Distributed execution of workflow

Khademi Hedayat, Maryam

2009

Khademi Hedayat, M. (2009). Distributed execution of workflow. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/19300>

<https://doi.org/10.32657/10356/19300>

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

**DISTRIBUTED EXECUTION OF
WORKFLOW**

**MARYAM KHADEMI HEDAYAT
SCHOOL OF COMPUTER ENGINEERING
2009**

Distributed Execution of Workflow

Maryam Khademi Hedayat

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Master of Engineering

2009

Acknowledgments

I would like to use this opportunity to offer my gratitude to my supervisors Prof. Wentong Cai. and Prof. Steve Turner, for their great guidance through my study and research, and also for their patience and knowledge. They allowed me to choose and act according to my own decision and ideas in different matters.

Great thanks go to our kind and helpful laboratory technician, Irene Goh, for her support in many problems I faced with.

During this almost two and half years, I have been blessed with a friendly attitude of my colleagues in PDCC. My special thanks should go to Yuhong Feng for her useful ideas toward my work.

I heartily thank my family, specially my mother and my father for supporting me throughout all my studies, with their love, passion and always being behind me.

And the last person to whom I deeply owe my gratitude is my husband, Hossein, whose presence brought a second spirit to my life. It is not easy to forget his consistent encouragement during the last months of writing up the thesis.

Contents

Acknowledgments	i
List of Notations	x
Abstract	xiii
1 Introduction	1
1.1 Introduction	1
1.1.1 Grid Computing	1
1.1.2 Scientific Workflow	2
Abstract Workflow	2
Concrete Workflow	3
1.2 Objective and Scope	3
1.3 Organization of the Thesis	5
2 Background and Literature Survey	7
2.1 Background	7
2.1.1 Globus Toolkit	7

Information Services	8
Intermediate Data Movement	8
Execution Management	9
2.1.2 Condor, Condor-G, DAGMan	9
2.1.3 Pegasus	10
2.2 Grid Workflow System Taxonomy	10
2.2.1 Mapping	11
2.2.2 Enactment	13
2.2.3 Data Movement	14
2.3 Map of Taxonomy to the Existing Systems	15
2.3.1 Mapping	15
2.3.2 Enactment	16
2.3.3 Data Movement	19
2.4 Partitioning	21
2.5 Summery	23
3 Distributed Execution of Workflow	26
3.1 Overview of the System	26
3.2 Design Layer	31
3.2.1 Modifier	31
3.2.2 Partition Assignment	32
3.3 Enactment Layer	33
3.3.1 Dispatcher	33
3.3.2 Engine	35
3.4 Middleware Layer	35

3.4.1	Local and Global Data Movement	36
3.5	Summery	37
4	Partitioning	38
4.1	Problem Statement	39
4.1.1	Input	39
4.1.2	Output	40
4.2	Cost Model	41
4.2.1	DAG Model	41
4.2.2	Grid Model	43
4.3	Completion Time	44
4.4	Binary-Division Partitioning Approach	47
4.4.1	Motivation	47
4.4.2	Binary Transformer	47
4.4.3	Partitioning Algorithm	51
	Phase I: One-To-One Transformation	58
	Phase II: MP-Decision	59
	Phase III: Division	62
	Phase IV: Partition Assignment	64
	Phase V: CT-Decision	70
4.4.4	Time Complexity Analysis	72
4.5	Summery	76
5	Experiments and Results	77
5.1	DAG Model	77
5.2	Grid Model	81

5.3	Execution Model	83
5.4	Experiment Sets	84
5.4.1	Experiment Set I	84
5.4.2	Experiment Set II	86
5.4.3	Experiment Set III	87
5.4.4	Experiment Set IV	91
5.4.5	Comparing Make-span of MPSE and MPME Execution Models	93
5.5	Summery	94
6	Conclusions	96
6.1	Summery of Contributions and Results	97
6.2	Recommendations of Future Work	98
	Appendices	100
	A: Sample of RCG DAGs Used as System's Input	101
	B: Sample of DAG Submit Files with Pre/Post-Scripts	105
	C: Publication List for this Research	107
	Bibliography	108

List of Figures

2.1	A Grid workflow taxonomy for our focused area	12
3.1	System layer components	27
3.2	PDWMS' sequence diagram	30
3.3	An overview of workflow enactment through engines' collaboration in PDWMS	34
4.1	Different relation types in a binary-DAG	48
4.2	Applying the <i>Join-transform</i> routine to task V and its six parents .	52
4.3	Applying the <i>Branch-transform</i> routine to task V and its five children	53
4.4	Flowchart of the partitioning algorithm	55
4.5	One-to-one transformation	59
4.6	Applying one-to-one transformation to a sample DAG	60
4.7	Different states of the left/right/common groups in the division phase	65
4.8	Sample DAG d and its two partitions after division	66
4.9	An overview of using partition assignment algorithm	69
4.10	Gantt chart of completion time of DAG d before division	73
4.11	Gantt chart of completion time of DAG d's partitions after division	74

5.1	Experiment Set I: Effect of computing resources' performance on make-span	85
5.2	Experiment Set II: Effect of computing resource combinations on make-span	87
5.3	Experiment Set III-1: Performance of MPSE on various DAGs . . .	88
5.4	Experiment Set III-2: Performance of MPSE on various DAGs . . .	90
5.5	Experiment Set IV: Performance of MPME on various DAGs	91
1	RCG	102
2	RCG+w	103
3	RCG+l	104

List of Tables

2.1	Common architecture of Grid workflow systems	8
2.2	Mapping schemes used by the available Grid workflow systems . . .	16
2.3	Enactment architecture adopted by the available Grid workflow systems	17
2.4	Intermediate data movement adopted by the available Grid workflow systems	20
2.5	Systems and their partitioning algorithm	21
2.6	PDWMS summery	25
4.1	Fan-in and fan-out of different relation types in a DAG	56
5.1	RCG groups specification	81
5.2	Clusters configuration	82
5.3	Experiment sets' specification	85
5.4	Comparing make-span of MPSE and MPME execution models-I . .	93
5.5	Comparing make-span of MPSE and MPME execution models-II . .	93
1	A part of partition RCG+w.dag file	106

List of Algorithms

1	Pseudo-code of binary-transform algorithm	50
2	Pseudo-code of one-to-one transformation phase	61
3	Pseudo-code of MP-decision phase	63
4	Pseudo-code of division phase	67
5	Pseudo-code of partition assignment phase	68
6	Pseudo-code of CT-decision phase	71

List of Notations

In this table the notation along with its description comes in first and second columns. Note that the first column entry in the format of $X_{[m1.m2.m3]}$ means that notation X first appears in Section m1.m2.m3. For example $DG_{[4.1.2]}$ means that the notation DG first appears in Section 4.1.2.

Notation	Description
$MS_{[1.2]}$	This denotes the make-span of workflow
$D_{[4.1]}$	This denotes a DAG
$V_{[4.1]}$	This denotes the set of tasks (vertices) of a DAG
$E_{[4.1]}$	This denotes the set of edges of a DAG
$v_{i[4.1]}$	This denotes a task (vertex) of a DAG
$e_{i,j[4.1]}$	This denotes an edge of a DAG connecting tasks v_i and v_j
$C_{[4.1]}$	This denotes the set of clusters
$P(c)_{[4.1]}$	This denotes the set of resource pools in cluster c
$DG_{[4.1.2]}$	This denotes a directed graph
$N_l(d)_{[4.2.1]}$	This denotes the number of levels in DAG d
$N_t(i, d)_{[4.2.1]}$	This denotes the number of tasks in level i of DAG d
$MP_{[4.2.1]}$	This denotes the maximum parallelism

Notation	Description
$MP(d)_{[4.2.1]}$	This denotes the maximum parallelism of DAG d
$CP(d)_{[4.2.1]}$	This denotes the critical path of DAG d, which is the heaviest path between entry/exit tasks of DAG d
$CT(d)_{[4.2.1]}$	This denotes the completion time of a DAG/sub-DAG d
$R/C(d)_{[4.2.1]}$	This denotes the avg ratio of task weights over edge weights in DAG d
$W(v_i)_{[4.2.1]}$	This denotes the weight of task v_i
$O(v_i, v_j)_{[4.2.1]}$	This denotes the weight of edge $e_{i,j}$ (connecting tasks v_i and v_j)
$Par(v_i)_{[4.2.1]}$	This denotes the parents of task v_i
$Chi(v_i)_{[4.2.1]}$	This denotes the children of task v_i
$FI(v_i)_{[4.2.1]}$	This denotes the fan-in of task v_i
$FO(v_i)_{[4.2.1]}$	This denotes the fan-out of task v_i
$L(v_i)_{[4.2.1]}$	This denotes the level of task v_i
$R(p_c)_{[4.2.2]}$	This denotes the set of resources of pool p in cluster c
$Trans-cost(c_i, c_j)_{[4.2.2]}$	This denotes the cost of transferring unit of data (1 MB) from cluster c_i to cluster c_j
$OS(r)_{[4.2.2]}$	This denotes the operating system of resource r
$Arch(r)_{[4.2.2]}$	This denotes the architecture of resource r
$Mem(r)_{[4.2.2]}$	This denotes the memory of resource r
$Sp(r)_{[4.2.2]}$	This denotes the speed of resource r
$Comp-cost(v_i)_{[4.2.2]}$	This denotes the computation cost of task v_i

Notation	Description
Comm- $\text{cost}(c_i, v_j, c_k, v_z)_{[4.2.2]}$	This denotes the communication cost of transferring output of task v_j in cluster c_i to task v_z in cluster c_k
$BFT_{[4.4.2]}$	This denotes the Breadth First Traverse
$Wid_{[5.1]}$	This denotes the width of graph
$Len_{[5.1]}$	This denotes the length of graph
$RCL_{[5.1]}$	This denotes the DAG category with R/C range of 0.1-0.3
$RCE_{[5.1]}$	This denotes the DAG category with R/C range of 0.8-1.2
$RCG_{[5.1]}$	This denotes the DAG category with R/C range of 3-10
$RCG + w_{[5.1]}$	This denotes the added-width version of RCG
$RCG + l_{[5.1]}$	This denotes the added-length version of RCG
$SPSE_{[5.3]}$	This denotes the Single-Pool-Single-Engine execution model
$MPSE_{[5.3]}$	This denotes the Multiple-Pool-Single-Engine execution model
$MPME_{[5.3]}$	This denotes the Multiple-Pool-Multiple-Engine execution model

Abstract

Grid computing is a fundamental technology for large scale distributed resource sharing. *Workflow management* is becoming one of the most important Grid services. A lot of research work has been done on different issues involved in workflow management systems. The focus of this project is on three areas: *workflow partitioning*, *enactment* and *data movement*. The main reasons which led us to this selection are: (i) there is few existing work on partitioning workflow, specially as a means of accelerating workflow execution; (ii) several systems have been developed using three main enactment models (i.e., *centralized*, *hierarchical*, and *distributed* models), but most of them focus on centralized execution control while the existing hierarchical and distributed models still have problems such as high overhead in communication amongst the distributed parties; and (iii) data movement mechanisms are divided into: *centralized*, *mediated*, and *peer-to-peer* models. A good combination of the three mentioned categories (partitioning, enactment, and data movement) with the purpose of reducing workflow *make-span* (MS) became a motivation to propose our system.

A new workflow management system called *parallel and distributed workflow management system* (PDWMS) is proposed. To achieve the main target of the system

(i.e., minimization of workflow MS), a new partitioning algorithm is also proposed. Using this partitioning algorithm, the original abstract workflow (input of the system) is partitioned into parallel partitions. The generated partitions are then assigned to a network of collaborative engines to be executed in parallel. Therefore, the PDWMS uses a distributed enactment and a peer-to-peer data movement model. The proposed workflow management system (i.e., PDWMS), the partitioning algorithm, and their evaluation are described in detail in this thesis.

Chapter 1

Introduction

1.1 Introduction

1.1.1 Grid Computing

The Grid [25] has emerged as a global cyber-infrastructure for the next-generation of e-Science applications, by integrating large-scale, distributed and heterogeneous resources. Scientific communities, such as high-energy physics, gravitational-wave physics, geophysics, astronomy, and bioinformatics, are utilizing the Grid to share, manage and process large data sets. In order to support complex scientific experiments, distributed resources such as computational devices, data, applications, and scientific instruments need to be orchestrated to support workflow operations within Grid environments [30].

1.1.2 Scientific Workflow

Scientific workflow is concerned with the automation of scientific processes in which tasks are structured based on their control and data dependencies. The workflow paradigm for scientific applications on Grid offers several advantages, such as: (i) increasing ability to build dynamic applications; (ii) utilizing resources that are located in a particular domain to increase throughput or reduce execution costs; (iii) enabling execution that span multiple administrative domains to obtain specific processing capabilities; and (iv) integrating of multiple teams involved in the management of different parts of the experiment workflow thus promoting inter-organizational collaborations.

With the advent of Grid and application technologies, scientists and engineers are building more and more complex applications to manage and process large data sets, and execute scientific experiments on distributed resources. Such application scenarios require means for composing and executing complex workflows.

In general, a workflow can be described in two formats: *abstract* and *concrete*. Here we explain what each exactly means.

Abstract Workflow

Abstract workflow describes the computation in terms of logical file names and transformations. It also indicates dependencies between different workflow components, but activities are independent of Grid resources. Abstracting away the resource descriptions allows the workflows to be portable. This level of abstrac-

tion also enables the workflow tasks to be efficiently assigned to the existing Grid resources at the time that the workflow tasks can be executed.

Concrete Workflow

We refer to the executable workflow as the concrete workflow, in which the workflow activities are bound to specific Grid resources. Concrete workflow also includes the necessary data movement to stage data in and out of the computing resources.

1.2 Objective and Scope

In this project, a workflow management system, i.e., PDWMS (Parallel and Distributed Workflow Management System) is proposed, which includes the following characteristics and components: (i) a partitioner which is able to separate parallel parts of a workflow represented by a *Directed Acyclic Graph* (DAG); (ii) a just in-time mapping mechanism that transforms the abstract to concrete workflow; (iii) a series of collaborative engines to execute different partitions of a whole workflow in parallel; and (iv) a data movement mechanism, which enables data transmission in peer-to-peer manner.

Since PDWMS has a decentralized architecture rather than centralized one and uses collaborative engines to execute workflow with corporation of each other, it would be different from other available workflow management systems.

The major objective of PDWMS is to reduce the make-span (MS) of workflow. This will be achieved by: (i) improving parallelism; (ii) reducing communication

overhead; and (iii) dynamic mapping.

By improving parallelism, we mean executing the workflow's independent tasks in parallel. This becomes our motivation to propose an algorithm which generates partitions of tasks which can fit the available resource pools while having the minimum dependencies. For the next step, one way is to use one engine to execute these partitions. In this way, we will have parallelism just at the task-level. Besides, the single engine could become the bottleneck of the system and its load would be high. But we could achieve even better performance by adding another level of parallelism which is the level of engines. By providing several engines to execute partitions, we can reduce the make-span even more and decrease the engines' load. So, in this way, parallelism can be achieved at both engine-level and task-level.

Other than the above-mentioned reason, to execute large workflows, we usually face with insufficient resources using only one cluster. More precisely, the number of computing resources may not be sufficient (i.e., maximum parallelism of workflow is greater than the number of resources), and thus the MS will be increased. This becomes another motivation to partition the workflow in order to share the power of several clusters (i.e., access to more resources at the same time) by running multiple workflow engines.

To sum up, in order to achieve the objective of this project (i.e., reducing workflow's overall make-span) it is needed to implement a partitioning algorithm to transform the original workflow to a number of partitions which can be run in parallel. To execute these partitions, a number of engines should work together. Issues regarding

coordinating the engines to communicate with each other in a distributed manner and transfer data among themselves by a peer-to-peer approach, will be addressed in this thesis.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows:

- Chapter 2 develops a taxonomy of workflow management systems. It identifies different models in each category. There is also a mapping of the existing systems to this taxonomy. It should be noticed that the presented taxonomy and related systems are all based on the focus of this project.
- Chapter 3 discusses preliminary system architecture and different parts of our proposed system. They include: (i) partitioning algorithm (i.e., how parallel partitions should be extracted out of the original workflow); (ii) enactment model (i.e., how engines should collaborate with each other); and (iii) data movement model.
- Chapter 4 talks about our approach toward partitioning the workflow. First it defines the partitioning problem. Then it develops a cost model on which our DAG (i.e., workflow) and Grid model are described. After that, it proposes a heuristic to partition the workflow into smaller groups of tasks, which are later called sub-DAG or partition.
- Chapter 5 shows the experiments we run to evaluate the system and justify our claims. It also reveals the yielded results.

- Chapter 6 summarizes all the steps done through building PDWMS and what we achieved in this project.

Chapter 2

Background and Literature Survey

2.1 Background

There is a common three-layer architecture found in almost all Grid workflow systems. In the first or above layer, workflow is composed with user involvement or automatically. Then it is transformed to concrete format and is passed on to the second layer which is in charge of the execution of workflow. Bottom layer provides the access to the computational resources which are normally distributed on the Grid. Table 2.1 presents this model. For each layer a real example is also provided. In the following subsections, we go through a brief description of each system mentioned in Table 2.1 and their relation to our work.

2.1.1 Globus Toolkit

We use some of the Globus Toolkit's services in order to provide the required middleware infrastructure for our project. These services include MDS [35] for information services, more precisely for resource discovery and management, GridFTP [20] for

Table 2.1: Common architecture of Grid workflow systems

Compositor (Pegasus [39])
Meta Scheduler-Executor (Condor, Condor-G, DAGMan [5, 4, 7])
Infrastructure Middleware (Globus Toolkit [16])

intermediate data movement, and GRAM [18] for execution management.

Information Services

MDS: The Monitoring and Discovery System (MDS) is a suite of web services to monitor and discover resources and services on the Grid [35]. This system allows users to discover what resources are considered part of a Virtual Organization (VO) and to monitor those resources. Presence of a service like MDS is necessary for us because available resources in the Grid should be determined and then monitored to always get the current state of the resources.

Intermediate Data Movement

GridFTP: GridFTP is a high-performance, secure, reliable data transfer protocol optimized for high-bandwidth wide-area networks [20]. It is actually based on FTP, the highly-popular internet file transfer protocol. In workflow execution, tasks are assigned to different computing resources and correspondingly, may also need to be staged-in and staged-out from the assigned resources. GridFTP is used as a means for all these data transmissions.

2.1 Background

9

Execution Management

GRAM: The Grid Resource Allocation and Management (GRAM) service provides a single interface for requesting and using remote system resources for the execution of “tasks” [18]. The most common use of GRAM is remote task submission and control. It is designed to provide a uniform, flexible interface to task scheduling systems. In this project, GRAM can be used for the same purpose (i.e., remote task submission and control).

2.1.2 Condor, Condor-G, DAGMan

Condor [5] is a specialized scheduling system for compute-intensive tasks. Like other full-featured batch systems, Condor provides a task queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management.

Moreover, Condor-G [4] is the Grid-enabled version of Condor. Similar to condor, it lets the user submit tasks into a queue, has a log detailing the life cycle of the tasks and manages input and output files. Instead of using the Condor-developed protocols to start running a task on a remote machine, it uses the Globus Toolkit to start the task.

DAGMan (Directed Acyclic Graph Manager) [7] is a task-sequencer for Condor. It manages dependencies between tasks at a higher level than the Condor. DAGMan submits tasks to Condor in an order represented by a DAG (Directed Acyclic Graph). In this project we can use DAGMan as our workflow execution engine.

In this project, Condor can be used as a local resource-task scheduler while DAG-Man takes care of tasks sequencing. Although Condor-G can be used as our system's meta-scheduler, we will develop a distributed approach for coordinating local task scheduler-engines (i.e., Condor-DAGMans). In Chapter 5 we use Condor-G as a centralized execution model and compare its performance against the execution model proposed in this thesis.

2.1.3 Pegasus

Pegasus (Planning for Execution in Grid) [39] is a workflow mapping engine developed and used as part of several projects (e.g., GriPhyN [21], NVO [37], and SCEC-CME [43]). It automatically maps the high-level workflow descriptions onto distributed infrastructures such as the TeraGrid, the Open Science Grid, and others. Using Pegasus, scientists construct workflows in abstract terms without worrying about the details of the underlying cyber infrastructure. It is currently being used in a variety of scientific applications ranging from astronomy, biology, earthquake science, gravitational-wave physics and others.

2.2 Grid Workflow System Taxonomy

Nowadays, Grid workflow management systems are widely used. They usually provide services for workflow design, mapping, enactment, data movement, monitoring, and fault tolerance. Thus, research in workflow management involves many research areas.

Workflow life cycle includes four main phases: *design*, *deployment*, *enactment* and

2.2 Grid Workflow System Taxonomy

11

monitoring [41]. This life cycle can be a good criterion for categorizing and studying workflow systems. On the other hand, there is a comprehensive survey done by the Melbourne university, which classifies the Grid workflow systems based on the following four aspects: *design*, *scheduling*, *fault tolerance* and *data movement* [55].

Taking both into consideration, we use the following taxonomy to study Grid workflow systems: *workflow design*, *mapping*, *enactment (i.e., execution coordination)*, and *data movement*. In this project, workflow design is not considered. Figure 2.1 shows our focused areas in Grid workflow taxonomy. In the following subsections, we explain our focused parts in the above-mentioned taxonomy in details.

2.2.1 Mapping

Workflow needs to be translated from its abstract form to the concrete one before it can be executed on distributed resources over the Grid.

Schemes for translating abstract models to concrete models can be categorized into either *static* or *dynamic* schemes [56]. In a static scheme, concrete models have to be generated before the execution, according to the current information about the execution environment. Therefore, the dynamically changing state of the resources is not taken into account. In contrast, a dynamic scheme uses both dynamic and static information about resources to make mapping decisions at run-time.

Static schemes can be classified as *user-directed* or *simulation-based* where dynamic schemes are divided into *prediction-based* or *just in-time*. In user-directed scheme,

Figure 2.1: A Grid workflow taxonomy for our focused area

all the mapping decisions are made based on users' knowledge and preference. But in simulation-based scheme, a simulation of the whole scenario before real execution is made. The simulation results can then be used in decision making. Prediction-based dynamic scheme uses dynamic information in conjunction with some results based on prediction. It is similar to simulation-based static scheme. However, it changes the initial mapping dynamically during the execution. In just in-time scheme, decisions are made at execution time. In this case, since the mapping decisions can be made according to the current state of Grid at run-time, it will be more accurate than a full-ahead static decision making.

2.2.2 Enactment

Workflow Enactment involves in execution of workflow (i.e., how engines are coordinated and collaborate with each other to execute a workflow). The architecture under which the workflow is executed, is very important for the scalability, autonomy, quality and performance of the system [56]. There are three major common approaches for workflow enactment: *centralized*, *hierarchical* and *distributed schemes*.

In the centralized approach, one central engine makes decisions for all tasks in the workflow. For hierarchical enactment, there is a central manager and multiple lower-level sub-workflow engines. This central manager is in charge of controlling the workflow execution and assigning sub-workflows to the lower-level engines. Against these two schemes, i.e., centralized and hierarchical models, there is a distributed approach, in which every engine can communicate with each other and the whole duty of enactment is divided up amongst them with no preference.

Efficiency is believed as the advantage of centralized approach while weak scalability should be considered as its flaw. Decentralized approach is more scalable but faces more challenges in communication and synchronization issues. Hierarchical approach is scalable to some extent. In addition, communication and synchronization can be done easier, because each upper level can be assumed as a controller for lower levels (i.e., it can take care of communication and synchronization issues of lower levels). Single point of failure may also be noticed as a problem. However, there are methods for resolving that (e.g., keeping a back-up coordinator in case of coordinator failure).

2.2.3 Data Movement

In Grid workflow systems, we need to stage-in and stage-out data to remote sites because tasks have some input and output data. Input data is needed to be staged in to the sites on which the task is going to be executed, while the output data is required to be staged out to the site where they are required. The transmission process can be done automatically or by user involvement. The automatic approach is divided into three categories: *centralized*, *mediated* and *peer-to-peer*.

A centralized approach transfers intermediate data between resources via a central point. In a mediated approach, there is a third party in charge of keeping logical and physical address of data, i.e., it acts like an index directory. Therefore, instead of transferring data between nodes and this third party, only logical/physical addresses of data are exchanged. A peer-to-peer approach transfers data between

resources directly.

In general, centralized approaches can be easily implemented but are not scalable. They are suitable for workflow applications in which large scale data-flow is not required. Mediated approaches are more scalable and suitable for applications which need to keep intermediate data for later use. Peer-to-peer approaches save transmission time significantly and reduce the bottleneck caused by additional communication in the centralized and mediated approaches. Thus, peer-to-peer approaches are more suitable for large-scale intermediate data transfer. However, they have more difficulties in deployment because they require a cluster to be capable of providing both data management and movement services [55].

2.3 Map of Taxonomy to the Existing Systems

2.3.1 Mapping

Table 2.2 shows different mapping schemes of the available systems. In the following, we explain two of the available systems' mapping mechanism.

Pegasus uses both user-directed static and just in-time dynamic schemes for mapping. In order to provide just in-time mapping, a partitioner is added. This partitioner divides the abstract workflow into smaller workflows of partitions. The dependencies between the partitions reflect the original dependencies between the tasks of the abstract workflow. Pegasus then maps the partial workflows following these dependencies. So after each partition's execution, next partition will be mapped and executed. In this way, mapping of each partition is delayed until its

Table 2.2: Mapping schemes used by the available Grid workflow systems

Mapping Scheme	System
Static (User-directed)	Pegasus [39], UNICORE [52], Gridbus [19]
Static (Simulation-based)	GridFlow [3]
Dynamic (Prediction-based)	ICENI [23], GrADS [17], Askalon [2]
Dynamic (Just in-time)	DAGMan [7], Pegasus [39], Triana [51], Taverna [48], Gridbus [19]

execution time. This makes the workflow execution to be more adjustable to the changes in the Grid environment.

Askalon has a prediction-based scheme toward mapping. As already explained in Section 2.2.1, prediction-based dynamic mapping uses dynamic information in conjunction with some results based on prediction. Similarly, Askalon provides a service for predicting the time required by an activity. It employs active history-based predictions as follows: initially, all activity types are selected. Then for each activity type, different parameter combinations on different Grid computers are tested and the execution times are stored in a performance database [13]. Askalon uses these information in the database for prediction and thus improves mapping decisions.

2.3.2 Enactment

In case of enactment, different models are adopted by different systems. Table 2.3 presents a map of three main models of scheduling and execution architecture

2.3 Map of Taxonomy to the Existing Systems

17

used by the available systems. In the following, we explain some of the systems mentioned in Table 2.3 regarding their enactment mechanism.

Table 2.3: Enactment architecture adopted by the available Grid workflow systems

Enactment Architecture	System
Centralized	Pegasus [8], DAGMan [47], ICENI [23], Taverna [48], GrADS [17], UNICORE [52], Karajan [28], Kepler [1]
Hierarchical	Askalon [10], GridFlow [3]
Decentralized	Gridbus Workflow [19], Triana [49]

Pegasus [44] uses Condor as the enactment engine. It schedules tasks on a set of distributively owned machines known as the condor pool (we use the term resource pool in PDWMS). It uses two major components: a task sequencing engine which is DAGMan and a task scheduler of Condor (i.e., *schedd*). The whole scheduling architecture of Pegasus is centralized.

The enactment engine adopted in Askalon [13] uses a hierarchical architecture organized in a master-slave communication model. A partitioner is provided, which resides in master engine and divides the workflow into smaller partitions that can be executed by slave engines. The master engine monitors the execution of the entire workflow and the state of the slave engines. Slave engines (usually one for each cluster) monitor the execution of individual workflow partitions and report to the master whenever individual activities change their state or when the partitions produce some intermediate data. Askalon's partitioning algorithm is based on graph transformation which transforms the original DAG to a number of partitions

according to some predefined rules applied in several steps (to be discussed later in Section 2.4).

Gridbus [19] has a decentralized architecture which is implemented in event-based approach. Every task has a *Task Manager* (TM) which is in charge of task monitoring and execution, and there is a *workflow coordinator* (WCO) which controls the lifetime of all TMs as well as the whole process of workflow execution. Gridbus makes use of *Tuple Space*¹ as a means for event communication between TMs. Task managers can subscribe to the events of their interest and will be notified about these events whenever they happen. For example, in a DAG with a child task (i.e., child node) and parent task, first parent task defines the “output-ready” event in the tuple space. Since child task is dependent on the parent task’s output, it subscribes for that event. Whenever the execution of parent task finishes, the child task will be notified.

The advantage of this method is to provide an immediate task execution. Once the required input data becomes available, the dependent nodes are informed, and they can start executing immediately. The weak point of Gridbus architecture is that, although Gridbus claims to have a decentralized architecture, WCO plays as a centralized controller to monitor TMs’ lifetime as well as the whole workflow execution process. Besides, having a TM per task (i.e., having a workflow executioner per task) makes some extra overhead for the system. The number of TMs can be reduced by clustering a group of tasks and specifying only one manager to all of them.

¹A tuple is simply a vector of typed values (fields). A tuple space is a collection of tuples which can be shared by multi-parties using operations such as read, write and delete.

GridFlow [3] is another workflow management system that we take into consideration because of its hierarchical architecture. The workflow management is conducted by a hierarchical approach including two levels: *global workflow management* and *local sub-workflow schedulers*. An XML-form workflow is composed through a portal and sent to the global manager which makes a simulation accordingly to get an optimal schedule. Then local schedulers assign sub-workflows to local sites through an agent-based resource management system (ARMS). Each agent in ARMS is in charge of the resources in a single administrative domain.

2.3.3 Data Movement

Table 2.4 shows the data movement mechanisms used in the available systems. Note that since some of the systems may support more than one data movement mechanism, they are categorized in more than one model as well. In the following, a more detailed explanation on some of the most important systems' data movement mechanism is given.

DAGMan does not provide automatic intermediate data movement, so all stage-in/out data is user's responsibility. For example, user may need to use pre/post-scripts of condor submission file to stage the data in/out respectively.

In Askalon, its Enactment Engine provides flexible management of large collections of intermediate data generated by hundreds of parallel activities that are typical to scientific workflows. Additionally, it provides a mechanism to automatically track data dependencies between activities and performs static and run-time workflow

Table 2.4: Intermediate data movement adopted by the available Grid workflow systems

Intermediate data movement	System
User-directed	DAGMan [47], Karajan [28], Askalon [10]
Automated (Centralized)	Askalon [10], Taverna [38], Gridbus [19], Kepler [1]
Automated (Mediated)	Pegasus [8], ICENI [34], UNICORE [52]
Automted (Peer-to-Peer)	Triana [49], GrADS [17], GridFlow [3], Gridbus [19]

optimizations, including archiving and compressing of multiple files to be transferred between two clusters. But since it uses centralized model where slaves carry out the stage-in/out data from/to a common data repository, this could result in extra communication overhead.

Pegasus uses a mediated approach for data movement. The Replica Location Service (RLS) [42] of Globus Toolkit is used for this purpose. RLS allows the registration and discovery of replicas. It maintains and provides access to mapping information from logical names of data items to target names (that is, their physical locations). In Pegasus, in case of data stage-in, a request is made to RLS containing the logical name of the input data. If the data is available, its physical address is sent back by RLS to be later used by GridFTP to transfer the data. Moreover, in case of data stage-out, the logical name of data along with its physical address are registered in RLS for later usage.

Gridbus Workflow provides both peer-to-peer and centralized approach for intermediate data movement. As already mentioned in Section 2.3.2, Gridbus has a decentralized architecture in which a TM is in charge of a task. So, TMs need to exchange data (i.e., input and output of tasks). Therefore a peer-to-peer approach would be more appropriate. On the other hand, there is also an option to use a centralized approach for data transmission.

2.4 Partitioning

There are many works on assigning graphs and workflows tasks on computing resources (i.e., different algorithms and methods are available on how and with which order computing resources should be used by the tasks in a workflow) [24,33,59,60], But limited research has been done on partitioning workflows (specially DAG-based ones) to make the mentioned assignment easier. Table 2.5 shows a few existing work.

Table 2.5: Systems and their partitioning algorithm

System	Algorithm
MCCF [57]	Sub-task Grouping
Pegasus [8]	Level-Wise
Askalon [10]	Graph Transformation
[31]	Temporal Decomposition

There is a sub-task grouping algorithm proposed by [58] to preprocess the abstract workflow to be executed by agents in a mobile code collaboration framework (MCCF). It uses *minimum spanning tree* and *depth first search* traversal algorithm

to generate the partitions. The objective is to determine the group of tasks to be executed by a mobile agent.

Pegasus [8] uses a simple method for partitioning the workflow in its just in-time mapping scheme, which is called *level-wise* model. In this model, each level of DAG is selected as a separate partition. According to [53], this workflow partitioning strategy appears to have no advantage over the full-graph scheduling and may produce even less effective results for unbalanced workflows. In fact performance of the level-wise method is dependent on the workflow structure. For example it can get an acceptable result in case of a workflow with large proportion of parallel tasks rather than sequential parts [31]. Moreover, this kind of level-based partitioning cannot be used for decentralized workflow enactment. Because although the workflow is divided into partitions, the partitions are sequentially dependent on each other.

Askalon [53] provides a partitioning algorithm based on graph transformation. It transforms the original DAG into a number of partitions according to some predefined rules applied in several steps. In the first step of their method, workflow is mapped to resources by using three well-known algorithms: namely genetic, HEFT, and simple myopic. Therefore, the abstract workflow is transformed to a concrete one. Then a four-phase partitioning method is applied to the resulted concrete workflow to group tasks into partitions. Both control-flow and data-flow are considered in the partitioning phases. For example, activities which are part of a “while loop” (i.e., a control-flow constraint) could be assumed in the same partition. As a data-flow constraint example, the activities which are in different partitions should have minimum data dependency to each other. Askalon’s partitioning method is a

novel one, but since the control/data-flow information used in the partitioning is provided statically, resulted partitions are not optimal. Therefor, several optimizers are developed to refine the partitions.

[31] suggests a temporal decomposition approach which presents a heuristic for workflow partitioning by reducing the problem size. So instead of spacial decomposition, the workflow is divided according to the estimated start/end time of tasks. The advantage of this work is the adaptability of the scheme to dynamic resources. In other words, the sub-workflow's task assignment is chosen according to the latest states of Grid resources. This work also supports complex workflow components like loops or conditional structures. However, it has several flaws. For instance, estimated start/end time of tasks may go wrong. Also all the results are based on simulations done on SimGrid, not a real existing system.

2.5 Summery

Most of the known workflow management systems use partitioning as part of the assignment (i.e., task to computing resource assignment) mechanism. However, it could be considered separately. Among the studied systems, we found two systems which are similar to ours with regard to partitioning: Pegasus [39] and Askalon [12].

Pegasus uses partitioning before the assignment step (i.e., on the abstract workflow) and as a means of deferred assignment [8] (which is an approach to delay the mapping of partitions from abstract to concrete form until their execution times). Its partitioning method (i.e., *level-wise*) is only useful for a centralized workflow en-

actment not a distributed one. The resulting partitions are sequentially dependent on each other and cannot be executed in parallel by collaborative engines.

Askalon uses partitioning after the mapping step (i.e., on concrete workflow) and makes use of the resulting partitions in the execution step [10]. This means static mapping information is used for partitioning. Given the dynamic nature of Grid environments, this approach may not be suitable. That is the main reason why Askalon uses optimizers to refine the partitions. However, the use of optimization may introduce additional overhead to the whole system.

Among different studied systems regarding their enactment architecture, Askalon and Gridbus are good examples of hierarchical and distributed approaches. Askalon uses a master-slave architecture model, in which slave engines are responsible for executing workflow partitions, while the master as a higher-level, centralized engine monitors their work. Gridbus executes the workflow in a distributed manner, using agent-based task managers (TMs) to control execution per task. Although Gridbus has decentralized workflow execution, the use of a TM per task creates a lot of overhead.

To resolve the shortcomings of the above-mentioned systems regarding partitioning and enactment, we propose a workflow management system which is able to partition the abstract workflow into groups of tasks that can be executed in a parallel manner. These partitions are distributed among collaborative engines which can communicate with each other, using GridFTP to transfer data as well as control messages.

Our proposed system has a decentralized architecture approach to coordinate engines (to be discussed later in Section 3). Therefore the problem of poor scalability as well as a single point of failure, which is usually seen in centralized systems, may not appear. Moreover, our system makes use of peer-to-peer data movement for the purpose of transferring intermediate data amongst computing resources.

It should also be noted that the mapping scheme of our proposed system is just in-time, i.e., decision making needed for mapping of tasks in each partition is delayed until their execution time. In this way the workflow execution is more adjustable to the changes of the Grid environment. Table 2.6 presents a summery of our proposed system's features.

Table 2.6: PDWMS summery

System Features	PDWMS
Partitioning	Binary partitioning algorithm
Mapping Scheme	Just in-time
Enactment (Implementation Model)	Distributed
Data Movement	Peer-to-peer

Chapter 3

Distributed Execution of Workflow

3.1 Overview of the System

The main components of the proposed system (i.e., PDWMS) are shown in Figure 3.1. They are mapped onto the common three-layer architecture of workflow management systems. The design layer at the top has three components, called: (i) *Task Graph for Free* (TGFF) [50]; (ii) *partitioner*; and (iii) *modifier*. Layer two, which is the enactment layer, has: (i) a *dispatcher*; and (ii) an *engine* whose internal component is a Condor-DAGMan. In layer three, which is the middleware layer, there are Globus Toolkit services (e.g., GridFTP) and Hawkeye [22] (a tool for monitoring the Condor resource pool).

The components that do not previously exist and are implemented as part of this project are distinguished by a circle in their corner. They are: (i) *partitioner*; (ii)

3.1 Overview of the System

27

Figure 3.1: System layer components

modifier; and (iii) dispatcher. Following is a brief description of each component. More elaboration on the system's components are given in the following sections.

- **TGFF:** TGFF is an open-source software used in PDWMS to generate abstract workflow. The user can adjust the input parameter to shape the DAG as she/he prefers (e.g., in PDWMS, we adjust TGFF's parameters to have binary-DAGs as output).
- **Partitioner:** This takes the binary-DAG (output of TGFF) as its input, and transforms it to several partitions (called *sub-DAGs*) as its output. It is also in charge of deciding the resource pool onto which each partition should be mapped.
- **Modifier:** This takes the partitions produced by the partitioner and modifies their internal tasks by adding pre/post-scripts. These pre/post-scripts will enable engines to collaborate with each other later in the execution step.
- **Dispatcher:** This takes the modified sub-DAG partitions from the modifier as its input and submits them to the engines. In other words, the dispatcher's role is to stage the partition to the proper engine that is in charge of the resource pool.
- **Engine:** The main role of an engine is to execute a workflow partition. This is done by using a Condor-DAGMan as its main component.
- **Globus Toolkit services:** They have been discussed in Section 2.1.1. Here, we use GridFTP as a means of data movement among resources of different clusters.

- Hawkeye: Hawkeye is a monitoring and management tool for distributed systems. It utilizes the technologies already present in Condor to provide rich mechanisms for collecting, storing, and using information about computers. A Hawkeye system can be used to monitor various attributes within a resource pool such as disk space, memory used, system load, users and so on [22].

In this project, a cluster of resources refers to a hardware and software infrastructure of heterogeneous resources under the same administrative control. It is assumed that each cluster has Globus Toolkit installed. One or more resource pools are available in each cluster. A resource pool is actually a pool of resources (from the same cluster), where the tasks are executed. In PDWMS, each resource pool is administrated by a Condor in terms of assigning tasks to computing resources of the resource pool.

As depicted in Figure 3.1, other than system components, there are some black arrows which show the existing inter/intra-layer relations among them. For example, for inter-layer relations, the arrow between TGFF and partitioner represents the fact that output of TGFF is used as the input of the partitioner. Similarly, the arrow between partitioner and modifier, and the one between dispatcher and engine represent the input-output relation existing between them. On the other hand, as an example for intra-layer relations, we have one between the modifier (design layer) and the dispatcher (enactment layer) (note that all the intra-layer relations are shown by a double-sided arrow between their layers).

To understand the whole sequence of workflow execution process as well as the

mentioned relations, Figure 3.2, shows the sequence diagram of PDWMS, including the components in all the layers. For more elaboration on the system components, we discuss them in the following sections under their own layers.

Figure 3.2: PDWMS' sequence diagram

3.2 Design Layer

As mentioned, components of this layer includes: TGFF, partitioner and modifier. Further description on TGFF, and the graphs we generate as input of PDWMS, are given in Section 5. Since partitioning is one of the major focuses of this thesis, it is described in detail in Chapter 4.

3.2.1 Modifier

The duty of the modifier is to add pre/post-scripts to the partition's description file to support the extra actions required for collaborative workflow execution.

If both parent and child tasks are in the same partition, they will be assigned to the resources of the same resource pool. In this case, a *local data movement* (to be explained in Section 3.4.1) is needed. On the other hand, when parent and child tasks are in different partitions, if they are assigned to the different clusters, a *global data movement* (to be discussed in Section 3.4.1) mechanism is required. However, if they are assigned to different resource pools in the same cluster, a local rather than global data movement is assumed. To enable data transmission through global data movement among the dependent tasks, we add pre/post-scripts to their partition description file. To see a sample of pre/post-scripts added to a partition description file, refer to Appendix B.

Usually, the procedure of a pre-script is to check the availability of an input. If the input is ready, the task starts its execution immediately. Otherwise, it should wait until the specified input is generated. On the other hand, a post-script is in

charge of output data transfer. For example, assume task A (from partition $part_i$ which is already assigned to a resource pool of cluster C_i) has finished its execution and A.out (output of A) is generated. If A.out is required by task B (from partition $part_j$ which is also already assigned to a resource pool of cluster C_j), as soon as A.out is released, it should be transferred to cluster C_j . Further details on how the pre/post-scripts are used will be given in Section 3.3.2.

3.2.2 Partition Assignment

Making a decision on the best resource pool for executing a partition could be a challenging problem. For example, one problem scenario could be as follows: one of the generated partitions might have a *Maximum Parallelism* (MP) (to be defined in Section 4.2.1) which is greater than the number of computing resources in any of the resource pools. Thus, some of its parallel tasks have to be mapped to the same resources.

To solve the partition assignment problem, the workflow tasks are grouped into partitions based on the resource pools and number of their computing resources. Using monitoring tools such as Hawkeye, we are provided with a full environment report, including the information such as total number of clusters, the number of resource pools which each cluster has, as well as the number of resources that any of the resource pools encompasses.

In other words, to get a better overall result, partitioning is done based on the monitoring information obtained. And as a consequence of having both partition-

ing and partition assignment at the same time, output of partitioner module consists of both the groups of tasks (i.e., partitions) and the resource pool to which they should be mapped.

3.3 Enactment Layer

In the proposed system, workflow execution is performed through collaboration of a network of engines. Figure 3.3 shows a general overview of workflow enactment through engines' collaboration in the system. In a very general view, there are several workflow engines which are connected to each other to support parallel execution of workflow partitions. These engines might be in the same or different clusters. Every engine is attached to one resource pool to which the partition's tasks can be mapped. Hence, whenever the engine needs to map its partition's tasks to computing resources, it can only use resources of its attached resource pool.

3.3.1 Dispatcher

As mentioned in the partition assignment section 3.2.2, the partitioner module in the design layer is not only in charge of grouping tasks to partitions, but also deciding the resource pool to which a partition should be mapped. Having these two pieces of information, the dispatcher's duty is to transfer the partition's tasks code from the code repository and staging them in the specified cluster. The dispatcher gets help of GridFTP on this matter (this transfer is considered as a global data movement, which will be discussed in Section 3.4.1).

Figure 3.3: An overview of workflow enactment through engines' collaboration in PDWMS

3.3.2 Engine

Each resource pool has a Condor-DAGMan that executes the group of tasks assigned to the resource pool. The scheduling and assigning of individual tasks to resources as well as keeping track of their dependencies are carried out entirely by the Condor-DAGMan. The engines' collaboration is then possible via a static configuration setup which is provided for each engine while initializing them using an XML file. Therefore, the only part to be implemented is how several engines should collaborate with each other. The coordination between engines is implemented, using the pre/post-scripts, as explained in Section 3.2.1.

Note that, these pre/post-scripts only need to be added to those dependent tasks which exist in different partitions. Another point which should be remembered here is that the component which is in charge of adding the pre/post-scripts to tasks' description file is the modifier that has already been introduced in Section 3.2.1.

3.4 Middleware Layer

The middleware layer consists of two modules: (i) Hawkeye; and (ii) GridFTP. The first one is in charge of monitoring the health of resource pools; whereas the second one is a means of data transfer.

The information provided by Hawkeye is used by the partitioner as discussed in Section 3.2.2. GridFTP as a means of data movement is required in both engine collaboration (Section 3.3.2) and code staging (Section 3.3.1).

3.4.1 Local and Global Data Movement

Data movement can be either local or global. In local data movement, data is transmitted between computing resources of the same cluster; whereas global data movement transfers data between computing resources of different clusters. More precisely, in local data movement, dependent tasks can be from the same as well as different resource pools as long as both resource pools belong to the same cluster. In contrast, global data movement occurs only when intermediate data of dependent tasks need to be transmitted between the resource pools of two different clusters.

In case of global data movement, peer-to-peer approach is used. The parent tasks involving in the global data movement are individually in charge of data transfer, using the post-scripts of their submit files, which have already been modified by modifier. The source, the destination, and the output data to transfer are specified in the post-scripts. Therefore, output data can be transmitted among peers without any exterior control.

For the case of local data movement, though, data transfer is handled using a centralized approach. In fact, it is not recommended to have the same peer-to-peer method as in case of global data movement, although it is possible. The reason is that the transfer cost of local data movement is negligible, compared to global communication cost, due to the higher communication speed inside a cluster.

The communication cost of global data movement can be calculated as follows:

$$\text{Comm-cost}(c_i, v_j, c_k, v_z) = \text{Trans-cost}(c_i, c_k) \times O(v_j, v_z)$$

where $\text{Comm-cost}(c_i, v_j, c_k, v_z)$ denotes the communication cost of transferring output of task v_j from cluster c_i to task v_z in cluster c_k . $\text{Trans-cost}(c_i, c_k)$ is the cost of transferring a unit of data (1MB) from cluster c_i to c_k and $O(v_j, v_z)$ denotes the weight of the edge connecting task v_j to task v_z .

3.5 Summery

To sum up, there are some points about PDWMS' architecture which makes it different from other systems. First, the architecture is distributed, which makes the system very well scalable and prevents single point of failure from occurring. Second point is the use of pre/post-scripts for engine collaboration in order to reduce the communication overhead amongst engines. Data movement which is a key part of PDWMS is done in a peer-to-peer manner using GridFTP as data transmission protocol provided by Globus Toolkit.

The internal components of PDWMS can be placed in three layers designated as: (i) design; (ii) enactment; and (iii) middleware. These three layers work in relation with each other and the information provided by components of each can be used by others.

Chapter 4

Partitioning

The act of partitioning is to map groups of tasks (partitions) onto clusters (more precisely onto the resource pools of the clusters). A partition consists of a set of tasks; whereas a task is an indivisible unit of computation. We aim to generate partitions of tasks which can fit the available resource pools while having the minimum dependencies. In this way, we can achieve parallelism at both engine-level (by using collaborative engines to execute partitions in parallel as explained in Section 3.3) and task-level. This leads to the end objective of partitioning, which is to minimize the make-span of workflow execution.

The problem of partitioning tasks and mapping them to clusters is an *NP-hard* problem in its general form [11]. The time complexity of an algorithm refers to its execution time as a function of its input. Usually an algorithm whose time complexity is bounded by a polynomial is called a polynomial-time algorithm. Besides, an optimal algorithm is considered to be efficient if it runs in polynomial time. On the other hand, inefficient optimal algorithms are those which require a search of

the whole enumerated space and have an exponential time complexity. A heuristic produces an answer in less than exponential time, but does not guarantee an optimal solution. Intuition is most often used to come up with heuristics that make use of special parameters that affect the system in an indirect way.

An effective optimal algorithm may be found if some restrictions are imposed on the problem. There are two main cases in the literature where optimal solutions can be found efficiently: (i) the case where the distributed system is composed of only two processors [45]; and (ii) the case where the distributed system is composed of a linear array of any number of processors [29]. Since these are very restrictive situations, we prefer to use a sub-optimal heuristic for general cases.

4.1 Problem Statement

First, we explain the partitioning problem by determining its input and its output. Then, we describe a cost model, followed by the methods for DAG completion time calculation which supports our proposed partitioning algorithm given in Section 4.4.

4.1.1 Input

The input of partitioning problem consists of two parts: (i) DAG model; and (ii) Grid model.

- **DAG Model:** Given a DAG $D = (V, E)$ of a workflow representation of an application, let the set of workflow tasks be $V = \{v_i, 1 \leq i \leq |V|\}$, where $|V|$

denotes the number of tasks. These tasks are connected to each other through some edges represented by $E = \{e_{i,j} = \langle v_i, v_j \rangle, 1 \leq i, j \leq |V|\}$. Further, there are parameters which affect the shape of the DAG, such as R/C (the average ratio of task weights over edge weights), and fan-in and fan-out of each task (i.e., number of its direct parents and children). In case of binary-DAG, fan-in and fan-out of each task can be 0, 1, or 2. Based on these parameters, the shape of DAG can be different such as: tall-thin, short-wide, balanced, dense, sparse and so on.

- **Grid Model:** In this project, we have a multi-cluster environment. These clusters can reside in either of the same or different domains. Set of clusters is denoted by $C = \{c_i, 1 \leq i \leq |C|\}$. Each cluster may include several resource pools. $P(c) = \{p_i, 1 \leq i \leq |P(c)|\}$ denotes a set of resource pools in cluster c .

4.1.2 Output

The partitioning problem is to obtain an output in the form of *Directed Graph (DG)* whose nodes are sub-DAGs of the original DAG, while having the least dependencies in between (it could be in terms of fewer number of links between nodes or lighter weights on the links). The mentioned directed graph can be denoted by $D' = (V', E')$. V' is a set of partitions or sub-DAGs¹ which is denoted by $V' = \{v'_i, 1 \leq i \leq |V'|\}$. $E' = \{e'_{i,j}, 1 \leq i, j \leq |V'|\}$ is a set of dependencies between the sub-DAGs. Notice that, the result of the partitioning algorithm would not necessarily be a DAG, i.e., D' can be a cyclic graph.

¹these two terms are interchangeably used in the text

4.2 Cost Model

To make a cost model, there are some parameters which should be declared. These parameters include input, output, and the criteria that should be met to achieve the output. Input parameters are categorized into DAG model and Grid model parameters. DAG model parameters are determined by users and they will not be changed during the execution. The Grid model parameters are based on the Grid environment and its resources. Since the Grid is not a stable environment, some of its parameters should be updated according to the reports received from Grid monitors (we are using Hawkeye for this purpose).

The purpose of this cost model is to help make decision on when the partitioning in the division phase of the partitioning algorithm should be stopped. It also helps to select a resource pool to which a specific partition should be submitted. It could be noteworthy to mention that these two decisions are made at the same time. In fact, we stop the partitioning process based on the number of computing resources which are available in different resource pools. Therefore, the results of partitioning are the partitions as well as the resource pools to which they should be submitted.

4.2.1 DAG Model

1. DAG $D = (V, E)$, where $V = \{v_i, 1 \leq i \leq |V|\}$ represents tasks and $E = \{e_{i,j} = \langle v_i, v_j \rangle, 1 \leq i, j \leq |V|\}$ represents the edges which connect tasks together. $|V|$ and $|E|$ denote the number of tasks and dependencies in the DAG respectively.

- *Entry Task*: A task with no input edges
- *Exit Task*: A task with no output edges
- $N_l(d)$: Number of levels in DAG d . In a DAG, the tasks at the same distance from the entry task are said to be at the same level. The entry task is considered to be at level 0 and the exit task is at the maximum level.
- $N_t(i, d)$: Number of tasks in level i of DAG d
- $MP(d)$: Maximum parallelism of DAG d is defined as the maximum number of tasks that a level of DAG d can have:

$$MP(d) = \max_{1 \leq i \leq N_l(d)} N_t(i, d)$$
- $CP(d)$: A path between entry and exit tasks of DAG d is called critical path if the sum of the weights associated with the tasks and edges in this path is the maximum one among all available paths.
- $CT(d)$: Completion time of DAG d is the total time required for its tasks to finish their execution.
- $R/C(d)$: Average ratio of task weights over edge weights. A graph is called *fine-grain* (i.e., communication-intensive) if R/C is below 1; otherwise it is called *coarse-grain* (i.e., computation-intensive). (In Chapter 5, we will use this parameter to categorize the generated random DAGs.)

2. Task:

- $W(v_i)$: Weight of task v_i
- $O(v_i, v_j)$: Output size of task v_i to task v_j . This is the weight of the edge connecting tasks v_i and v_j

- $Par(v_i)$: Parents of task v_i , denoted by set the set $Par(v_i) = \{v_j \mid \exists e_{v_j, v_i} \in E\}$
- $Chi(v_i)$: Children of task v_i , denoted by the set $Chi(v_i) = \{v_j \mid \exists e_{v_i, v_j} \in E\}$
- $FI(v_i)$: Fan-in of task v_i , denoted by $FI(v_i) = |Par(v_i)|$
- $FO(v_i)$: Fan-out of task v_i , denoted by $FO(v_i) = |Chi(v_i)|$
- $L(v_i)$: Level of task v_i in a DAG. This is defined as the length of the longest path from the entry task (root of the DAG) to the task.

4.2.2 Grid Model

1. Cluster: Clusters are denoted by set $C = \{c_i, 1 \leq i \leq |C|\}$, where they can reside in either the same or different domain.
 - $P(c)$: Resource pools in cluster c , which can be represented by set $P(c) = \{p_i, 1 \leq i \leq |P(c)|\}$. $|P(c)|$ is the number of resource pools in cluster c (note that a cluster may include multiple resource pools).
 - $Trans-cost(c_i, c_j)$: This is the cost in terms of time required to transfer unit of data (1MB) from cluster c_i to cluster c_j .
2. Resource pool: A resource pool consists of a set of resources:
 - $R(p_c)$: Resources of resource pool p in cluster c , represented by set $R(p_c) = \{r_i, 1 \leq i \leq |R(p_c)|\}$, where $|R(p_c)|$ is the number of available computing resources in resource pool p .
3. Computing resource: These parameters which characterize the computing resources are used at the task definition level to specify tasks requirements:
 - $OS(r)$: Operating system of resource r

- $Arch(r)$: Architecture of resource r
- $Mem(r)$: Memory of resource r
- $Sp(r)$: Speed of resource r

We can obtain the computation cost and communication cost using the given parameters of DAG and Grid model as follows:

- $Comp-cost(v_i)$: Computation cost of task v_i . Since computing resources are different in terms of speed and memory, the computation cost (i.e. execution time) of two tasks with the same weight executing on different resources may not be equal. Computation cost of task v_i on resource r_j can be determined by:

$$Comp-cost(v_i) = W(v_i)/Sp(r_j)$$

- $Comm-cost(c_i, v_j, c_k, v_z)$: Communication cost is produced when the output of the task v_j on cluster c_i is required by the task v_z on cluster c_k . So, we have the following equation:

$$Comm-cost(c_i, v_j, c_k, v_z) = O(v_j, v_z) \times Trans-cost(c_i, c_k)$$

4.3 Completion Time

It has been known that critical path could be a good measure to calculate the completion time (CT) (i.e., the overall execution time) of a partition (i.e., a group of tasks) on a cluster of computing resources. But the critical path only reflects the

actual completion time if parallelism of the partition is less than the number of computing resources in the specified cluster. This means:

$$\text{If } (MP(d) < |R(P_c)|) \text{ then } CT(d) = CP(d)$$

However, this is not always the case in reality, because it is very common to have parallelism exceeds the number of resources (e.g., in case of having very large DAGs with a high MP to execute). Solving this problem, resources should be shared among independent tasks which may end up with an increase in completion time.

To calculate the completion time, we face with scheduling problem and its NP-hard nature [11] imposes us to give a near optimal solution using a good heuristic.

In the meantime, it should be remarked that partitioning is a preprocessing step of the whole workflow execution procedure whereas scheduling comes as the next step. It means at the partitioning step we can only come up with an approximation of the completion time of the partition before its real scheduling time arrives.

To obtain an approximation on completion time of a partition, we need to know some implementation issues of the system such as the scheduler and scheduling method which will be used in later phases. In case of PDWMS, Condor is used as our scheduler. According to [46], condor uses FIFO method for MPI jobs whereas it uses priority queue for the rest of jobs. PDWMS is not specifically targeted at MPI jobs, therefore the priority queue is assumed. Priority queue or list scheduling is a class of scheduling heuristics, in which tasks are assigned priorities and placed

in a list ordered in decreasing magnitude of priority. Whenever tasks contend for processors, the selection of tasks to be immediately processed is done on the basis of priority with the higher-priority tasks being assigned processors first. If there is more than one tasks of a given priority, ties are broken randomly [11].

In condor, job prioritization is based on user, task, and submission time respectively. However, for our purpose (i.e., calculating the completion time of a partition), we are dealing with submission of a group of tasks (i.e., a partition) at the same time and by only one user. Hence, the tasks' priorities could not be differentiated by the above-mentioned characteristics. The partition's tasks are placed in the ready queue based on their position in the DAG, i.e., their levels. In other words, condor puts a task in the ready queue only when all its immediate predecessors finished their execution. For the cases where there exists more than one task to be positioned in the queue (e.g., when the predecessors are the same), topological order of tasks in the graph would break the tie.

We use the priority queue heuristic as explained in [11] to calculate the DAG completion time. The original algorithm considers communication delay among the computing resources within a resource pool. However, in our system, we neglect the communication delay among the computing resources of the same cluster and, instead, consider the data transmission time between resource pools of different clusters. Therefore, the original algorithm needs minor modification to be used for our purpose.

Following the above-mentioned priority queue list scheduling algorithm to schedule

a DAG, we can come up with a Gantt chart which includes the start/finish and execution time of all tasks plus the overall completion time required for the execution of a specified DAG on multiple resource pools each with certain number of resources (an example of completion time calculation will be presented later in Section 4.4.3).

4.4 Binary-Division Partitioning Approach

Here we introduce our partitioning algorithm. It is called *binary-division* partitioning, which is a heuristic specially designed for binary-DAGs. However, the criterion of having binary-DAG as the input will not cause loss of generality as we propose a *binary transformer* which can transform a DAG to its binary version beforehand.

4.4.1 Motivation

To apply our proposed algorithm, we need to have the binary version of the input DAG. The motivation for using the binary version of the DAG is to simplify the process of the division phase in partitioning. In fact, binary-DAGs could be handled more simply because their dependencies between tasks have only limited types such as *branch*, *join*, *one-to-one*, *entry/exit*, and *join-branch*. These are depicted in Figure 4.1. To have the binary version of the original DAG, we can use a transformation algorithm which is explained in the following section.

4.4.2 Binary Transformer

The general idea behind DAG transformation is to substitute *complex dependency* patterns of a DAG by simple ones. This can be done by different ways of adding

Figure 4.1: Different relation types in a binary-DAG. a, b, c, d, and e represent a branch, join, one-to-one, entry/exit and join-branch relation respectively.

4.4 Binary-Division Partitioning Approach

49

virtual tasks to the original DAG. To understand the above idea, the following terms should be defined:

- Complex and simple dependency: Any combination of dependencies between a task and its parents or a task and its children which satisfies the following:

$$\forall v_i \in V, FI(v_i) \geq 3 \vee FO(v_i) \geq 3$$

is called a complex dependency between v_i and its parents or v_i and its children respectively. The other combination of dependencies between a task and its parents or children is distinguished as simple dependencies.

- Virtual task: It is a task with zero computation. There are some rules to use it:
 - An original task cannot be substituted with a virtual task and vice versa
 - Adding/deleting any number of virtual tasks to/from a DAG will not change the final output of the DAG

Algorithm 1 shows the pseudo-code of the binary-transform algorithm. As can be seen, tasks are examined in a *breadth first traversal (BFT)* order to see whether they have complex dependency with their parents/children. In case of complex dependency, i.e., if they have more than two parents/children, a *Join-transform/Branch-transform* routine is called respectively; otherwise, i.e., if the dependencies between the task and its parents/children are not complex, transformation is not required.

```

Input: A DAG with root  $r$ 
Output: A binary-DAG with root  $r$ 
begin
   $v \leftarrow r$ ;
  start BFT;
  while all tasks are not visited do
    if  $FI(v) \leq 2$  and  $FO(v) \leq 2$  then
      | the number of task's parents and children are less than 2;
      | transformation is not required;
    else
      if  $FI(v) > 2$  then
        | Join-transform( $v, \text{Par}(v)$ );
      end
      if  $FO(v) > 2$  then
        | Branch-transform ( $v, \text{Chi}(v)$ );
      end
    end
     $v \leftarrow$  next task in BFT order to visit;
  end
end

```

Algorithm 1: Pseudo-code of binary-transform algorithm

As mentioned, *Join-transform* routine is invoked when there is a complex dependency between a task and its parents. To remove the complex dependency, it adds virtual tasks in a way that every two of its parent tasks has only one virtual task as their direct child. The virtual task becomes a parent of the task concerned. Then, the connecting edges are updated accordingly. After updating the edges, if number of direct parents of the main task is still more than two, the *Join-transform* routine will be invoked again (with the updated parents as its arguments). This procedure, i.e., adding virtual tasks and updating the related edges are repeated recursively until the task concerned has only two direct parents. Figure 4.2 illustrates the different steps of applying the *Join-transform* routine on a task with six direct parents.

Similarly, the *Branch-transform* routine is invoked when a complex dependency between a task and its children (i.e., the task has more than two children) is found. Therefore, in a similar procedure as *Join-transform* routine, virtual tasks are added in a recursive way until the task has only two direct children. Figure 4.3 shows a sample of using the *Branch-transform* routine to remove the complex dependency of a task with 5 children.

Note that since generating input for PDWMS is not the focus of this project, we use TGFF (to be introduced later in Chapter 5) to generate the binary-DAGs directly. However, the discussed binary transformer can be implemented and used instead.

4.4.3 Partitioning Algorithm

Our partitioning algorithm aims at grouping tasks into partitions through five phases as follows:

Figure 4.2: Applying the *Join-transform* routine to task V and its six parents. The virtual tasks are marked as “VT”.

4.4 Binary-Division Partitioning Approach

53

Figure 4.3: Applying the *Branch-transform* routine to task V and its five children. The virtual tasks are marked as “VT”.

1. *One-to-one transformation phase*
2. *MP-decision phase*
3. *Division phase*
4. *Partition assignment phase*
5. *CT-decision phase*

The flowchart of the algorithm can be seen in Figure 4.4. The cost model described earlier in this chapter is used in all the phases. In addition, the phases I and III (i.e., the one-to-one transformation and the division phases) are topology-based. That is, the structural information of the DAG such as tasks, their position in the DAG and their dependencies with each other are involved in these phases. The last phase (i.e., the CT-decision phase) uses the completion time calculation with the help of the priority queue list scheduling as mentioned in Section 4.3.

As can be seen in the partitioning flowchart, during partitioning a DAG, the one-to-one transformation phase is executed only once; whereas the other phases (i.e., phase II, III, IV, and V) are repeated as long as the conditions determined in phases II and V are met. Further details on each phase is given in the following.

Before explaining the algorithm's phases, there are a few points to be noted. First, it should be reminded that in a binary-DAG, there are limited types of dependencies existed among a task and its direct parents/children. Hence, the task of concern is called as branch, join, one-to-one, entry/exit, and join-branch task depending on the number of direct parents and children it has. Table 4.1 shows the number of

4.4 Binary-Division Partitioning Approach

55

Figure 4.4: Flowchart of the partitioning algorithm

fan-in/fan-out for each type of task (see Figure 4.1 for the graph representation).

Relation Type	Fan-in	Fan-out
Branch	1	2
Join	2	1
One-to-one	1	1
Entry	0	1/2
Exit	1/2	0
join-branch	2	2

Table 4.1: Fan-in and fan-out of different relation types in a DAG

The second point is that, to traverse the DAG in both phases I and III, a BFT is used. Therefore, the time complexity of these two phases would be affected by the order of BFT which is $O(|V| + |E|)$ [11] (the time complexity analysis of the whole algorithm will be discussed in detail in Section 4.4.4). Just as a reminder, BFT algorithm is a traverse method, which traverses a graph in level order (level by level). Intuitively it starts at the entry task and visits all the tasks directly connected to the entry and continues the procedure until it visits all the tasks of the graph.

Terminology:

There are a few terms which are commonly used in the pseudo-code of different phases of the algorithm. They are defined as follows:

- *Partition-arr*: Array of partitions whose members either require to be further partitioned or no resource pool has yet been assigned to them. Initially, we put the original DAG as its first member. In the next round, it is updated by

4.4 Binary-Division Partitioning Approach

57

temp-partition-arr (to be introduced in the following), if the newly generated partitions are accepted (in the CT-decision phase).

- *Temp-partition-arr*: A copy of *partition-arr* which is made at the MP-decision phase in each round. The division and the partition assignment phases use it. It is updated by removing a partition from it whenever a resource pool is assigned to the partition (in the partition assignment phase); or by adding the newly generated partitions (in the division phase) to it.
- *Pool-arr*: Array of resource pools which have not yet been assigned to any partitions. Initially, we add all the resource pools to it. In the next round, it is updated by *temp-pool-arr* (to be introduced in the following), if the newly generated partitions are accepted (in the CT-decision phase).
- *Temp-pool-arr*: A copy of *pool-arr* which is made at the MP-decision phase in each round. The division and the partition assignment phases use it. It is updated by removing the resource pools, when they have been assigned to partitions (in the partition assignment phase).
- *Assigned-arr*: Array of partitions and their assigned resource pools. Each member of it is a pair of “partition, pool”. It is updated by *temp-assigned-arr*, if the newly generated partitions are accepted (in the CT-decision phase).
- *Temp-assigned-arr*: A copy of *assigned-arr* which is made at the MP-decision phase in each round. The division and the partition assignment phases use it. It is updated by adding the pairs of “partition, pool” (in the partition assignment phase).

Phase I: One-To-One Transformation

In the first phase (refer to Algorithm 2 for its pseudo-code), the one-to-one relations are omitted by combining the middle task with either its parent or child task. The objective of this transformation is to shrink the workflow vertically. The act of partitioning is to put every task into a group. Tasks of the type one-to-one relation have no other choice but to be placed in the same partition as their parent or child task. To decide whether the parent or child task should be selected, the weight of the edges connecting the task to its parent and child is compared and the heavier one is removed.

In Figure 4.5-(a), task (A), (B), and (C) represent a one-to-one relation. Due to minimization of communication costs, the heavier edge has higher privilege to be omitted (zeroed). Therefore, depending on whether $O(A, B)$ (i.e., weight of the edge connecting tasks (A) and (B)) is larger than $O(B, C)$ (i.e., weight of the edge connecting tasks (B) and (C)), task (B) joins its parent task (A) (see Figure 4.5-(b)) or its child task (C) (see Figure 4.5-(c)). The result of one-to-one transformation phase would be a shrunk graph which has less number of tasks and edges (i.e., less total communication cost).

Figure 4.6 shows the result of applying the one-to-one transformation to a DAG. Traversing the DAG downward, task (B) is first distinguished as the one-to-one relation type task which is combined with its parent task (A) (because $O(A, B) > O(B, D)$). Then task (D) is marked as a one-to-one relation type task which should join task (G) (because $O(D, G) > O(D, A-B)$).

Figure 4.5: One-to-one transformation: (a), (b) and (c) show the one-to-one relation, the result of transformation if ($O(A, B) > O(B, C)$), and the result of transformation if ($O(B, C) > O(A, B)$) respectively.

It is notable that, in case of using a binary transformer and having virtual tasks in the DAG, one-to-one algorithm simply by-passes virtual tasks while traversing. In fact, a virtual task cannot have one-to-one relation with its parent or child task. As shown in the binary transformation algorithm in Section 4.4.2, all the added virtual tasks can only have branch or join relation types with their parent or child task.

Phase II: MP-Decision

This phase is to check some conditions which are required to be satisfied to continue the partitioning. First, if initially a resource pool with enough computing resources (i.e., more than the MP of the original DAG) is available, further partitioning is not required and a better make-span is achieved by executing the original DAG on a single resource pool. Therefore, the partitioning is not necessary. The second condition is that the number of partitions should not exceed the number of resource pools. Since in each round of division, the number of partitions is increased by one,

Figure 4.6: Applying one-to-one transformation to a sample DAG. One-to-one relation-type tasks are (B) and (D). In step (a), task (B) joins its parents task (A). Then in step (b), task (D) joins its child task (G).

4.4 Binary-Division Partitioning Approach

61

Input: A binary-DAG with root r
Output: A shrunk DAG with the same root

```

begin
   $v_j \leftarrow r$ ;
  start BFT;
  while all tasks are not visited do
    if  $v_j$  is a task with one-to-one relation then
      if  $O(v_i, v_j) > O(v_j, v_k)$  then
        combine tasks  $v_i$  and  $v_j$  and name it as  $v_{i-j}$ ;
        remove task  $v_j$ ;
        remove edges  $e_{i,j}$  and  $e_{j,k}$  and replace them with a new edge
         $e_{i-j,k}$  which has the same weight as  $O(v_j, v_k)$ , connecting tasks
         $v_{i-j}$  and  $v_k$ ;
      else
        combine tasks  $v_j$  and  $v_k$  and name it as  $v_{j-k}$ ;
        remove task  $v_j$ ;
        remove edges  $e_{i,j}$  and  $e_{j,k}$  and replace them with a new edge
         $e_{i,j-k}$  with the same weight as  $O(v_i, v_j)$ , connecting tasks  $v_i$ 
        and  $v_{j-k}$ ;
      end
    end
     $v_j \leftarrow$  next task in BFT order to visit;
  end
end

```

Algorithm 2: Pseudo-code of one-to-one transformation phase

we should stop partitioning when it is reached to the number of resource pools.

Besides, in this phase, the partition-arr, the pool-arr, and the assigned-arr are copied into three temporary arrays namely, temp-partition-arr, temp-pool-arr, and temp-assigned-arr respectively. As earlier mentioned in terminology (refer to Section 4.4.3), these temporary arrays are used by the division and the partition assignment phases; while the three main arrays are kept unchanged. Then in the CT-decision phase (to be described in Section 4.4.3), the temporary and main arrays are used to make decision on whether to continue or stop the partitioning.

Phase III: Division

In the third phase (refer to Algorithm 4 for its pseudo-code), tasks are divided into groups, following a heuristic algorithm that determines which task should belong to which group. Every binary-DAG can be divided into a left and a right group of tasks, starting from the root task which is initially put into a temporary group called *common group*. The direct left child of the root task goes to a group called *left group* while the direct right child of the root task goes to a group called *right group*. By traversing the graph level by level and left to right, every task is examined to see to which group its parents belong. If both parents are in the left group, it will be put into the left group as well. Similarly, if both parents are in the right group, it will be marked as a member of the right group. If one parent is a member of the left group whilst the other parent is a member of the right group, the task is placed in the common group.

Hence, while traversing lower levels of graph we may visit some tasks which have

4.4 Binary-Division Partitioning Approach

63

```

Input: (i) partition-arr; (ii) pool-arr; (iii) assigned-arr
Output: true/false which means either to continue or stop division
begin
  continue-division  $\leftarrow$  true;
  if this is the first round of partitioning then
    pool  $\leftarrow$  Find-largest-pool(pool-arr);
    if  $MP(original-DAG) < |R(pool)|$  then
      continue-division  $\leftarrow$  false;
      assigned-arr.Add(original-DAG,pool);
    end
  end
  if  $partition-arr.size = pool-arr.size$  then
    continue-division  $\leftarrow$  false;
    Sort(partition-arr);
    Sort(pool-arr);
    for  $i = 1; i < partition-arr.size; i = i + 1$  do
      assigned-arr.Add(partition-arr[i],pool-arr[i]);
    end
  end
  if continue-division then
    temp-partition-arr  $\leftarrow$  partition-arr;
    temp-pool-arr  $\leftarrow$  pool-arr;
    temp-assigned  $\leftarrow$  assigned-arr;
  end
  return continue-division;
end

```

Algorithm 3: Pseudo-code of MP-decision phase

their parents (one of them or both) as member of the common group. When one of the task's parents belongs to the common group, the decision is made based upon the other parent's group which could be either the left or the right group. But, when both parents are from the common group, the child should be placed in the common group to be later judged.

After finishing classification of all the tasks, we examine the remaining tasks in the common group. The common group's tasks are examined based on the information provided by the cost model (introduced in Section 4.2) to see whether putting them into the left or the right group can reduce the communication cost between the two generated partitions. Figure 4.7 depicts the states of the three left/right/common groups while partitioning the shown sample DAG d. Following the state of the three groups in Figure 4.7-(b), task (A) in the common group is categorized as member of the right group because $O(A, C) > O(A, B)$. In next step (shown in Figure 4.7-(c)), task (I) is removed from the common group and is added to the left group, due to $O(F, I) + O(I, M) < O(E, I)$. Similarly, in the final step (shown in Figure 4.7-(d)), task (O) is moved from the common group to the left group. The result of partitioning can be seen in Figure 4.8.

Phase IV: Partition Assignment

As mentioned in Section 3.2.2, the partition assignment is done at the same time as the partitioning. Therefore, this phase is in charge of assigning the resource pools to the partitions. The assignment is based on the number of computing resources of the resource pools (note that as described in Section 3.1, the information regarding the resource pools and number of their computing resources are provided by the

4.4 Binary-Division Partitioning Approach

65

Figure 4.7: Different states of the left/right/common groups in the division phase to make two partitions out of the sample DAG d

Figure 4.8: Sample DAG d and its two partitions after division

4.4 Binary-Division Partitioning Approach

67

```

Input: temp-partition-arr
Output: updated temp-partition-arr
begin
    select the largest partition part from temp-partition-arr;
    make three groups called left, right, and common;
    put the root task r of partition part into the common group;
    put the left child of r into the left group;
    put the right child of r into the right group;
     $v_i \leftarrow r$ ;
    start BFT;
    while all tasks are not visited do
        switch the number and the group of parents of task  $v_i$  are do
            case one, from the left/right group
                | place it in the left/right group respectively;
            case two, both from the left/right group
                | place it in the left/right group respectively;
            case two, one from the left and the other from the right group
                | place it in the common group;
            case one, from the common group
                | place it in the common group;
            case two, both from the common group
                | place it in the common group;
            case two, one from the common group and the other from the
                left/right group
                | place it in the left/right group respectively;
        end
         $v_i \leftarrow$  next task in BFT order to visit;
    end
    foreach task  $v_i$  in the common group do
        switch the number and the group of children of task  $v_i$  are do
            case one, from the left/right group
                | place it in the left/right group respectively;
            case two, both from the left/right group
                | place it in the left/right group respectively;
            case two, one from the left and the other from the right group
                | calculate the added communication cost by placing  $v_i$  in the
                | left/right group;
                | choose the option which results in less cost and assign the task
                | accordingly;
        end
    end
    remove part from temp-partition-arr;
    add the left and the right groups as new partitions into
    temp-partition-arr;
end

```

Algorithm 4: Pseudo-code of division phase

Hawkeye).

As can be seen in Algorithm 5, there are two input arrays: *temp-partition-arr* and *temp-pool-arr*. For each partition in the *temp-partition-arr*, we need to find a resource pool with the closest number of computing resources to the MP of the partition. To find such a resource pool, the two sets of partitions and resource pools are sorted. Then, using a binary search per partition, the mentioned resource pool can be found. However, in some cases, for example, a partition that has larger MP than the number of resources in any resource pools, a suitable resource pool might not be found for the partition. In these conditions, the partitions are remained at *temp-partition-arr* to be later decided if further division is required.

```

Input: (i) temp-partition-arr; (ii) temp-pool-arr
Output: temp-assigned-arr
begin
  Sort(temp-partition-arr);
  Sort(temp-pool-arr);
  for part  $\in$  temp-partition-arr do
    pool  $\leftarrow$  Find-pool(part,temp-pool-arr);
    if pool  $\neq$  null then
      temp-assigned-arr.Add(part,pool);
      temp-partition-arr.Remove(part);
      temp-pool-arr.Remove(pool);
    end
  end
  return temp-assigned-arr;
end

```

Algorithm 5: Pseudo-code of partition assignment phase

In the following, an example of using the partition assignment algorithm is explained. Assume, there are four resource pools of p_1 , p_2 , p_3 , and p_4 which have

4.4 Binary-Division Partitioning Approach

69

7, 3, 5, and 3 computing resources respectively. We start partitioning the DAG d with MP of 16. Assume, in the first division round, d is divided into two partitions: d_1 and d_2 with MP of 6 and 10. Using the partition assignment algorithm, p_1 with 7 computing resources is assigned to d_1 (because p_1 has the closest number of computing resources to the MP of d_1 , i.e., 6); whereas due to insufficient number of computing resources in the rest of resource pools, none of them can be assigned to d_2 .

Therefore, partition d_2 should be further divided. In second division round, d_2 is divided into two partitions: d_{2-1} and d_{2-2} with MPs equal to 3 and 7 respectively. Resource pool p_2 is assigned to d_{2-1} ; whereas partition d_{2-2} needs to be further divided. In third division round, d_{2-2} is divided into two partitions: d_{2-2-1} and d_{2-2-2} with MPs of 4 and 3 respectively. Resource pools p_3 and p_4 are assigned to partitions d_{2-2-1} and d_{2-2-2} accordingly. Figure 4.9 shows an overview of the four partitions and their assigned resource pools.

Figure 4.9: An overview of using partition assignment algorithm to assign the resource pools to the partitions of a sample DAG

Phase V: CT-Decision

The CT-decision is made after the division and the partition assignment phases to ensure whether the partitions' CT (completion time) after the last partitioning round is reduced compared to the one before that. If the CT is reduced, the partitioning can be continued; otherwise the previous partitions rather than the new ones should be given as the output of the partitioning algorithm and the partitioning should be stopped. The information provided by the cost model (refer to Section 4.2) and the completion time calculation (mentioned in Section 4.3) are used in this phase. The pseudo-code of the CT-decision phase is illustrated in Algorithm 6. Note that the assigned-arr and the temp-assigned-arr are used to calculate the CT of partitions before and after the last partitioning round.

To illustrate the CT-decision phase, we use the example shown in Figures 4.7 and 4.8. We calculate the completion time of the partitions in two situations: (i) before division: when the whole DAG d is executed on one resource pool (resource pool #1) with two computing resources: processor #1 and processor #2; (ii) after division: when DAG d is divided into two partitions whose assigned resource pools are resource pool #1 and resource pool #2 (each with two computing resources: processor #1 and processor #2).

Gantt charts of the above situations are illustrated in Figures 4.10 and 4.11 respectively. It is assumed that all processors have the same computing power. To help understand the start/finish time of the tasks in Figure 4.10, a ready queue is also presented. Note that the time slots in gray color show the idle times. In case of Figure 4.10, the idle time occurs when there is no ready task in the ready queue,

4.4 Binary-Division Partitioning Approach

71

Input: (i) assigned-arr, (ii) temp-assigned-arr
Output: true/false which means either CT is decreased/increased after the last division round

```

begin
  remaining-assigned-arr  $\leftarrow$  assign the remaining partitions in partition-arr
  to the remaining pools in pool-arr using first fit;
  remaining-temp-assigned-arr  $\leftarrow$  assign the remaining partitions in
  temp-partition-arr to the remaining pools in temp-pool-arr using first fit;
  CT-reduced  $\leftarrow$  false;
  CT-before  $\leftarrow$  completion time of the partitions in the assigned-arr  $\cup$ 
  remaining-assigned-arr;
  CT-after  $\leftarrow$  completion time of the partitions in the temp-assigned-arr
   $\cup$  remaining-temp-assigned-arr;
  if  $CT\text{-}before > CT\text{-}after$  then
    CT-reduced  $\leftarrow$  true;
    partition-arr  $\leftarrow$  temp-partition-arr;
    pool-arr  $\leftarrow$  temp-pool-arr;
    assigned-arr  $\leftarrow$  temp-assigned-arr;
  else
    CT-reduced  $\leftarrow$  false;
  end
  return CT-reduced;
end

```

Algorithm 6: Pseudo-code of CT-decision phase

while there exists free processors. However, in Figure 4.11, the idle time occurs not only in the above case, but also when there is a communication delay. For example, task (B) cannot be started until the output of task (A) arrives (because $\text{Comm-cost}(\text{resource pool\#2}, A, \text{resource pool\#1}, B) = 3$). A similar situation also happens for the task (O). Communication between resource pools can also be overlapped with task processing, for example, the output of task (F) sent from resource pool #2 to resource pool #1 and the output of task (I) sent from resource pool #1 to resource pool #2. Finally, as can be seen, partitioning caused a reduction of 25 ($122 - 97 = 25$) in the completion time of the DAG, thus the partitions are accepted.

4.4.4 Time Complexity Analysis

The overall time complexity of the binary-partitioning algorithm is the summation of its five phases' time complexity as follows:

- Phase I (one-to-one transformation): It is equal to the time complexity of BFT, that is, $O(|V| + |E|)$. The reason is all the tasks should be visited and for those with the one-to-one relations, some transformation actions with $O(1)$ time should be performed.
- Phase II (MP-decision): It takes $O(1)$ time to check whether the two conditions are met. The cost of making copy of the three arrays (i.e., *partition-arr*, *pool-arr*, and *assigned-arr*) is equal to the cost of the largest one which is $O(|pools|)$ (where $|pools|$ denotes the size of *pool-arr*). In the case of partition assignment for the remaining partitions and resource pools (i.e., when the number of partitions has reached to the number of resource pools), it

4.4 Binary-Division Partitioning Approach

73

Figure 4.10: Gantt chart of completion time of DAG d before division over resource pool #1 along with the ready queue of the its tasks

4.4 Binary-Division Partitioning Approach

75

needs to sort two arrays which is in order of $O(|parts| \times \log |parts|)$ and $O(|pools| \times \log |pools|)$, and then there is a “for” loop which takes $O(|parts|)$ (where $|parts|$ denotes the size of *partition-arr*). $|parts|$ cannot exceed the $|pools|$, and $|pools|$ compared to $|V|$ is a very small number. Therefore, the cost of this phase is much less than that of phase I.

- Phase III (division): The complexity is the summation of $O(|V| + |E|)$ for the main “while” loop (which is to traverse the whole DAG using BFT), and $O(|V_c|)$ for the “for” loop (which is to examine members of the common group), assuming $|V_c|$ is the number of tasks in the common group. We have $|V_c| \ll |V|$ (that is the number of the common group members is definitely less than the total number of tasks). Therefore, the total time taken to do the division is equal to $O(|V| + |E|) + O(|V|)$, which is in the order of $O(|V| + |E|)$.
- Phase IV (partition assignment): Sorting the two arrays of partitions and pools (i.e., *temp-partition-arr* and *temp-pool-arr*) takes $O(|parts| \times \log |parts|)$ and $O(|pools| \times \log |pools|)$ respectively (note that the size of the temporary arrays is in the same order as the size of the main arrays). Besides, to find the proper pool for each partition, a binary search should be done which takes $O(|parts| \times \log |pools|)$. Since $|parts| \leq |pools|$ and $|pools| \ll |V|$, the cost of this phase is much less than $O(|V| + |E|)$.
- Phase V (CT-decision): We compare the completion time of the DAG’s partitions before and after a division round. As mentioned in Section 4.3, a modified version of the priority queue list scheduling algorithm is used for this purpose. However, the modification in the algorithm does not change its complexity. Therefore, it is in the order of $O(|V| \times \log |V|)$ as reported in [40].

As mentioned earlier in the binary-partitioning algorithm (refer to Section 4.4.3), only phase I of the algorithm is executed once; whereas the rest of phases iterate at most $|pools|$, which is much less than $|V|$ and can be regarded as a constant. According to the above time complexity analysis, the phase V (i.e., the CT-decision phase) is the dominating phase. Therefore, the total time complexity of the binary-partitioning algorithm would be equal to $K \times O(|V| \times \log |V|)$.

4.5 Summery

In this chapter, a full discussion on workflow partitioning was done. We started with motivation of partitioning and problem statement. Then we showed how a DAG can be transformed to a binary-DAG. Later a binary partitioning heuristic was proposed. This heuristic is able to partition the workflow in five phases, namely, one-to-one transformation, MP-decision, division, partition assignment, and CT-decision.

The first and third phases (i.e., the one-to-one and the division phases) help to put dependent tasks in the same partition; whereas the second and fifth phases (i.e., MP-decision and CT-decision) are to make decisions on whether further partitioning is required or it should be stopped. The forth phase (i.e., the partition assignment phase) is to assign a resource pool to each partition. The information required for the fifth phases of binary partitioning algorithm is provided by the cost model (refer to Sections 4.2) and the completion time calculation using the priority queue list scheduling algorithm as mentioned in Section 4.3.

Chapter 5

Experiments and Results

In this chapter, we evaluate our system and justify the claims stated previously. To achieve this, we design several experiment sets, including different DAG, Grid and execution models. There are several parameters involved in each of these models which will be explained in the following sections.

5.1 DAG Model

We have generated several random DAGs, using the open source software TGFF (introduced in Section 3.1). TGFF provides a flexible and standard way of generating pseudo-random task-graphs for use in scheduling and allocation research. This includes the areas of embedded systems, hardware/software co-design, operating systems (both real-time and general-purpose), parallel or distributed hardware or software studies, flow-shop scheduling, as well as any other area which requires problem instances consisting of partially-ordered or directed acyclic graphs (DAGs) of tasks [50].

There are several assumptions which are taken into consideration in the DAG generation:

- The input DAG accepted by our partitioning algorithm should be in binary format. The binary-DAGs can be generated either directly or by using a binary transformer that converts original DAGs to the binary ones (refer to Section 4.4.2 for more details). In PDWMS, we generate binary-DAGs directly using TGFF rather than the binary transformer. For this purpose, a parameter in the TGFF input file (denoted as “task_degree”) is set to 2.
- All the required input data for a task are provided by other tasks in the DAG. That is, we do not consider data input directly from data repository.

To generate different DAGs, there are a number of parameters which could be set and used as the input to the TGFF. We use a set of fixed parameters for all our samples while changing a few of them as follows (note that the actual attribute name in the input file of the TGFF is provided in the parentheses):

- *Wid*: The range of independent tasks in each level, which approximates the maximum parallelism (MP) of the DAG (denoted as `series_wid` in the input file)
- *Len*: The number of levels of a DAG (denoted as `series_len`)
- *R/C(d)*: The average ratio of tasks’ weights over edges’ weights, which could be specified by setting the proper average weights for tasks and edges (denoted as `task_weight` and `edge_weight`)

To classify the generated DAGs, we first divide them into three groups of different R/C values, called as RCL, RCE, and RCG. The idea of this classification came from granularity categorization used by many researchers in this area (e.g., [54]). It is common to define DAGs based on the ratio of their average weight of tasks over average weight of edges. This classification helps identify the application of the system and algorithm since it may have the best results in only one or more of these categories. The classification is as follows:

- *RCL*: The R/C range is 0.1-0.3. The average weight of tasks is a lot less than the average weight of edges. These graphs are called *fine-grain*.
- *RCE*: The R/C range is 0.8-1.2. The average weights of tasks and edges are close.
- *RCG*: The R/C range is 3-10. The average weight of tasks is more than the average weight of edges by 3 to 10 times. These graphs are called *coarse-grain*.

As described in Chapter 4, partitioning may reduce the make-span. This is due to grouping dependent tasks into partitions and executing the partitions in parallel. On the other hand, communication cost, added for data transfer among the partitions, could diminish the effect of partitioning in make-span reduction. By using coarse-grain DAGs, i.e., DAGs with average weight of tasks much higher than average weight of edges, the overhead of communication cost might be decreased.

It should be noted that even in the case that partitions are load-balanced according to the available resource pools, having idle time slots is inevitable. Although there are enough computing resources to execute the tasks, they might not start

execution for a certain amount of time due to the existing dependencies between their assigned tasks and the tasks of other partitions (mapped onto the different clusters) which have not finished their execution yet.

According to the above reason, in our experiments, we focus on coarse-grain graphs (denoted by RCG) which have an R/C range set to 3-10. We generate three types RCG graphs named as: (i) normal (RCG); (ii) added-width ($RCG+w$); and (iii) added-length ($RCG+l$). RCG+w is a variation of RCG, which has a greater width in comparison with the normal RCG. Similarly, RCG+l is the representative of DAGs which have the same parameters as RCG but with longer length (i.e., more levels). In Appendix A, three generated samples of RCG, RCG+w, and RCG+l DAGs are demonstrated.

Moreover, Table 5.1 shows the specification of RCG groups including the values assigned to their attributes. The R/C of these RCG, RCG+w, and RCG+l graphs is set to 8, while the average weight of tasks (in terms of computation cost) is 40 sec and the average weight of edges (in terms of communication cost) is 5 sec (note that the weight can be stated in either data or time unit; here we use time unit). As shown in the table, the width of graph in case of the RCG and RCG+l graphs is in the range of [5..15], while it is increased to [15..25] for the RCG+w group. Similarly, the length of graph in case of RCG and RCG+w is set to [1..11], whilst it is increased to [7..19] in case of the RCG+l group.

DAG	Wid	Len
RCG	[5-15]	[1-11]
RCG+w	[15-25]	[1-11]
RCG+l	[5-15]	[7-19]

Table 5.1: RCG groups specification

5.2 Grid Model

The next step is to set up the Grid model. We use three clusters to run the experiments (denoted as *HPC*, *PDCBL*, and *PDCBL2*), each of which has only one resource pool. A combination of these three clusters with different number of resources is used in our experiments.

Table 5.2 shows the specification of these clusters, such as their resources' specification (including operating system, architecture, speed, RAM, whether they are single/dual core, and number of CPUs per core), and the version of Globus Toolkit and Condor installed. In our experiments, computing resources are considered at the core-level. Therefore, according to Table 5.2, the total number of cores in *PDCBL* is equal to that in *PDCBL2*.

There are several assumptions taken into consideration to ease setting up the Grid model:

- There is only one resource pool per cluster.
- The number of resources in each cluster is assumed not to be changed during execution.
- The computing resources within the resource pools are homogeneous whereas clusters are heterogeneous.

Cluster	HPC	PDCBL	PDCBL2
Speed (GHz)	2.59	3	3
OS	Linux-Redhat2.7	Linux-Redhat3	Linux-Redhat Enterprise Server 5
Arch	X86 (32bit)	X86_64-LINUX_RHEL3	X86_64
RAM (GB)	1	4	4
Condor Platform	I386-LINUX_RH72	X86_64-LINUX_RHEL3	X86_64-LINUX_RHEL5
Globus Version	4.0.4	4.0.4	4.2.1
Max # of Machines	11	16	8
Condor Version	6.8.5	6.8.0	7.0.4
CPU/ Machine	2	2	2
Core/ CPU	1	2	4
Total # of Cores	22	64	64

Table 5.2: Clusters configuration

- The computing resources of HPC, PDCBL, and PDCBL2 are assumed to satisfy the requirements of the examined DAGs' tasks in terms of operating system, architecture, memory and so on.
- The connecting links between the clusters have the same transmission rate.
- The cost of data transfer among resources of the same cluster (i.e., inter-cluster data transfer) are ignored. That is, the only communication cost is caused by transferring data between various clusters (i.e., intra-cluster data transfer). This communication delay is calculated by estimating the time taken to transmit data (i.e., output of tasks) between resource pools of the two clusters.

5.3 Execution Model

The experiments were conducted using three execution models as follows:

- *SPSE: Single-Pool-Single-Engine* which refers to the normal execution of a DAG over one resource pool, using condor-DAGMan as a single engine.
- *MPSE: Multiple-Pool-Single-Engine* which refers to having one engine to execute a DAG but over multiple resource pools. Condor-G is used as a meta-scheduler for the execution of the DAG. In this model, Condor-G dispatches the tasks among different resource pools. Note that despite the fact of having several clusters participating in the execution, this model is still considered as a centralized model as the submitter (i.e., Condor-G) is the central point which could become the bottle-neck of the system.

- *MPME: Multiple-Pool-Multiple-Engine* which refers to having several resource pools as well as multiple engines in the execution of a DAG. This is the execution model used in PDWMS. In other words, in the MPME model, DAGs are executed in a distributed manner, using several resource pools, each of which has one condor-DAGMan.

5.4 Experiment Sets

We conduct several experiment sets with different objectives. Table 5.3 shows the list of experiments including their execution, DAG, and Grid models. Note that each data point used in the diagrams is an average over three runs. Besides, the standard deviation of the data points are also shown in their plot. Finally, since the main target of the experiments is to illustrate the effect of using MPME (the proposed execution model of PDWMS) on the make-span, a comparison of MPSE and MPME is also presented at the end.

5.4.1 Experiment Set I

Figure 5.1 focuses on the effect of computing resource performance on make-span. In this experiment, the DAG model includes: RCG, RCG+w, and RCG+l DAGs, while the execution model is fixed to SPSE. The DAGs are executed on HPC, PD-CBL, and PDCBL2 clusters with seven cores respectively.

The difference between make-spans of clusters come from their different specifications. As can be seen, HPC has the longest make-span in all three cases of the DAGs, because its machines have less power such as 2.59 GHz CPU, 1GB RAM,

5.4 Experiment Sets

85

Set	EXE Model	DAG Model	Grid Model	Objective
I	SPSE	RCG RCG+w RCG+l	HPC/PDCBL/PDCBL2 (with fixed # computing resources)	Study the effect of computing resource performance on MS
II	SPSE	RCG RCG+w RCG+l	HPC (with # computing resources=3 and 7)	Study the effect of computing resource configuration on MS
III	MPSE	RCG RCG+w RCG+l	Different combination of resource pools	Study the performance of MPSE on various DAGs and cluster combinations
IV	MPME	RCG RCG+w RCG+l	Different combination of resource pools	Study the performance of MPME on various DAGs and cluster combinations

Table 5.3: Experiment sets' specification

Figure 5.1: Experiment Set I (EXE: SPSE, DAG: [RCG/RCG+w/RCG+l], Grid: [HPC=7/PDCBL=7/PDCBL2=7]): Effect of computing resource performance on make-span

and 32bit processor; whereas PDCBL and PDCBL2 both have 3GHz CPU, 4 GB RAM, and 64bit processors. Besides, PDCBL and PDCBL2 have 4 and 8 cores per machine respectively. The results reveal that PDCBL2's performance is always worse than PDCBL in case of executing our sample DAGs. The reason is that our DAG tasks are not CPU-bound. The performance of PDCBL2 is less than PDCBL, because RAM, network and other I/O resources in its machines are shared among its cores (i.e., 8 cores). This creates a more competitive environment than 4-core machines, and thus causing the non CPU-bound (e.g., I/O bound) tasks to take even longer time for execution.

5.4.2 Experiment Set II

To show the effect of changing the computing resource configuration on make-span, experiment set II was conducted. The execution model on this experiment was fixed to SPSE, and different types of DAGs were used. As can be seen, by adding more computing resources (i.e., from 3 to 7), the make-span decreases in all the three DAGs. As earlier mentioned, the MP of the examined RCG, RCG+w, and RCG+l is 9, 18, and 11 respectively. It means, at least, 9, 18, and 11 computing resources are required in each case respectively to ensure no ready task is kept waiting (in the queue) for a computing resource to become available. Here, although the minimum required computing resources of these DAGs are provided neither in case of "HPC=3" nor in case of "HPC=7", more tasks can be executed in parallel in case of "HPC=7" and thus its make-spans are reduced.

Figure 5.2: Experiment Set II (EXE: SPSE, DAG: [RCG/RCG+w/RCG+l], Grid: [HPC=3/HPC=7]): Effect of computing resource configuration on make-span

5.4.3 Experiment Set III

In this set, the execution model is fixed to MPSE. As explained earlier, Condor-G is used in this execution model as a meta-scheduler/dispatcher to distribute tasks among the clusters. Each of the clusters has a resource pool, thus we have a multiple resource pool environment. On the other hand, since the Condor-G manages the whole process of execution, we call it single engine model. To have multiple clusters participating, all the HPC, PDCBL, and PDCBL2 clusters are involved. We examine the RCG, RCG+w, and RCG+l DAGs against two computing resource combinations: (i) “HPC=3, PDCBL=7, PDCBL2=7”; and (ii) “HPC=7, PDCBL=7, PDCBL2=7”. Figure 5.3 shows the result.

Compared with experiment I, although more computing resources were used, longer

Figure 5.3: Experiment Set III-1 (EXE: MPSE, DAG: [RCG/RCG+w/RCG+l], Grid: [HPC=3, PDCBL=7, PDCBL2=7]/[HPC=7, PDCBL=7, PDCBL2=7]): Performance of MPSE on various DAGs and cluster combinations

make-spans were obtained. The reason is the communication overhead. Besides, in Condor-G data transfer is done in a centralized manner. It means all the intermediate data which are required to be transferred between clusters are passed through the submitter cluster. (In MPSE model, one of the clusters is considered as a dispatcher of tasks to other clusters, including itself. This cluster is called the submitter cluster). In other words, the submitter cluster has an intermediary role for data communication which results in a larger make-span.

When the total number of computing resources is increased from “HPC=3, PDCBL=7, PDCBL2=7” to “HPC=7, PDCBL=7, PDCBL2=7”, all the three make-span values, corresponding to the RCG, RCG+w, and RCG+l DAGs, increase. There are two reasons: (i) the more resources in HPC, the more tasks are assigned to its machines. Since HPC’s machines are less powerful (in terms of CPU speed and memory) in comparison with machines of PDCBL and PDCBL2, the overall make-span will increase; and (ii) Condor-G does not care about the tasks’ dependencies (in contrast with PDWMS which takes care of them). Having more computing resources in HPC, more tasks might be assigned to different clusters and thus increasing the communication costs.

On the other hand, by increasing the number of computing resources in PDCBL with more powerful machines compared to those of HPC or PDCBL2, i.e., from “HPC=3, PDCBL=3, PDCBL2=7” to “HPC=3, PDCBL=7, PDCBL2=7”, the make-span in case of the three examined DAGs has a decrease. Figure 5.4 illustrates the new results.

Figure 5.4: Experiment Set III-2 (EXE: MPSE, DAG: [RCG/RCG+w/RCG+l], Grid: [HPC=3, PDCBL=3, PDCBL2=7]/[HPC=3, PDCBL=7, PDCBL2=7]): Performance of MPSE on various DAGs and cluster combinations

5.4.4 Experiment Set IV

In the forth group of experiments, we focus on the MPME model as the fixed execution model. The same set of DAGs used in experiment set III is adopted. In this experiment, we obtain the make-span of different DAG models under the same cluster combinations as used in experiment III-1 but with MPME execution model.

Figure 5.5: Experiment Set IV (EXE: MPME, DAG: [RCG/RCG+w/RCG+l], Grid:[HPC=3,PDCBL=7,PDCBL2=7]/[HPC=7,PDCBL=7,PDCBL2=7]): Performance of MPME on various DAGs and cluster combinations

Figure 5.5 demonstrates the results. An increase in make-span is observable by changing the DAG from RCG to RCG+w and RCG+l. This is due to more number of tasks and greater MP of the RCG+w and the RCG+l DAGs compared to the RCG DAG. Besides, as can be seen, changing the computing resource combina-

tion from “HPC=3,PDCBL=7,PDCBL2=7” to “HPC=7,PDCBL=7,PDCBL2=7” causes a lower make-span for all the three DAGs. As our results show, in case of RCG+w DAG, according to number of computing resources of the three clusters, three partitions with MP of 5, 8, and 5 are generated. HPC, PDCBL, and PDCBL2 are assigned to them respectively in both case of computing resource combinations. Adding computing resource of HPC from 3 to 7 decreases the make-span because the first partition with MP of 5 has more resources to be executed. There are similar reasons for the case of RCG, and RCG+l DAG.

Moreover, observing the results of the MPME model, it can be seen how the generated partitions fit the available resource pools, while having the minimum dependencies. Take the example of the RCG+w DAG with 138 tasks and 162 edges, R/C equal to 8, and width (MP) fixed at 18. With the Grid setup of [HPC=7, PDCBL=7, PDCBL2=7], the results of partitioning and mapping the above DAG reveal that three partitions are generated with 5, 8, and 5 tasks and are mapped onto HPC, PDCBL, and PDCBL2 respectively. As can be seen, the number of tasks in each partition is best fit the resource pool onto which it is mapped. This is actually due to monitoring the size of partitions to fit the resource pools, carried out by the partition assignment phase of the partitioning algorithm (refer to Section 4.4.3).

On the other hand, the number of overall dependencies among the above three partitions is 25, which is a great reduction compared to the original 162 dependencies (edges). The minimum number of dependencies among partitions can be achieved through the other two phases of the partitioning algorithm, i.e., the one-to-one transformation phase and the division phase. This is actually owing to grouping

the tasks along with their immediate parents/children in one partition, while the lower communication cost breaks the tie if two tasks share a common parent/child (refer to Sections 4.4.3 for more details).

5.4.5 Comparing Make-span of MPSE and MPME Execution Models

In this section, we compare the results of previous experiments. Tables 5.4 and 5.5 compare the make-span values obtained from executing the RCG, RCG+w, and RCG+l DAGs under the MPSE and MPME execution models. They correspond to the different computing resource combinations of “HPC=3, PDCBL=7, PDCBL2=7” and “HPC=7, PDCBL=7, PDCBL2=7” respectively.

DAG	MPSE	MPME
RCG	56.14	30.67
RCG+w	67.58	40.45
RCG+l	71.68	43.2

Table 5.4: DAG: [RCG/RCG+w/RCG+l], Grid:[HPC=3,PDCBL=7,PDCBL2=7]: Comparing make-span of MPSE and MPME execution models-I

DAG	MPSE	MPME
RCG	73.62	29.8
RCG+w	75.35	38.11
RCG+l	83.2	42.1

Table 5.5: DAG: [RCG/RCG+w/RCG+l], Grid:[HPC=7,PDCBL=7,PDCBL2=7]: Comparing make-span of MPSE and MPME execution models-II

Comparing the results of the MPSE and the MPME model, make-span of MPME

is always less than the values obtained from MPSE.

In other words, the results show that PDWMS, used as a workflow management system, with the support of: (i) partitioning; (ii) collaborative execution with multiple engines; and (iii) peer-to-peer data movement, gives better performance for the DAGs under study in comparison with the MPSE model using Condor-G. Although the MPSE model takes advantage of sharing resources over multiple clusters, it may not reduce the overall make-span because the scheduling does not consider dependencies when scheduling tasks amongst clusters.

5.5 Summery

In this chapter, we presented a set of experiments with different objectives to evaluate our system. First the factors that affect the performance of PDWMS were categorized into DAG, Grid, and execution models. For the DAG model, we generated several random DAGs in RCG DAG category, namely, RCG, RCG+w, and RCG+l. For the Grid model, we prepared three clusters with adjustable number of resources to provide us with different combination of computing resources. For the execution model, three models of SPSE, MPSE and MPME were used. Combining the DAG, Grid, and execution models, different experiment sets were defined and their obtained data points were studied.

The final results reveal that overall, the MPME execution model proposed by the PDWMS gives the better make-span in executing different DAGs used in the experiments. These make-spans were compared to the results gained from the MPSE

model for executing the same DAGs and under the same Grid model.

Chapter 6

Conclusions

In this thesis, we discussed the role which current workflow management systems play in the world of Grid computing. Among different involved issues in workflow management systems (which are classified in a taxonomy [55]) such as workflow design, mapping, enactment, monitoring and fault-tolerance, some of them are selected as the focus of this thesis. Therefore, the main focus of the system proposed in this thesis is on workflow partitioning, enactment and data movement. The decision for such a selection was made by studying the existing systems, mapping them on the mentioned taxonomy and investigating their pros and cons.

The result of the above-mentioned study can be summarized as follows:

- Workflow partitioning is a noteworthy feature in workflow management systems, which has rarely been considered by researchers (e.g., even the comprehensive taxonomy mentioned above does not include the partitioning as a feature, based on which workflow management systems can be categorized). However, in the proposed system, partitioning is used as an important factor

to reduce the workflow make-span.

- The existing workflow enactment approaches are mostly centralized. However, we know that due to the distributed nature of Grid, it is not proper to use such enactment models. Besides, having centralized enactment may cause some common problems such as single point of failure. Even in some existing distributed models such as Gridbus, they still rely on a coordinator to a certain degree [10].
- A good combination of partitioning method along with a proper enactment and also data movement model can lead to a system which is able to execute workflows more efficiently.

6.1 Summery of Contributions and Results

According to the three mentioned focused areas (i.e., partitioning, enactment, and data movement), a parallel and distributed workflow management system (PDWMS) was proposed. It executes the workflow using a network of collaborative engines. Therefore, different parts of the workflow can be executed on different engines concurrently. For this purpose, parallel partitions of the workflow should be determined using a partitioning algorithm which is also described in the thesis.

Eventually, after completing the design and implementation of the system, we evaluated our system. For that purpose, a series of test scenarios were designed which were fully explained in chapter 5. There, we categorized DAG, Grid and execution models into several subgroups, trying to study their individual effect on the make-

span.

The final results show that execution of the RCG category of DAGs in a multiple cluster environment (each with a limited number of resources) using the MPME execution model gives the least make-span in comparison with other scenarios such as executing the same DAGs under the execution model MPSE.

6.2 Recommendations of Future Work

Although we obtained the results in the defined scope of this thesis, there are still many open ways for further study as well as improving the performance of PDWMS. These could happen in the design, implementation and experiment level such as:

- Developing more partitioning heuristics and examining them against the one currently used in the PDWMS. This can be carried out by modifying (or even directly using) available algorithms such as: *reverse-clustering* [60], *edge-zeroing* [15], and so on.
- Generalizing the current partitioning algorithm to be applicable to not only binary-DAGs but also the n-ary ones (here by a general partitioning algorithm, we mean the one which can deal with all type of DAGs directly, i.e., without using any further step such as binary transformation as introduced in Section 4.4.2).
- Further analyzing partitions to understand the trade-off between computation cost and communication cost in the cost model, for example when the communication costs are substantially different between clusters.

- Running the experiments with other DAG categories including RCL and RCE. Moreover, testing of the partitioning algorithm on the following benchmark DAGs of known application is also recommended: Random Graphs with Pre-Determined Optimal Schedules (RGPOS); trace set of benchmarks by Kwok and Ahmand [27]; and graphs from applications in the trace graph set which include Cholesky decomposition, Fast Fourier Transform, Gaussian Elimination, LU decomposition and Mean Value Analysis [26].
- Running the experiments over WAN.
- Expanding the generated task graphs to have more partitions and running the experiments using more clusters.

Appendices

Appendix A

Sample of RCG DAGs Used as System's Input

Figure 1: RCG

Figure 2: RCG+w

Appendix B

Sample of DAG Submit Files with Pre/Post-Scripts

```

...
JOB t0_47 t0_47.submit
JOB t0_55 t0_55.submit
JOB t0_63 t0_63.submit
JOB t0_75 t0_75.submit
...
PARENT t0_47 CHILD t0_48
PARENT t0_48 CHILD t0_49
PARENT t0_49 CHILD t0_50
PARENT t0_50 CHILD t0_51
PARENT t0_51 CHILD t0_52
PARENT t0_52 CHILD t0_53
PARENT t0_53 CHILD t0_54
PARENT t0_55 CHILD t0_56
PARENT t0_56 CHILD t0_57
...

SCRIPT POST t0_54 /PDWMS/bash_files/add_dummy_tasks/dummy_parent pd-
cbl.ntu.edu.sg /PDWMS/input_output/metadag/RCG+w/RCG+w/a0_61.out

SCRIPT POST t0_112 /PDWMS/bash_files/add_dummy_tasks/dummy_parent pd-
cbl.ntu.edu.sg /PDWMS/input_output/metadag/RCG+w/RCG+w/a0_124.out

SCRIPT POST t0_90 /PDWMS/bash_files/add_dummy_tasks/dummy_parent pd-
cbl.ntu.edu.sg /PDWMS/input_output/metadag/RCG+w/RCG+w/a0_100.out

SCRIPT POST t0_85 /PDWMS/bash_files/add_dummy_tasks/dummy_parent pd-
cbl.ntu.edu.sg /PDWMS/input_output/metadag/RCG+w/RCG+w/a0_153.out
...
SCRIPT PRE t0_57 /PDWMS/bash_files/add_dummy_tasks/dummy_child a0_156.out
...

```

Table 1: A part of partition RCG+w.dag file

Appendix C

Publication List for this Research

Maryam Khademi Hedayat, W. Cai, S.J. Turner, and Sh. Shahand. Distributed Execution of Workflow Using Parallel Partitioning. In proceedings of the IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA2009), Chengdu and Jiuzhai Valley, China, August 2009.

Bibliography

- [1] I. Altintas, A. Birnbaum, K.K. Baldridge, W. Sudholt, M. Miller, C. Amoreira, Y. Potier, and B. Ludaescher. A framework for the design and reuse of grid workflows. In *International Workshop on Scientific Aspects of Grid Computing*, pages 120–133. Springer-Verlag, 2005.
- [2] Askalon. <http://www.dps.uibk.ac.at/askalon>.
- [3] J. Cao, S.A. Jarvis, S. Saini, and G.R.Nudd. Gridflow: workflow management for grid computing. In *Proceedings of CCGrid 2003. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003.*, pages 198–205, May 2003.
- [4] Condor-G. <http://www.cs.wisc.edu/condor/condorg/>.
- [5] Condor. <http://www.cs.wisc.edu/condor/>.
- [6] K. Cooper, A. Dasgupta, K. Kennedy, C. Koelbel, M. Mazina, A. Mandal, J. Mellor-crummey, F. Berman, H. Casanova, A. Chien, H. Dail, X. Liu, A. Olugbile, O. Sievert, H. Xia, L. Johnsson, B. Liu, M. Patel, D. Reed, W. Deng, and C. Mendes. New grid scheduling and rescheduling methods in the grads project. In *Proceedings of NSF Next Generation Software Work-*

- shop: International Parallel and Distributed Processing Symposium. Santa Fe, USA: IEEE CS*, pages 209–229. Press, 2004.
- [7] DAGMan. <http://www.cs.wisc.edu/condor/dagman/>.
- [8] E. Deelman, J. Blythe, A. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi, and M. Livny. *Pegasus: Mapping scientific workflows onto the grid*, volume 3165/2004 of *Lecture Notes in Computer Science*, pages 11–20. Springer Berlin/Heidelberg, December 2004.
- [9] R. Duan, T. Fahringer, R. Prodan, J. Qin, A. Villazn, and M. Wiczorek. *Advances in Grid Computing - EGC 2005*, volume 3470/2005, chapter Real World Workflow Applications in the Askalon Grid, pages 454–463. Springer Berlin / Heidelberg, 2005.
- [10] R. Duan, R. Prodan, and Th. Fahringer. Run-time optimization of grid workflow applications. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing GRID '06*, pages 33–40, Washington, DC, USA, 2006. IEEE Computer Society.
- [11] H. El-Rewini, Th.G. Lewis, and H.H. Ali. *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
- [12] T. Fahringer, A. Jugravu, S. Pillana, R. Prodan, C. Seragiotto, and H. Truong. Askalon: a tool set for cluster and grid computing: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):143–169, 2005.
- [13] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H. Truong, A. Villazon, and M. Wiczorek. Askalon: A grid application

- development and computing environment. *International Workshop on Grid Computing, IEEE/ACM*, pages 122–131, 2005.
- [14] Y. Feng, W. Cai, and J. Cao. Communication partner identification in distributed job workflow execution over the grid. In *25th IEEE International Conference on Distributed Computing Systems Workshops, 2005.*, pages 587–593, June 2005.
- [15] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4:686–701, 1993.
- [16] Globus Toolkit. <http://www.globus.org/toolkit/>.
- [17] GrADS. <http://www.iges.org/grads/>.
- [18] GRAM. <http://www.globus.org/toolkit/docs/4.0/execution/wsgram/>.
- [19] Gridbus. <http://www.gridbus.org/>.
- [20] GridFTP. <http://www.globus.org/toolkit/docs/4.0/data/gridftp/>.
- [21] GriPhyN. <http://www.griphyn.org/>.
- [22] Hawkeye. <http://www.cs.wisc.edu/condor/hawkeye/>.
- [23] ICENI. <http://www.lesc.ic.ac.uk/iceni/>.
- [24] M. Iverson and F. Ozguner. Dynamic, competitive scheduling of multiple dags in a distributed heterogeneous environment. In *Proceedings of the Seventh Heterogeneous Computing Workshop HCW '98*, pages 70–79, Washington, DC, USA, 1998. IEEE Computer Society.

- [25] C. Kesselman and I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.
- [26] Y. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, 1996.
- [27] Y. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. In *Proceedings of the 12th International Parallel Processing Symposium on International Parallel Processing Symposium IPPS '98*, pages 531–537, Washington, DC, USA, 1998. IEEE Computer Society.
- [28] G. Von Laszewski, M. Hategan, and D. Kodeboyina. *Java CoG Kit Workflow Concepts for Scientific Experiments*, pages 340–356. Springer London, Dec 2007.
- [29] C.H. Lee, D. Lee, and M. Kim. Optimal task assignment in linear array networks. *IEEE Transactions on Computers*, 41(7):877–880, Jul 1992.
- [30] Y. Liu, W. Qu, Z. Feng, D. Zhao, and W. Zuo. A novel method for realizing grid workflow modeling. In *11th International Conference on Computer Supported Cooperative Work in Design, 2007. CSCWD 2007.*, pages 812–817, April 2007.
- [31] F. Long and H. K. Pung. Temporal decomposition and scheduling of grid workflow. In *Proceeding of the 3rd International Workshop on Grid Computing and Applications GCA2007*, pages 53–64, 2007.

- [32] A. Mayer and et al. Workflow expression: Comparison of spatial and temporal approaches. In *Workflow in Grid Systems Workshop, GGF-10, Berlin*, pages 1–7, March 2008.
- [33] C.L. McCreary, A.A. Khan, J. Thompson, and M.E. McArdle. A comparison of heuristics for scheduling dags on multiprocessors. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 446–451, 1994.
- [34] S. McGough, L. Young, A. Afza, S. Newhouse, and J. Darlington. Workflow enactment in iceni. In *UK e-science all-hands meeting, AHM 2004, (September 2004), Nottingham, UK*, pages 894–900, September 2004.
- [35] MDS. <http://www.globus.org/toolkit/docs/4.0/info/key-index.html>.
- [36] Mean ValueAnalysis. .
- [37] NVO. <http://www.us-vo.org/>.
- [38] T. Oinn, M. Addis, J. Ferris, D. Marvin, T. Carver, M.R. Pocock, and A. Wipat. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20:3045 – 3054, 2004.
- [39] Pegasus. <http://pegasus.isi.edu/>.
- [40] A. Radulescu and A.J.C. van Gemund. On the complexity of list scheduling algorithms for distributed-memory systems. In *Proceedings Int’l ACM Conf. on Supercomputing (ICS’99)*, July 1999.
- [41] O.F. Rana. Creating and managing distributed scientific workflows: Techniques and tools, 2005.

- [42] RLS. <http://www.globus.org/rls/>.
- [43] SCEC-CME. <http://epicenter.usc.edu/cmeportal/>.
- [44] G. Singhj, C. Kesselman, and E. Deelman. Optimizing grid-based workflow execution. *Journal of Grid Computing*, 3:201–219, 2005.
- [45] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. Softw. Eng.*, 3(1):85–93, 1977.
- [46] T. Tannenbaum. Scheduling algorithm in condor. condor-users forum, Feb 2005.
- [47] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – A distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.
- [48] Taverna. <http://taverna.sourceforge.net/>.
- [49] I. Taylor, M. Shields, and I. Wang. *Resource management for the Triana peer-to-peer services*, pages 451–462. Kluwer Academic Publishers, Norwell, MA, USA, 2004.
- [50] TGFF. <http://ziyang.eecs.umich.edu/dickrp/tgff/>.
- [51] Triana. <http://www.trianacode.org/>.
- [52] Unicore. <http://www.unicore.eu/>.
- [53] M. Wicczorek, R. Prodan, and T. Fahringer. Scheduling of scientific workflows in the askalon grid environment. *SIGMOD Rec.*, 34(3):56–62, 2005.

- [54] Tao Yang and Apostolos Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5:951–967, 1994.
- [55] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34(3):44–49, 2005.
- [56] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. Technical report, Journal of Grid Computing, 2005.
- [57] F. Yuhung. *A Mobile Code Collaboration Framework for Grid Computing*. PhD thesis, Nanayang Technological University, 2008.
- [58] F. Yuhung and W. Cai. *Parallel and Distributed Processing and Applications*, volume 3358/2005 of *Lecture Notes in Computer Science*, chapter MCCF: A Distributed Grid Job Workflow Execution Framework, pages 274–279. Springer, 2005.
- [59] Y. Zhang, C. Koelbel, and K. Kennedy. Relative performance of scheduling algorithms in grid environments. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGRID'07)*, pages 521–528, Washington, DC, USA, 2007. IEEE Computer Society.
- [60] H. Zhou. Scheduling dags on a bounded number of processors. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 1996, August 9-11, 1996, Sunnyvale, California, USA*, volume 2, pages 823–834. CSREA Press, August 1996.