality of the planning. The inaccuracy of the estimated ExeTime is propagated in the planning process along with the workflow DAG retrieval; low quality or even unworkable planning schedule may be produced. So this approach is suitable only for workflows that have predictable and accurate ExeTime for each task and it is the best approach for scheduling workflows with deadlines in heavily-loaded environments.

#### 6.2.2.2 Just-In-Time Scheduling

In this policy, the Planner discovers and allocates resources for a workflow task only when the executions of all its parent tasks complete. The advantage of this policy is that the StartTime that is calculated based on the EndTimes of the parent tasks is much more accurate than that in a full or partial workflow planning because this EndTime is not an estimated time value, but the real EndTime of the parent task. The inaccuracy in, or the inability of calculating, the ExeTime of the task would not impact the quality of planning for later tasks. The disadvantage of this policy is that the Planner may not acquire a resource reservation because the time between when the reservation is requested and when the reservation is activated is too short, which is the time for transferring the immediate files. The Planner finds a resource that has the shortest queue waiting time and queues up the task in the resource's local scheduler. In heavily-loaded environments, this waiting may delay the whole workflow execution dramatically. So this approach is suitable for lightly

load environments or workflows that do not have strict deadlines. For workflows that do not have enough information for the Planner to predict the ExeTime of each task, the Planner has to use this policy.

#### 6.2.2.3 Partial Workflow Planning

In this policy, the depth of workflow DAG retrieval during the planning process is between the other two policies. The Planner makes scheduling decisions for a subset of the workflow tasks starting from the starting task(s), and then launches those tasks according to the schedules that have been made. Near or upon the completion of those tasks, the Planner makes scheduling decisions for the next subset of the workflow tasks based on the information about the completed tasks. This iteration continues until all the workflow tasks are completed. This policy is very suitable for deep workflows that have large number of tasks, and the negative impacts of the disadvantages of the other two policies on the planning quality can be alleviated if the task subset is chosen properly.

#### 6.2.2.4 Two Prerequisites to Apply The Three Policies

As a summary of this section, we note that the three scheduling policies require two supports from the underlying local schedulers and other middlewares. First, the resource advanced reservation support should be available and enabled in the local scheduler. This feature can always be exploited by the three policies to reduce queue waiting time. But the full or partial workflow planning policy requires it to be

available in order to make the best-effort execution planning. If this feature is not available, we recommend using just-in-time scheduling policy instead. Second, middlewares that provide the functionality of performance prediction of task execution time on a resource should be available. This is required by the workflow scheduler in order to make the resource allocation decisions in advance in full and partial workflow planning policies. As we mentioned, the accuracy of the prediction impacts the quality of scheduling and planning. So if the performance of an application is not predictable, or the prediction is not accurate enough, only the just-in-time scheduling policy can be applied. If those applications have regular executions, a common approach in scheduling would be to apply the just-in-time scheduling policy first, and then gather the execution history for each of its executions. Based on this historical information, the scheduler makes predictions of its future executions, and compares the predictions with the real execution scenarios to see how accurate these predictions are. If the predictions become accurate enough to make good workflow planning, the full or partial workflow planning policy can be applied. This is a topic of machine learning that can be explored in order to improve the performance of applications using our scheduling policies.

## 6.3 Summary

In this chapter, we presented our grid workflow system architecture and the algorithms of workflow resource allocation and execution planning in the GRACCE project. The project aims to provide an end-to-end solution for automatic workflow

execution in grid environments. Using the GRACCE middleware, domain scientists are only required to specify the application logic and resource requirements; GRACCE is responsible for allocating grid resources for the application workflow, for launching the workflow, and for the delivery of the results back to the users.

The GRACCE workflow system architecture provides and integrates solutions to grid scheduling related problems. The three subsystems in the architecture, the Scheduler, the GridDAG workflow engine, and the EPExec runtime system, constitute an extensible platform for the integration of grid middleware and applications. Grid middleware solutions can be easily interfaced with one of the subsystems without changing the other subsystems. From this platform, end users without any in-depth grid knowledge are able to deploy their applications on the grid easily.

The GRACCE scheduling algorithms apply advanced scheduling techniques, such as look-ahead resource co-allocation, execution planning and performance prediction, during the workflow resource allocation and scheduling process. By taking into account the various factors that may impact the workflow execution performance, such as the dependency relationships of workflow tasks, the network bandwidth and intermediate file size between dependent tasks, the GRACCE scheduler makes scheduling decisions with the goal of improving the overall workflow performance, instead of individual task performance. We believe by applying those techniques aggressively, the users' expectations of quality of services, such as workflow execution time, can be easily met.

# Chapter 7

# Experiment and Performance Evaluation

In this chapter, we show the performance results of scheduling workflow applications on a simulated grid environment using the GRACCE scheduling algorithms.

## 7.1 Simulation Environment Setup

The GRACCE workflow scheduler targets computational grid environments that comprise parallel computing resources owned by different organizations. But there are difficulties in evaluating a workflow scheduler in a real grid environment. These include the limited number of resources available for testing purpose, and the impossibility of creating a repeatable and traceable environment for evaluating different scheduling strategies under different resource loads. For these reasons, we perform

our experiments in a simulated grid environment that models a real grid very closely in those respects that are important for workflow scheduling. The experimental environment consists of simulated grid resources with a variety of computational capabilities and different network bandwidth between resources, simulated grid jobs and the local schedulers of resources, and a random job generator that models the resource users. We have also developed a random workflow generator that is able to create workflows with different task specifications and dependency relationships for evaluating our scheduling methods.

### 7.1.1 Simulation of Grid Resources and the Local Schedulers

In our simulation, a resource is configured with a total number of PEs and a total number of MIPS (Million Instructions Per Second) to represent its computational capability, an approach borrowed from the GridSim toolkit [104]. GridSim has limitations in its manner of scheduling parallel applications, and thus cannot be directly exploited for our work. In GridSim, a parallel job that requires more than one PE is normalized to a single-PE job (Gridlet in GridSim term). When a job requests 10 PEs and 15 minutes execution time, GridSim allocates one PE and sets the execution time to 10*15 minutes. For this reason, we have developed our own local scheduler for the simulated resources. It allows us to allocate or reserve multiple PEs for a parallel job, and it keeps track of currently used PEs (and thus the available PEs) in order to serve other requests.

The scheduling policy of the local scheduler is first-come-first-serve (FCFS) space-sharing with resource advanced reservation. The scheduler maintains a job queue for the resource, and a newly submitted job is put at the tail of the queue. The local scheduler allocates PEs for the job at the head of the queue when another job completes its execution and releases the PEs. If there are not enough available PEs for the job at the head of the queue, the job (and all those behind it) has to wait until another job completes and releases enough PEs. If an advanced reservation is used when submitting a job, the job does not need to wait and is launched when the reservation becomes active. So we basically implement a gang scheduling policy [105] that is widely used for parallel application scheduling.

The simulated grid consists of eight resources. Their configurations are shown in Table 7.1. The MIPS per PE (MIPS/PE) is calculated to represent the computation speed of the resource's PEs. The greater the MIPS/PE, the faster the resource. The network bandwidth between every pair of resources in the simulation is calculated by generating a random bandwidth between the minimum bandwidth (0 MB/s) and the maximum bandwidth (10 MB/s).

Table 7.1: Resource Specification in the Simulated Grid Environment

| Resource | Total PEs | Total MIPS | MIPS/PE |
|---|---|---|---|
| R1 | 8 | 200 | 25 |
| R2 | 64 | 750 | 11.719 |
| R3 | 48 | 310 | 6.458 |
| R4 | 32 | 400 | 12.5 |
| R5 | 32 | 500 | 15.625 |
| R6 | 48 | 800 | 16.667 |
| R7 | 16 | 320 | 20 |
| R8 | 32 | 400 | 12.5 |

### 7.1.2 Simulation of Job Execution

A grid job (or a workflow task) in our simulation environment requests resources in terms of the total number of instructions (in the unit of Millions Instructions (MIs)), and the number of PEs. We assume that the instructions are evenly distributed and executed on the PEs, thus we have the parameter of MIs/PE for a job. In this modeling schema, the base execution time (*bExeTime*) of a job on a resource is equal to the job's MIs/PE divided by the resource's MIPS/PE. The base execution time is the ideal execution time of the job on the resource without considering the impacts of those factors, such as cache misses, or disk accesses that may stall the CPU calculation. We call those factors non-CPU factors with regard to their impacts on the job's performance.

A job execution is simulated using a timer thread: when the thread starts, the job starts; when it times out, the job completes. The timeout interval of the thread, corresponding to the execution time (*ExeTime*) of the job, is calculated by adding an additional time to the *bExeTime* that represents the impact of those non-CPU factors on the job execution. To simplify our simulation, this additional execution time (*aExeTime*) is modeled as *bExeTime * Random (0, extFactor)*, where *extFactor* is a number between 0.0 and 1.0 representing the impacts of these factors. The *Random (0, extFactor)* is a random number between 0 and *extFactor*.

In this schema, the job execution time includes both the CPU time and the time for memory and disk/network accesses, thus closely models a real job execution on

a real computational resource. Furthermore, we can easily define a schema for estimating the execution time of the job on a resource by using one property of the *Random(0, extFactor)* function: the arithmetic mean of an infinite number of numbers generated by *Random(0, executor)* is equal to *exeFactor / 2*. So the estimated execution time (*eExeTime*) can be modeled as *bExeTime * (1 + exeFactor / 2)*. In this way, the estimated execution time equals the real execution time statistically; and the difference between *Random(0, extFactor)* and *exeFactor / 2* introduces the unpredictable part of the execution time. We formulate the two schemas as follows:

$$\mathsf{ExeTime} = \mathsf{bExeTime} * (1 + \mathsf{Random}(0, \mathsf{extFactor}))$$

$$\mathsf{eExeTime} = \mathsf{bExeTime} * (1 + \mathsf{extFactor} / 2)$$

### 7.1.3 Random Job Generator and Random Workflow Generator

To mimic grid resource users, we have created a random job generator that creates and submits jobs with different resource requirements to the local scheduler of a resource. The job generator is able to maintain the average resource load at a specific value between 0.0 and 1.0. The resource load at a given time is calculated by dividing the number of occupied PEs by the total number PEs of the resource. If the current resource load is less than the expected load, the job generator creates and submits jobs; otherwise it waits and monitors the resource load. As a multi-threaded program, the job generator is independent of the resource simulator, the local scheduler and the workflow scheduler. It, hence, models the multiple users of

the resources.

We have also created a random workflow generator to create different workflows for evaluating workflow scheduling algorithms. The workflow generator creates workflow tasks as the grid jobs, and then creates the dependency relationships between these tasks and sets the size of the intermediate files between the dependent tasks. The task specifications and dependency relationships, such as the number of PEs and the MIs, the number of parent and child tasks, and the intermediate file sizes between its parent/child tasks, are all controllable. The generator gives users options to specify the maximum and minimum of a parameter, and uses the *Random(min, max)* operation to generate a random value for the parameter. The number of tasks allowed in the generator is 0 to 200. A generated workflow can be simply visualized using yFiles graph library [106].

## 7.2 Performance Evaluation of Workflow Execution

The performance of a workflow execution is measured by the time to complete the execution, i.e., from the time when users submit a workflow to the time that the results are produced. The execution of a workflow involves both the execution of the workflow tasks and the transfer of immediate files between dependent tasks. The execution time is not simply the sum of the times for task execution and for the file transfers. Some executions and/or transfers may be overlapped, and additional

times may be introduced due to the unavailability of resources for ready tasks. As a result, the total workflow execution time consists mainly of three parts: the task execution time, data transfer time and the time spent waiting for resources to be available.

### 7.2.1 Task Execution Time

Obviously, the task execution time is not simply the sum of times spent carrying out all tasks because some of them are executed concurrently. For a workflow that can be modeled as a DAG, critical tasks are those that must be started on their earliest start times in order to achieve the best performance of the workflow execution. The sum of the execution times of critical tasks is the time spent for workflow task execution. So if we know the critical tasks of a workflow, we can easily calculate its task execution time. Yet during the workflow execution, the critical path is changing because the time spent on a task execution is not fixed. This is the issue of dynamic critical path. In our scheduler, an initial critical path is calculated based on the estimated task execution time. During workflow execution, the scheduler compares the subworkflow of completed tasks with the subworkflow of uncompleted tasks and adjusts the critical path so that it matches with the real execution pattern.

The time spent executing a task depends on what resource is allocated for it. In general, a workflow scheduler searches for the fastest available resource for a task. So in a computational grid, the overall grid load impacts the choice of resource. The higher the load is, the more likely a slower resource is allocated for the task. But

if the workflow scheduler is able to reserve resources for a task in advance, a fast resource can be allocated even in a high-load environment.

### 7.2.2 Queue Waiting Time

The workflow scheduler, working on top of the local schedulers of grid resources, cannot launch a workflow task on the allocated resource directly. It has to submit it in the form of a job to the local scheduler, which schedules the job based on its own policies. This local scheduler may queue the job for any reason, for example, if the resource is heavily loaded or higher-priority jobs come in and have to be scheduled earlier. So even when a task is ready, it may be queued by the local scheduler. The time period from when the task is ready to when it is launched by the local scheduler is often referred to as the "queue waiting time".

For a task that is submitted to the local scheduler, the length of queue waiting time depends on many factors, including the resource load, the number of CPUs requested by the job, the characteristics of the jobs currently running and those queued in the local scheduler, and the scheduling policies applied. It is difficult to predict [107]; there have been several efforts [108, 109, 110] to develop prediction mechanisms in the local schedulers. In general, the higher the resource load is, the longer the queue waiting will be. Again, if a task is submitted to a resource that has been reserved in advance, the queue waiting time can be greatly reduced.

In our simulation environment, as long as we know the allocated resource for a task, the execution time can be easily predicted, i.e., the *eExeTime* calculated

using the formula in the last section. Also, since we use the FCFS space-sharing scheduling policy with advanced reservation, we can estimate the start time for a newly submitted job by performing a simulated execution of current running jobs and the queued jobs, a process that sums up the estimated execution time of the tasks that occupy the majority period of the resource while considering the advanced reservation created for the resource. We can then calculate the queue waiting time easily. In our workflow scheduler, this prediction schema is used when planning the workflow execution. Lastly, we want to note that the queue waiting for a non-critical task does not delay the overall workflow execution if it does not delay the launching of critical tasks.

### 7.2.3 Data Transfer Time

If the parent task and child task in a data dependency relationship are allocated on different resources, intermediate files need to be transferred between the two resources. If the network is slow and the intermediate files are large, the data transfer time may become a significant part of the overall workflow execution time. But at run time, file transfers are overlapped with the execution of tasks. So not all data transfers impact the workflow performance; only those that delay the launching of critical tasks, directly or indirectly, do so.

## 7.3 Simulation Results

Using the simulation environment and the resource local scheduler we have developed, we have implemented two workflow schedulers based on the algorithms we discussed in the last chapter: the just-in-time workflow scheduler that is widely used in most of current workflow systems, and the GRACCE scheduler with full workflow planning and resource reservation policy. Using the just-in-time scheduler, when the parent tasks of a task are completed, the scheduler allocates a resource for it, and then submits it the local scheduler of the allocated resource. In the GRACCE workflow scheduler, the execution of the workflow is planned in advance before it is launched. Resources are allocated and reserved for the workflow tasks during the planning process. The GRACCE scheduler launches each workflow task according to the planned schedule for the task. Both of the two schedulers take into account the time spent to transfer immediate data file between the parent task and the child task when allocating a resource for the child task. In the rest of this section, we evaluate the performance of the workflow execution using the two schedulers in our simulated grid environment. The assumptions made in using the two schedulers and in discussing the performance results are as follows:

1. The capability of resource advanced reservation is available in the local scheduler. This is reasonable because this feature is supported by most existing local schedulers, such as SGE, LSF, and PBS Pro.

2. The scheduler is evaluated based on the execution of a single workflow a time. In other words, it does not schedule multiple workflows at the same time,

which may create race conditions in the handshaking process during multiple reservation requests to the local scheduler.

3. The job generators of grid resources maintain the resource load in the same level. In doing this, we isolate the impact on the workflow performance by resource load from other factors.

### 7.3.1 Performance Evaluation of a 7-Task Workflow

Our first workflow example is a 7-task workflow, as shown in Figure 7.1, and the task specifications and dependency specifications are shown in Table 7.2 and Table 7.3.

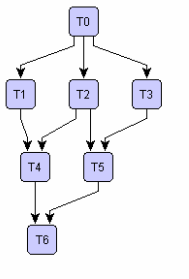



Figure 7.1: A 7-Task Workflow

Table 7.2: Task Specifications of the 7-Task Workflow

| Task | 100xMIs | NumPEs | 100xMIs/PE |
|---|---|---|---|
| T0 | 250 | 16 | 15.625 |
| T1 | 400 | 32 | 12.5 |
| T2 | 200 | 32 | 6.25 |
| T3 | 300 | 32 | 9.375 |
| T4 | 450 | 64 | 7.03 |
| T5 | 340 | 16 | 21.25 |
| T6 | 420 | 32 | 13.125 |

Table 7.3: Dependency Specifications of the 7-Task Workflow

| Parent Task | Child Task | File Size (MByte) |
|---|---|---|
| T0 | T1 | 100 |
| T0 | T2 | 230 |
| T0 | T3 | 300 |
| T1 | T4 | 400 |
| T2 | T4 | 1000 |
| T2 | T5 | 490 |
| T3 | T5 | 1600 |
| T4 | T6 | 400 |
| T5 | T6 | 380 |

Under our grid simulation environment, both of the two schedulers can easily find the optimal schedule for the workflow in Figure 7.1 and using this schedule, the best performance can be achieved. This schedule is made under the assumption that all the grid resources are available for the workflow, i.e., the resource load is 0 and the network bandwidth is the maximum one (10 MB/s). There is no queue waiting when submit a task to the local scheduler. The optimal schedule for the 7-task workflow is shown in Table 7.4. In this schedule, the critical tasks are *T0*, *T2*, *T5*, *T76*. Of the workflow execution, the total time spent for executing tasks, i.e., 344.4 seconds, is the sum of the execution times of the critical tasks; the executions of non-critical tasks overlap the executions of critical tasks. The total time spent for transferring immediate files, i.e., 110 seconds, is the sum of the transfer times for the input files to the critical tasks, while considering the overlapping transfers of multiple files to the same task.

Figure 7.2 shows the execution time of the 7-task workflow under resource load from 0.0 to 0.9 using the just-in-time scheduler and the GRACCE scheduler. The

Table 7.4: The Optimal Schedule of the 7-Task Workflow

| Task | Resource | Task ExeTime (s) | Data Transfer From | Data Transfer Time (s) | Task Completion Time (s) |
|---|---|---|---|---|---|
| **T0** | R7 | 78.2 | N/A | 0 | 78.2 |
| T1 | R6 | 75 | R7 (T0) | 10 | 163.2 |
| **T2** | R4 | 50 | R7 (T0) | 23 | 151.2 |
| T3 | R5 | 60 | R7 (T0) | 30 | 168.2 |
| T4 | R2 | 60 | R6 (T1)<br>R4 (T2) | 40<br>100 | 311.2 |
| **T5** | R5 | 136.8 | R4 (T2)<br>R5 (T3) | 49<br>0 | 337 |
| **T6** | R6 | 79.4 | R2 (T4)<br>R5 (T5) | 40<br>38 | 454.4 |
| Total | | 344.4 | | 110 | 454.4 |

workflow execution time increases as the load of grid resources increases. But under the same resource load, the GRACCE scheduler achieves a significant reduction in execution time compared to the just-in-time scheduler. The biggest difference is the queue waiting time in the two schedulers. The data transfer time and the task execution time do not change a lot as the resource load changes.

The distribution of the workflow execution time is shown in Figure 7.3. The time spent in task execution increases generally as the resource load increases, but not uniformly. The scheduler does not necessarily allocate the fastest resource available for a workflow task. The two schedulers consider both the queue waiting time and the time to transfer the intermediate files when allocating a resource. The GRACCE scheduler also takes into account whether an advanced reservation has been granted. So it is very common that the scheduler allocates a slower resource to a task, either because it has been reserved or because it leads to a shorter queue waiting time.

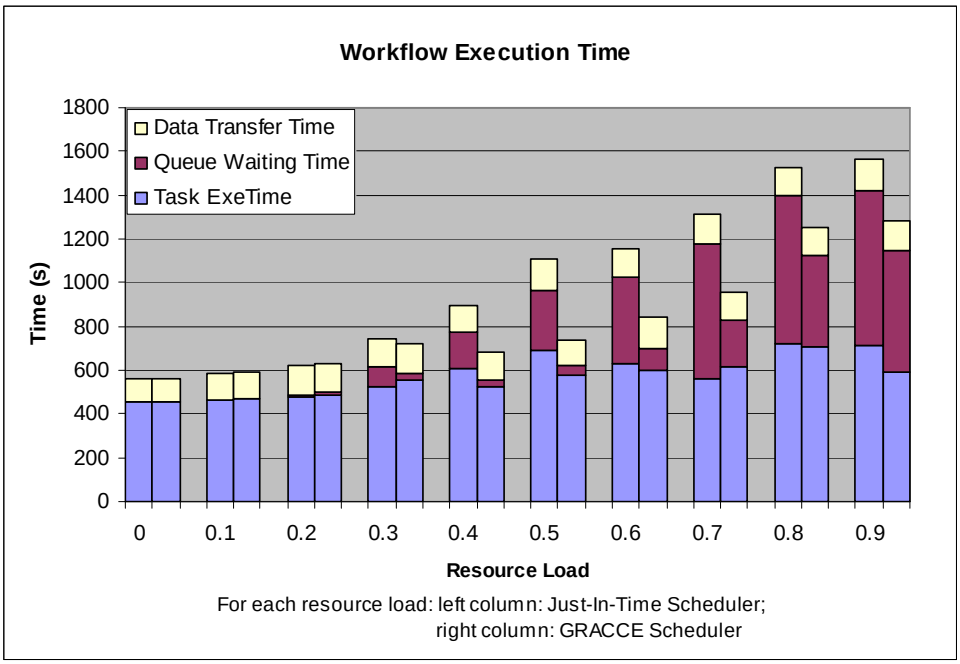



Figure 7.2: Execution Time of the 7-Task Workflow

When the resource load is above 0.2, the queue waiting time is reduced greatly under the GRACCE scheduler. Although the queue waiting time increases when the resource load rises above 0.3 under both schedulers, the savings achieved using the GRACCE scheduler are significant, from 35% to 85%. The reason why the GRACCE scheduler cannot cut down the queue waiting time to the minimum under higher resource load is because most of these queue waiting times are spent by the first several tasks, and it is too late to reserve a resource without introducing any queue waiting.

Similar to the task execution time, the data transfer time does not change a lot under different resource loads. That is because the data transfers for the same level of tasks, such as *T1*, *T2* and *T3*, or *T4* and *T5*, or *T6* are overlapped, and

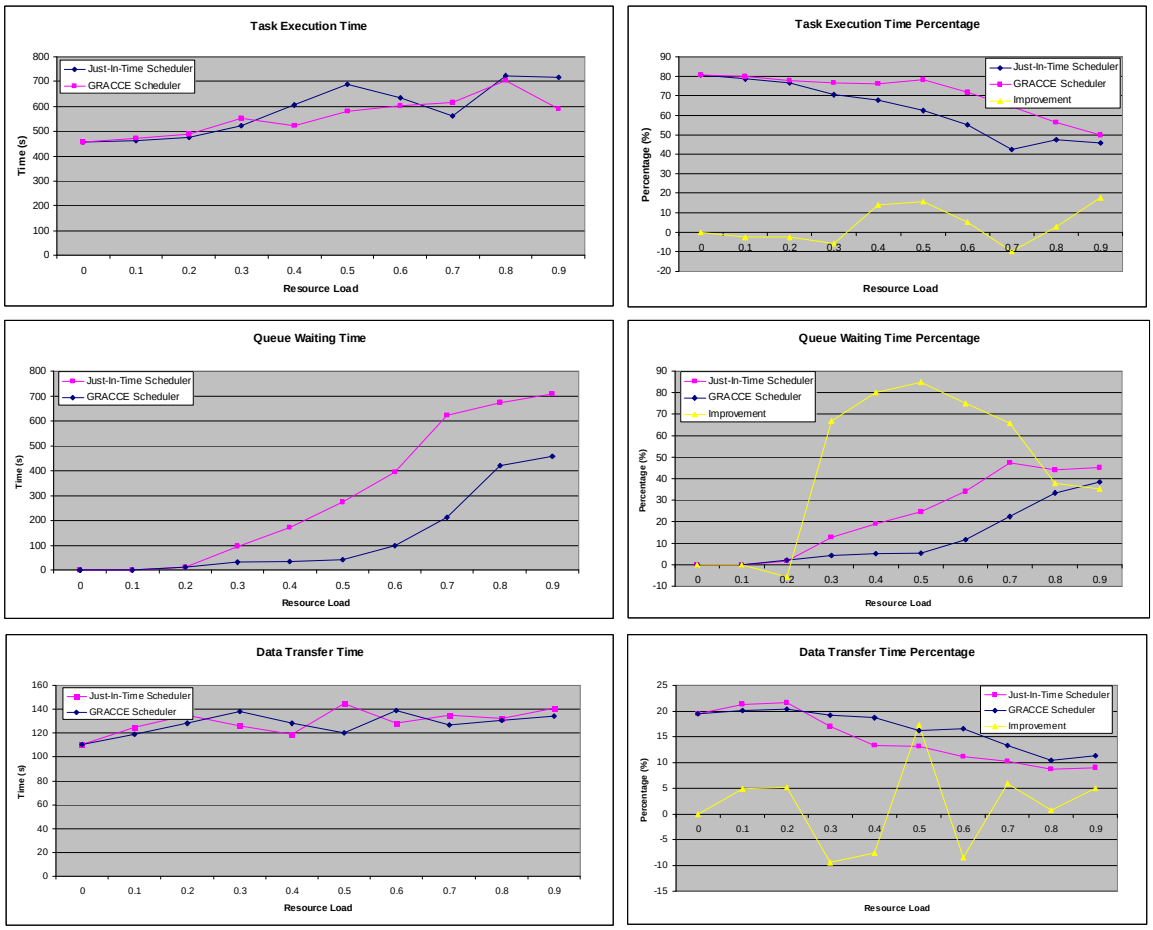



Figure 7.3: Execution Time Distribution of the 7-Task Workflow

of the total 7 tasks, such three overlapped transfers do not introduce much of the performance increase or decrease in the overall workflow execution time. In terms of the data transfer time percentage, it is similar to the task execution time percentage. They both decrease as the resource load increases, which is mainly because the total workflow execution time increases greatly.

In Figure 7.4, we show the distribution of the improvement obtained for the 7-task workflow by using the GRACCE workflow scheduler rather than the just-in-time

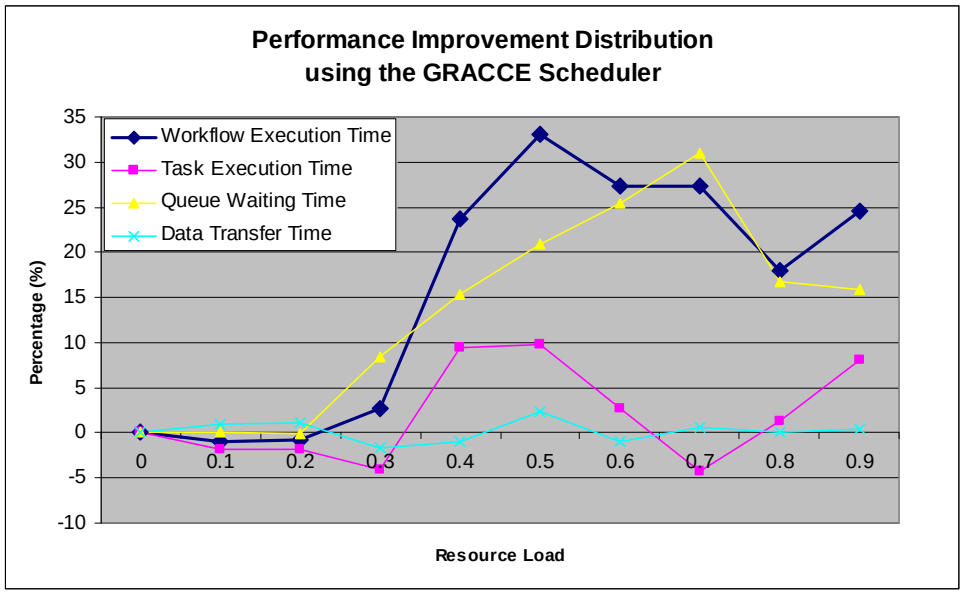



Figure 7.4: Performance Improvement Distribution of the 7-Task Workflow

scheduler. The performance improvements come mostly from the reduction in queue waiting time. We can also see from the figure that the biggest improvement comes when the resource load is 0.5. Before this point, the minimum queue waiting time is reached. With resource load above 0.5, even the GRACCE scheduler cannot reduce the queue waiting time to the minimum because it is too late to make immediate resource reservation for the first several tasks under higher resource load. That is why we cannot see the similar performance improvement under higher resource load.

### 7.3.2 Performance Evaluation of a 20-Task Workflow

We next consider a 20-task workflow, shown in Figure 7.5. It has much higher MIs for each task than in the 7-task workflow, which means that the task execution time is a more significant contributor to the workflow execution. The workflow execution

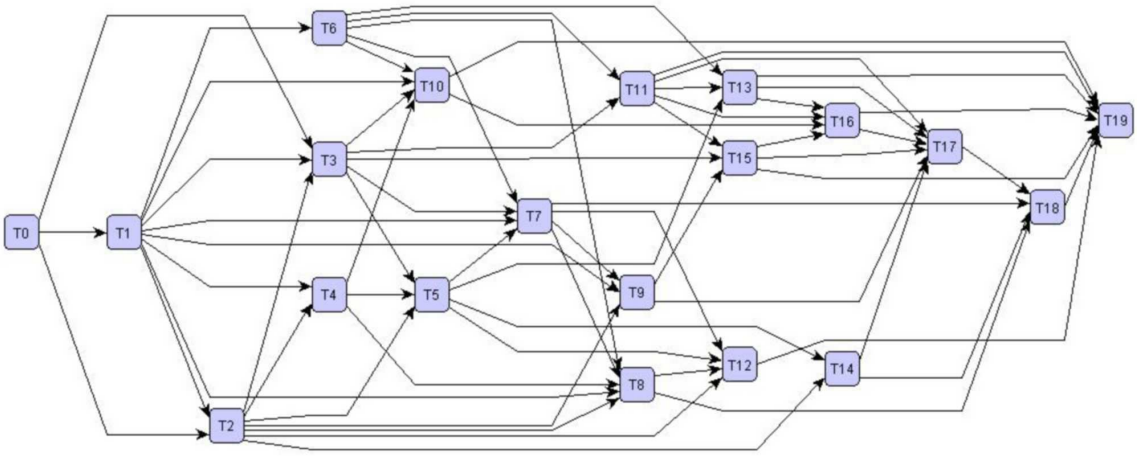



Figure 7.5: A 20-Task Workflow

times under the two schedulers are plotted in Figure 7.6. It shows that the workflow execution time using the GRACCE scheduler is less than that using the just-in-time scheduler when the resource load is above 0.3. The same as the 7-task workflow execution, most of the reduction in execution time comes from the lower queue waiting time.

The Figure 7.7 shows the distribution of the workflow execution time and the percentage of each individual processing time in the total workflow execution time using the two schedulers. The task execution time using the GRACCE scheduler is greater than that using the just-in-time scheduler when the resource load is above 0.1. We have mentioned the reason before when we evaluated the 7-task workflow. In terms of the queue waiting time, we see the GRACCE scheduler is able to reduce it greatly when the resource load is above 0.4. For the data transfer time, using the GRACCE scheduler improves by about 10% to 25% with resource load above 0.2. But using both schedulers, the data transfer time is only a very small portion of the

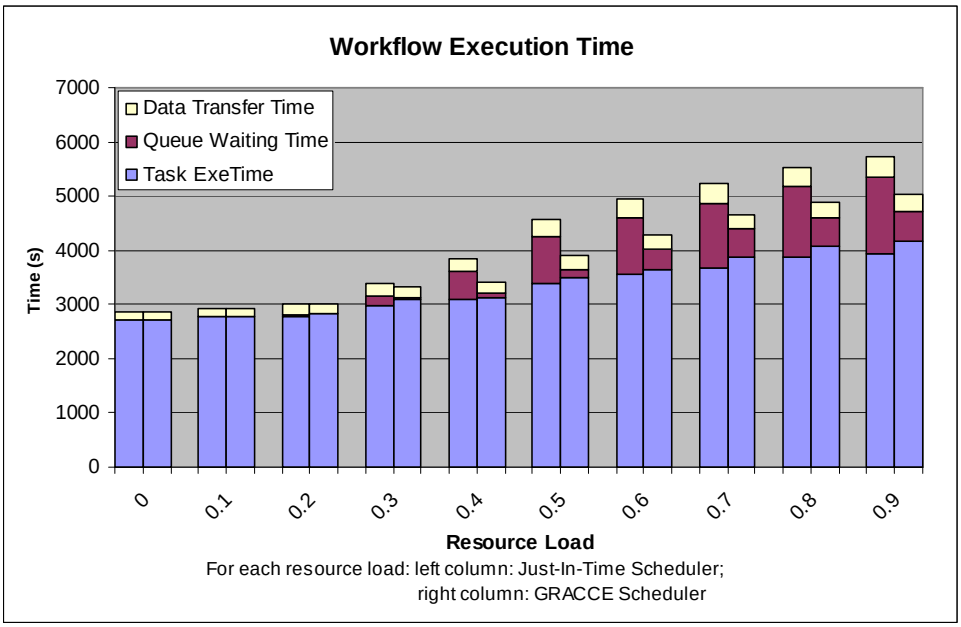



Figure 7.6: Execution Time of the 20-Task Workflow

workflow execution, about 5%, so the performance improvement using the GRACCE scheduler does not substantially reduce the overall execution time.

The distribution of performance improvement under the GRACCE scheduler is shown in Figure 7.8. For a moderate to high resource load (0.4 to 0.9), the GRACCE scheduler is able to reduce the workflow execution time by about 4% to 9%, the queue waiting time by about 4% to 12% and data transfer time by about 1% to 2%. However, it slows down the workflow task execution by 2% to 5%. As for the 7-task workflow, the reduction of queue waiting time contributes most of the performance improvement of the workflow execution.

### 7.3.3 Summary

In order to compare the performance obtained under the two scheduling approaches, we have collected the execution times of 60 workflows that were generated by our random workflow generators. The queue waiting time percentages in the total workflow execution times of these workflows using the two schedulers are plotted in Figure 7.9. Using the just-in-time scheduler, the queue waiting time is in the range of 0% to 55%. If using the GRACCE scheduler, the range is of 0% to 25%. So our GRACCE scheduler with workflow planning and reservation is able to reduce the queue waiting time significantly under different resource loads.

The average performance improvement distribution obtained by using the GRACCE scheduler is shown in Figure 7.10. The workflow execution time is reduced from 2% to 20% if the resource load is above 0.3. As in our examples, the reduction of queue waiting time is the biggest contributor. Again the reduction of data transfer time is within 3% and the task execution time increases by about 3% to 5%. We can also see that the biggest performance improvement is obtained with a load of 0.6 to 0.7.

In summary, compared to the widely-used just-in-time scheduler, our GRACCE scheduler is able to improve the workflow execution performance by about 20% under moderate and high resource load. This performance improvement is achieved by the reduction of the task queue waiting time using the techniques of workflow execution planning and resource advanced reservation. In order to reduce the overall queue waiting time, the scheduler may allocate slower resources to some workflow tasks, causing the increase of the task execution time.

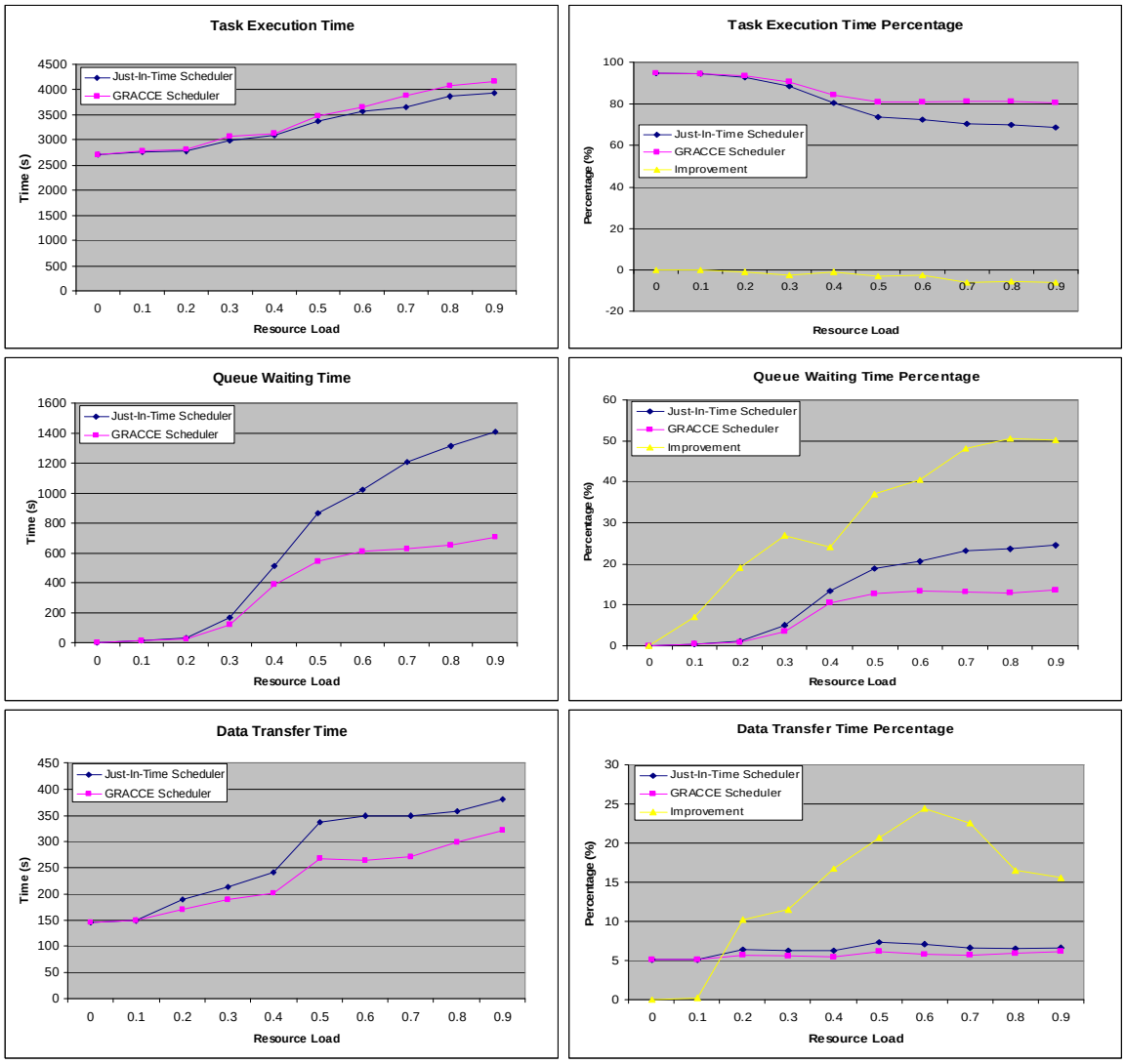



Figure 7.7: Execution Time Distribution of the 20-Task Workflow

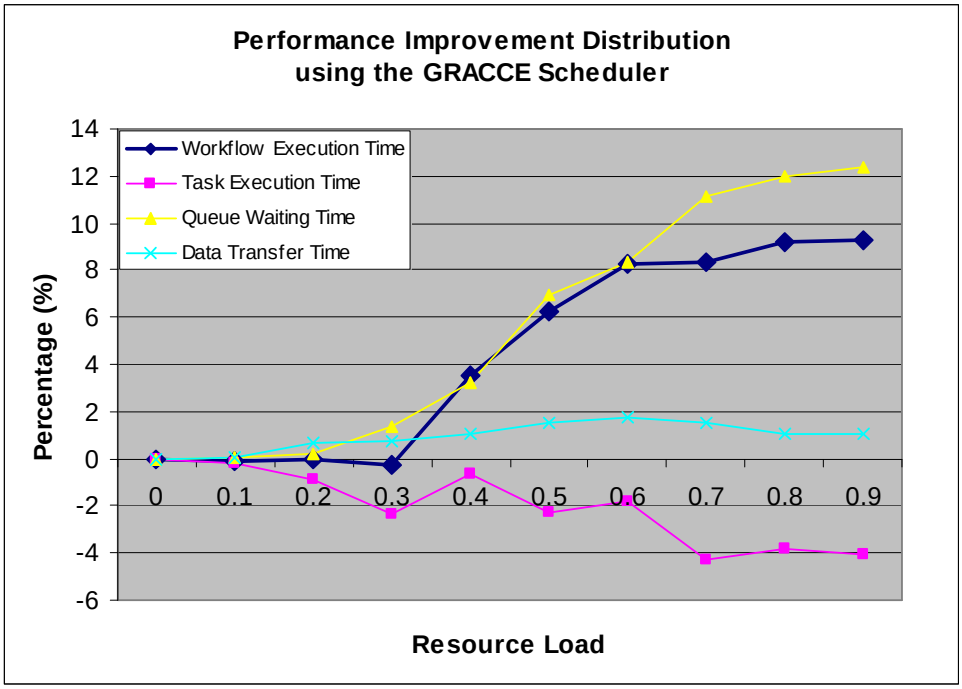



Figure 7.8: Performance Improvement Distribution of the 20-Task Workflow

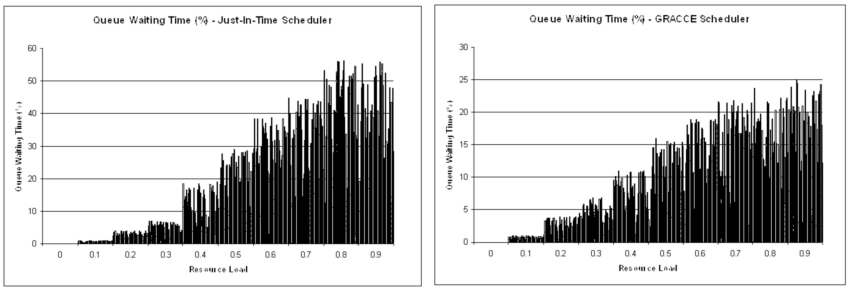



Figure 7.9: Queue Waiting Time Summary

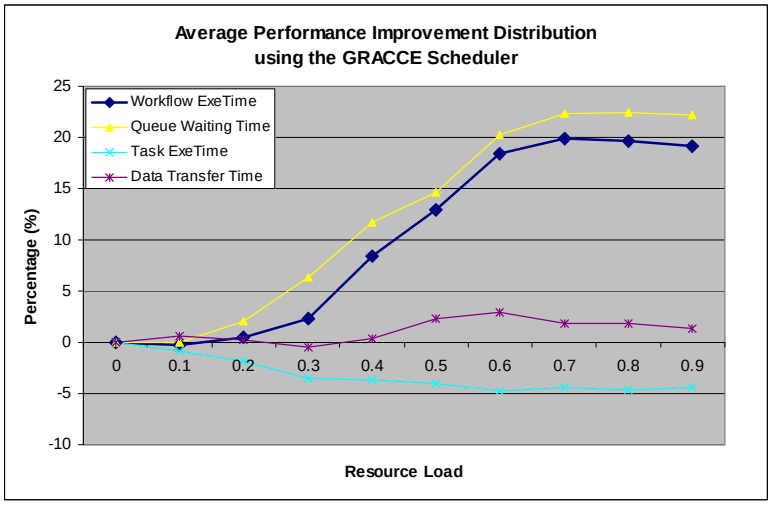



Figure 7.10: The Distribution of Performance Improvement

# Chapter 8

# Conclusion

Scientific workflow applications in distributed and grid environments involve multiple modules coupled together in execution to accomplish certain domain goals. An uninterrupted and coordinated execution of such an application requires both the mapping and scheduling of the application modules onto grid resources, and the on-time and automatic processing of dependencies between coupled modules. Furthermore, users expect not only the correct and uninterfered execution of the workflow, but also have certain quality of services requirements, commonly, the performance. This is the topic of grid workflow scheduling, which has been an active and open research area recently.

This research studies the topic of executing scientific workflow applications in the dynamic distributed and grid environments. The dissertation researches the issues of workflow description and scheduling, and defines a full-featured workflow description language and an integrated architecture of a grid workflow system. In defining the

architecture, an advanced scheduler with features of workflow planning, resource advanced reservation and performance prediction is proposed. Our simulation results show that using our workflow scheduler, the workflow execution performance gains about 20% improvement under high resource load, compared to the regular, widely used workflow scheduling approach.

The workflow description language defined in this work addresses the issue of lacking support for workflow resource allocation in current workflow description languages. Using our language, users no longer need to manually specify resource multi-request for the workflow tasks, nor need to resort to another resource specification language for describing task resource requests, which are proved to be inflexible and inconvenient for end users. Other than that, our language introduces lots of other features that are not found in other languages and are very convenient to end users and workflow developers.

## 8.1 Future Work

We will first refine and enhance our simulation environment with more features and capabilities, and make it as close as possible to a real grid environment. Based on that, we will further study the behavior of our scheduler. Along with that, we will set up a real grid environment that has the widely-used local scheduler, such as LSF, PBS, or SGE, installed on resources, and test our scheduler on this environment to study the performance improvement. In terms of the workflow scheduler, in current scheduling policy, the guideline of allocating a resource for a task is to improve

the performance of this task as much as possible in order to improve the workflow execution performance. We believe a more intelligent scheduler should also consider the performance impacts of the task resource allocation on its child and sibling tasks. We will study what algorithms can be used in such a scheduler and enhance our scheduler with this feature.

# Bibliography


[1] I. Foster and C. Kesselman, Eds., *The Grid: Blueprint for a New Computing Infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[2] T. Prudhomme, C. Kesselman, T. Finholt, I. Foster, D. Parsons, D. Abrams, J.-P. Bardet, R. Pennington, J. Towns, R. Butler, J. Futrelle, N. Zaluzec, and J. Hardin, "NEESgrid: A Distributed Virtual Laboratory for Advanced Earthquake Experimentation and Simulation: Scoping Study," Tech. Rep., 2001, http://it.nees.org/library.

[3] W. E. Allcock, I. T. Foster, V. Nefedova, A. L. Chervenak, E. Deelman, C. Kesselman, J. Lee, A. Sim, A. Shoshani, B. Drach, and D. Williams, "High-Performance Remote Access to Climate Simulation Data: A Challenge Problem for Data Grid Technologies," in *Supercomputing 2001*, 2001, p. 46.

[4] E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams, and S. Koranda, "GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists," in *Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2002, p. 225.

[5] Kelvin K. Droegemeier, V. Chandrasekar, R. Clark, D. Gannon, S. Graves, E. Joseph, M. Ramamurthy, R. Wilhelmson, K. Brewster, B. Domenico, T. Leyton, V. Morris, D. Murray, B. Plale, R. Ramachandran, D. Reed, J. Rushing, D. Weber, A. Wilson, M. Xue, and S. Yalda, "Linked Environments for Atmospheric Discovery (LEAD): A Cyberinfrastructure for Mesoscale Meteorology Research and Education," in *20th International Conference on Interactive Information and Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology*, 2004.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Topological Sort," in *Introduction to algorithms, 2nd Edition*. Cambridge, MA, USA: MIT Press, 2001, ch. Section 22.4, pp. 549–552.

[7] "The Grid Workflow Forum," http://www.gridworkflow.org.

[8] "DAGMan (Directed Acyclic Graph Manager)," http://www.cs.wisc.edu/condor/dagman.

[9] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe, "Taverna: Lessons in Creating a Workflow Environment for the Life Sciences," *Concurrency and Computation: Practice & Experience*, vol. 18, no. 10, pp. 1067–1100, 2006.

[10] "Java CoG Kit Karajan/Gridant Workflow Guide," http://www.cogkit.org/release/4_0_a1/manual/workflow.pdf.

[11] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang, "Programming Scientific and Distributed Workflow with Triana Services," *Concurrency and Computation: Practice & Experience*, vol. 18, no. 10, pp. 1021–1037, 2006.

[12] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, J. Clovis Seragiotto, and H.-L. Truong, "ASKALON: A Tool Set for Cluster and Grid Computing," *Concurrency and Computation: Practice & Experience*, vol. 17, no. 2-4, pp. 143–169, 2005.

[13] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda, "Mapping Abstract Complex Workflows onto Grid Environments," *Journal of Grid Computing*, vol. 1, no. 1, pp. 25–39, 2003.

[14] J. Yu, R. Buyya, and C. K. Tham, "Cost-based Scheduling of Scientific Workflow Applications on Utility Grids," in *Proceedings of the 1st IEEE International Conference on e-Science and Grid Computing*. Melbourne, Australia: IEEE Computer Society, 2005.

[15] A. M. Weinberg, "Impact of Large-Scale Science on the United States," *Science*, vol. 134, pp. 161–164, July 1961.

[16] A. Berson, *Client/Server Architecture*. New York, NY, USA: McGraw-Hill, Inc., 1992.

[17] "Common Object Request Broker Architecture (CORBA)," http://www.corba.org.

[18] "OpenSSL: The Open Source toolkit for SSL/TLS," http://www.openssl.org.

[19] "Secure Shell," http://www.openssh.org.

[20] "The Message Passing Interface (MPI) standard," http://www-unix.mcs.anl.gov/mpi/.

[21] "The OpenMP (Open Multi-Processing)," http://www.openmp.org.

[22] "Sun Grid Engine, Sun Microsystems," http://gridengine.sunsource.net.

[23] "Load Sharing Facility, Resource Management and Job Scheduling System," http://www.platform.com/products/HPC.

[24] "Portable Batch System (PBS) Professional," http://www.altair.com/software/pbspro.htm.

[25] "TOP500 Supercomputing Sites," http://www.top500.org.

[26] "Big Science, in Encyclopedia Britannica," http://www.britannica.com/eb/article-9117806/Big-Science.

[27] "European Organization for Nuclear Research," http://www.cern.ch/.

[28] "Hubble Space Telescope (HST)," http://hubble.nasa.gov/index.php.

[29] C. Murray and C. B. Cox, *Apollo: The Race to the Moon.* Simon & Schuster, 1989.

[30] "World Wide Web Consortium," http://www.w3.org.

[31] "Big Computers For Big Science," http://www.physorg.com/news903.html.

[32] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International Journal of High Performance Computing Applications*, vol. 15, no. 3, pp. 200–222, 2001.

[33] D. E. Atkins, K. K. Droegemeier, S. I. Feldman, H. Garcia-Molina, M. L. Klein, D. G. Messerschmitt, P. Messina, J. P. Ostriker, and M. H. Wright, "Revolutionizing Science and Engineering Through Cyberinfrastructure," *Report of the National Science Foundation Blue Ribbon Advisory Panel on CyberInfrastructure*, 2003.

[34] I. Foster, J. Geisler, B. Nickless, W. Smith, and S. Tuecke, "Software Infrastructure for the I-WAY High-Performance Distributed Computing Experiment," in *Proceedings of the High Performance Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 1996, p. 562.

[35] "Grid Computing Info Centre (GRID Infoware)," http://www.gridcomputing.com.

[36] "Grid Application and Deployment Projects in Science and Engineering," http://www-fp.mcs.anl.gov/~foster/grid-projects/.

[37] I. Foster and C. Kesselman, "The Globus Project: A Status Report," in *Proceedings of the Seventh Heterogeneous Computing Workshop*. Washington, DC, USA: IEEE Computer Society, 1998, p. 4.

[38] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, "A Security Architecture for Computational Grids," in *Proceedings of the 5th ACM conference on Computer and communications security*. New York, NY, USA: ACM Press, 1998, pp. 83–92.

[39] K. Czajkowski, I. T. Foster, N. T. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, "A Resource Management Architecture for Metacomputing Systems," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. London, UK: Springer-Verlag, 1998, pp. 62–82.

[40] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy, "A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation," in *Proceedings of the International Workshop on Quality of Service*, 1999.

[41] K. Czajkowski, C. Kesselman, S. Fitzgerald, and I. Foster, "Grid Information Services for Distributed Resource Sharing," vol. 00. IEEE Computer Society, 2001, p. 0181.

[42] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke, "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets," *Journal of Network and Computer Applications*, vol. 23, no. 3, pp. 187–200, July 2000.

[43] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "Grid Services for Distributed System Integration," *Computer*, vol. 35, no. 6, pp. 37–46, 2002.

[44] M. Baker, R. Buyya, and D. Laforenza, "Grids and Grid Technologies for Wide-Area Distributed Computing," *SoftwarePractice & Experience*, vol. 32, no. 15, pp. 1437–1466, 2002.

[45] *Workload Management with LoadLeveler*, IBM Redbook Abstract, 2001.

[46] "Ganglia Monitoring System," http://ganglia.sourceforge.net.

[47] "Simple Object Access Protocol," http://www.w3.org/TR/soap/.

[48] D. K. Barry, *The Savvy Manager's Guide to Web Services and Service-Oriented Architectures*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[49] "The Globus Project," http://www.globus.org.

[50] W. E. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link, "The Globus Striped GridFTP Framework and Server," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 54.

[51] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, M. Ripeanu, B. Schwartzkopf, H. Stockinger, K. Stockinger, and B. Tierney, "Giggle: a Framework for Constructing Scalable Replica Location Services," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–17.

[52] *Web Services Description Language (WSDL)*, World Wide Web Consortium, http://www.w3.org/TR/wsdl.

[53] T. L. Casavant and J. G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 141–154, 1988.

[54] J. Nabrzyski, J. M. Schopf, and J. Weglarz, Eds., *Grid Resource Management: State Of the Art and Future Trends*. Norwell, MA, USA: Kluwer Academic Publishers, 2004.

[55] Y. Yan and B. Chapman, "Campus Grids Meet Applications: Modeling, Metascheduling and Integration," *Journal of Grid Computing*, vol. 4, no. 2, pp. 159–175, June 2006.

[56] I. Foster, E. Alpert, A. Chervenak, B. Drach, C. Kesselman, V. Nefedova, D. Middleton, A. Shoshani, A. Sim, D. Williams, "Earth System Grid II (ESG): Turning Climate Model Datasets Into Community Resources," *Proceedings of the American Meterologcal Society Conference*, 2001.

[57] L. Pearlman, C. Kesselman, S. Gullapalli, B. F. Spencer, J. Futrelle, K. Ricker, I. Foster, P. Hubbard, and C. Severance, "Distributed Hybrid Earthquake Engineering Experiments: Experiences with a Ground-Shaking Grid Application," in *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 14–23.

[58] T. Sun, J. Liu, I. Shen, and Y. Ma, "Numerical Simulation of Car Crash Analysis Based on Distributed Computational Environment," in *Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2002, p. 334.

[59] E. Deelman, R. Plante, C. Kesselman, G. Singh, M.-H. Su, G. Greene, R. Hanisch, N. Gaffney, A. Volpicelli, J. Annis, V. Sekhri, T. Budavari, M. Nieto-Santisteban, W. O'Mullane, D. Bohlender, T. McGlynn, A. Rots, and O. Pevunova, "Grid-Based Galaxy Morphology Analysis for the National Virtual Observatory," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 47.

[60] K. Keahey, T. Fredian, Q. Peng, D. P. Schissel, M. Thompson, I. Foster, M. Greenwald, and D. McCune, "Computational Grids in Action: the National Fusion Collaboratory," *Future Generation Computer Systems*, vol. 18, no. 8, pp. 1005–1015, 2002.

[61] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, *Workflows for e-Science: Scientific Workflows for Grids*. Springer, 2006, ch. 1.

[62] "Workflow Management Coalition," http://www.wfmc.org/.

[63] *The Workflow Reference Model*, Workflow Management Coalition, 1995.

[64] T. Murata, "Petri Nets: Properties, Analysis and Applications," in *Proceedings of the IEEE*, vol. 77, Apr. 1989, pp. 541–580.

[65] W.M.P. van der Aalst, "The Application of Petri Nets to Workflow Management," *The Journal of Circuits, Systems and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

[66] N. R. Adam, V. Atluri, and W. Huang, "Modeling and Analysis of Workflows Using Petri Nets," *Journal of Intelligent Information Systems*, vol. 10, no. 2, pp. 131–158, 1998.

[67] M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[68] I. Foster, J.-S. Vöckler, M. Wilde, and Y. Zhao, "Chimera: A Virtual Data System for Representing, Querying and Automating Data Derivation," in *14th International Conference on Scientific Database Management*, 2002, pp. 37–46.

[69] "The Virtual Data Language Reference Manual," http://www.griphyn.org/workspace/VDS/langref/index.html.

[70] "XScufl Language Reference," http://taverna.sourceforge.net/docs/xscuflspecification.html.

[71] M. Wieczorek, R. Prodan, and T. Fahringer, "Scheduling of Scientific Workflows in the ASKALON Grid Environment," *ACM SIGMOD Record Journal*, vol. 34, no. 3, pp. 56–62, 2005.

[72] W. M. P. van der Aalst and A. H. M. ter Hofstede, "YAWL: Yet Another Workflow Language," *Information System*, vol. 30, no. 4, pp. 245–275, 2005.

[73] "BPEL4WS: Business Process Execution Language for Web Services v1.0," http://www.106.ibm.com/developerworks/webservices/library/wsbpel.

[74] D. D. Roure and J. A. Hendler, "E-Science: The Grid and the Semantic Web," *IEEE Intelligent Systems*, vol. 19, no. 1, pp. 65–71, 2004.

[75] "Resource Description Framework (RDF)," http://www.w3.org/RDF.

[76] "OWL Web Ontology Language Overview," http://www.w3.org/TR/owl-features.

[77] "The Globus Resource Specification Language RSL v1.0," http://www-fp.globus.org/gram/rsl_spec1.html.

[78] B. Ludscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific Workflow Management and the Kepler System," *Concurrency and Computation: Practice & Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[79] M. Wieczorek, M. Siddiqui, A. Villazon, R. Prodan, and T. Fahringer, "Applying Advance Reservation to Increase Predictability of Workflow Execution on the Grid," *Second IEEE International Conference on e-Science and Grid Computing*, vol. 0, p. 82, 2006.

[80] J. Yu and R. Buyya, "Scheduling Scientific Workflow Applications with Deadline and Budget Constraints using Genetic Algorithms," *Scientific Programming Journal, Special Issue: Scientific Workflows*, vol. 14, pp. 217–230, 2006.

[81] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd, "Gridflow: Workflow management for grid computing," in *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2003, p. 198.

[82] K. Czajkowski, I. Foster, and C. Kesselman, "Resource Co-Allocation in Computational Grids," in *Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 1999, p. 37.

[83] "Maui Moab Grid Scheduler," http://www.clusterresources.com/products/mgs.

[84] G. Mateescu, "Quality of Service on the Grid via Metascheduling with Resource Co-Scheduling and Co-Reservation," *International Journal of High Performance Computing Applications*, vol. 17, no. 3, pp. 209–218, 2003.

[85] W. Smith, I. Foster, and V. Taylor, "Scheduling with Advanced Reservations," *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, vol. 00, p. 127, 2000.

[86] K. Czajkowski, I. T. Foster, C. Kesselman, V. Sander, and S. Tuecke, "SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems," in *The 8th International Workshop on Job Scheduling Strategies for Parallel Processing*. London, UK: Springer-Verlag, 2002, pp. 153–183.

[87] A. Pugliese and D. Talia, "Application-Oriented Scheduling in the Knowledge Grid: A Model and Architecture," in *International Conference on Computational Science and its Applications (ICCSA)*. Berlin, Germany: Springer-Verlag, 2004, pp. 55–65.

[88] "Community Scheduler Framework," http://www.platform.com/products/Globus.

[89] R. Buyya, D. Abramson, and J. Giddy, "Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid," in *The 4th International Conference on High Performance Computing in Asia-Pacific Region*, vol. 01. Los Alamitos, CA, USA: IEEE Computer Society, 2000, p. 283.

[90] A. Bose, B. Wickman, and C. Wood, "MARS: A Metascheduler for Distributed Resources in Campus Grids," in *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 110–118.

[91] F. D. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao, "Application-Level Scheduling on Distributed Heterogeneous Networks," in *Proceedings of the ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 1996, p. 39.

[92] "GRACCE: GRid Application Coordination, Collaboration and Execution," http://www.cs.uh.edu/~gracce.

[93] B. Chapman, P. Raghunath, B. Sundaram, and Y. Yan, "Air Quality Prediction in a Production Quality Grid Environment," in *Engineering the Grid: Status and Perspective*, B. D. Martino, J. Dongarra, A. Hoisie, L. T. Yang, and H. Zima, Eds. American Scientific Publishers.

[94] B. Chapman, H. Donepudi, J. He, Y. Li, P. Raghunath, B. Sundaram, and Y. Yan, "Grid Environment with Web-Based Portal Access for Air Quality Modeling," in *Parallel and Distributed Scientific and Engineering Computing: Practice and Experience*, Y. Pan and L. T. Yang, Eds. Nova Science Publishers, vol. 15.

[95] B. M. Chapman, H. Donepudi, Y. Li, P. Raghunath, B. Sundaram, Y. Yan, and J. He, "An OGSI-Compliant Portal for Campus Grids," in *10th ISPE International Conference on Concurrent Engineering: Research and Applications*, 2003, pp. 987–994.

[96] Y. Yan, B. M. Chapman, and B. Sundaram, "Air Quality Forecasting on Campus Grid Environment," in *the Workshop on Grid Applications: From Early Adopters to Mainstream Users, GGF14*, June 2005.

[97] "W3C XML Schema," http://www.w3.org/XML/Schema.

[98] "The XMLBeans," http://xmlbeans.apache.org/.

[99] "Web Services Notification," http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn.

[100] "Grid Resource Allocation Agreement Protocol Working Group (GRAAP-WG)," https://forge.gridforum.org/sf/projects/graap-wg.

[101] S.-H. Jang, V. E. Taylor, X. Wu, M. Prajugo, E. Deelman, G. Mehta, and K. Vahi, "Performance Prediction-based versus Load-based Site Selection: Quantifying the Difference," in *Proceedings of the 18th International Conference on Parallel and Distributed Computing Systems*, 2005, pp. 148–153.

[102] R. Gibbons, "A Historical Application Profiler for Use by Parallel Schedulers," in *Proceedings of the Job Scheduling Strategies for Parallel Processing*. London, UK: Springer-Verlag, 1997, pp. 58–77.

[103] R. Wolski, "Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service," in *Proceedings of the 6th International Symposium on High Performance Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 1997, p. 316.

[104] R. Buyya and M. Murshed, "GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing," *The Journal of Concurrency and Computation: Practice and Experience (CCPE)*, vol. 14, 2002.

[105] M. A. Jette, "Performance Characteristics of Gang Scheduling in Multiprogrammed Environments," in *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 1997, pp. 1–12.

[106] "yFiles - Java Graph Layout and Visualization Library," http://www.yworks.com/en/products_yfiles_about.htm.

[107] H. Li, J. Chen, Y. Tao, D. Gro, and L. Wolters, "Improving a Local Learning Technique for Queue Wait Time Predictions," *Sixth IEEE International Symposium on Cluster Computing and the Grid*, vol. 0, pp. 335–342, 2006.

[108] W. Smith, V. E. Taylor, and I. T. Foster, "Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance," in *Proceedings of the Job Scheduling Strategies for Parallel Processing*. London, UK: Springer-Verlag, 1999, pp. 202–219.

[109] A. B. Downey, "Predicting Queue Times on Space-Sharing Parallel Computers," in *Proceedings of the 11th International Symposium on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 209–218.

[110] D. Nurmi, A. Mandal, J. Brevik, C. Koelbel, R. Wolski, and K. Kennedy, "Evaluation of a Workflow Scheduler Using Integrated Performance Modelling and Batch Queue Wait Time Prediction," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 2006, p. 119.