WORKFLOW RESTRUCTURING TECHNIQUES FOR IMPROVING THE PERFORMANCE OF SCIENTIFIC WORKFLOWS EXECUTING IN DISTRIBUTED ENVIRONMENTS

by

Weiwei Chen

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

Jun 2014

Dedication

To my family.

Acknowledgments

First, I would like to thank my family: my parents Xueyi Chen and Cuilan Zhou, for giving birth to me and passing their value on hard work and education to me. My sister Li Chen and my brother Yuwei Chen, who have continuously encouraged me to pursuit my dreams.

I would like to express my sincere gratitude to my advisor Ewa Deelman for the continuous support of my Ph.D. study and research. She provided me with excellent guidance, encouraged me investigate my own ideas, connected me with many experts in different disciplines and corrected my papers, posters, articles, presentations and thesis. I could not have imaged having a better advisor and mentor for my Ph.D. study.

Besides my advisor, I am deeply grateful to my committee members, Bob Lucas, Ann Chervenak, Aiichiro Nakano and Viktor Prasanna, who have all given me invaluable guidance and advice. I took Bob Lucas' Parallel Programming course and he helped me build a whole picture of the parallel and distributed world. I remember my first project at ISI was with Ann Chervenak and Muhammad Ali Amer. They helped me build the very skills for my study and research and I am grateful to them for enlightening me the first glance of research.

I am fortunate to have many good friends and colleagues at ISI. I would like to thank Karan Vahi, Gaurang Mehta, Mats Rynge. They helped me learn many softwares and tools I needed in my research. Their patience, enthusiasm and immense knowledge helped me overcome many problems I have met in practice.

I would like to offer my special thanks to Gideon Juve, who have been a wise counselor and a great source of ideas and discussion. Gideon have continuously offered me his great advice not only on my research and study, but also on my career planning and selection.

I am fortunate to meet Rafael Ferreira da Silva at the last year of my Phd, who provided me a sounded support of my research. And I would like to offer my special appreciation to Rafael for all of his work on the research described in Chapter 5 and Chapter 6. Rafael helped me finalize the algorithms and metrics used for balanced task clustering and fault tolerant clustering.

I thank my other labmates in Pegasus group: Rajiv Mayani, Prasanth Thomas, Fabio Silva, Jens Vöckler and Idafen Santana Pérez for the stimulating discussions and for all the fun we have had in the last five years.

I have greatly benefited from three great European scientists: Rizos Sakellariou, Thomas Fahringer and Radu Prodan. The discussion with them have been very insightful and inspiring. Radu Prodan's work on overhead analysis inspired my work on task clustering. Rizos Sakellariou helped me finalize the algorithms and metrics used for balanced task clustering. Thomas Fahringer worked with me on the chapter of fault tolerant clustering and provided his insightful ideas.

I thank my friends in the networking and AI division: Congxing Cai, Lin Quan, Zi Hu, Xun Fan, Xue Cai, Chengjie Zhang, Xiyue Deng, Hao Shi and Lihang Zhao for all the fun we have had at ISI and their advice on my thesis.

Finally, I would like to thank everyone at ISI, the Viterbi School of Engineering, and the Department of Computer Science, who have helped me reach this point: Larry Godinez, Liszl DeLeon, Ruth Heron, Raphael Bolze, Rubing Duan, Claire Bono and Mei-Hui Su.

Table of Contents

List of Tables

List of Figures

Abstract

Scientific workflows are a means of defining and orchestrating large, complex, multi-stage computations that perform data analysis, simulation, visualization, etc. Scientific workflows often involve a large amount of data transfers and computations and require efficient optimization techniques to reduce the runtime of the overall workflow. Today, with the emergence of large-scale scientific workflows executing in modern distributed environments such as grids and clouds, the optimization of workflow runtime has introduced new challenges that existing optimization methods do not tackle. Traditionally, many existing runtime optimization methods are confined to the task scheduling problem. They do not consider the refinement of workflow structures, system overheads, the occurrence of failures, etc. Refining workflow structures using techniques such as workflow partitioning and task clustering represent a new trend in runtime optimization and can result in significant performance improvement. The runtime improvement of these workflow restructuring methods depends on the ratio of application computation time to the system overheads. Since system overheads in modern distributed systems can be high, the potential benefit of workflow restructuring can be significant.

This thesis argues that workflow restructuring techniques can significantly improve the runtime of scientific workflows executing in modern distributed environments. In particular, we innovate in the area of workflow partitioning and task clustering techniques. Several previous studies also utilize workflow partitioning and task clustering techniques to improve the performance of scientific workflows. However, existing methods are based on the trial-and-error approach and require users' knowledge to tune the workflow performance. For example, many workflow partitioning techniques do not consider constraints on resources used by the workflows such as the data storage. Also, many task clustering methods optimize task granularity at the workflow level without considering data dependencies between tasks. We distinguish our work from other research by modeling a realistic distributed system with overheads and failures and we use real world applications that exhibit load imbalance in their structure and computations.

We investigate the key concern of refining workflow structures and propose a series of innovative workflow partitioning and task clustering methods to improve the runtime performance. Simulation-based and real system-based evaluation verifies the effectiveness of our methods.

Chapter 1

Introduction

In recent years, with the emergence of the fourth paradigm of scientific discovery [38], computational workflows continue to gain in popularity among many science disciplines, including physics [26], astronomy [73], biology [49, 62], chemistry [95], seismology [53] and etc. A **workflow** is a high-level specification of a set of tasks that represent a computational science or engineering processes and the dependencies between the tasks that must be satisfied in order to accomplish a specific goal.

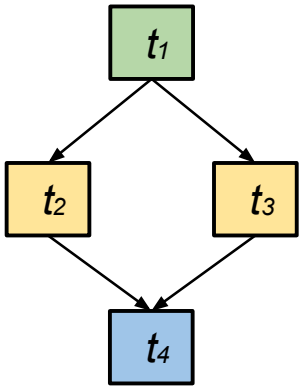

Figure 1.1: A simple DAG with four tasks (t_1, t_2, t_3, t_4). The edges represent data dependencies between tasks.

Traditionally, a workflow is modeled as a Directed Acyclic Graph (DAG). As shown in Figure 2.1, each node in the DAG represents a workflow task, and the edges represent dependencies between the tasks (t) that constrain the order in which the tasks are executed. Each task is a program and a set of parameters that need to be executed. The dependencies typically represent data flow dependencies in the application, where the output files produced by one task are needed as inputs of another task.

Scientific workflows increasingly require tremendous amounts of data processing and workflows with up to a million tasks are not uncommon [11]. For example, the CyberShake workflow [55] composed of more than 800,000 jobs have been executed on the TeraGrid [86]. Among these large-scale, loosely-coupled applications, the majority of the tasks within these applications are often relatively small (from a few seconds to a few minutes in runtime). However, in aggregate they represent a significant amount of computation and data [26]. Most existing systems consider the allocation of computation resources and the scheduling of tasks, but do not effectively take into account system overheads, failure occurrence, or the possibility to restructure the workflow. An **overhead** is defined as the time of performing miscellaneous work other than executing the user's computational activities. Overheads adversely influence runtime of large-scale scientific workflows and cause significant resource underutilization [15].

In order to minimize the impact of system overheads, workflow restructuring techniques such as task clustering [79, 39, 104] and workflow partitioning [46, 37] have been developed to reduce the system overheads and improve their scalability. **Task clustering** [79] is a technique that merges small tasks into larger jobs so that the number of computational activities is reduced and the ratio of application computation to the system overheads is increased. A **job** is a single execution unit in a workflow management system such as Pegasus [24], Askalon [65] and Taverna [88].

Workflow partitioning is another technique that refines workflow structures by dividing a large workflow into several sub-workflows such that the overall number of tasks is reduced and the resource requirements of these sub-workflows can be satisfied in the execution environments. A **sub-workflow** is a workflow and also a job of a higher-level workflow. Sub-workflows are often scheduled to run in different execution sites. The performance of workflow partitioning has been evaluated in [55] and it has shown significant runtime improvement for an astronomy application.

However, these existing methods are using a trial-and-error approach to optimize the workflow structures. For example, Horizontal Clustering (HC) [79] merges multiple tasks that are at the same horizontal level of the workflow, in which the horizontal level of a task is defined

as the longest distance from the entry task of the DAG to this task. The clustering granularity (number of tasks within a cluster) of a clustered job is controlled by the user, who defines either the number of tasks per clustered job (*clusters.size*), or the number of clustered jobs per horizontal level of the workflow (*clusters.num*). The user would try a group of task clustering parameters and pick the ”best” one. Such an approach of tuning task granularity has ignored or underestimated the dynamic characteristics of distributed environments. First of all, such a naïve setting of clustering granularity may cause significant imbalance between runtime and data transfer since it heavily relies on the users' knowledge of the applications and systems. Second, these techniques have ignored the occurrence of task failures, which can counteract the benefit gained from task clustering. The workflow partitioning approach [55] is used to break up a large workflow into several smaller sub-workflows. However, workflow partitioning techniques have also ignored the fact that the execution environment has limited resources to host some large-scale scientific workflows, such as the data storage.

These ad-hoc methods have introduced many challenges for the management of large-scale scientific workflows. In next section, we generalize these drawbacks to the three research challenges.

1.1 Problem Space

In this thesis, we identify new challenges when executing complex scientific applications in distributed environments:

The first challenge deals with the **data management within a workflow** [94, 93, 92]. Because of the distributed nature of computational and storage resources in grids and clouds, and the fact that scientific applications usually need collaborations of scientists from different institutions, it is sometimes hard to perform computations close to the data. Additionally, compute cycles and application data are distributed across multiple resources. Thus the issue of efficient data transfers between different execution sites is becoming critical. Communication-aware scheduling [81, 41] has taken the communication cost into the scheduling/clustering

model and have achieved some significant improvement. The workflow partitioning approach [37, 100, 95, 30] represents the partitioning problem as a global optimization problem and aims to minimize the overall runtime of a graph. However, the complexity of solving such an optimization problem does not scale well. Heuristics [54, 10] are used to select the right parameters and achieve better runtime performance but the approach is not automatic and requires considerable knowledge of the applications.

The second challenge stems from the **imbalance of runtime and data dependency** when merging workflow tasks. Tasks may have diverse runtimes and such diversity may cause significant load imbalance. To address this challenge, researchers have proposed several approaches. Bag-of-Tasks [39, 14, 64] dynamically groups tasks together based on the task characteristics but it assumes tasks are independent, which limits its usage in scientific workflows. Singh [79] and Rynge [55] examine the workflow structures and group tasks together into jobs in a static way for Pegasus Workflow Management Systems [24]. However, this work ignores the computational requirements of tasks and may result in an imbalanced load on the resources. A popular technique in workload studies to address the load balancing challenge is over-decomposition [51]. This method decomposes computational work into medium-grained tasks. Each task is coarse-grained enough to enable efficient execution and to reduce scheduling overheads, while being fine-grained enough to expose higher application-level parallelism than that is offered by the hardware.

The third challenge deals with **fault tolerance**. Existing clustering strategies ignore or underestimate the influence of the occurrence of failures on system behavior, despite the fact that failures are commonplace in large-scale distributed systems. Many researchers [103, 84, 76, 72] have emphasized the importance of fault tolerant design and indicated that the failure rates in modern distributed systems are significant. The major concern has to do with transient failures because they are expected to be more prevalent than permanent failures [103]. For example, denser integration of semiconductor circuits and lower operating voltage levels may increase the likelihood of bit-flips when circuits are bombarded by cosmic rays and other particles [103]. In a faulty execution environment, there are usually three approaches for managing task failures.

First, one can simply retry the entire job when its computation is not successful as in the Pegasus Workflow Management System [24]. However, some of the tasks within the job may have completed successfully and it could be a waste of time and resources to retry all of the tasks. Second, the application process can be periodically check-pointed so that when a failure occurs, the amount of work to be retried is limited. However, the overheads of checkpointing can limit its benefits [103]. Third, tasks can be replicated to different nodes to avoid location-specific failures [102]. However, this approach increases resource cost since it has duplicate tasks.

1.2 Thesis Statement

This thesis states that **workflow restructuring techniques can significantly improve the performance of scientific workflows executing in distributed environments**. We distinguish our work from prior work in the following way: First, we propose data aware workflow partitioning to divide large workflows into sub-workflows that satisfy the data storage limit in the execution environments. Second, we propose a series of balanced task clustering strategies to address the tradeoff between the dependency imbalance and the runtime imbalance. Third, we propose fault tolerant clustering algorithms to automatically adjust the task granularity in a faulty execution environment and thereby improve the overall runtime of scientific workflows.

This thesis uses a wide range of scientific workflows to demonstrate the efficiency and effectiveness of our approaches. We believe that our methods can be used by many domain scientists to optimize their scientific workflows and will inspire new ways for future optimization of runtime performance in scientific workflows.

1.3 Supporting the Thesis Statement

In this section, we substantiate the thesis statement through four specific studies, each gaining new insights and knowledge about the optimization of workflow structures in scientific workflows.

In our first work [15] (Chapter 3), we **extend the existing Directed Acyclic Graph (DAG) model to be overhead-aware** and analyze the relationship between the workflow performance and overheads. Previous research has established models to describe system overheads in distributed systems and has classified them into several categories [69, 70]. In contrast, we investigate the distributions and patterns of different system overheads and discuss how the system environment (system configuration, etc.) influences the distribution of system overheads. Furthermore, we have developed a workflow simulator based on our overhead-aware DAG model and evaluated its effectiveness.

In our second work [17, 18] (Chapter 4), we introduce **data aware workflow partitioning** to refine workflow structures and improve data management within large-scale scientific workflows. Data-intensive workflows require significant amount of storage and computation. For these workflows, we need to use multiple execution sites and consider their available storage. Data aware partitioning aims to reduce the intermediate data transfer in a workflow while satisfying the storage constraints. Heuristics and algorithms are proposed to improve the efficiency of partitioning. Experiment-based evaluation is performed to validate its effectiveness.

In our third work [20, 21] (Chapter 5), we **solve the runtime and data dependency imbalance problem** and introduce a series of balanced clustering methods. We identify the two challenges: runtime imbalance due to the inefficient clustering of independent tasks and dependency imbalance that is related to dependent tasks. What makes this problem even more challenging is that solutions to address these two problems are usually conflicting. For example, balancing runtime may aggravate the dependency imbalance problem, and vice versa. A quantitative measurement of workflow characteristics is required to serve as a criterion to select and balance these solutions. To achieve this goal, we propose a series of imbalance metrics to reflect the internal structure (in terms of runtime and dependency) of workflows.

In our fourth work [16] (Chapter 6), we propose **fault tolerant clustering** techniques that dynamically adjusts the clustering strategy based on the current trend of failures. During the

runtime, this approach uses Maximum Likelihood Estimation to estimate the failure distribution among all the resources. We then dynamically merge tasks into jobs of moderate size and recluster failed jobs to avoid further failures.

Overall, this thesis aims to **improve the quality of task clustering and workflow partitioning methods in large-scale scientific workflows**. We present both real system-based and simulation-based evaluations of our algorithms on a wide range of scientific workflows.

1.4 Research Contributions

The main contribution of this thesis is a framework for task clustering and workflow partitioning in executing scientific workflows. Specially:

1. We developed an overhead-aware workflow model to investigate the performance of task clustering techniques in distributed environments. We present the system overhead characteristics for a wide range of widely used workflows.

2. We have developed workflow partitioning algorithms that use heuristics to divide large-scale workflows into sub-workflows to satisfy resource constraints at execution sites such as the data storage constraint.

3. We have built a statistical model to demonstrate that transient failures can have a significant impact on the runtime of scientific workflows. We have developed a Maximum Likelihood Estimation based parameter estimation approach to integrate both prior knowledge and runtime feedback about task runtime, system overheads and transient failures. We have proposed fault tolerant clustering algorithms to dynamically adjust the task granularity and improve the runtime.

4. We have examined the reasons that cause runtime imbalance and dependency imbalance in task clustering. We have proposed quantitative metrics to evaluate the severity of the two imbalance problems and a series of balanced clustering methods to address the imbalance problems for five widely used scientific workflows.

5. We have developed an innovative workflow simulator called WorkflowSim that includes popular task scheduling and task clustering algorithms. We have built an open source community for the users and developers of WorkflowSim. Furthermore, our balanced algorithms have been implemented in the Pegasus Workflow Management Systems [27].

Chapter 2

Background

In this chapter, we introduce the background knowledge and our observed data about a wide range of scientific workflows used in this thesis. First, we introduce the workflow and system model used in workflow execution. Second, we introduce the classification and distribution of system overheads during the workflow executions. Third, we describe the scientific workflow applications used in this thesis.

2.1 Workflow and System Model

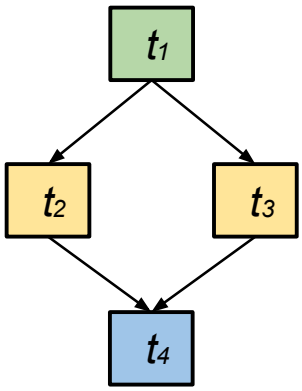

Figure 2.1: A simple DAG with four tasks (t_1, t_2, t_3, t_4). The edges represent data dependencies between tasks.

A workflow is usually modeled as a Directed Acyclic Graph (DAG). As shown in Figure 2.1, each node in the DAG represents a workflow task, and the edges represent dependencies between the tasks (t) that constrain the order in which the tasks are executed. Each task is a program and a set of parameters that need to be executed. The dependencies typically represent data flow

dependencies in the application, where the output files produced by one task are needed as inputs of another task.

Scientific workflows increasingly require tremendous amounts of data processing and workflows with up to a million tasks are not uncommon [11]. For example, the CyberShake workflow [55] composed of more than 800,000 jobs have been executed on the TeraGrid [86]. Among these large-scale, loosely-coupled applications, the majority of the tasks within these applications are often relatively small (from a few seconds to a few minutes in runtime). However, in aggregate they represent a significant amount of computation and data [26]. Most existing systems consider the allocation of computation resources and the scheduling of tasks, but do not effectively take into account system overheads, failure occurrence, or the possibility to restructure the workflow. An **overhead** is defined as the time of performing miscellaneous work other than executing the user's computational activities. Overheads adversely influence runtime of large-scale scientific workflows and cause significant resource underutilization [15].

In order to minimize the impact of system overheads, workflow restructuring techniques such as task clustering [79, 39, 104] and workflow partitioning [46, 37] have been developed to reduce the system overheads and improve their scalability. **Task clustering** [79] is a technique that merges small tasks into larger jobs so that the number of computational activities is reduced and the ratio of application computation to the system overheads is increased. A **job** is a single execution unit in a workflow management system such as Pegasus [24], Askalon [65] and Taverna [88]. Figure 2.2 shows a typical execution environment for scientific workflows. The components of this environment are listed below:

Workflow Mapper generates an executable workflow based on an abstract workflow provided by a user or a workflow composition system.

Workflow Engine executes the jobs defined by the workflow in order of their dependencies. Only free jobs that have all their parent jobs completed are submitted to Job Scheduler.

Job Scheduler and **Local Queue** manage individual workflow jobs and supervise their execution on local and remote resources.

Job Wrapper extracts tasks from clustered jobs and executes them at the worker nodes.

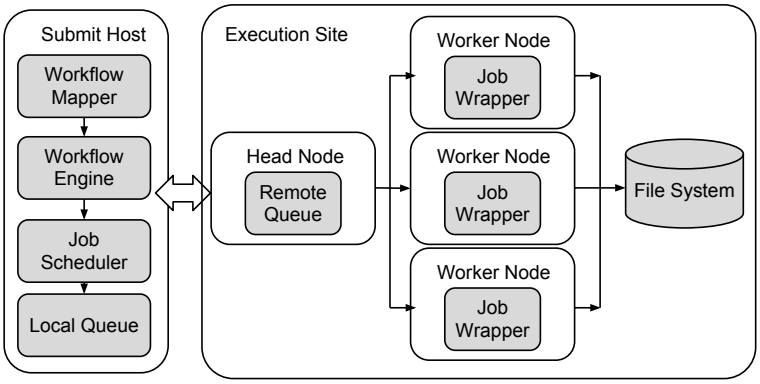

Figure 2.2: System Model

2.2 System Overheads

2.2.1 Overhead Classification

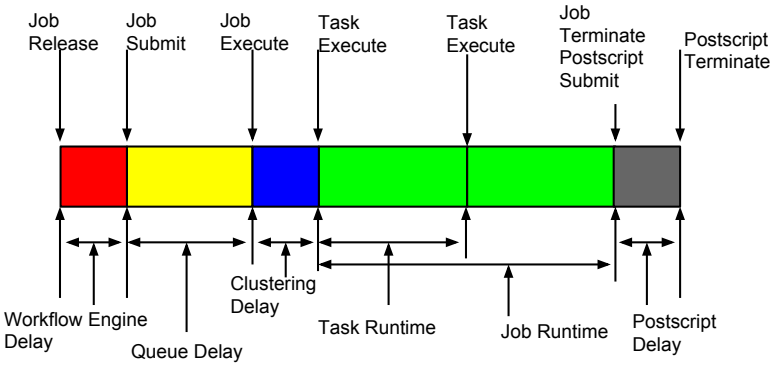

Figure 2.3: Workflow Events

The execution of a job is comprised of a series of events as shown in Figure 2.3 and they are defined as:

1. Job Release is defined as the time when the workflow engine identifies that a job is ready to be submitted (when its parents have successfully completed).

2. Job Submit is defined as the time when the workflow engine submits a job to the local queue.

3. Job Execute is defined as the time when the workflow engine sees a job is being executed.

4. Task Execute is defined as the time when the job wrapper sees a task is being executed.

5. Postscript Start is defined as the time when the workflow engine starts to execute a postscript.

6. Postscript Terminate is defined as the time when the postscript returns a status code (success or failure).

Figure 2.3 shows a typical timeline of overheads and runtime in a job. We do not specify the data transfer delay in this timeline because data transfer is handled by data transfer jobs (stage-in and stage-out jobs).

We have classified workflow overheads into five categories as follows.

1. Workflow Engine Delay measures the time between when the last parent job of a job completes and the time when the job gets submitted to the local queue. The completion time of the last parent job means this job is released to the ready queue and is waiting for resources to be assigned to it. The workflow engine delay reflects the efficiency of a workflow engine (i.e., DAGMan [23]).

2. Queue Delay is defined as the time between the submission of a job by the workflow engine to the local queue and the time the local scheduler sees the job running. This overhead reflects the efficiency of the local workflow scheduler (e.g. Condor [34]) to execute a job and the availability of resources for the execution of this job.

3. Postscript Delay is the time taken to execute a lightweight script under some execution systems after the execution of a job. Postscripts examine the status code of a job after the computational part of this job is done.

4. Data Transfer Delay happens when data is transferred between nodes. It includes three different types of processes: staging data in, cleaning up, and staging data out. Stage-in jobs transfer input files from source sites to execution sites before the computation starts. Cleanup jobs delete intermediate data that is no longer needed by the remainder of the

workflow. Stage-out jobs transfer workflow output data to archiving sites for storage and analysis.

5. Clustering Delay measures the difference between the sum of the actual task runtime and the job runtime seen by the job wrapper. The cause of Clustering Delay is usually because we use a job wrapper in worker nodes to execute a clustered job that requires some delay to extract the list of tasks.

2.2.2 Overhead Distribution

We examined the overhead distributions of a widely used astronomy workflow called Montage [5] that is used to construct large image mosaics of the sky. Montage was run on FutureGrid [33]. The details of FutureGrid will be introduced in Section 3.2. Figure 2.5, 2.4 and 2.6 show the overhead distribution of the Montage workflow run on the FutureGrid. The postscript delay concentrates at 7 seconds, because the postscript is only used to locally check the return status of a job and is not influenced by the remote execution environment. The workflow engine delay tends to have a uniform distribution, which is because the workflow engine spends a constant amount of time to identify that the parent jobs have completed and insert a job that is ready at the end of the local queue. The queue delay has three decreasing peak points at 8, 14, and 22 seconds. We believe this is because the average postscript delay is about 7 seconds (see details in Figure 2.5) and the average runtime is 1 second. The local scheduler spends about 8 seconds finding an available resource and executing a job; if there is no resource idle, it will wait another 8 seconds for the current running jobs to finish, and so on.

Figure 2.7 shows the average value of Clustering Delay of mProjectPP, mDiffFit, and mBackground. It is clear that with the increase of *clusters.num* (the maximum number of jobs per horizontal level), since there are less and less tasks in a clustered job, the Clustering Delay for each job decreases. For simplicity, we use an inverse proportional model in Equation 2.1 to describe this trend of Clustering Delay with *clusters.num*. Intuitively we assume that the average delay per task in a clustered job is constant (n is the number of tasks in a horizontal level). An inverse proportional model can estimate the delay when $clusters.num = i$ directly if we have known the

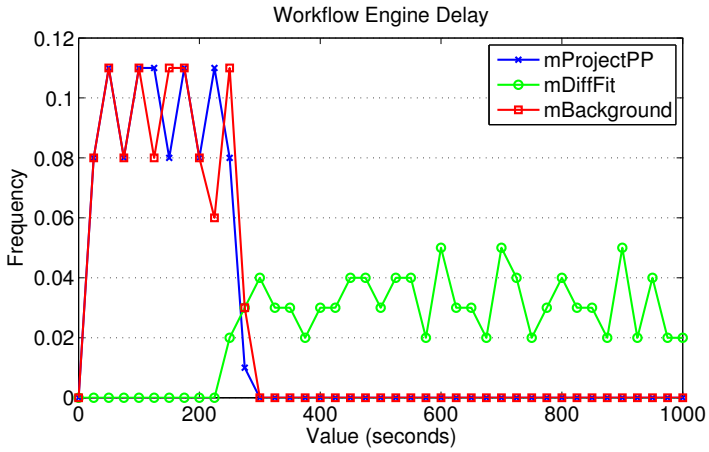

Figure 2.4: Distribution of Workflow Engine Delay in the Montage workflow

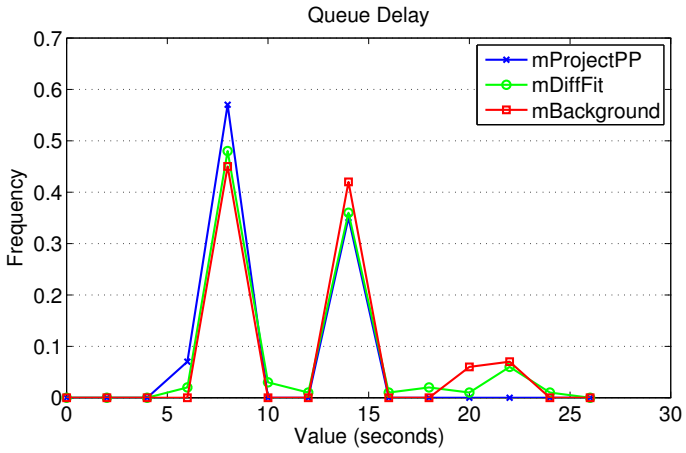

Figure 2.5: Distribution of Queue Delay in the Montage workflow

delay when $clusters.num = j$. Therefore we can predict all the clustering cases as long as we have gathered one clustering case. This feature will be used to evaluate the performance of a workflow simulator in Section 3.2.

$$\frac{ClusteringDelay|_{clusters.num=i}}{ClusteringDelay|_{clusters.num=j}} = \frac{n/i}{n/j} = \frac{j}{i} \tag{2.1}$$

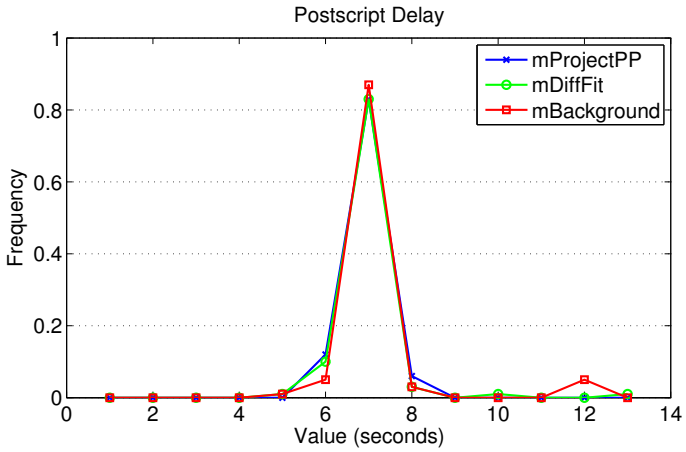

Figure 2.6: Distribution of Postscript Delay in the Montage workflow

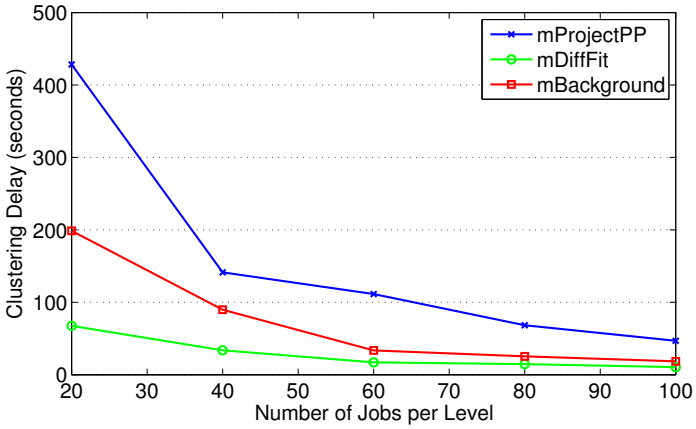

Figure 2.7: Clustering Delay of mProjectPP, mDiffFit, and mBackground

2.3 Scientific Workflow Applications

Five widely used scientific workflow applications are used in this thesis: LIGO Inspiral analysis [48], Montage [5], CyberShake [35], Epigenomics [91], and SIPHT [80]. In this section, we describe each workflow application and present their main characteristics and structures. For simplicity, system overheads are not displayed.

LIGO Laser Interferometer Gravitational Wave Observatory (LIGO) [48] workflows are used to search for gravitational wave signatures in data collected by large-scale interferometers. The

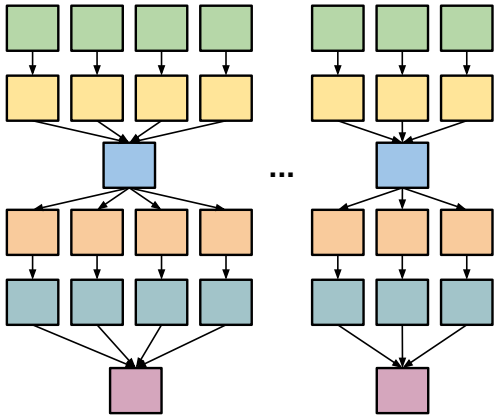

Figure 2.8: A simplified visualization of the LIGO Inspiral workflow.

observatories' mission is to detect and measure gravitational waves predicted by general relativity (Einstein's theory of gravity), in which gravity is described as due to the curvature of the fabric of time and space. The LIGO Inspiral workflow is a data-intensive workflow. Figure 2.8 shows a simplified version of this workflow. The LIGO Inspiral workflow is separated into multiple groups of interconnected tasks, which we call branches in this thesis. However, each branch may have a different number of pipelines as shown in Figure 2.8.

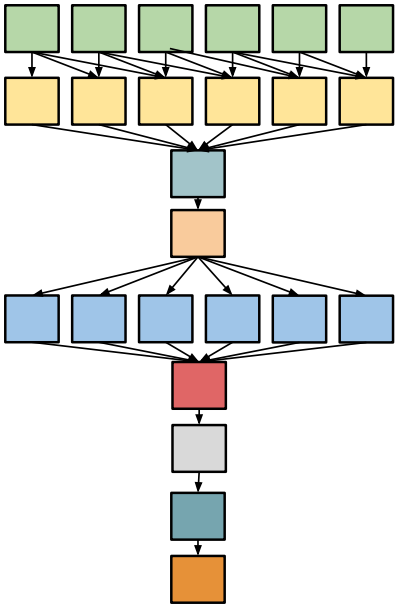

Figure 2.9: A simplified visualization of the Montage workflow.

Montage Montage [5] is an astronomy application that is used to construct large image mosaics of the sky. Input images are reprojected onto a sphere and overlap is calculated for each input image. The application reprojects input images to the correct orientation while keeping background emission level constant in all images. The images are added by rectifying them to a common flux scale and background level. Finally the reprojected images are co-added into a final mosaic. The resulting mosaic image can provide a much deeper and detailed understanding of the portion of the sky in question. Figure 2.9 illustrates a small Montage workflow. The size of the workflow depends on the number of images used in constructing the desired mosaic of the sky.

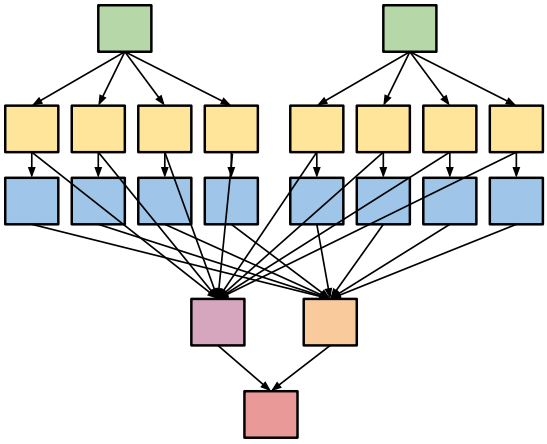

Figure 2.10: A simplified visualization of the CyberShake workflow.

Cybershake CyberShake [35] is a seismology application that calculates Probabilistic Seismic Hazard curves for geographic sites in the Southern California region. It identifies all ruptures within 200km of the site of interest and converts rupture definition into multiple rupture variations with differing hypocenter locations and slip distributions. It then calculates synthetic seismograms for each rupture variance, and peak intensity measures are then extracted from these synthetics and combined with the original rupture probabilities to produce probabilistic seismic hazard curves for the site. Figure 2.10 shows an illustration of the Cybershake workflow.

Epigenomics The Epigenomics workflow [91] is a CPU-intensive application. Initial data are acquired from the Illumina-Solexa Genetic Analyzer in the form of DNA sequence lanes. Each

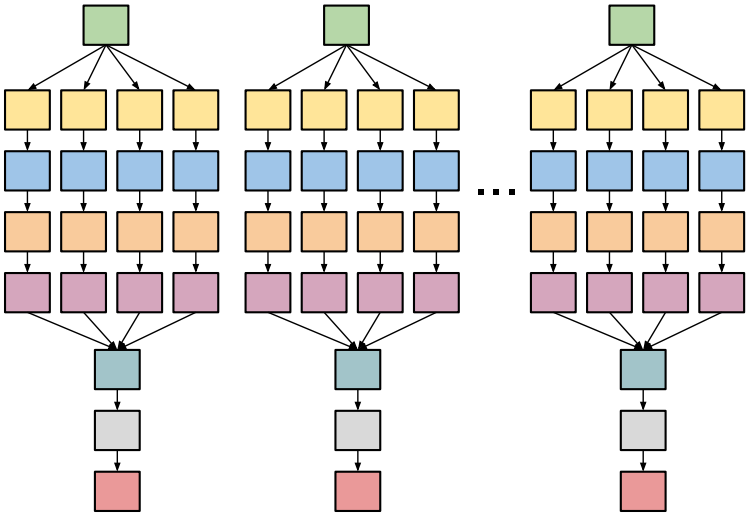

Figure 2.11: A simplified visualization of the Epigenomics workflow with multiple branches.

Solexa machine can generate multiple lanes of DNA sequences. Then the workflow maps DNA sequences to the correct locations in a reference Genome. This generates a map that displays the sequence density showing how many times a certain sequence expresses itself on a particular location on the reference genome. A simplified structure of Epigenomics is shown in Figure 2.11.

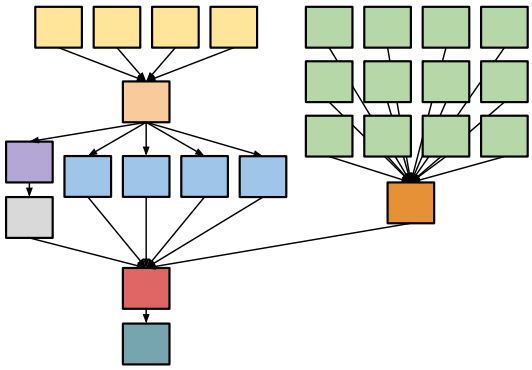

Figure 2.12: A simplified visualization of the SIPHT workflow.

SIPHT The SIPHT workflow [80] conducts a wide search for small untranslated RNAs (sRNAs) that regulates several processes such as secretion or virulence in bacteria. The kingdom-wide prediction and annotation of sRNA encoding genes involves a variety of individual programs

that are executed in the proper order using Pegasus [24]. These involve the prediction of ρ-independent transcriptional terminators, BLAST (Basic Local Alignment Search Tools [4]) comparisons of the inter genetic regions of different replicons and the annotations of any sRNAs that are found. A simplified structure of the SIPHT workflow is shown in Figure 2.12.

Workflow	Number of Tasks	Average Data Size	Average Task Runtime
LIGO	800	5 MB	228s
Montage	300	3 MB	11s
CyberShake	700	148 MB	23s
Epigenomics	165	355 MB	2952s
SIPHT	1000	360 KB	180s

Table 2.1: Summary of the scientific workflows characteristics.

Table 2.1 shows the summary of the main **workflow characteristics**: number of tasks, average data size, and average task runtimes for the five workflows.

2.4 Summary

In this chapter, we have introduced the background knowledge and our observed data about a wide range of scientific workflows, including a widely used workflow DAG model and the system model. In the next chapter, we are going to introduce the overhead-aware workflow model that provides better abstraction of scientific workflows executing in distributed environments.

Chapter 3

Overhead-aware Workflow Model

In this chapter, we first introduce how we extend the existing DAG model to be overhead-aware. Second, we use our proposed model to describe the process of task clustering. Third, we introduce the experiment environments used in this thesis including FutureGrid, a distributed platform and a workflow simulator as an example of the importance of an overhead model when simulating workflow execution. The simulation of a widely used workflow verifies the necessity of taking overheads into consideration.

3.1 Overhead-aware DAG Model

A DAG models the computational activities and data dependencies within a workflow and it fits with most workflow management systems such as Pegasus [24] and DAGMan [44]. However, the preparation and execution of a scientific workflow in distributed environments often involve multiple components and the system overheads within and between these components cannot be ignored. An overhead is defined as the time of performing miscellaneous work other than executing computational activities. To address this challenge, in this thesis, we extend the existing Directed Acyclic Graph (DAG) model to be overhead-aware (o-DAG), in which an overhead is also a node in the DAG and the control dependencies are added as directed edges. We utilize o-DAG to provide a systematic analysis of the performance of workflow optimization methods and provide a series of novel optimization methods to further improve the overall workflow performance.

Fig 3.1 shows how we augment a DAG to be an o-DAG with the capability to represent scheduling overheads (s) such as workflow engine delay, queue delay, and postscript delay. The

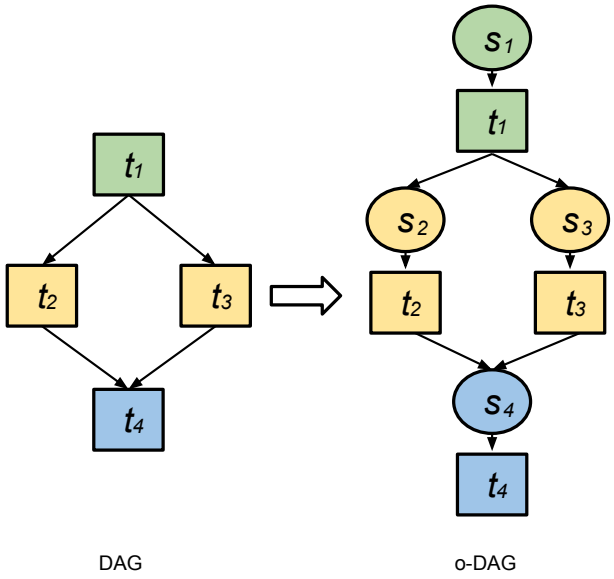

Figure 3.1: Extending DAG to o-DAG

classification of these system overheads is based on the model of a typical workflow management system shown in Fig 2.2 and will be introduced in Section 2.2.

3.1.1 Modeling Task Clustering

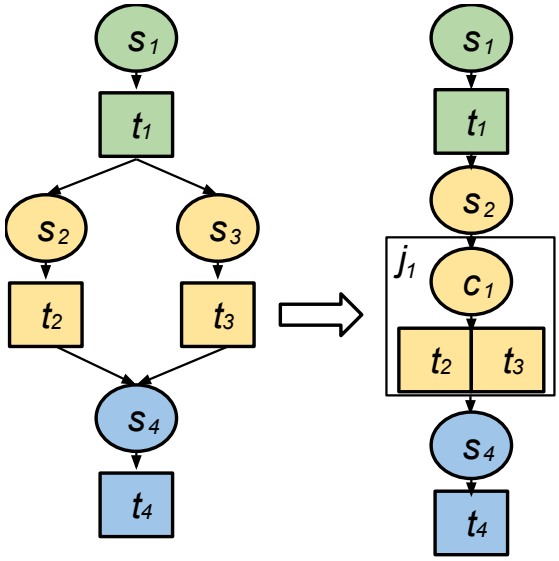

Figure 3.2: An example of task clustering (horizontal clustering)

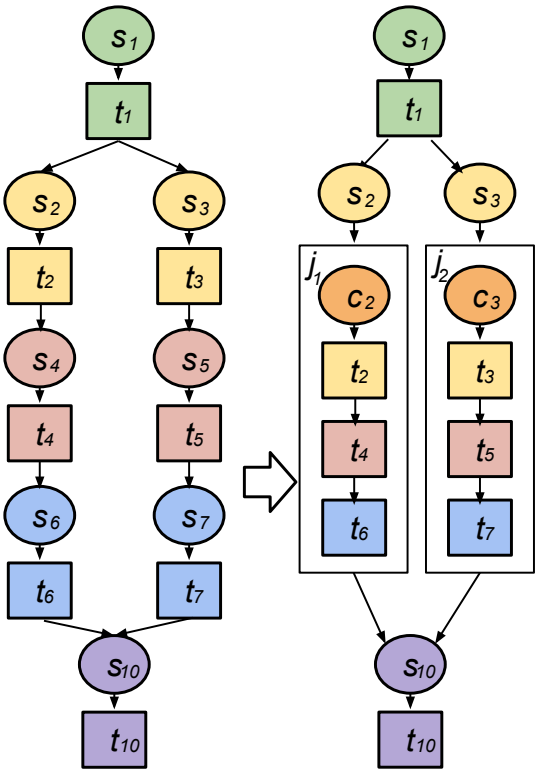

Figure 3.3: An example of vertical clustering.

With an o-DAG model, we can explicitly express the process of task clustering. In this thesis, we address task clustering horizontally and vertically. **Horizontal Clustering** (HC) merges multiple tasks that are at the same horizontal level of the workflow, in which the horizontal level of a task is defined as the longest distance from the entry task of the DAG to this task. **Vertical Clustering** (VC) merges tasks within a pipeline of the workflow. Tasks at the same pipeline share a single-parent-single-child relationship, which means a task t_a is the unique parent of a task t_b, which is the unique child of t_a.

Figure 3.2 shows a simple example of how to perform HC, in which two tasks t_2 and t_3, without a data dependency between them, are merged into a clustered job j_1. A job is a single execution unit composed by one or multiple task(s). Job wrappers are commonly used to execute clustered jobs, but they add an overhead denoted by the clustering delay c. The clustering delay measures the difference between the sum of the actual task runtimes and the job runtime seen by

the job scheduler. After horizontal clustering, t_2 and t_3 in j_1 can be executed in sequence or in parallel, if parallelism in one compute node is supported. In this work, we consider sequential executions only. Given a single resource, the overall runtime for the workflow in Figure 3.2 (left) is $runtime_l = \sum_{i=1}^{4}(s_i + t_i)$, and the overall runtime for the clustered workflow in Figure 3.2 (right) is $runtime_r = s_1 + t_1 + s_2 + c_1 + t_2 + t_3 + s_4 + t_4$. $runtime_l > runtime_r$ as long as $c_1 < s_3$, which is often the case in many distributed systems since the clustering delay within a single execution node is usually shorter than the scheduling overhead across different execution nodes.

Figure 3.3 illustrates an example of vertical clustering, in which tasks t_2, t_4, and t_6 are merged into j_1, while tasks t_3, t_5, and t_7 are merged into j_2. Similarly, clustering delays c_2 and c_3 are added to j_1 and j_2 respectively, but system overheads s_4, s_5, s_6, and s_7 are removed.

3.2 Evaluation Methodologies

In this section, we introduce the experiment environments to evaluate our approaches in this thesis. To evaluate our techniques in Chapter 5 and Chapter 6, we have developed WorkflowSim, a trace based workflow simulator extended from CloudSim [9]. Below we introduce the details of WorkflowSim and we verify the effectiveness of WorkflowSim.

3.2.1 Overview of WorkflowSim

We have developed WorkflowSim as an open source workflow simulator that extends CloudSim [9] by providing a workflow level support of simulation. It models workflows with an o-DAG model with support of an elaborate model of node failures, a model of delays occurring in the various levels of the WMS stack [15], the implementations of several most popular dynamic and static workflow schedulers (e.g., HEFT[90] and MinMin[7]) and task clustering algorithms (e.g., runtime-based algorithms [21], data-oriented algorithms [21] and fault tolerant clustering algorithms [16]). Parameters are imported from traces of workflow executions that were run on FutureGrid [33].

In distributed systems, workflows may experience different types of overheads. Since the causes of overheads differ, the overheads have diverse distributions and behaviors. For example, the time to run a post-script that checks the return status of a computation is usually a constant. However, queue delays incurred while tasks are waiting in a batch scheduling systems can vary widely. By classifying these workflow overheads in different layers and system components, our simulator can offer a more accurate result than simulators that do not include overheads in their system models.

Whats more, many researchers [103, 84, 76, 72, 63, 56] have emphasized the importance of fault tolerant design and concluded that the failure rates in modern distributed systems should not be ignored. A simulation with support of random failures with different distributions is implemented in WorkflowSim to promote such studies.

Finally, progress in workflow research also requires a general-purpose framework that can support widely accepted features of workflows and optimization techniques. Existing simulators such as CloudSim/GridSim [9] fail to provide fine granularity simulations of workflows. For example, they lack the support of task clustering. The simulation of task clustering requires two layers of execution model, on both task and job levels. It also requires a workflow-clustering engine that launches algorithms and heuristics to cluster tasks. Other techniques such as workflow partitioning and task retry are also ignored in these simulators. These features have been implemented in WorkflowSim.

To the best of our knowledge, none of the current distributed system simulators support these rich-features and techniques. Below, we introduce our early work on simulating scientific workflows satisfying these requirements. We evaluate the performance of WorkflowSim with an example of task clustering.

3.2.2 Components and Functionalities

Deelman et. al. [25] have provided a survey of popular workflow systems for scientific applications and have classified their components into four categories: composition, mapping,

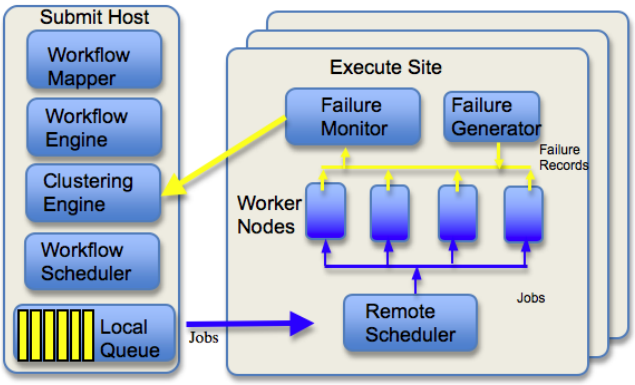

Figure 3.4: Overview of WorkflowSim.

execution, and provenance. Based on this survey, we identified the mandatory functionalities/components and designed the layers in our WorkflowSim. In our design as shown in Figure 3.4, we add multiple layers on top of the existing task scheduling layer of CloudSim, which include Workflow Mapper, Workflow Engine, Clustering Engine, Workflow Scheduler, Failure Generator and Failure Monitor etc.

1. Workflow Mapper is used to import DAG files formatted in XML (called DAX in WorkflowSim) and other metadata information such as file size from Workflow Generator [96]. Workflow Mapper creates a list of tasks and assigns these tasks to an execution site. A task is a program/activity that a user would like to execute.

2. Workflow Engine manages tasks based on their dependencies between tasks to assure that a task may only be released when all of its parent tasks have completed successfully. Workflow Engine will only release free tasks to Clustering Engine.

3. Clustering Engine merges tasks into jobs such that the scheduling overhead is reduced. A job is an atomic unit seen by the execution system, which contains multiple tasks to be executed in sequence or in parallel if applicable. Different to the clustering engine in Pegasus WMS, Clustering Engine in WorkflowSim also performs task reclustering in a faulty execution environment. The details will be introduced in Chapter 6.

4. Workflow Scheduler is used to match jobs to a worker node based on the criteria selected by users. WorkflowSim relies on CloudSim to provide an accurate and reliable job-level execution model, such as time-shared model and space-shared model. However, WorkflowSim has introduced different layers of system overheads and failures, which improves the accuracy of simulation.

5. Failure Generator is introduced to inject task/job failures at each execution site during the simulation. After the execution of each job, Failure Generator randomly generates task/job failures based on the distribution that a user has specified.

6. Failure Monitor collects failure records (e.g., resource id, job id, task id) and returns these records to Clustering Engine to adjust the clustering strategies dynamically.

3.2.3 Results and Validation

We use task clustering as an example to illustrate the necessity of introducing system overheads into workflow simulation. The goal was to compare the simulated overall runtime of workflows in case the information of job runtime and system overheads are known and extracted from prior traces. In this example, we collected real traces generated by the Pegasus Workflow Management System while executing workflows on FutureGrid [33]. We built an execution site with 20 worker nodes and we executed the Montage workflow five times in every single configuration of *clusters.num*, which is the maximum number of clustered jobs in a horizontal level. These five traces of workflow execution with the same *clusters.num* is a training set or a validation set. We ran the Montage workflow with a size of 8-degree squares of sky. The workflow has 10,422 tasks and 57GB of overall data. We tried different *clusters.num* from 20 to 100, leaving us 5 groups of data sets with each group having 5 workflow traces. First of all, we adopt a simple approach that selects a training set to train WorkflowSim and then use the same training set as validation set to compare the predicted overall runtime and the real overall runtime in the traces.

We define accuracy in this section as the ratio between the predicted overall runtime and the real overall runtime:

$$Accuracy = \frac{Predicted\ Overall\ Runtime}{Real\ Overall\ Runtime} \tag{3.1}$$

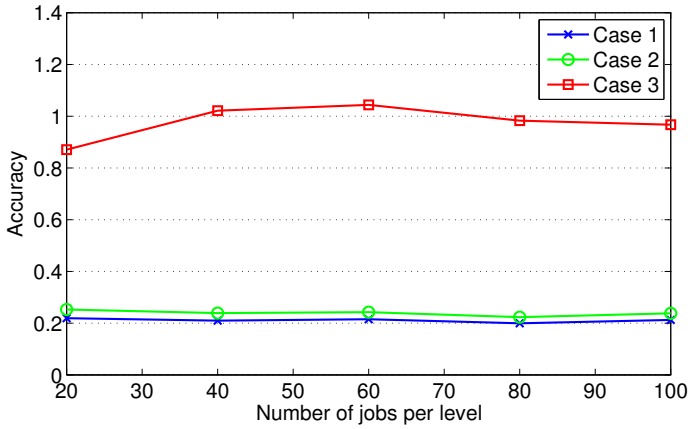

Figure 3.5: Performance of WorkflowSim with different support levels

To train WorkflowSim, from the traces of workflow execution (training sets), we extracted information about job runtime and overheads, such as average/distribution and, for example, whether it has a cyclic increase. We then added these parameters into the generation of system overheads and simulated them as close as possible to the real cases.

To present an explicit comparison, we simulated the cases using WorkflowSim that has no consideration of workflow dependencies or overheads (Case 1), WorkflowSim with Workflow Engine that has considered the influence of dependencies but ignored system overheads (Case 2), and WorkflowSim, that has covered both aspects (Case 3). Intuitively, we expect that the order of the accuracy of them should be Case 3 > Case 2 > Case 1.

Fig 3.5 shows that the performance of WorkflowSim with different support levels is consistent to our expectation. The accuracy of Case 3 is quite close to but not equal to 1.0 in most points. The reason is that to simulate workflows, WorkflowSim has to simplify models with a few parameters, such as the average value and the distribution type. It is not efficient to recur every overhead as is present in the real traces. It is also impossible to do so since the traces within

the same training set may have much variance in each run. Fig 3.5 also shows that the accuracy of both Case 1 and Case 2 are much lower than that of Case 3. The reason is that it ignores both dependencies and multiple layers of overheads. By ignoring data dependencies, it releases tasks that are not supposed to run since their parents have not completed (a real workflow system should never do that) and thereby reducing the overall runtime. At the same time, it executes jobs/tasks irrespective of the actual overheads, which further reduces the simulated overall runtime. In Case 2, with the help of Workflow Engine, WorkflowSim is able to control the release of tasks and thereby the simulated overall runtime is closer to the real traces. However, since it has ignored most overheads, jobs are completed and returned earlier than that in real traces. The low accuracy of Case 1 and Case 2 confirms the necessity of introducing overhead design into our simulator.

3.3 Summary

In this chapter, we have introduced an overhead-aware DAG model that we have proposed and the evaluation methodologies including a workflow simulator that we have developed. In next chapter, we will introduce the first proposed approach called Workflow Partitioning to address the optimization of workflows with resource constraints.

Chapter 4

Data Aware Workflow Partitioning

In this chapter, we introduce our work on data aware workflow partitioning that divides large-scale workflows into several sub-workflows that are fit for execution within a single execution site. A sub-workflow is a workflow and also a job of a higher-level workflow. Three widely used workflows have been used to evaluate the effectiveness of out methods and the experiments show an runtime improvement of up to 48.1%.

4.1 Motivation

Data movement between tasks in scientific workflows has received less attention compared to task execution. Often the staging of data between tasks is ignored compared to task execution, which is not true in many cases, especially in data-intensive applications. In this chapter, we take the data transfer into consideration and propose to partition large workflows into several sub-workflows where each sub-workflows can be executed within one execution site.

The motivation behind workflow partitioning starts from a common scenario where a researcher at a research institution typically has access to several research clusters, each of which may consist of a small number of nodes. The nodes in one cluster may be very different from those in another cluster in terms of file system, execution environment, and security systems. For example, we have access to FutureGrid [33], Teragrid/XSEDE [86] and Amazon EC2 [1] but each cluster imposes a limit on the resources, such as the maximum number of nodes a user can allocate at one time or the maximum storage. If these isolated clusters can work together, they collectively become more powerful.

For example, FutureGrid is built out of a number of clusters of different types and sizes that are interconnected with up to a 10GB Ethernet between its sites. In our experiments of workflow

partitioning, we have used five execution sites including Indiana University (IU), University of Chicago (UC), San Diego Supercomputing Center (SDSC), Texas Advanced Computing Center (TACC), and University of Florida (UF). The FutureGrid policy allows us to launch 20 virtual machines in each site. Table 4.1 shows the overview of these clusters.

Table 4.1: Overview of the Clusters

Site	Name	Nodes	CPUs	Cores	RAM(GB)	Storage (TB)
IU	india	128	256	1024	3072	335
UC	hotel	84	168	672	2016	120
SDSC	sierra	84	168	672	2688	96
UF	foxtrot	32	64	256	768	0
TACC	alamo	96	192	768	1152	30

Additionally, the input dataset could be very large and widely distributed across multiple clusters. Data-intensive workflows require significant amount of storage and computation and therefore the storage system becomes a bottleneck. For these workflows, we need to use multiple execution sites and consider their available storage. For example, the entire CyberShake earthquake science workflow has 16,000 sub-workflows and each sub-workflow has more than 24,000 individual jobs and requires 58 GB of data. In this chapter, we assume we have Condor installed at the execution sites. A Condor pool [44] can be either a physical cluster or a virtual cluster.

The first benefit of workflow partitioning is that this approach reduces the complexity of workflow mapping. For example, the entire CyberShake workflow has more than 3.8×10^8 tasks, which is a significant load for workflow management tools to maintain or schedule. In contrast, each sub-workflow has 24,000 tasks, which is acceptable for workflow management tools. What is more, workflow partitioning provides a fine granularity adjustment of workflow activities so that each sub-workflow can be adequate for one execution site. In the end, workflow partitioning allows us to migrate or retry sub-workflows efficiently. The overall workflow can be partitioned into sub-workflows and each sub-workflow can to be executed in different execution environments such as a hybrid platform of Condor/DAGMan [23] and MPI/DAGMan [55])

while the traditional task clustering technique requires all the tasks can be executed in the same execution environment.

4.2 Related Work

Workflow Partitioning. Because of the limited resources at an execution site, scientists execute scientific workflows [6, 30] in distributed large-scale computational environments such as multi-cluster grids, that is, grids comprising multiple independent execution sites. Topcuoglu [90] presented a classification of widely used task scheduling approaches. Such scheduling solutions, however, cannot be applied directly to multi-cluster grids. First, the data transfer delay between multiple execution sites is more significant than that within an execution site and thus it should be considered. Second, they do not consider the dynamic resource availability in grids, which also makes accurate predictions of computation and communication costs difficult. Sonmez [82] extended the traditional scheduling problem to multiple workflows on multi-cluster grids and presented a performance of a wide range of dynamic workflow scheduling policies in multi-cluster grids. Duan [30] and Wieczorek [95] have discussed the scheduling and the partitioning of scientific workflows in dynamic grids in the presence of a broad set of unpredictable overheads and possible failures. Duan [30] then developed a distributed service-oriented Enactment Engine with a master-slave architecture for de-centralized coordination of scientific workflows. Kumar [46] proposed the use of graph partitioning to divide the resources of a distributed system, but not the workflow itself, which means the resources are provisioned from different execution sites but the workflows are not partitioned at all. Dong [28] and Kalayci [44] have discussed the use of graph partitioning algorithms for workflows according to resource requirements of the workflow tasks and the availability of selected clusters. Our work focuses on the workflow partitioning problem with resource constraints, in particular, the data storage constraint. Compared to Dong [28] and Kalayci [44], we extend their work to estimate the overall runtime of sub-workflows and then schedule these sub-workflows based on the estimates.

Data Placement techniques try to strategically manage placement of data before or during the execution of a workflow. Kosar et al. [45] presented Stork, a scheduler for data placement activities on grids and proposed to make data placement activities as first class citizens in the Grid. In Stork, data placement is a job and is decoupled from computational jobs. Amer et al. [2] studied the relationship between data placement services and workflow management systems for data-intensive applications. They proposed an asynchronous mode of data placement in which data placement operations are performed as data sets become available and according to the policies of the virtual organization and not according to the directives of the workflow management system (WMS). The WMS can however assist the placement services with the staging of data based on information collected during task executions and data transfers. Shankar [78] presented an architecture for Condor in which the input, output and executable files of jobs are cached on the local disks of the machines in a cluster. Caching can reduce the amount of pipelines and batch I/O that is transferred across the network. This in turn significantly reduces the response time for workflows with data-intensive workloads. In contrast, we mainly focus on the workflow partitioning and task clustering problem but our work can be extended to consider the proposed data placement strategies.

Data Throttling. Park et al. [66] limits the amount of parallel data transfer to avoid overloading supporting services such as data servers, which is called data throttling. Throttling is especially useful for unbalanced workflows in which one task might be idle while waiting for data to arrive. However, as discussed in [66], data throttling has an impact on the overall workflow performance depending on the ratio between computational and data transfer tasks. Therefore, performance analysis is necessary after the profiling of data transfers so that the relationship between computation and data transfers can be identified more explicitly. Rodriguez [71] proposed an automated and trace-based workflow structural analysis method for DAGs. File transfers are completed as fast as the network bandwidth allows, and once transferred, the files are buffered/stored at their destination. To improve the use of network bandwidth and buffer/storage within a workflow, they adjusted the speeds of some data transfers and assured that tasks have all their input data arriving at the same time. Compared to our work, data throttling has a limit

in performance gain by the amount of data transfer that can be reduced, while our partitioning approach can improve the overall workflow runtime and resource usage.

4.3 Approach

To efficiently partition workflows, we propose a three-phase scheduling approach integrated with the Pegasus Workflow Management System to partition, estimate, and schedule workflows onto distributed resources. Our contribution includes three heuristics to partition workflows respecting storage constraints and internal job parallelism. We utilize three methods to estimate and compare runtime of sub-workflows and then we schedule them based on two commonly used algorithms (HEFT[90] and MinMin[7]).

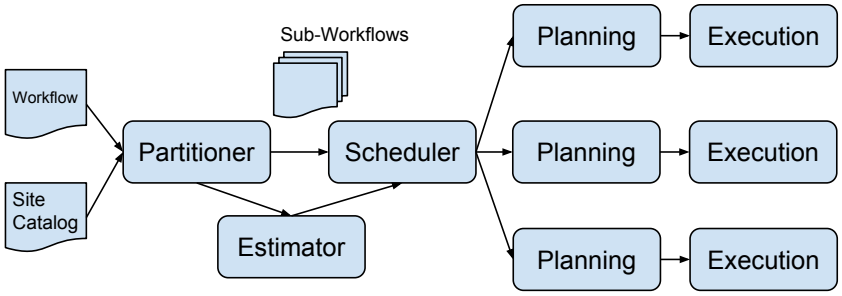

Figure 4.1: The steps to partition and schedule a workflow

Our approach (see Figure 4.1) has three phases: partition, estimate and schedule. The partitioner takes the original workflow and site catalog as input, and outputs various sub-workflows that respect the storage constraints, which means that the data requirements of a sub-workflow are within the data storage limit of a site. The site catalog provides information about the available resources. The estimator provides the runtime estimation of the sub-workflows and supports three estimation methods. The scheduler maps these sub-workflows to resources considering storage requirement and runtime estimation. The scheduler supports two commonly used algorithms. We first find a valid mapping of sub-workflows satisfying storage constraints. Then we

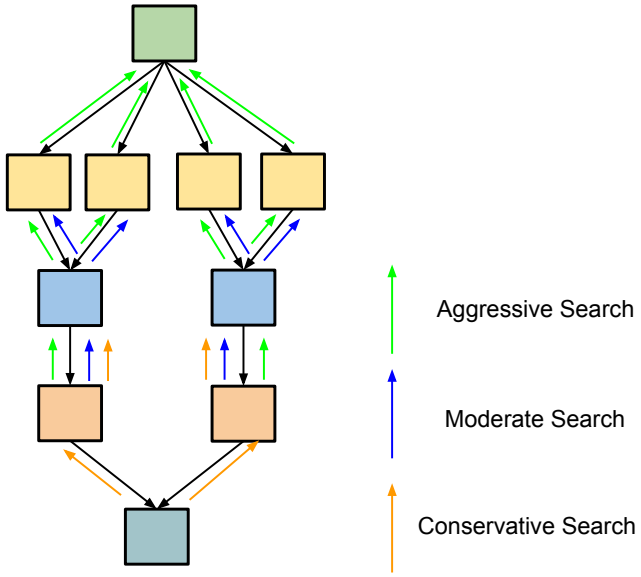

Figure 4.2: Three Steps of Search

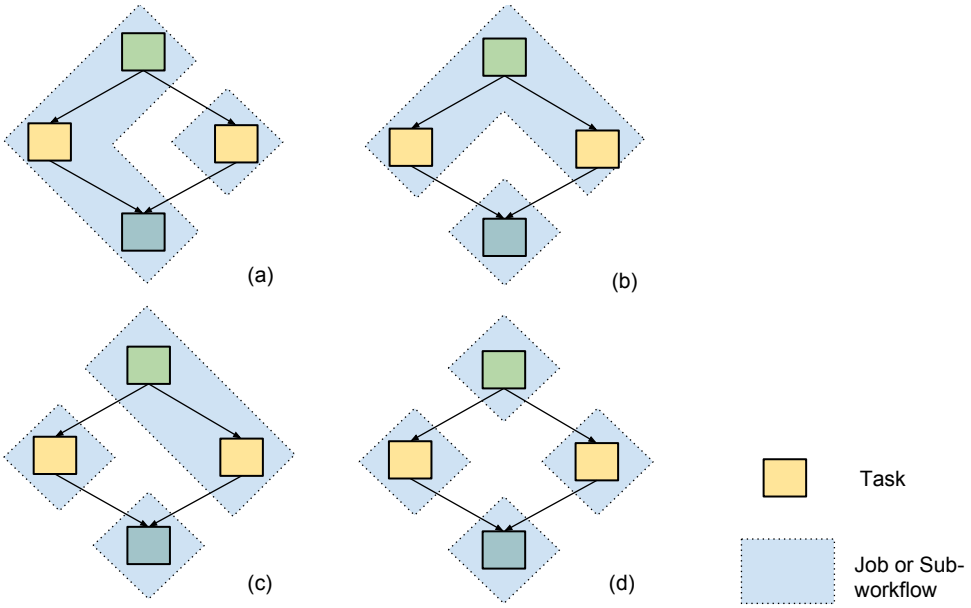

Figure 4.3: Four Partitioning Methods

optimize performance based on these generated sub-workflows and schedule them to appropriate execution sites if runtime information for individual jobs is already known. If not, a static scheduler maps them to resources merely based on storage requirements.

The major challenge in partitioning workflows is to avoid cross dependency, which is a chain of dependencies that forms a cycle in graph (in this case cycles between sub-workflows). With cross dependencies, workflows are not able to proceed since they form a deadlock loop. For a simple workflow depicted in Figure 4.3, we show the result of four different partitioning. Partitioning (a) does not work in practice since it has a deadlock loop. Partitioning (c) is valid but not efficient compared to Partitioning (b) or (d) that have more parallelism.

Usually jobs that have parent-child relationships share a lot of data since they have data dependencies. Its reasonable to schedule such jobs into the same partition to avoid extra data transfer and also to reduce the overall runtime. Thus, we propose Heuristic I to find a group of parents and children. Our heuristic only checks three particular types of nodes: the fan-out job, the fan-in job, and the parents of the fan-in job and searches for the potential candidate jobs that have parent-child relationships between them. The check operation means checking whether one particular job and its potential candidate jobs can be added to a sub-workflow while respecting storage constraints. Thus, our algorithm reduces the time complexity of check operations by n folds, while n is the average depth of the fan-in-fan-out structure. The check operation takes more time than the search operation since the calculation of data usage needs to check all the data allocated to a site and see if there is data overlap. Similar to [90], the algorithm starts from the sink job and proceeds upward.

To search for the potential candidate jobs that have parent-child relationships, the partitioner tries three steps of searches. For a fan-in job, it first checks if it is possible to add the whole fan structure into the sub-workflow (aggressive search). If not, similar to Figure 4.3(d), a cut is issued between this fan-in job and its parents to avoid cross dependencies and increase parallelism. Then a moderate search is performed on its parent jobs, which includes all of its predecessors until the search reaches a fan-out job. If the partition is still too large, a conservative search is performed, which includes all of its predecessors until the search reaches a fan-in job or a fan-out job. Figure 4.2 depicts an example of three steps of search while the workflow in it has an average depth of 4. Pseudo-code of Heuristic I is depicted in Algorithm 1 .

We propose two other heuristics to solve the problem of cross dependency. The motivation for Heuristic II is that Partitioning (c) in Figure 4.3 is able to solve the problem. The motivation for Heuristic III is an observation that partitioning a fan structure into multiple horizontal levels is able to solve the problem. Heuristic II adds a job to a sub-workflow if all of its unscheduled children can be added to that sub-workflow without causing cross dependencies or exceed the storage constraint. Heuristic III adds a job to a sub-workflow if two conditions are met:

1. For a job with multiple children, each child has already been scheduled.

2. After adding this job to the sub-workflow, the data size does not exceed the storage constraint.

To optimize the workflow performance, runtime estimation for sub-workflows is required assuming runtime information for each job is already known. We provide three methods.

1. Critical Path is defined as the longest depth of the sub-workflow weighted by the runtime of each job.

2. Average CPU Time is the quotient of cumulative CPU time of all jobs divided by the number of available resources (it is the number of Condor slots in our experiments).

3. The HEFT estimator uses the calculated earliest finish time of the last sink job as makespan of sub-workflows assuming that we use HEFT to schedule sub-workflows.

The scheduler selects appropriate resources for the sub-workflows satisfying the storage constraints and optimizes the runtime performance. Since the partitioning step has already guaranteed that there is a valid mapping, this step is called re-ordering or post-scheduling. We select HEFT[90] and MinMin[7], which represent global and local optimizations respectively. There are two differences compared to their original versions. First, the data transfer cost within a sub-workflow is ignored since we use a shared file system in our experiments. Second, the data constraints must be satisfied for each sub-workflow. The scheduler selects an optimal set of resources in terms of available Condor slots since it is the major factor influencing the performance. This work can be easily extended to considering more factors. Although some more

Algorithm 1 Workflow Partitioning algorithm

Require: G: workflow; $SL[index]$: site list, which stores all information about a compute site
Ensure: Create a subworkflow list SWL that does not exceed storage constraints

```
1:  procedure ParWorkflow(G, SL)
2:      index ← 0
3:      Q ← new Queue()
4:      Add the sink job of G to Q
5:      S ← new subworkflow()
6:      while Q is not empty do
7:          j ← the last job in Q
8:          Aggressive-Search(j)                                  ▷ for fan-in job
9:          C ← the list of potential candidate jobs to be added to S in SL[index]
10:         P ← the list of parents of all candidates
11:         D ← the data size in SL[index] with C
12:         if D > storage constraint of SL[index] then
13:             Less-Aggressive-Search(j), update C, P, D
14:             if D > storage constraint of SL[index] then
15:                 Conservative-Search(j), update C, P, D
16:             end if
17:         end if
18:         ...                                                   ▷ for other jobs
19:         if S causes cross dependency in SL[index] then
20:             S = new subworkflow()
21:         end if
22:         Add all jobs in C to S
23:         Add all jobs in P to the head of Q
24:         Add S to SWL[index]
25:         if S has no enough space left then
26:             index + +
27:         end if
28:         ...                                                   ▷ for other situations
29:         Remove j from Q
30:     end while
31:     return SWL
32: end procedure
```

comprehensive algorithms can be adopted, HEFT or MinMin are able to find an efficient schedule in terms that the sub-workflows are already generated since the number of sub-workflows has been greatly reduced compared to the number of individual jobs.

4.4 Results and Discussion

In order to quickly deploy and reconfigure computational resources, we use a private cloud computing resource called FutureGrid [33]. In all the experiments, each VM has 4 CPU cores, 2 Condor slots, 4GB RAM and has a shared file system mounted to make sure data staged into a site is accessible to all compute nodes. In the initial experiments we build up four clusters, each with 4 VMs, 8 Condor slots. In the last experiment of site selection, the four virtual clusters are reconfigured and each cluster has 4, 8, 10 and 10 Condor slots respectively. The submit host that performs workflow planning and sends jobs to the execution sites is a Linux 2.6 machine equipped with 8GB RAM and an Intel 2.66GHz Quad CPUs. We use Pegasus to plan the workflows and then submit them to Condor DAGMan [23], which provides the workflow execution engine. Each execution site contains a Condor pool and a head node visible to the network.

Table 4.2: CyberShake with Storage Constraint

storage constraint	site	Disk Usage(GB)	Percentage
35GB	A	sub0:0.06; sub1:33.8	97%
	B	sub2:28.8	82%
30GB	A	sub0:0.07;sub1:29.0	97%
	B	sub2:29.3	98%
	C	sub3:28.8	96%
25GB	A	sub0:0.06;sub1:24.1	97%
	B	sub2:24.4	98%
	C	sub3:19.5	78%
20GB	A	sub0:0.06;sub1:18.9	95%
	B	sub2:19.3	97%
	C	sub3:19.6	98%
	D	sub4:15.3	77%

Performance Metrics. To evaluate the performance, we use two types of metrics. Satisfying the Storage Constraints is the main goal of our work in order to fit the sub-workflows into the available storage resources. We compare the results of different storage constraints and heuristics. Improving the Runtime Performance is the second metric in order to minimize the overall makespan. We compare the results of different partitioners, estimators and schedulers.

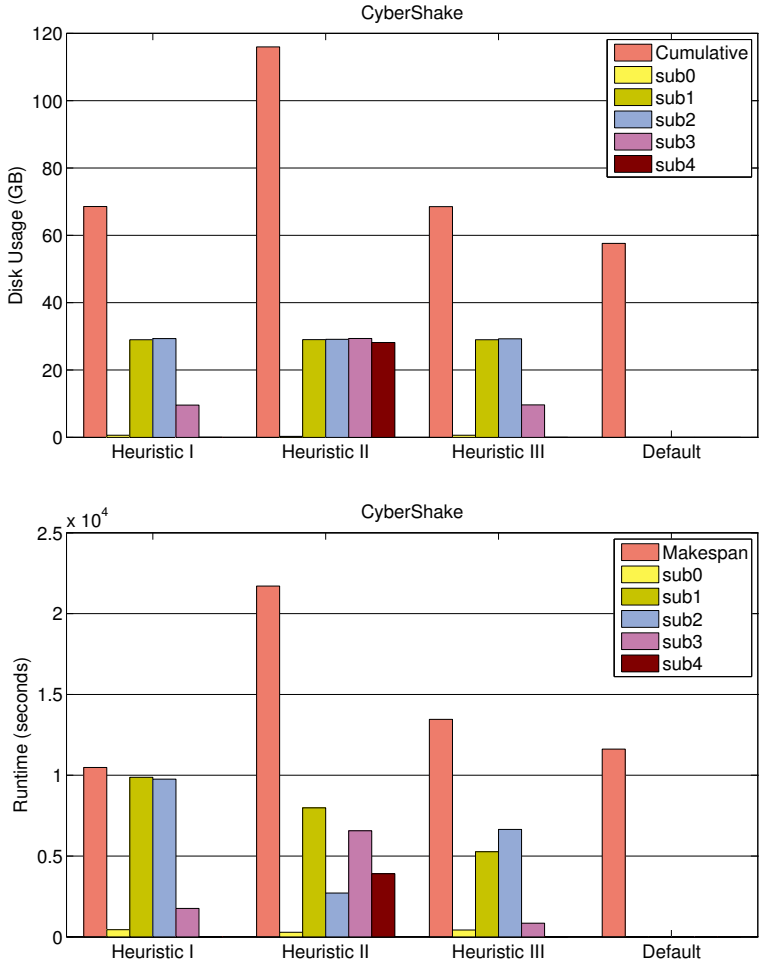

Figure 4.4: Performance of the three heuristics. The default workflow has one execution site with 4 VMs and 8 Condor slots and has no storage constraint.

Workflows Used. We ran three different workflow applications: an astronomy application (Montage), a seismology application (CyberShake) and a bioinformatics application (Epigenomics). They were chosen because they represent a wide range of application domains and a variety of resource requirements [43].

Performance of Different Heuristics. We compare the three heuristics with the CyberShake application. The storage constraint for each site is 30GB. Heuristic II produces 5 sub-workflows with 10 dependencies between them. Heuristic I produces 4 sub-workflows and 3 dependencies. Heuristic III produces 4 sub-workflows and 5 dependencies. The results are shown in Figure 4.4

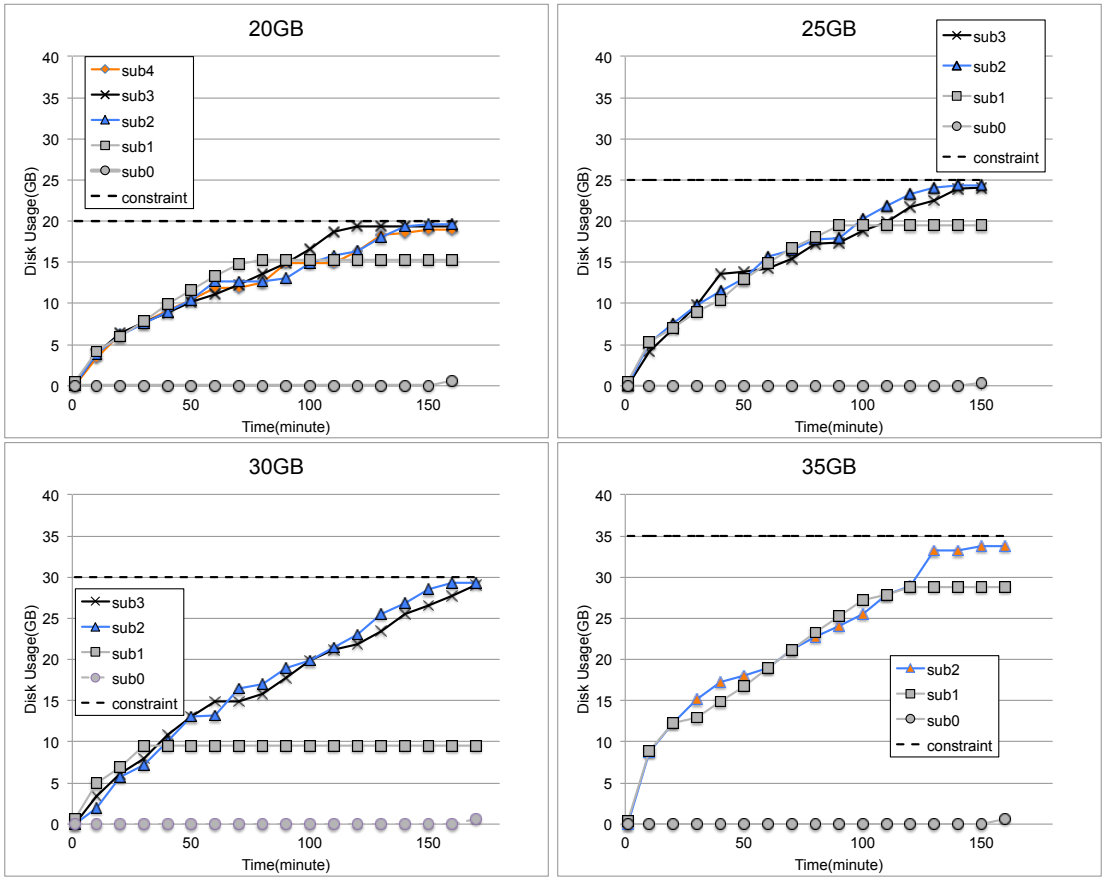

Figure 4.5: CyberShake with storage constraints of 35GB, 30GB, 25GB, and 20GB. They have 3, 4, 4, and 5 sub-workflows and require 2, 3, 3, and 4 sites to run respectively.

and Heuristic I performs better in terms of both runtime reduction and disk usage. This is due to the way it handles the cross dependency. Heuristic II or Heuristic III simply adds a job if it does not violate the storage constraints or the cross dependency constraints. Furthermore, Heuristic I puts the entire fan structure into the same sub-workflow if possible and therefore reduces the dependencies between sub-workflows. From now on, we only use Heuristic I in the partitioner in our experiments below.

Performance with Different Storage Constraints. Figure 4.5 and Table 4.2 depict the disk usage of the CyberShake workflows over time with storage constraints of 35GB, 30GB, 25GB, and 20GB. They are chosen because they represent a variety of required execution sites. Figure 4.6 depicts the performance of both disk usage and runtime. Storage constraints for

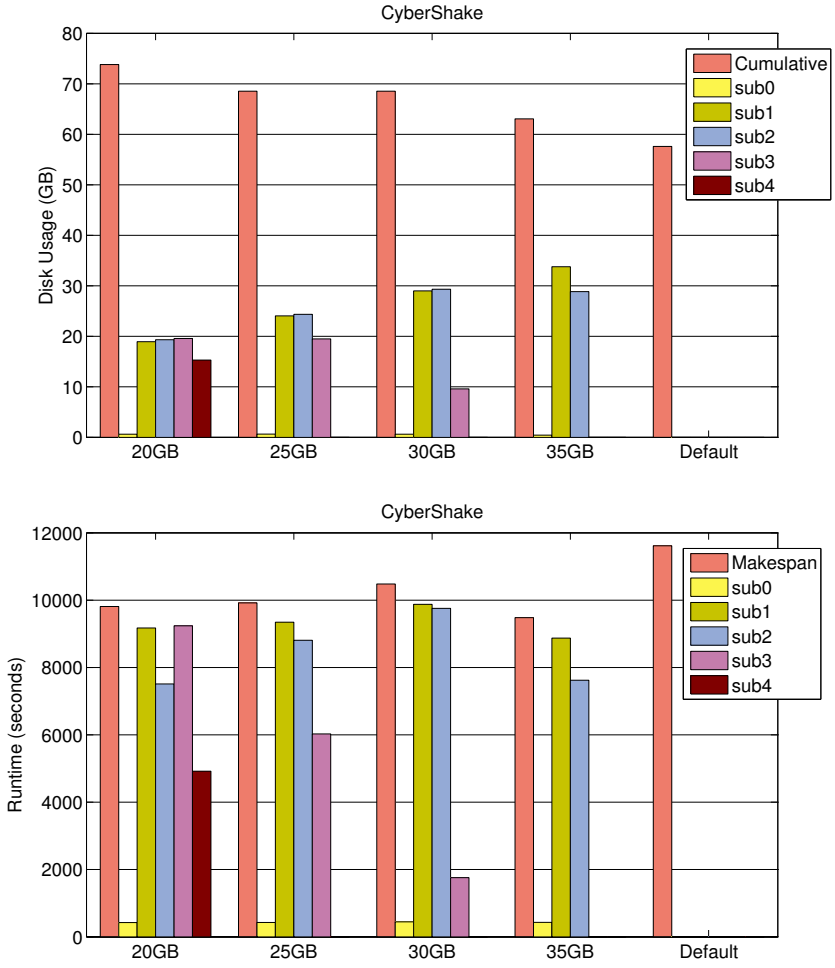

Figure 4.6: Performance of the CyberShake workflow with different storage constraints

Table 4.3: Performance of estimators and schedulers

Combination	Estimator	Scheduler	Makespan(second)
HEFT+HEFT	HEFT	HEFT	10559.5
PATH+HEFT	Critical Path	HEFT	12025.4
CPU+HEFT	Average CPU Time	HEFT	12149.2
HEFT+MIN	HEFT	MinMin	10790
PATH+MIN	Critical Path	MinMin	11307.2
CPU+MIN	Average CPU Time	MinMin	12323.2

all of the sub-workflows are satisfied. Among them sub1, sub2, sub3 (if exists), and sub4 (if exists) are run in parallel and then sub0 aggregates their work. The CyberShake workflow across two sites with a storage constraint of 35GB performs best. The makespan (overall completion

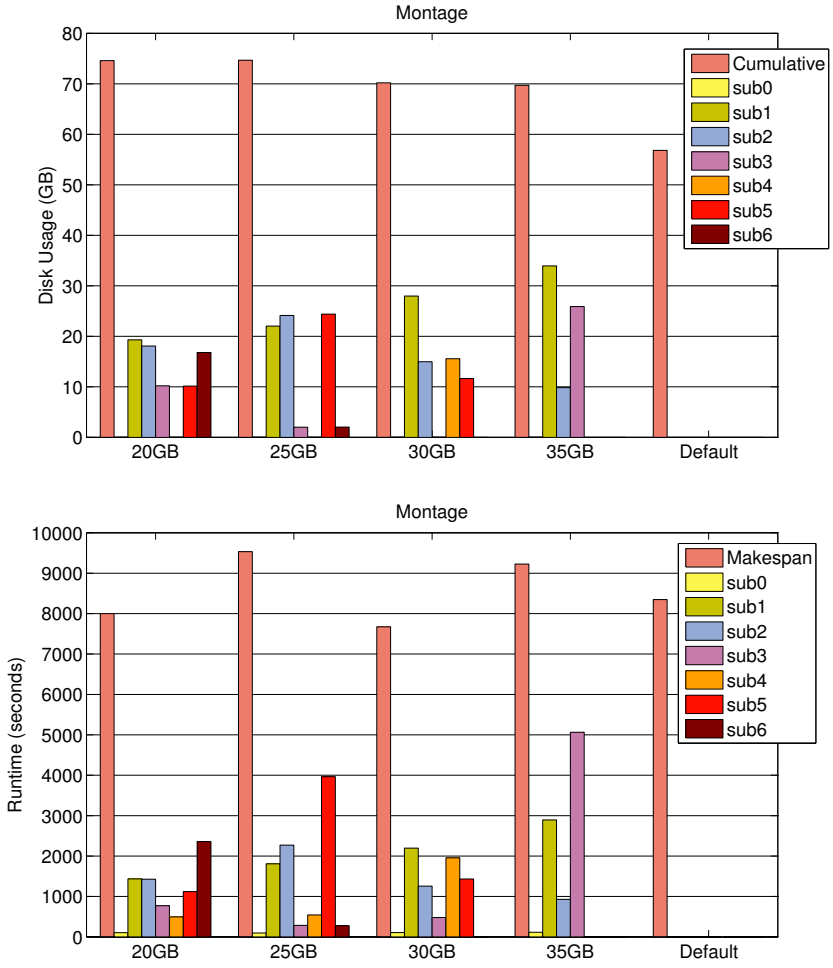

Figure 4.7: Performance of the Montage workflow with different storage constraints

time) improves by 18.38% and the cumulative disk usage increases by only 9.5% compared to the default workflow without partitioning or storage constraints. The cumulative data usage is increased because some shared data is transferred to multiple sites. Workflows with more sites to run on do not have a smaller makespan because they require more data transfer even though the computation part is improved.

Figure 4.8 depicts the performance of Montage with storage constraints ranging from 20GB to 35GB and Epigenomics with storage constraints ranging from 8.5GB to 15GB. The Montage workflow across three sites with 30GB disk space performs best with 8.1% improvement in makespan and the cumulative disk usage increases by 23.5%. The Epigenomics workflow across

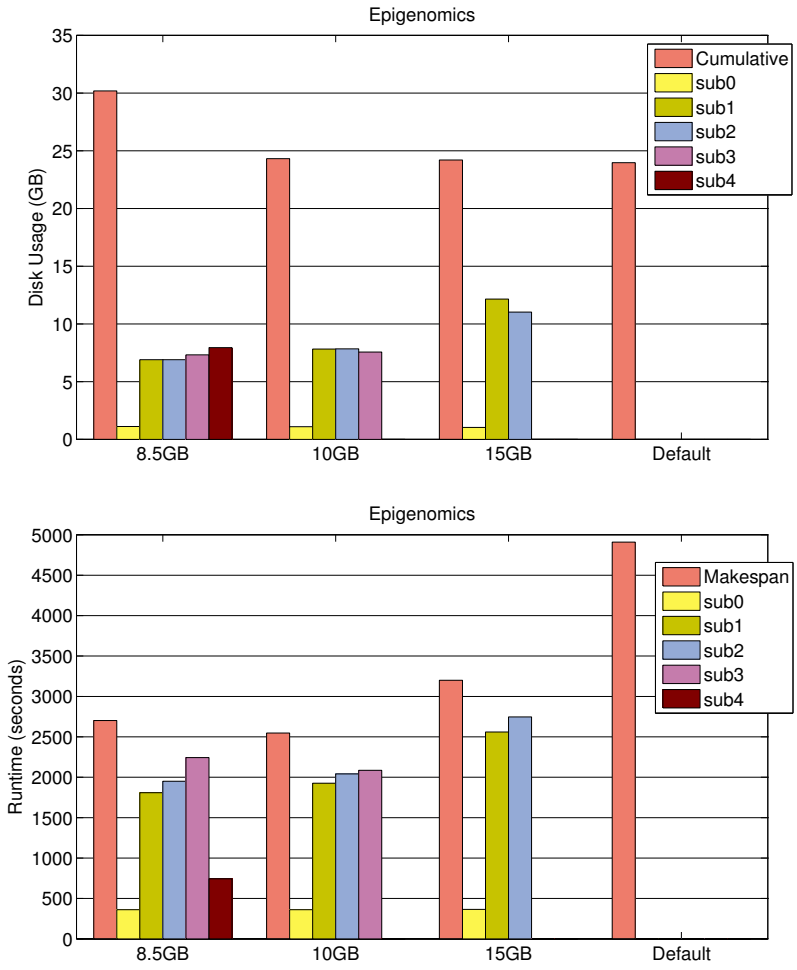

Figure 4.8: Performance of the Epigenomics workflow with different storage constraints

three sites with 10GB storage constraints performs best with 48.1% reduction in makespan and only 1.4% increase in cumulative storage. The reason why Montage performs worse is related to its complex internal structures. Montage has two levels of fan-out-fan-in structures and each level has complex dependencies between them.

Site selection. To show the performance of site selection for each sub-workflow, we use three estimators and two schedulers together with the CyberShake workflow. We build four execution sites with 4, 8, 10 and 10 Condor slots respectively. The labels in Figure 4.9 and Table 4.3 are defined in a way of Estimator + Scheduler. For example, HEFT+HEFT denotes a combination of HEFT estimator and HEFT scheduler, which performs best as we expected. The

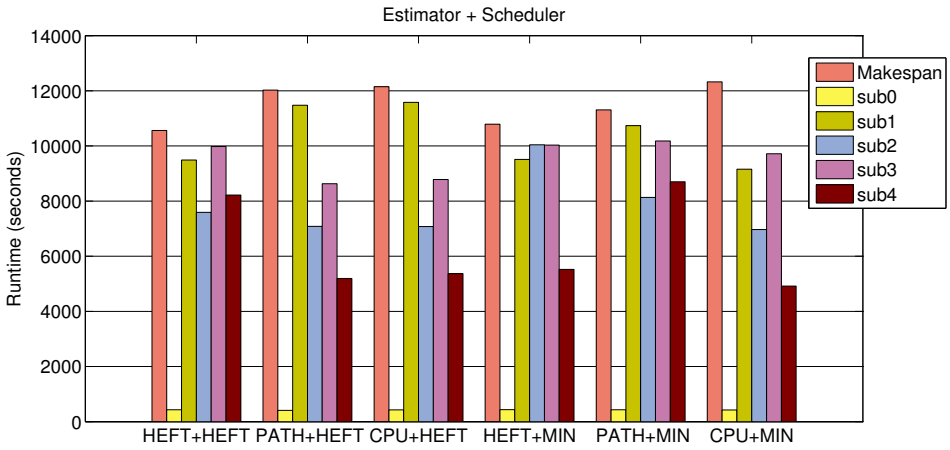

Figure 4.9: Performance of estimators and schedulers

Average CPU Time (or CPU in Figure 4.7) does not take the dependencies into consideration and the Critical Path (or PATH in Figure 4.9) does not consider the resource availability. The HEFT scheduler is slightly better than the MinMin scheduler (or MIN in Figure 4.9). Although the HEFT scheduler uses a global optimization algorithm compared to the local optimization in MinMin, the complexity of scheduling sub-workflows has been greatly reduced compared to scheduling a vast number of individual tasks. Therefore, both local and global optimization algorithms are able to handle such situations well.

4.5 Summary

In this chapter, we propose a framework of partitioning and scheduling large workflows to provide a fine granularity adjustment of workflow activities. The approach relies on partitioning the workflow into valid sub-workflows. Three heuristics are proposed and compared to show the close relationship between cross dependency and runtime improvement. The performance with three workflows shows that this approach is able to satisfy the storage constraints and reduce the makespan significantly especially for Epigenomics which has fewer fan-in (synchronization) jobs. For the workflows we used, scheduling them onto two or three execution sites is best due to a tradeoff between increased data transfer and increased parallelism. Site selection shows that

the global optimization and local optimization perform almost the same. Even though we are using data constraint as an example in this chapter, our approach can be simply modified to apply to other constraints such as CPU, memory and bandwidth.

After workflow partitioning, we satisfy the resource constraints within an execution site. However, resources within an execution site are not evenly distributed and the balancing between computation and communication is another challenge. In next chapter, we will introduce our balanced task clustering to address this issue.

Chapter 5

Balanced Clustering

In this chapter, we examine the reasons that cause load imbalance in task clustering. Furthermore, we propose a series of balanced task clustering methods to address these imbalance problems. A trace-based simulation shows our methods can significantly improve the runtime performance of five widely used workflows compared to the naïve implementation of task clustering.

5.1 Motivation

Existing task clustering strategies have demonstrated their effect in some scientific workflows such as CyberShake [55] and LIGO [26]. However, there are several challenges that are not yet addressed.

The first challenge users face when executing workflows is task runtime variation. In a scientific workflow, tasks within a level (or depth within a workflow directed acyclic graph) may have different runtimes. Merging tasks within a level without considering the runtime variance may cause load imbalance, i.e., some clustered jobs may be composed of short running tasks while others of long running tasks. This imbalance delays the release of tasks from the next level of the workflow, penalizing the workflow execution with an overhead produced by the use of inappropriate task clustering strategies. A common technique to handle load imbalance is overdecomposition [51]. This method decomposes computational work into medium-grained balanced tasks. Each task is coarse-grained enough to enable efficient execution and reduce scheduling overheads, while being fine-grained enough to expose significantly higher application-level parallelism than what is offered by the hardware.

The second challenge deals with the complex data dependencies within a workflow. Merging tasks that have no intermediate data between them seems safe at the first sight. However, the

subsequent tasks that rely on the output data that their parent tasks produce may suffer a data locality problem since data may be distributed poorly and the data transfer time is increased. As a result, data transfer times and failure probabilities increase. Therefore, we claim that data dependencies of subsequent tasks should be considered.

We generalize these two challenges (Runtime Imbalance and Dependency Imbalance) to the general load balance problem. We introduce a series of balancing methods to address these challenges. However, there is a tradeoff between runtime and data dependency balancing. For instance, balancing runtime may aggravate the Dependency Imbalance problem, and vice versa. Therefore, we propose a series of quantitative metrics that reflect the internal structure (in terms of task runtimes and dependencies) of the workflow and use them as a criterion to select and balance these solutions.

In particular, we provide a novel approach to capture these metrics. Traditionally, there are two approaches to improve the performance of task clustering. The first one is a top-down approach [17] that represents the task clustering problem as a global optimization problem and aims to minimize the overall runtime of a workflow. However, the complexity of solving such an optimization problem does not scale well. The second one is a bottom-up approach [59][52] that only examines free tasks to be merged and optimizes the task clustering results locally. In contrast, our work extends these solutions to consider the neighboring tasks including siblings, parents, children and so on because such a family of tasks has strong connections between them.

In this chapter, we address the balancing problem by studying (*i*) the performance gain of using our balancing methods over a baseline execution on a larger set of workflows; (*ii*) the performance gain over two additional task clustering methods in literature; (*iii*) the performance impact of the variation of the average data size and the number of resources; and (*iv*) the performance impact of combining our balancing methods with vertical clustering.

5.2 Related Work

Workflow Scheduling. There have been a considerable amount of work trying to solve workflow-mapping problem using DAG scheduling heuristics such as HEFT [90], Min-Min [7], MaxMin [8], MCT [8], etc. Duan [30] and Wieczorek [95] have discussed the scheduling and the partitioning of scientific workflows in dynamic grids. The emergence of cloud computing [3] has made it possible to easily lease large-scale homogeneous computing resources from commercial resource providers and satisfy QoS requirements. In our case, since we assume resources are homogeneous, the resource selection is not a major concern and thus the scheduling problem can be simplified as a task clustering problem. More specifically, a plethora of scheduling algorithms have been developed in the networking and operating system domains. Many of these schedulers have been extended to perform in a hierarchical fashion [51], which divides the tasks into independent autonomous groups and organizes the groups in a hierarchy. Lifflander et al. [51] proposed to use work stealing and a hierarchical persistence-based rebalancing algorithm to address the imbalance problem in scheduling. Zheng et al. [105] presented an automatic hierarchical load balancing method that overcomes the scalability challenges of centralized schemes and poor solutions of traditional distributed schemes. There are other scheduling algorithms [8] (e.g. list scheduling) that indirectly achieve load balancing of workflows through makespan minimization. However, the benefit that can be achieved through traditional scheduling optimization is limited by its complexity. The performance gain of task clustering is primarily determined by the ratio between system overheads and task runtime, which is more substantial in modern distributed systems such as clouds and grids.

Task Granularity Control has also been addressed in scientific workflows. For instance, Singh et al. [79] proposed a level- and label-based clustering. In level-based clustering, tasks at the same workflow level can be clustered together. The number of clusters or tasks per cluster is specified by the user. In the label-based clustering, the user labels tasks that should be clustered together. Although their work considers data dependencies between workflow levels, it is done manually by the users, a process that is prone to errors. Recently, Ferreira da Silva et al. [32]

proposed task grouping and ungrouping algorithms to control workflow task granularity in a non-clairvoyant and online context, where none or few characteristics about the application or resources are known in advance. Their work significantly reduced scheduling and queuing time overheads, but did not consider data dependencies. The low performance of *fine-grained* tasks is a common problem in widely distributed platforms where the scheduling overhead and queuing times at resources are high, such as grid and cloud systems. Several works have addressed the control of task granularity of bags of tasks. For instance, Muthuvelu et al. [59] proposed a clustering algorithm that groups bags of tasks based on their runtime—tasks are grouped up to the resource capacity. Later, they extended their work [58] to determine task granularity based on task file size, CPU time, and resource constraints. Recently, they proposed an online scheduling algorithm [57, 60] that groups tasks based on resource network utilization, user's budget, and application deadline. Ng et al. [61] and Ang et al. [87] introduced bandwidth in the scheduling framework to enhance the performance of task scheduling. Longer tasks are assigned to resources with better bandwidth. Liu and Liao [52] proposed an adaptive fine-grained job scheduling algorithm to group fine-grained tasks according to processing capacity and bandwidth of the current available resources. Although these techniques significantly reduce the impact of scheduling and queuing time overhead, they did not consider data dependencies.

Load Balancing. With the aim of dynamically balancing the computational load among resources, some jobs have to be moved from one resource to another and/or from one period of time to another, which is called task reallocation [89]. Caniou [12] presented a reallocation mechanism that tunes parallel jobs each time they are submitted to the local resource manager. They only needed to query batch schedulers with simple submission or cancellation requests. They also presented different reallocation algorithms and studied their behavior in the case of a multi-cluster environment. In [101], a preemptive process migration method was proposed to dynamically migrate processes from overloaded computational nodes to lightly-loaded nodes. However, this approach can achieve a good balance only when there are some idle compute nodes (e.g. when the number of task processes is less than that of compute nodes). In our case, since

we deal with large-scale scientific workflows, usually we have more tasks than available compute nodes. Guo et al. [36] presented mechanisms to dynamically split and consolidate tasks to cope with load imbalance. They have proposed algorithms to dynamically split unfinished tasks to fill idle compute nodes. Similarly, Ying et al. [99] proposed a load-balancing algorithm based on collaborative task clustering. The algorithm divides the collaborative computing tasks into subtasks and then dynamically allocates them to the servers. Compared to these approaches, our work selects tasks to be merged based on their task runtime distribution and their data dependencies initially, without introducing additional overheads during the runtime. Also, a quantitative approach of imbalance measurement guides the selection of tasks to be merged without causing runtime imbalance or dependency imbalance.

5.3 Approach

In this section, we introduce metrics that quantitatively capture workflow characteristics to measure runtime and dependence imbalances. We then present methods to address the load balance problem.

5.3.1 Imbalance metrics

Runtime Imbalance describes the difference of the task/job runtime of a group of tasks/jobs. In this work, we denote the **Horizontal Runtime Variance** (*HRV*) as the ratio of the standard deviation in task runtime to the average runtime of tasks/jobs at the same horizontal level of a workflow. At the same horizontal level, the job with the longest runtime often controls the release of the next level jobs. A high *HRV* value means that the release of next level jobs has been delayed. Therefore, to improve runtime performance, it makes sense to reduce the standard deviation of job runtime. Figure 5.1 shows an example of four independent tasks t_1, t_2, t_3 and t_4 where the task runtime of t_1 and t_2 is 10 seconds, and the task runtime of t_3 and t_4 is 30 seconds. In the Horizontal Clustering (HC) approach, a possible clustering result could be merging t_1 and t_2 into a clustered job, and t_3 and t_4 into another. This approach results in imbalanced runtime,

i.e., $HRV > 0$ (Figure 5.1-top). In contrast, a balanced clustering strategy should try its best to evenly distribute task runtime among jobs as shown in Figure 5.1 (bottom). A smaller *HRV* means that the runtime of tasks within a horizontal level is more evenly distributed and therefore it is less necessary to use runtime-based balancing algorithms. However, runtime variance is not able to describe how symmetric the structure of the dependencies between tasks is.

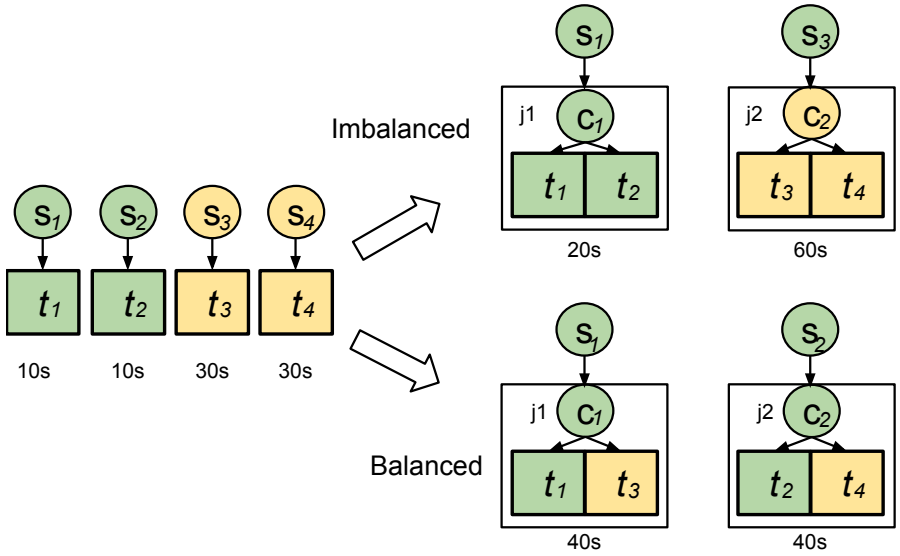

Figure 5.1: An example of Horizontal Runtime Variance.

Dependency Imbalance means that the task clustering at one horizontal level forces the tasks at the next level (or even subsequent levels) to have severe data locality problems and thus loss of parallelism. For example, in Figure 5.2, we show a two-level workflow composed of four tasks in the first level and two in the second. Merging t_1 with t_3 and t_2 with t_4 (imbalanced workflow in Figure 5.2) forces t_5 and t_6 to transfer files from two locations and wait for the completion of t_1, t_2, t_3, and t_4. A balanced clustering strategy groups tasks that have the maximum number of child tasks in common. Thus, t_5 can start to execute as soon as t_1 and t_2 are completed, and so can t_6. To measure and quantitatively demonstrate the Dependency Imbalance of a workflow, we propose two metrics: (*i*) Impact Factor Variance, and (*ii*) Distance Variance.

We define the **Impact Factor Variance** (IFV) of tasks as the standard deviation of their impact factors. The **Impact Factor** (IF) of a task t_u is defined as follows:

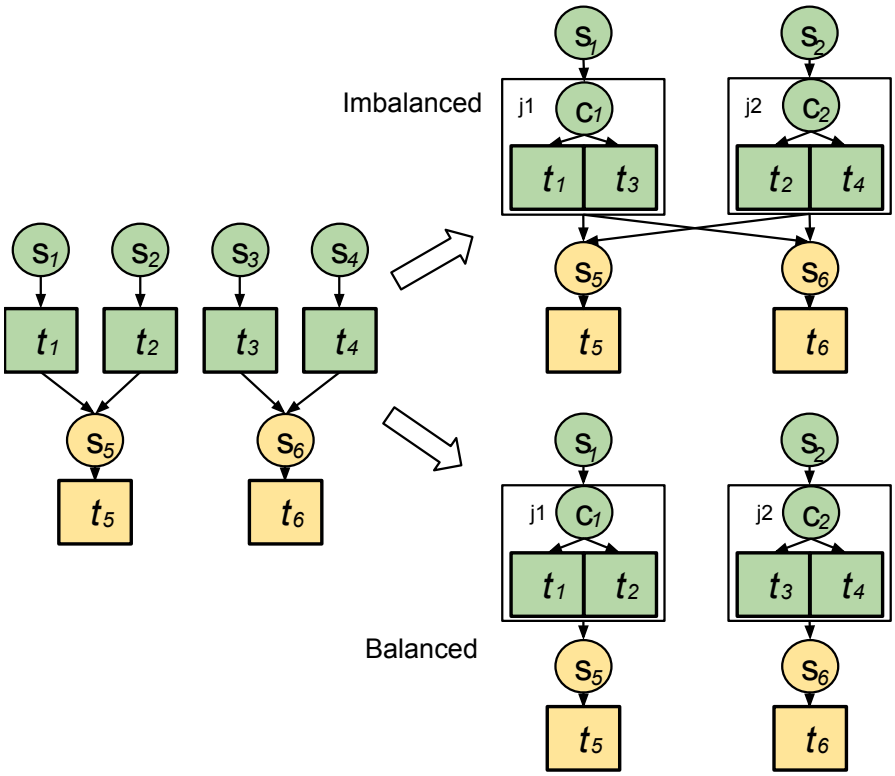

Figure 5.2: An example of Dependency Imbalance.

$$IF(t_u) = \sum_{t_v \in Child(t_u)} \frac{IF(t_v)}{||Parent(t_v)||} \tag{5.1}$$

where $Child(t_u)$ denotes the set of child tasks of t_u, and $||Parent(t_v)||$ the number of parent tasks of t_v. The Impact Factor aims at capturing the similarity of tasks/jobs in a graph by measuring their relative impact factor or importance to the entire graph. Tasks with similar impact factors are merged together, so that the workflow structure tends to be more 'even' or symmetric. For simplicity, we assume the IF of a workflow exit task (e.g. t_5 in Figure 5.2) is 1.0. For instance, consider the two workflows presented in Figure 5.3. The IF for each of t_1, t_2, t_3, and t_4 is computed as follows:

$$IF(t_7) = 1.0, IF(t_6) = IF(t_5) = IF(t_7)/2 = 0.5$$

$$IF(t_1) = IF(t_2) = IF(t_5)/2 = 0.25$$

$$IF(t_3) = IF(t_4) = IF(t_6)/2 = 0.25$$

Thus, IFV(t_1, t_2, t_3, t_4) = 0. In contrast, the IF for $t_{1'}$, $t_{2'}$, $t_{3'}$, and $t_{4'}$ is:

$$IF(t_{7'}) = 1.0, IF(t_{6'}) = IF(t_{5'}) = IF(t_{1'}) = IF(t_{7'})/2 = 0.5$$

$$IF(t_{2'}) = IF(t_{3'}) = IF(t_{4'}) = IF(t_{6'})/3 = 0.17$$

Therefore, the IFV value for $t_{1'}$, $t_{2'}$, $t_{3'}$, $t_{4'}$ is 0.17, which predicts it is likely to be less symmetric than the workflow in Figure 5.3 (left). In this chapter, we use **HIFV** (Horizontal IFV) to indicate the IFV of tasks at the same horizontal level. The time complexity of calculating the IF of all the tasks of a workflow with n tasks is $O(n)$.

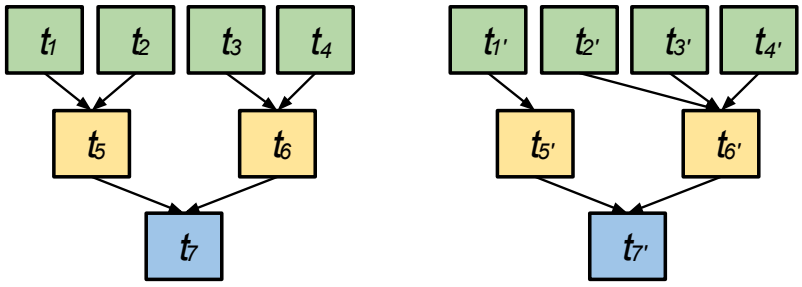

Figure 5.3: Example of workflows with different data dependencies

Distance Variance (DV) describes how 'close' tasks are to each other. The distance between two tasks/jobs is defined as the cumulative length of the path to their closest common successor. If they do not have a common successor, the distance is set to infinity. For a group of n tasks/jobs, the distance between them is represented by a $n \times n$ matrix D, where an element $D(u, v)$ denotes

the distance between a pair of tasks/jobs u and v. For any workflow structure, $D(u, v) = D(v, u)$ and $D(u, u) = 0$, thus we ignore the cases when $u \geq v$. Distance Variance is then defined as the standard deviation of all the elements $D(u, v)$ for $u < v$. The time complexity of calculating all the values of D of a workflow with n tasks is $O(n^2)$.

Similarly, HDV indicates the DV of a group of tasks/jobs at the same horizontal level. For example, Table 5.1 shows the distance matrices of tasks from the first level for both workflows of Figure 5.3 (D_1 for the workflow in the left, and D_2 for the workflow in the right). HDV for t_1, t_2, t_3, and t_4 is 1.03, and for $t_{1'}, t_{2'}, t_{3'}$, and $t_{4'}$ is 1.10. In terms of distance variance, D_1 is more 'even' than D_2. A smaller HDV means that the tasks at the same horizontal level are more equally 'distant' from each other and thus the workflow structure tends to be more 'even' and symmetric.

D_1	t_1	t_2	t_3	t_4
t_1	0	2	4	4
t_2	2	0	4	4
t_3	4	4	0	2
t_4	4	4	2	0

D_2	t'_1	t'_2	t'_3	t'_4
t'_1	0	4	4	4
t'_2	4	0	2	2
t'_3	4	2	0	2
t'_4	4	2	2	0

Table 5.1: Distance matrices of tasks from the first level of workflows in Figure 5.3.

In conclusion, runtime variance and dependency variance offer a quantitative and comparable tool to measure and evaluate the internal structure of a workflow.

5.3.2 Balanced clustering methods

In this subsection, we introduce our balanced clustering methods used to improve the runtime and dependency balance in task clustering. We first introduce the basic runtime-based clustering method, and then two other balancing methods that address the dependency imbalance problem.

Horizontal Runtime Balancing (HRB) aims to evenly distribute task runtime among clustered jobs. Tasks with the longest runtime are added to the job with the shortest runtime. Algorithm 2 shows the pseudocode of HRB. This greedy method is used to address the load balance problem caused by runtime variance at the same horizontal level. Figure 5.4 shows an example of HRB where tasks in the first level have different runtimes and should be grouped into two

Algorithm 2 Horizontal Runtime Balancing algorithm.

Require: W: workflow; C: number of tasks per jobs; R: number of jobs per horizontal level

```
1:  procedure CLUSTERING(W, C)
2:      for level < depth(W) do
3:          TL ← GETTASKSATLEVEL(W, level)        ▷ Partition W based on depth
4:          CL ← MERGE(TL, C, R)                  ▷ Returns a list of clustered jobs
5:          W ← W − TL + CL                       ▷ Merge dependencies as well
6:      end for
7:  end procedure
8:  procedure MERGE(TL, C, R)
9:      for i < R do
10:         J_i ←{}                               ▷ An empty job
11:     end for
12:     CL ←{}                                    ▷ An empty list of clustered jobs
13:     Sort TL in descending of runtime
14:     for all t in TL do
15:         J ← the job with shortest runtime and less than C tasks
16:         J.add (t)                             ▷ Adds the task to the shortest job
17:     end for
18:     for i < R do
19:         CL.add( J_i)
20:     end for
21:     return CL
22: end procedure
```

jobs. HRB sorts tasks in decreasing order of runtime, and then adds the task with the highest runtime to the group with the shortest aggregated runtime. Thus, t_1 and t_3, as well as t_2 and t_4 are merged together. For simplicity, system overheads are not displayed.

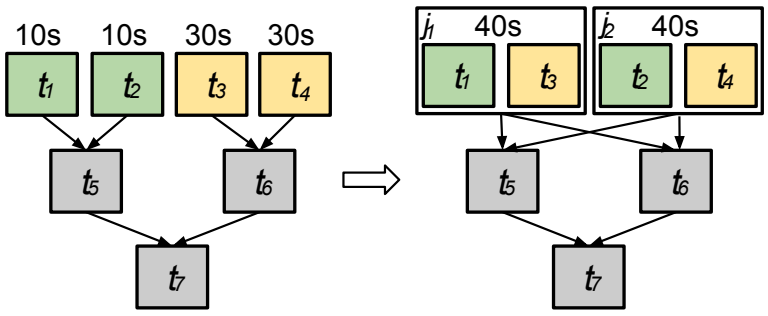

Figure 5.4: An example of the HRB (Horizontal Runtime Balancing) method. By solely addressing runtime variance, data locality problems may arise.

However, HRB may cause a dependency imbalance problem since the clustering does not take data dependency into consideration. To address this problem, we propose the **Horizontal Impact Factor Balancing** (HIFB) and the **Horizontal Distance Balancing** (HDB) methods.

Algorithm 3 Horizontal Impact Factor Balancing algorithm.

Require: W: workflow; C: number of tasks per jobs; R: number of jobs per horizontal level

procedure CLUSTERING(W, C)
  for $level < depth(W)$ **do**
    $TL \leftarrow$ GETTASKSATLEVEL($W, level$) ▷ Partition W based on depth
    $CL \leftarrow$ MERGE(TL, C, R) ▷ Returns a list of clustered jobs
    $W \leftarrow W - TL + CL$ ▷ Merge dependencies as well
  end for
end procedure
procedure MERGE(TL, C, R)
  for $i < R$ **do**
    $J_i \leftarrow \{\}$ ▷ An empty job
  end for
  $CL \leftarrow \{\}$ ▷ An empty list of clustered jobs
  Sort TL in descending of runtime
  for all t in TL **do**
    $L \leftarrow$ Sort all J_i with the similarity of impact factors with t
    $J \leftarrow$ the job with shortest runtime and less than C tasks in L
    J.add (t)
  end for
  for $i < R$ **do**
    CL.add(J_i)
  end for
  return CL
end procedure

Algorithm 4 Horizontal Distance Balancing algorithm.

Require: W: workflow; C: number of tasks per jobs; R: number of jobs per horizontal level

procedure CLUSTERING(W, C)
  for $level < depth(W)$ **do**
    $TL \leftarrow$ GETTASKSATLEVEL($W, level$) ▷ Partition W based on depth
    $CL \leftarrow$ MERGE(TL, C, R) ▷ Returns a list of clustered jobs
    $W \leftarrow W - TL + CL$ ▷ Merge dependencies as well
  end for
end procedure
procedure MERGE(TL, C, R)
  for $i < R$ **do**
    $J_i \leftarrow \{\}$ ▷ An empty job
  end for
  $CL \leftarrow \{\}$ ▷ An empty list of clustered jobs
  Sort TL in descending of runtime
  for all t in TL **do**
    $L \leftarrow$ Sort all J_i with the closest distance with t
    $J \leftarrow$ the job with shortest runtime and less than C tasks in L
    J.add (t)
  end for
  for $i < R$ **do**
    CL.add(J_i)
  end for
  return CL
end procedure

In HRB, candidate jobs within a workflow level are sorted by their runtime, while in HIFB jobs are first sorted based on their similarity of IF, then on runtime. Algorithm 3 shows the pseudocode of HIFB. For example, in Figure 5.5, t_1 and t_2 have $IF = 0.25$, while t_3, t_4, and t_5 have $IF = 0.16$. HIFB selects a list of candidate jobs with the same IF value, and then HRB is performed to select the shortest job. Thus, HIFB merges t_1 and t_2 together, as well as t_3 and t_4.

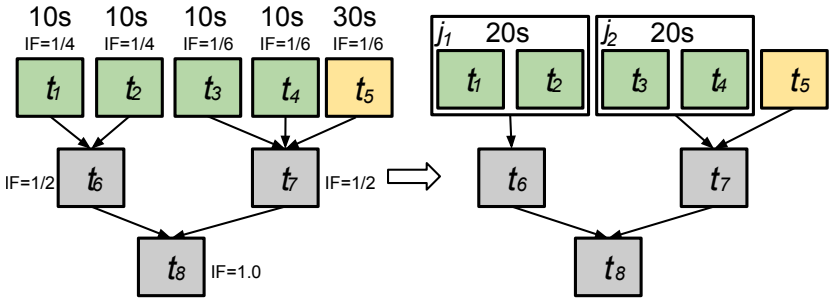

Figure 5.5: An example of the HIFB (Horizontal Impact Factor Balancing) method. Impact factors allow the detection of similarities between tasks.

However, HIFB is suitable for workflows with asymmetric structure. A symmetric workflow structure means there exists a (usually vertical) division of the workflow graph such that one part of the workflow is a mirror of the other part. For symmetric workflows, such as the one shown in Figure 5.4, the IF value for all tasks of the first level will be the same ($IF = 0.25$), thus the method may also cause dependency imbalance. In HDB, jobs are sorted based on the distance between them and the targeted task t, then on their runtimes. Algorithm 4 shows the pseudocode of HDB. For instance, in Figure 5.6, the distances between tasks $D(t_1, t_2) = D(t_3, t_4) = 2$, while $D(t_1, t_3) = D(t_1, t_4) = D(t_2, t_3) = D(t_2, t_4) = 4$. Thus, HDB merges a list of candidate tasks with the minimal distance (t_1 and t_2, and t_3 and t_4). Note that even if the workflow is asymmetric (Figure 5.5), HDB would obtain the same result as with HIFB.

There are cases where HDB would yield lower performance than HIFB. For instance, let t_1, t_2, t_3, t_4, and t_5 be the set of tasks to be merged in the workflow presented in Figure 5.7. HDB does not identify the difference in the number of parent/child tasks between the tasks, since $d(t_u, t_v) = 2, \forall u, v \in [1, 5], u \neq v$. On the other hand, HIFB does distinguish them since their

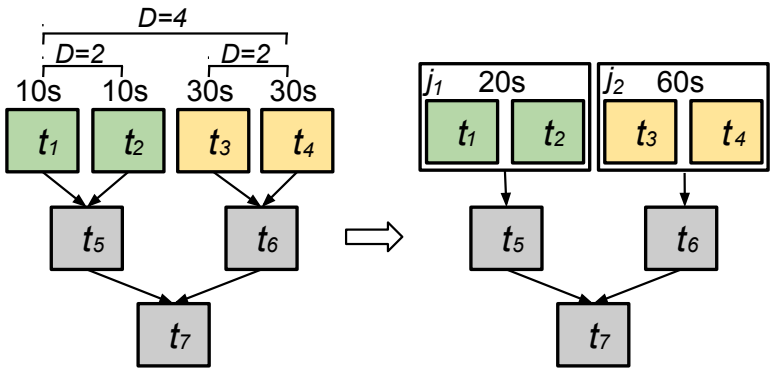

Figure 5.6: An example of the HDB (Horizontal Distance Balancing) method. Measuring the distances between tasks avoids data locality problems.

impact factors are slightly different. Example of such scientific workflows include the LIGO Inspiral workflow [48], which is used in the evaluation of this chapter (Section 5.4.3).

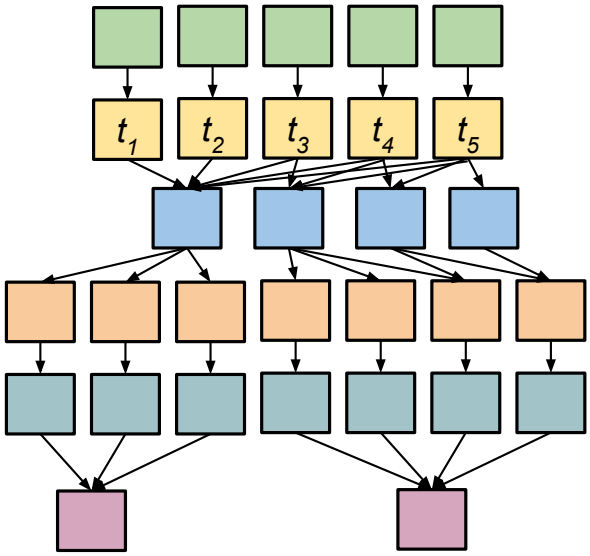

Figure 5.7: A workflow example where HDB yields lower performance than HIFB. HDB does not capture the difference in the number of parents/child tasks, since the distances between tasks (t_1, t_2, t_3, t_4, and t_5) are the same.

Table 5.2 summarizes the imbalance metrics and balancing methods presented in this chapter.

Imbalance Metrics	*abbr.*
Horizontal Runtime Variance	*HRV*
Horizontal Impact Factor Variance	*HIFV*
Horizontal Distance Variance	*HDV*
Balancing Methods	*abbr.*
Horizontal Runtime Balancing	HRB
Horizontal Impact Factor Balancing	HIFB
Horizontal Distance Balancing	HDB

Table 5.2: Summary of imbalance metrics and balancing methods.

5.3.3 Combining vertical clustering methods

In this subsection, we discuss how we combine the balanced clustering methods presented above with vertical clustering (VC). In pipelined workflows (single-parent-single-child tasks), vertical clustering always yields improvement over a baseline, non-clustered execution because merging reduces system overheads and data transfers within the pipeline. Horizontal clustering does not have the same guarantee since its performance depends on the comparison of system overheads and task durations. However, vertical clustering has limited performance improvement if the workflow does not have pipelines. Therefore, we are interested in the analysis of the performance impact of applying both vertical and horizontal clustering in the same workflow. We combine these methods in two ways: (*i*) *VC-prior*, and (*ii*) *VC-posterior*.

VC-prior In this method, vertical clustering is performed first, and then the balancing methods (HRB, HIFB, HDB, or HC) are applied. Figure 5.8 shows an example where pipelined-tasks are merged first, and then the merged pipelines are horizontally clustered based on the runtime variance.

VC-posterior

Here, balancing methods are first applied, and then vertical clustering. Figure 5.9 shows an example where tasks are horizontally clustered first based on the runtime variance, and then merged vertically. However, since the original pipeline structures have been broken by horizontal clustering, VC does not perform any changes to the workflow.

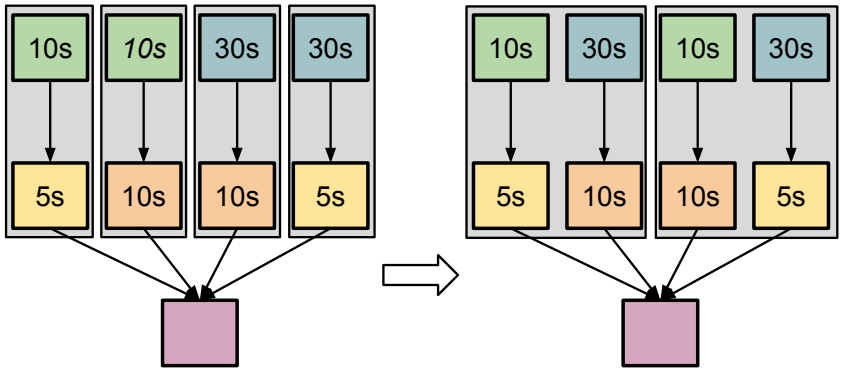

Figure 5.8: *VC-prior*: vertical clustering is performed first, and then the balancing methods.

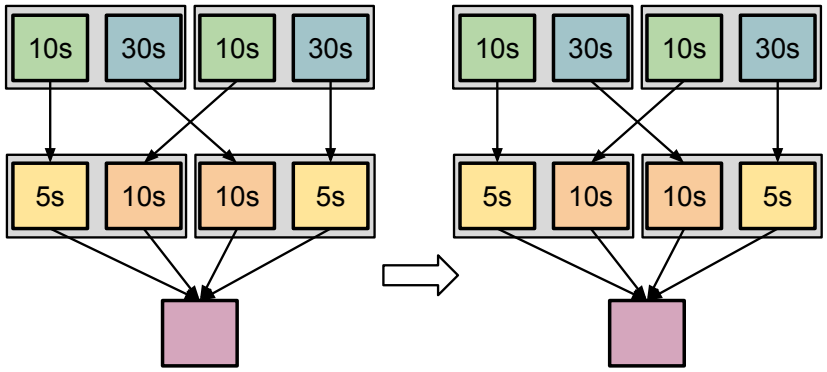

Figure 5.9: *VC-posterior*: horizontal clustering (balancing methods) is performed first, and then vertical clustering (but without changes).

5.4 Evaluation

The experiments presented hereafter evaluate the performance of our balancing methods when compared to an existing and effective task clustering strategy named Horizontal Clustering (HC) [79], which is widely used by workflow management systems such as Pegasus [24]. We also compare our methods with two heuristics described in literature: DFJS [59], and AFJS [52]. DFJS groups bags of tasks based on the task durations up to the resource capacity. AFJS is an extended version of DFJS that is an adaptive fine-grained job scheduling algorithm to group fine-grained tasks according to processing capacity of the current available resources and bandwidth between these resources.

5.4.1 Task clustering techniques

In the experiments, we compare the performance of our balancing methods to the Horizontal Clustering (HC) [79] technique, and with two methods well known from the literature, DFJS [59] and AFJS [52]. In this subsection, we briefly describe each of these algorithms.

HC Horizontal Clustering (HC) merges multiple tasks that are at the same horizontal level of the workflow. The clustering granularity (number of tasks within a cluster) of a clustered job is controlled by the user, who defines either the number of tasks per clustered job (*clusters.size*), or the number of clustered jobs per horizontal level of the workflow (*clusters.num*). This algorithm has been implemented and used in Pegasus [79]. For simplicity, we define *clusters.num* as the number of available resources. In our prior work [21], we have compared the runtime performance with different clustering granularity. The pseudocode of the HC technique is shown in Algorithm 5.

Algorithm 5 Horizontal Clustering algorithm.

```
Require: W: workflow; C: max number of tasks per job defined by clusters.size or clusters.num
 1: procedure CLUSTERING(W, C)
 2:     for level < depth(W) do
 3:         TL ← GETTASKSATLEVEL(W, level)          ▷ Partition W based on depth
 4:         CL ← MERGE(TL, C)                       ▷ Returns a list of clustered jobs
 5:         W ← W − TL + CL                         ▷ Merge dependencies as well
 6:     end for
 7: end procedure
 8: procedure MERGE(TL, C)
 9:     J ← {}                                      ▷ An empty job
10:     CL ←{}                                      ▷ An empty list of clustered jobs
11:     while TL is not empty do
12:         J.add (TL.pop(C)                        ▷ Pops C tasks that are not merged
13:         CL.add( J)
14:     end while
15:     return CL
16: end procedure
```

DFJS The dynamic fine-grained job scheduler (DFJS) was proposed by Muthuvelu et al. [59]. The algorithm groups bags of tasks based on their granularity size—defined as the processing time of the task on the resource. Resources are ordered by their decreasing values of capacity (in MIPS), and tasks are grouped up to the resource capacity. This process continues until all tasks are grouped and assigned to resources. Algorithm 6 shows the pseudocode of the heuristic.

Algorithm 6 DFJS algorithm.

```
Require: W: workflow; max.runtime: max runtime of clustered jobs
 1: procedure CLUSTERING(W, max.runtime)
 2:     for level <the depth of W do
 3:         TL ← GETTASKSATLEVEL(W, level)          ▷ Partition W based on depth
 4:         CL ← MERGE(TL, max.runtime)             ▷ Returns a list of clustered jobs
 5:         W ← W − TL + CL                         ▷ Merge dependencies as well
 6:     end for
 7: end procedure
 8: procedure MERGE(TL, max.runtime)
 9:     J ← {}                                      ▷ An empty job
10:     CL ←{}                                      ▷ An empty list of clustered jobs
11:     while TL is not empty do
12:         t ← TC.pop()                            ▷ Get a task that is not mereged
13:         if J.runtime + t.runtime > max.runtime then
14:             CL.add(J)
15:             J ←{}
16:         end if
17:         J.add(t)
18:     end while
19:     return CL
20: end procedure
```

AFJS The adaptive fine-grained job scheduler (AFJS) [52] is an extension of DFJS. It groups tasks not only based on the maximum runtime defined per cluster job, but also on the maximum data size per clustered job. The algorithm adds tasks to a clustered job until the job's runtime is greater than the maximum runtime or the job's total data size (input + output) is greater than the maximum data size. The AFJS heuristic pseudocode is shown in Algorithm 7.

DFJS and AFJS require parameter tuning (e.g. maximum runtime per clustered job) to efficiently cluster tasks into coarse-grained jobs. For instance, if the maximum runtime is too high, all tasks may be grouped into a single job, leading to loss of parallelism. In contrast, if the runtime threshold is too low, the algorithms do not group tasks, leading to no improvement over a baseline execution.

For comparison purposes, we perform a parameter study in order to tune the algorithms for each workflow application described in Section 2.3. Exploring all possible parameter combinations is a cumbersome and exhaustive task. In the original DFJS and AFJS works, these parameters are empirically chosen, however this approach requires deep knowledge about the workflow applications. Instead, we performed a parameter tuning study, where we first estimate

Algorithm 7 AFJS algorithm.

Require: W: workflow; *max.runtime*: the maximum runtime for a clustered jobs; *max.datasize*: the maximum data size for a clustered job

```
 1: procedure CLUSTERING(W, max.runtime)
 2:     for level <the depth of W do
 3:         TL ← GETTASKSATLEVEL(W, level)              ▷ Partition W based on depth
 4:         CL ← MERGE(TL, max.runtime, max.datasize)   ▷ Returns a list of clustered jobs
 5:         W ← W − TL + CL                             ▷ Merge dependencies as well
 6:     end for
 7: end procedure
 8: procedure MERGE(TL, max.runtime, max.datasize)
 9:     J ← {}                                          ▷ An empty job
10:     CL ←{}                                          ▷ An empty list of clustered jobs
11:     while TL is not empty do
12:         t ← TC.pop()                                ▷ Get a task that is not mereged
13:         if J.runtime + t.runtime > max.runtime OR J.datasize + t.datasize > max.datasize then
14:             CL.add(J)
15:             J ←{}
16:         end if
17:         J.add(t)
18:     end while
19:     return CL
20: end procedure
```

the upper bound of *max.runtime* (n) as the sum of all task runtimes, and the lower bound of *max.runtime* (m) as 1 second for simplicity. Data points are divided into ten chunks and then we sample one data point from each chunk. We then select the chunk that has the lowest makespan and set n and m as the upper and lower bounds of the selected chunk, respectively. These steps are repeated until n and m have converged into a data point.

To demonstrate the correctness of our sampling approach in practice, we show the relationship between the makespan and the *max.runtime* for an example Montage workflow application in Figure 5.10—experiment conditions are presented in Section 5.4.2. Data points are divided into 10 chunks of 250s each (for *max.runtime*). As the lower makespan values belongs to the first chunk, n is updated to 250, and m to 1. The process repeats until the convergence around *max.runtime*=180s. Even though there are multiple local minimal makespan values, these data points are close to each other, and the difference between their values (on the order of seconds) is small enough to be ignored.

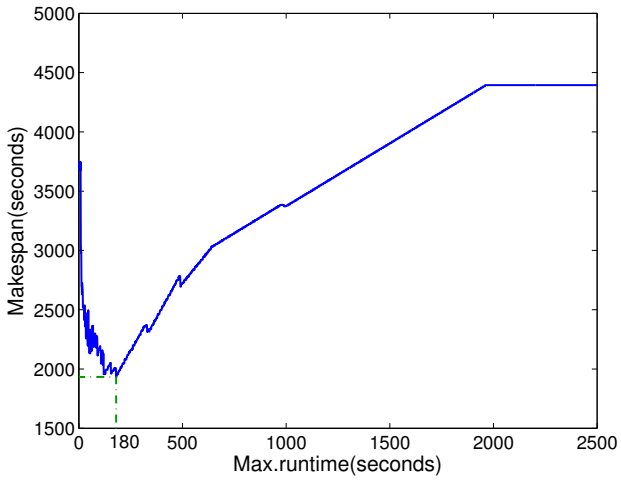

Figure 5.10: Relationship between the makespan of workflow and the specified maximum runtime in DFJS (Montage).

For simplicity, in the rest of this chapter we use DFJS* and AFJS* to indicate the best estimated performance of DFJS and AFJS respectively using the sampling approach described above.

5.4.2 Experiment conditions

We adopt a trace-based simulation approach, where we extended our WorkflowSim [19] simulator with the balanced clustering methods and imbalance metrics to simulate a controlled distributed environment. The simulated computing platform is composed by 20 single homogeneous core virtual machines (worker nodes), which is the quota per user of some typical distributed environments such as Amazon EC2 [1] and FutureGrid [33]. Each simulated virtual machine (VM) has 512MB of memory and the capacity to process 1,000 million instructions per second. The default network bandwidth is 15MB according to the real environment in FutureGrid from where our traces were collected. The task scheduling algorithm is data-aware, i.e. tasks are scheduled to resources which have the most input data available. By default, we merge

tasks at the same horizontal level into 20 clustered jobs, which is a simple selection of granularity control of the strength of task clustering. The study of granularity size has been done in [21], which shows such selection is acceptable.

We collected workflow execution traces [42, 15] (including overhead and task runtime information) from real runs (executed on FutureGrid and Amazon EC2) of the scientific workflow applications described in Section 2.3. The traces are used to feed the Workflow Generator toolkit [96] to generate synthetic workflows. This allows us to perform simulations with different configurations under controlled conditions. The toolkit uses the information gathered from actual scientific workflow executions to generate synthetic workflows resembling those used by real world scientific applications. The number of inputs to be processed, the number of tasks in the workflow, and their composition determine the structure of the generated workflow. Such an approach of traced based simulation allows us to utilize real traces and vary the system setting (i.e., the number of VMs) and workflow (i.e., avg. data size) to fully explore the performance of our balancing algorithms.

Three sets of experiments are conducted. Experiment 1 evaluates the performance gain (μ) of our balancing methods (HRB, HIFB, and HDB) over a baseline execution that has no task clustering. We define the performance gain over a baseline execution (μ) as the performance of the balancing methods related to the performance of an execution without clustering. Thus, for values of $\mu > 0$ our balancing methods perform better than the baseline execution. Otherwise, the balancing methods perform poorer. The goal of the experiment is to identify conditions where each method works best and worst. In addition, we also evaluate the performance gain of using workflow structure metrics (HRV, HIFV, and HDV), which require fewer *a-priori* knowledge from task and resource characteristics, over task clustering techniques in literature (HC, DFJS*, and AFJS*).

Experiment 2 evaluates the performance impact of the variation of average data size (defined as the average of all the input and output data) and the number of resources available in our balancing methods for one scientific workflow application (LIGO). The original average data size (both input and output data) of the LIGO workflow is about 5MB as shown in Table 2.1.

In this experiment, we increase the average data size up to 500MB to study the behavior of data-intensive workflows. We control resource contention by varying the number of available resources (VMs). High resource contention is achieved by setting the number of available VMs to 5, which represents less than 10% of the required resources to compute all tasks in parallel. On the other hand, low contention is achieved when the number of available VMs is increased to 25, which represents about 50% of the required resources.

Experiment 3 evaluates the influence of combining our horizontal clustering methods with vertical clustering (VC). We compare the performance gain under four scenarios: (*i*) *VC-prior*, VC is first performed and then HRB, HIFB, or HDB; (*ii*) *VC-posterior*, horizontal methods are performed first and then VC; (*iii*) *No-VC*, horizontal methods only; and (*iv*) *VC-only*, no horizontal methods. Table 5.3 shows the results of combining VC with horizontal methods. For example, VC-HIFB indicates we perform VC first and then HIFB.

Combination	HIFB	HDB	HRB	HC
VC-prior	VC-HIFB	VC-HDB	VC-HRB	VC-HC
VC-posterior	HIFB-VC	HDB-VC	HRB-VC	HC-VC
VC-only	VC	VC	VC	VC
No-VC	HIFB	HDB	HRB	HC

Table 5.3: Combination Results. '-' indidates the order of performing these algorithms, i.e., VC-HIFB indicates we perform VC first and then HIFB

5.4.3 Results and discussion

Experiment 1 Figure 5.11 shows the performance gain μ of the balancing methods for the five workflow applications over a baseline execution. All clustering techniques significantly improve (up to 48%) the runtime performance of all workflow applications, except HC for SIPHT. The reason is that SIPHT has a high HRV compared to other workflows as shown in Table 5.4. This indicates that the runtime imbalance problem in SIPHT is more significant and thus it is harder for HC to achieve performance improvement. Cybershake and Montage workflows have the highest gain but nearly the same performance independent of the algorithm. This is due to their symmetric structure and low values for the imbalance metrics and the distance metrics as shown

in Table 5.4. Epigenomics and LIGO have higher average task runtime and thus the lower performance gain. However, Epigenomices and LIGO have higher variance of runtime and distance and thus the performance improvement of HRB and HDB is better than that of HC, which is more significant compared to other workflows. In particular, each branch of the Epigenomics workflow (Figure 2.11) has the same number of pipelines, consequently the IF values of tasks in the same horizontal level are the same. Therefore, HIFB cannot distinguish tasks from different branches, which leads the system to a dependency imbalance problem. In such cases, HDB captures the dependency between tasks and yields better performance. Furthermore, Epigenomics and LIGO workflows have high runtime variance, which has higher impact on the performance than data dependency. Last, the performance gain of our balancing methods is better than the tuned algorithms DFJS* and AFJS* in most cases. The other benefit is that our balancing methods do not require parameter tuning, which is cumbersome in practice.

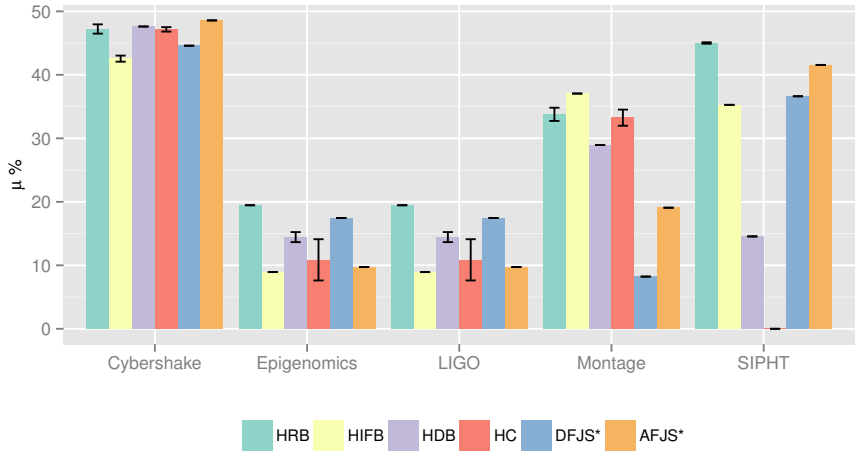

Figure 5.11: Experiment 1: performance gain (μ) over a baseline execution for six algorithms (* indicates the tuned performance of DFJS and AFJS). By default, we have 20 VMs.

Experiment 2 Figure 5.12 shows the performance gain μ of HRB, HIFB, HDB, and HC over a baseline execution for the LIGO Inspiral workflow. We chose LIGO because the performance improvement among these balancing methods is significantly different for LIGO compared to other workflows as shown in Figure 5.11. For small data sizes (up to 100 MB), the application is

	# of Tasks	HRV	HIFV	HDV
Level	(a) **CyberShake**			
1	4	0.309	0.03	1.22
2	347	0.282	0.00	0.00
3	348	0.397	0.00	26.20
4	1	0.000	0.00	0.00
Level	(b) **Epigenomics**			
1	3	0.327	0.00	0.00
2	39	0.393	0.00	578
3	39	0.328	0.00	421
4	39	0.358	0.00	264
5	39	0.290	0.00	107
6	3	0.247	0.00	0.00
7	1	0.000	0.00	0.00
8	1	0.000	0.00	0.00
9	1	0.000	0.00	0.00
Level	(c) **LIGO**			
1	191	0.024	0.01	10097
2	191	0.279	0.01	8264
3	18	0.054	0.00	174
4	191	0.066	0.01	5138
5	191	0.271	0.01	3306
6	18	0.040	0.00	43.70
Level	(d) **Montage**			
1	49	0.022	0.01	189.17
2	196	0.010	0.00	0.00
3	1	0.000	0.00	0.00
4	1	0.000	0.00	0.00
5	49	0.017	0.00	0.00
6	1	0.000	0.00	0.00
7	1	0.000	0.00	0.00
8	1	0.000	0.00	0.00
9	1	0.000	0.00	0.00
Level	(e) **SIPHT**			
1	712	3.356	0.01	53199
2	64	1.078	0.01	1196
3	128	1.719	0.00	3013
4	32	0.000	0.00	342
5	32	0.210	0.00	228
6	32	0.000	0.00	114

Table 5.4: Experiment 1: average number of tasks, and average values of imbalance metrics (HRV, HIFV, and HDV) for the 5 workflow applications (before task clustering).

CPU-intensive and runtime variations have higher impact on the performance of the application. Thus, HRB performs better than any other balancing method. When increasing the data average size, the application turns into a data-intensive application, i.e. data dependencies have higher

impact on the application's performance. HIFB captures both the workflow structure and task runtime information, which reduces data transfers between tasks and consequently yields better performance gain over the baseline execution. HDB captures the strong connections between tasks (data dependencies), while HIFB captures the weak connections (similarity in terms of structure). In some cases, HIFV is zero while HDV is less likely to be zero. Most of the LIGO branches are like the ones in Figure 2.8, however, as mentioned in Section 5.3.2, the LIGO workflow has a few branches that depend on each other as shown in Figure 5.7. Since most branches are isolated from each other, HDB initially performs well compared to HIFB. However, with the increase of average data size, the performance of HDB is more constrained by the interdependent branches, which is shown in Figure 5.12. HC has nearly constant performance despite of the average data size, due to its random merging of tasks at the same horizontal level regardless of the runtime and data dependency information.

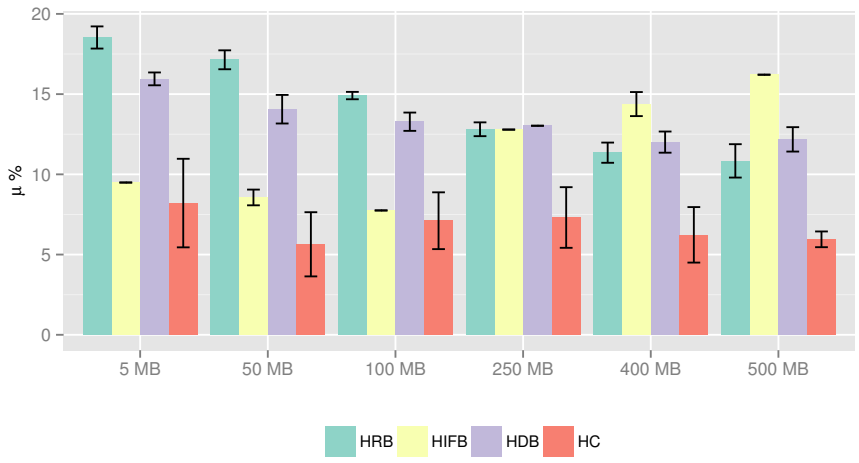

Figure 5.12: Experiment 2: performance gain (μ) over a baseline execution with different average data sizes for the LIGO workflow. The original avg. data size is 5MB.

Figures 5.13 and 5.14 show the performance gain μ when varying the number of available VMs for the LIGO workflows with an average data size of 5MB (CPU-intensive) and 500MB (data-intensive) respectively. In high contention scenarios (small number of available VMs), all methods perform similar when the application is CPU-intensive (Figure 5.13), i.e., runtime

variance and data dependency have smaller impact than the system overhead (e.g. queuing time). As the number of available resources increases, and the data size is too small, runtime variance has more impact on the application's performance, thus HRB performs better than the others. Note that as HDB captures strong connections between tasks, it is less sensitive to the runtime variations than HIFB, thus it yields better performance. For the data-intensive case (Figure 5.14), data dependencies have more impact on the performance than the runtime variation. In particular, in the high contention scenario HDB performs poor task clustering leading the system to data locality problems compared to HIFB due to the interdependent branches in the LIGO workflow. However, the method still improves the execution due to the high system overhead. Similar to the CPU-intensive case, under low contention, runtime variance increases its importance and then HRB performs better.

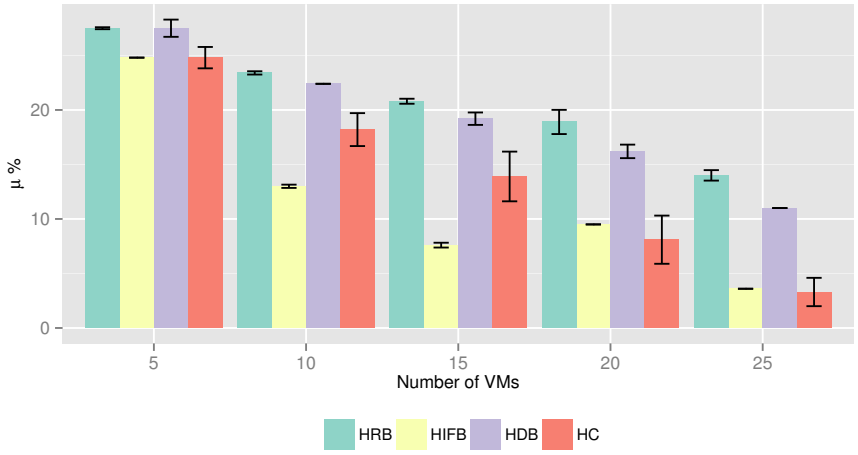

Figure 5.13: Experiment 2: performance gain (μ) over baseline execution with different number of resources for the LIGO workflow (average data size is 5MB).

Experiment 3 Figure 5.15 shows the performance gain μ for the Cybershake workflow over the baseline execution when using vertical clustering (VC) combined to our balancing methods. Vertical clustering does not aggregate any improvement to the Cybershake workflow ($\mu(VC\text{-}only) \approx 0.2\%$), because the workflow structure has no explicit pipeline (see Figure 2.10). Similarly, VC

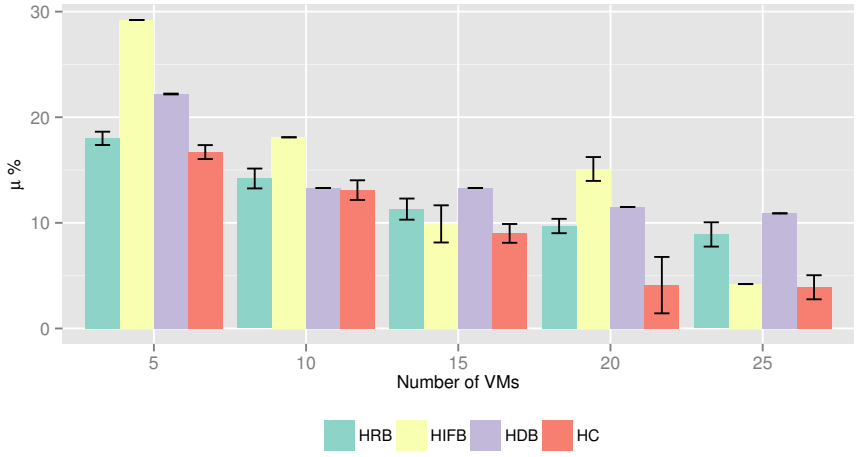

Figure 5.14: Experiment 2: performance gain (μ) over baseline execution with different number of resources for the LIGO workflow (average data size is 500MB).

does not improve the SIPHT workflow due to the lack of pipelines on its structure (Figure 2.12). Thus, results for this workflow are omitted.

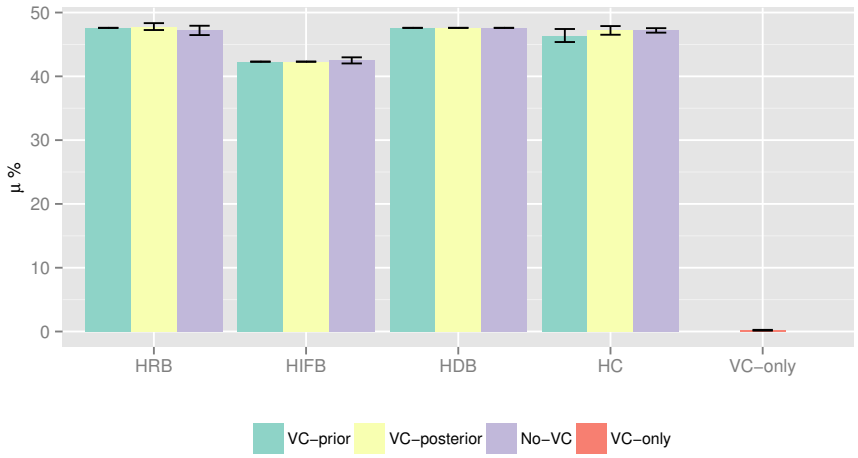

Figure 5.15: Experiment 3: performance gain (μ) for the Cybershake workflow over baseline execution when using vertical clustering (VC).

Figure 5.16 shows the performance gain μ for the Montage workflow. In this workflow, vertical clustering is often performed on the two pipelines (Figure 2.9). These pipelines are commonly single-task levels, thereby no horizontal clustering is performed on the pipelines. As

a result, whether performing vertical clustering prior or after horizontal clustering, the result is about the same. Since VC and horizontal clustering methods are independent with each other in this case, we still should do VC in combination with horizontal clustering to achieve further performance improvement.

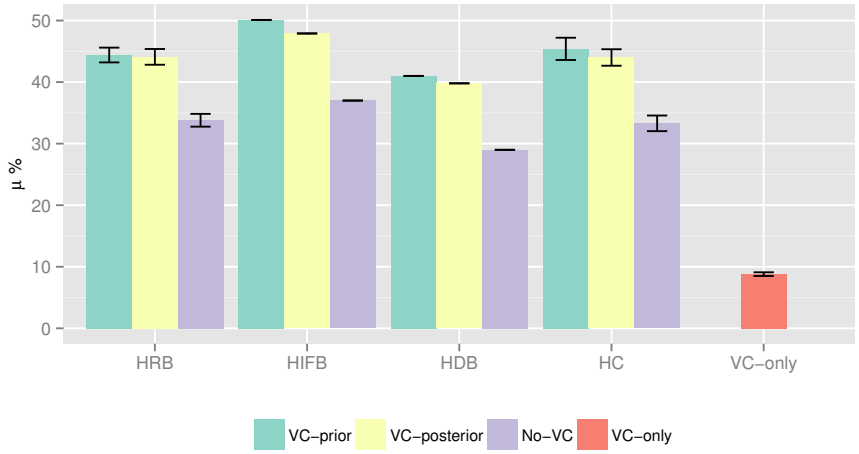

Figure 5.16: Experiment 3: performance gain (μ) for the Montage workflow over baseline execution when using vertical clustering (VC).

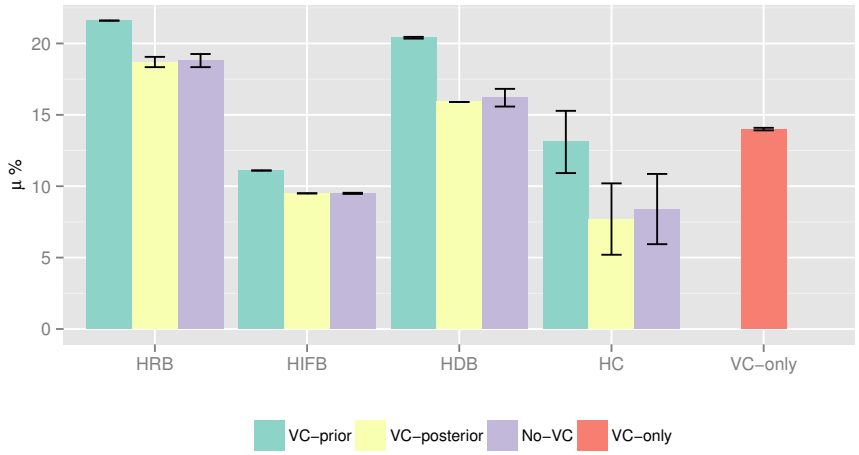

Figure 5.17: Experiment 3: performance gain (μ) for the LIGO workflow over baseline execution when using vertical clustering (VC).

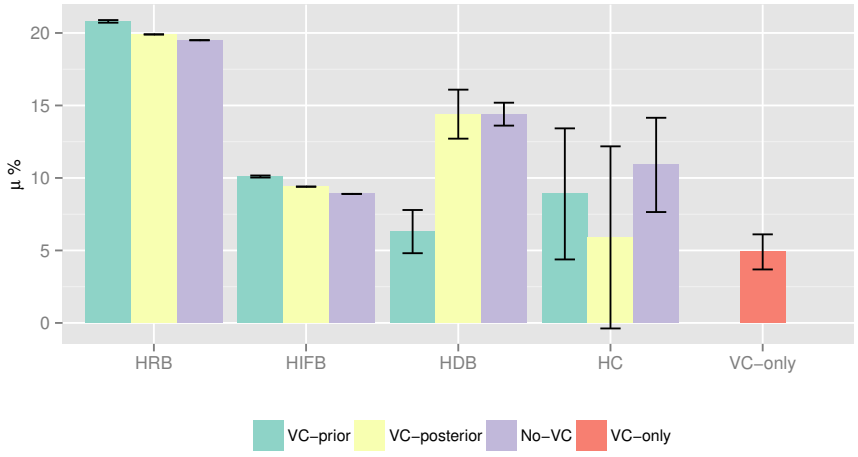

Figure 5.18: Experiment 3: performance gain (μ) for the Epigenomics workflow over baseline execution when using vertical clustering (VC).

The performance gain μ for the LIGO workflow is shown in Figure 5.17. Vertical clustering yields better performance gain when it is performed prior to horizontal clustering (*VC-prior*). The LIGO workflow structure (Figure 2.8) has several pipelines that are primarily clustered vertically and thus system overheads (e.g. queuing and scheduling times) are reduced. Furthermore, the runtime variance (HRV) of the clustered pipelines increases, thus the balancing methods, in particular HRB, can further improve the runtime performance by evenly distributing task runtimes among clustered jobs. When vertical clustering is performed *a posteriori*, pipelines are broken due to the horizontally merging of tasks between pipelines neutralizing vertical clustering improvements.

Similarly to the LIGO workflow, the performance gain μ values for the Epigenomics workflow (see Figure 5.18) are better when VC is performed *a priori*. This is due to several pipelines inherent to the workflow structure (Figure 2.11). However, vertical clustering has poorer performance if it is performed prior to the HDB algorithm. The reason is that the average task runtime of Epigenomics is much larger than other workflows as shown in Table. 2.1. Therefore, *VC-prior* generates very large clustered jobs vertically and makes it difficult for horizontal methods to improve further.

In a word, these experiments show strong connections between the imbalance metrics and the performance improvement of the balancing methods we proposed. HRV indicates the potential performance improvement for HRB. The higher HRV is, the more performance improvement HRB is likely to have. Similarly, for symmetric workflows (such as Epigenomics), their HIFV and HDV values are low and thus neither HIFB or HDB performs well.

5.5 Summary

We presented three balancing methods and two vertical clustering combination approaches to address the load balance problem when clustering workflow tasks. We also defined three imbalance metrics to quantitatively measure workflow characteristics based on task runtime variation (HRV), task impact factor (HIFV), and task distance variance (HDV).

Three experiment sets were conducted using traces from five real workflow applications. The first experiment aimed at measuring the performance gain over a baseline execution without clustering. In addition, we compared our balancing methods with three algorithms in literature. Experimental results show that our methods yield significant improvement over a baseline execution, and that they have acceptable performance when compared to the best estimated performance of the existing algorithms. The second experiment measured the influence of average data size and number of available resources on the performance gain. In particular, results show that our methods have different sensitivity to data- and computational-intensive workflows. Finally, the last experiment evaluated the interest of performing horizontal and vertical clustering in the same workflow. Results show that vertical clustering can significantly improve pipeline-structured workflows, and it is not suitable if the workflow has no explicit pipelines.

The simulation based evaluation also shows that the performance improvement of the proposed balancing algorithms (HRB, HDB and HIFB) is highly related to the metric values (HRV, HDV and HIFV) that we introduced. For example, a workflow with high HRV tends to have better performance improvement with HRB since HRB is used to balance the runtime variance.

This chapter has provided a novel approach to address the problem of load imbalance in task clustering. However, the dynamic features of modern distributed systems have brought us new challenges. For example, task clustering strategies in a faulty execution environment may fail to maintain their performance. In next chapter, we will introduce our fault tolerant clustering methods to address this issue.

Chapter 6

Fault Tolerant Clustering

Task clustering has been proven to be an effective method to reduce execution overhead and thereby the workflow makespan. However, a job composed of multiple tasks may have a greater risk of suffering from failures than a job composed of a single task. In this chapter, we demonstrate that transient failures can have a significant impact on the runtime performance of scientific workflows that use existing clustering policies that ignore failures. We optimize the workflow makespan by dynamically adjusting the clustering granularity in the presence of failures. We also propose a general task failure modeling framework and use a Maximum Likelihood Estimation based parameter estimation process to address these performance issues. We further propose three methods to improve the runtime performance of executing workflows in faulty execution environments. A trace based simulation is performed and it shows that our methods improve the workflow makespan significantly for five important applications.

6.1 Motivation

In task clustering, a clustered job consists of multiple tasks. A task is marked as failed (task failure) if it is terminated by unexpected events during the computation of this task. If a task within a job fails, this job has a job failure, even though other tasks within this job do not necessarily fail. In a faulty execution environment, there are several options for reducing the influence of workflow failures. First, one can simply retry the entire job when its computation is not successful as in the Pegasus [24], Askalon [31] and Chemomentum [77]. However, some of the tasks within the job may have completed successfully and it could be a waste of time and resources to retry all of the tasks. Second, the application process can be periodically checkpointed such that when

a failure occurs, the amount of work to be retried is limited. However, the overheads of checkpointing can limit its benefits [103]. Third, tasks can be replicated to different computational nodes to avoid failures that are related to one specific worker node [67]. However, inappropriate task clustering (and replication) parameters may cause severe performance degradation if they create long-running clustered jobs. As we will show, a long-running job that consists of many tasks has a higher job failure rate even when the inter-arrival time of failures is long.

We view the sequence of failure events as a stochastic process and study the distribution of its inter-arrival times, i.e. the time between failures. Our work is based on an assumption that the distribution parameter of the inter-arrival time is a function of the *type of task*. Tasks of the same type have the same computational program (executable file). In the five workflows we examine in this chapter, tasks at the same horizontal workflow level (defined as the longest distance from the entry task of the workflow) of the workflows have the same type. The characteristics of tasks such as the task runtime, memory peak, and disk usage are highly related to the task type [22, 42]. Samak [75] et al. have analyzed 1,329 real workflow executions across six distinct applications and concluded that the type of a task is among the most significant factors that impacted failures.

We propose two horizontal methods and one vertical methods to improve the existing clustering techniques in a faulty execution environment. The first horizontal method retries the failed tasks within a job. The second horizontal solution dynamically *adjusts the granularity or clustering size* (number of tasks in a job) according to the estimated inter-arrival time of task failures. The vertical method reduces the clustering size by half in each job retry. We assume a task-level monitoring service is available. A task-level monitor tells which tasks in a clustered job fail or succeed, while a job-level monitor only tells whether this job failed or not. The job-level fault tolerant clustering has been discussed in our prior work [16].

Compared to our prior work in [16], we add a parameter learning process to estimate the distribution of the task runtime, the system overhead, and the inter-arrival time of failures. We adopt an approach of prior and posterior knowledge based on Maximum Likelihood Estimation (MLE) that has been recently used in machine learning. Prior knowledge about the parameters is modeled as a distribution with known parameters. Posterior knowledge about the execution

is also modeled as a distribution with a known *shape parameter* and an unknown *scale parameter*. The shape parameter affects the shape of a distribution and the scale parameter affects the stretching or shrinking of a distribution. Both parameters control the characteristics of a distribution. The distribution of the prior and the posterior knowledge are in the same family if the likelihood distribution follows some specific distributions. These functions are then called *conjugate distributions*[1]. For example, if the likelihood is a Weibull distribution and we model prior knowledge as an Inverse-Gamma distribution, then the posterior is also an Inverse-Gamma distribution. This simplifies the estimation of parameters and integrates the prior knowledge and posterior knowledge gracefully. More specifically, we define the parameter learning process with only prior knowledge as the static estimation. The process with both prior knowledge and posterior knowledge is called the dynamic estimation since we update the MLE based on the information collected during the execution.

6.2 Related Work

Failure Analysis and Modeling [84] presents system characteristics such as error and failure distribution and hazard rates. They have emphasized the importance of fault tolerant design and concluded that the failure rates in modern distributed systems should not be ignored. Schroeder et al. [76] has studied the statistics of the data, including the root cause of failures, the mean time between failures, and the mean time to repair. They have verified that the inter-arrival time of task failures fits a Weibull distribution better than the lognormal and exponential distributions. Sahoo et al. [72] analyzed the empirical and statistical properties of system errors and failures from a network of heterogeneous servers. Their work concluded that the system error and failure patterns are comprised of time-varying behavior with long stationary intervals. Oppenheimer et al. [63] analyzed the causes of failures from three large-scale Internet services and they found that the configuration error was the largest single cause of failures in these services. McConnel [56] analyzed the transient errors in computer systems and showed that transient errors follow a

[1] `http://en.wikipedia.org/wiki/Conjugate_prior`

Weibull distribution. In [83, 40], the Weibull distribution is one of the best fit for the workflow traces they used. Similarly, we assume that the inter-arrival time of failures in a workflow follows a Weibull distribution. We then develop methods to improve task clustering in a faulty execution environment. An increasing number of workflow management systems are taking fault tolerance into consideration. The Pegasus workflow management system [24] has incorporated a task-level monitoring system and used job retries to address the issue of task failures. They also used provenance data to track the failure records and analyzed the causes of failures [75]. Plankensteiner et. al. [68] have surveyed fault detection, prevention and recovery techniques in current grid workflow management systems such as Askalon [31], Chemomentum [77], Escogitare [47] and Triana [85]. Recovery techniques such as replication, checkpointing, task resubmission, and task migration have been used in these systems. We are integrating the work on failure analysis with the optimization performed by task clustering. To be best of our knowledge, none of existing workflow management systems have provided such features.

Machine Learning in Workflow Optimization has been used to predict execution time [29, 13, 50, 22] and system overheads [15], and develop probability distributions for transient failure characteristics. Duan et.al. [29] used Bayesian network to model and predict workflow task runtimes. The important attributes (such as the external load, arguments etc.) are dynamically selected by the Bayesian network and fed into a radial basis function neural network to make further predictions. Ferreira da Silva et. al. [22] used regression trees to dynamically estimate task behavior including process I/O, runtime, memory peak and disk usage. Samak [74] modeled failure characteristics observed during the execution of large scientific workflows on Amazon EC2 and used a Naïve Bayes classifier to accurately predict the failure probability of jobs. Their results show that job type is the most significant factor to classify failures. We reuse the knowledge gained from prior work on failure analysis, overhead analysis and task runtime analysis. We then use prior knowledge based on Maximum Likelihood Estimation to integrate both the knowledge and runtime feedbacks and adjust the estimation accordingly.

Task Granularity Control has also been addressed in scientific workflows. For instance, Singh et al. [79] proposed a level- and label-based clustering. In level-based clustering, tasks at

the same workflow level can be clustered together. The number of clusters or tasks per cluster is specified by the user. In the label-based clustering, the user labels tasks that should be clustered together. Although their work considers data dependencies between workflow levels, it is done manually by the users, a process that is prone to errors. Recently, Ferreira da Silva et al. [32] proposed task grouping and ungrouping algorithms to control workflow task granularity in a non-clairvoyant and online context, where none or few characteristics about the application or resources are known in advance. Their work significantly reduced scheduling and queuing time overheads, but did not consider data dependencies. The low performance of *fine-grained* tasks is a common problem in widely distributed platforms where the scheduling overhead and queuing times at resources are high, such as grid and cloud systems. Several works have addressed the control of task granularity of bags of tasks. For instance, Muthuvelu et al. [59] proposed a clustering algorithm that groups bags of tasks based on their runtime—tasks are grouped up to the resource capacity. Later, they extended their work [58] to determine task granularity based on task file size, CPU time, and resource constraints. Recently, they proposed an online scheduling algorithm [57, 60] that groups tasks based on resource network utilization, user's budget, and application deadline. Ng et al. [61] and Ang et al. [87] introduced bandwidth in the scheduling framework to enhance the performance of task scheduling. Longer tasks are assigned to resources with better bandwidth. Liu and Liao [52] proposed an adaptive fine-grained job scheduling algorithm to group fine-grained tasks according to processing capacity and bandwidth of the current available resources. Although these techniques significantly reduce the impact of scheduling and queuing time overhead, they did not consider fault tolerance.

6.3 Approach

In this chapter, the *goal* is to reduce the workflow makespan in a faulty execution environment by adjusting the clustering size (k), which is defined as the number of tasks in a clustered job. In the ideal case without any failures, the task clustering size is usually equal to the number of all the parallel tasks divided by the number of available resources. Such a naïve setting assures that

the number of jobs is equal to the number of resources and the workflow can utilize the resources as much as possible. However, when transient failures exist, we claim that the clustering size should be set based on the inter-arrival time of task failures. Intuitively, if the inter-arrival time of task failure rates is short, the clustered jobs may need to be re-executed more often compared to the case without clustering. Such performance degradation will counteract the benefits of reducing scheduling overheads. In the rest of this chapter, we will show how to adjust k based on the estimated parameters of the task runtime $\boldsymbol{t}$, the system overhead $\boldsymbol{s}$ and the inter-arrival time of task failures $\boldsymbol{\gamma}$.

6.3.1 Task Failure Model

In our prior work [15], we have verified that system overheads $\boldsymbol{s}$ fits a Gamma or a Weibull distribution better than two other potential distributions (Exponential and Normal). Schroeder et. al. [76] have verified the inter-arrival time of task failures fits a Weibull distribution with a shape parameter of 0.78 better than lognormal and exponential distribution. We will reuse this shape parameter of 0.78 in our failure model. In [83, 40] Weibull, Gamma and Lognormal distribution are among the best fit to estimate task runtimes for the workflow traces they used. Without loss of generality, we choose the Gamma distribution to model the task runtime ($\boldsymbol{t}$) and the system overhead ($\boldsymbol{s}$) and use Weibull distribution to model the inter-arrival times of failures ($\boldsymbol{\gamma}$). $\boldsymbol{s}$, $\boldsymbol{t}$ and $\boldsymbol{\gamma}$ are all random variables of all tasks instead of one specific task.

Probability distributions such as Weibull and Gamma are usually described with two parameters: the *shape parameter* (ϕ) and the *scale parameter* (θ). The shape parameter affects the shape of a distribution and the scale parameter affects the stretching or shrinking of a distribution. Both of them control the characteristics of a distribution. For example, the mean of a Gamma distribution is $\phi\theta$ and the Maximum Likelihood Estimation or MLE is $(\phi - 1)\theta$.

Assume a, b are the parameters of the prior knowledge, D is the observed dataset and θ is the parameter we aim to estimate. In Bayesian probability theory, if the posterior distribution $p(\theta|D, a, b)$ is in the same family as the prior distribution $p(\theta|a, b)$, the prior and the posterior

distributions are then called conjugate distributions, and the prior distribution is called a conjugate prior for the likelihood function. For example, the Inverse-Gamma distribution is conjugate to itself (or self-conjugate) with respect to a Weibull likelihood function: if the likelihood function is Weibull, choosing an Inverse-Gamma prior over the mean will ensure that the posterior distribution is also Inverse-Gamma. This simplifies the estimation of parameters since we can reuse the prior work from other researchers [76, 40, 83, 15] in the area of failure analysis and performance analysis.

After we observe data D, we compute the posterior distribution of θ:

$$\begin{aligned} p(\theta|D,a,b) &= \frac{p(\theta|a,b) \times p(D|\theta)}{p(D|a,b)} \\ &\propto p(\theta|a,b) \times p(D|\theta) \end{aligned}$$

D can be the observed inter-arrival time of failures X, the observed task runtime RT or the observed system overheads S. $X = \{x_1, x_2, ..., x_n\}$ is the observed data of $\boldsymbol{\gamma}$ during the runtime. Similarly, we define $RT = \{t_1, t_2, ..., t_n\}$ and $S = \{s_1, s_2, ..., s_n\}$ as the observed data of $\boldsymbol{t}$ and $\boldsymbol{s}$ respectively. $p(\theta|D,a,b)$ is the posterior that we aim to compute. $p(\theta|a,b)$ is the prior, which we have already known from previous work. $p(D|\theta)$ is the likelihood.

More specifically, we model the inter-arrival time of failures ($\boldsymbol{\gamma}$) with a Weibull distribution as [76] that has a known shape parameter of ϕ_γ and an unknown scale parameter θ_γ: $\boldsymbol{\gamma} \sim W(\theta_\gamma, \phi_\gamma)$.

The conjugate pair of a Weibull distribution with a known shape parameter ϕ_γ is an Inverse-Gamma distribution, which means if the prior distribution follows an Inverse-Gamma distribution $\Gamma^{-1}(a_\gamma, b_\gamma)$ with the shape parameter as a_γ and the scale parameter as b_γ, then the posterior follows an Inverse-Gamma distribution:

$$\theta_\gamma \sim \Gamma^{-1}(a_\gamma + n, b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}) \tag{6.1}$$

The MLE (Maximum Likelihood Estimation) of the scale parameter θ_γ is:

$$MLE(\theta_\gamma) = \frac{b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}}{a_\gamma + n + 1} \tag{6.2}$$

The adoption of the MLE has two benefits: initially we do not have any data and thus the MLE is $\frac{b_\gamma}{a_\gamma + 1}$, which means it is determined by the prior knowledge; when $n \to \infty$, the MLE $\frac{\sum_{i=1}^{n} x_i^{\phi_\gamma}}{n+1} \to \overline{x^{\phi_\gamma}}$, which means it is determined by the observed data and it is close to the regularized average of the observed data. The static estimation process only utilizes the prior knowledge and the dynamic estimation process uses both the prior and the posterior knowledge.

We model the task runtime ($\boldsymbol{t}$) with a Gamma distribution as [83, 40] with a known shape parameter ϕ_t and an unknown scale parameter θ_t. The conjugate pair of Gamma distribution with a known shape parameter is also a Gamma distribution. If the prior follows $\Gamma(a_t, b_t)$, while a_t is the shape parameter and b_t is the rate parameter (or $\frac{1}{b_t}$ is the scale parameter), the posterior follows $\Gamma(a_t + n\phi_t, b_t + \sum_{i=1}^{n} t_i)$ with $a_t + n\phi_t$ as the shape parameter and $b_t + \sum_{i=1}^{n} t_i$ as the rate parameter. The MLE of θ_t is $\frac{b_t + \sum_{i=1}^{n} t_i}{a_t + n\phi_t - 1}$.

Similarly, if we model the system overhead $\boldsymbol{s}$ with a Gamma distribution with a known shape parameter ϕ_s and an unknown scale parameter θ_s, and the prior is $\Gamma(a_s, b_s)$, the MLE of θ_s is $\frac{b_s + \sum_{i=1}^{n} s_i}{a_s + n\phi_s - 1}$.

We have already assumed the task runtime, system overhead and inter-arrival time between failures are a function of task types. Since in scientific workflows, tasks at the different level (the longest distance from the entry task of the DAG to this task) are usually of different type, we model the runtime level by level. Given n independent tasks at the same level and the distribution of the task runtime, the system overhead, and the inter-arrival time of failures, we aim to reduce the cumulative runtime $\boldsymbol{M}$ of completing these tasks by adjusting the clustering size k (the number of tasks in a job). $\boldsymbol{M}$ is also a random variable and it includes the system overheads

and the runtime of the clustered job and its subsequent retry jobs if the first try fails. We also assume that task failures are independent for each worker node (but with the same distribution) without considering the failures that bring the whole system down such as a failure in the shared file system.

The runtime of a job is a random variable indicated by $\boldsymbol{d}$. A clustered job succeeds only if all of its tasks succeed. The job runtime is the sum of the cumulative task runtime of k tasks and a system overhead. We assume that the task runtime of each task is independent of each other, therefore the cumulative task runtime of k tasks is also a Gamma distribution since the sum of Gamma distributions with the same scale parameter is still a Gamma distribution. We also assume the system overhead is independent of all the task runtimes and it has the same scale parameter ($\theta_{ts} = \theta_t = \theta_s$) with the task runtime. In practice, if the task runtime and the system overhead have different scale parameters, we can always normalize them. For a Gamma distribution $X \sim \Gamma(\phi, \theta)$ we have $cX \sim \Gamma(\phi, c\theta)$. The job runtime (irrespective of whether it succeeds or fails) is:

$$\boldsymbol{d} \sim \Gamma(k\phi_t + \phi_s, \theta_{ts}) \tag{6.3}$$

$$MLE(\boldsymbol{d}) = (k\phi_t + \phi_s - 1)\theta_{ts} \tag{6.4}$$

Let the retry time of clustered jobs to be N. The process to run and retry a job is a Bernoulli trial with only two results: success or failure. Once a job fails, it will be re-executed until it is eventually completed successfully (since we assume the failures are transient). For a given job runtime d_i, by definition:

$$N_i = \frac{1}{1 - F(d_i)} = \frac{1}{e^{-(\frac{d_i}{\theta_\gamma})^{\phi_\gamma}}} = e^{(\frac{d_i}{\theta_\gamma})^{\phi_\gamma}} \tag{6.5}$$

$F(x)$ is the CDF (Cumulative Distribution Function) of $\boldsymbol{\gamma}$. The time to complete d_i successfully in a faulty execution environment is

$$M_i = d_i N_i = d_i e^{(\frac{d_i}{\theta_\gamma})^{\phi_\gamma}} \tag{6.6}$$

Equation 6.6 has involved two distributions $\boldsymbol{d}$ and θ_γ (ϕ_γ is known). From Equation 6.1, we have:

$$\frac{1}{\theta_\gamma} \sim \Gamma(a_\gamma + n, \frac{1}{b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}}) \tag{6.7}$$

$$MLE(\frac{1}{\theta_\gamma}) = \frac{a_\gamma + n - 1}{b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}} \tag{6.8}$$

M_i is a monotonic increasing function of both d_i and $\frac{1}{\theta_\gamma}$, and the two random variables are independent of each other, therefore:

$$MLE(M_i) = MLE(d_i) e^{(MLE(d_i) MLE(\frac{1}{\theta_\gamma}))^{\phi_\gamma}} \tag{6.9}$$

Equation 6.9 means to attain $MLE(M_i)$, we just need to attain $MLE(d_i)$ and $MLE(\frac{1}{\theta_\gamma})$ at the same time. In both dimensions (d_i and $\frac{1}{\theta_\gamma}$), M_i is a Gamma distribution and each M_i has the same distribution parameters, therefore:

$$
\begin{aligned}
\boldsymbol{M} &= \frac{1}{r}\sum_{i=1}^{\frac{n}{k}} M_i \sim \Gamma \\
MLE(\boldsymbol{M}) &= \frac{n}{rk} MLE(M_i) \qquad (6.10) \\
&= \frac{n}{rk} MLE(d_i) e^{(MLE(d_i)MLE(\frac{1}{\theta_\gamma}))^{\phi_\gamma}} \qquad (6.11)
\end{aligned}
$$

r is the number of resources. In this chapter, we consider a compute cluster as a homogeneous cluster, which is usually true in dedicated clusters and clouds.

Let k^* be the optimal clustering size that minimizes Equation 6.10. arg min stands for the argument (k) of the minimum[2], that is to say, the k such that $MLE(\boldsymbol{M})$ attains its minimum value.

$$
k^* = \arg min\{MLE(\boldsymbol{M})\} \qquad (6.12)
$$

It is difficult to find a analytical solution of k^*. However, there are a few constraints that can simplify the estimation of k^*: (*i*) k can only be an integer in practice; (*ii*) $MLE(\boldsymbol{M})$ is continuous and has one minimum. Methods such as Newton's method can be used to find the minimal $MLE(\boldsymbol{M})$ and the corresponding k. Figure 6.1 shows an example of $MLE(\boldsymbol{M})$ using static estimation with a low task failure rate (θ_γ = 40s), a medium task failure rate (θ_γ = 30s) and a high task failure rate (θ_γ = 20s). Other parameters are n = 50, θ_t = 5 sec, and θ_s = 50 sec and all the shape parameters are 2 for simplicity. These parameters are close to the parameters of tasks at the level of mProjectPP in the Montage workflow (Section 6.4). Figure 6.2 shows the relationship between the optimal clustering size (k^*) and θ_γ, which is a non-decreasing function. The optimal clustering size (marked with red dots in Figure 6.2) when $\theta_\gamma = 20, 30, 40$ is 4,5

[2]Wiki: http://en.wikipedia.org/wiki/Arg_max

and 6 respectively. It is consistent with our expectation since the longer the inter-arrival time of failures is, the lower the task failure rate is. With a lower task failure rate, a larger k assures that we reduce system overheads without retrying the tasks too many times.

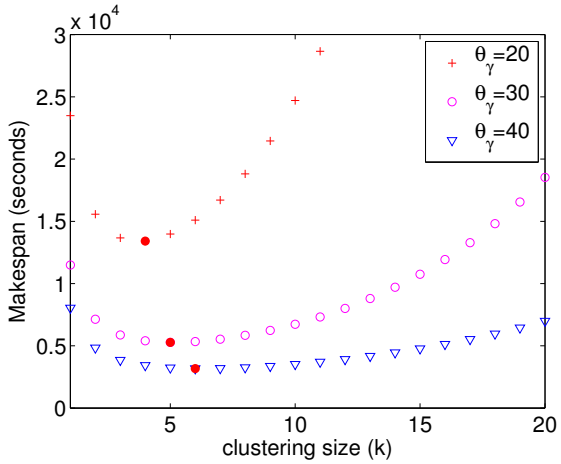

Figure 6.1: Makespan with different clustering size and θ_γ. (n = 1000, r = 20, θ_t = 5 sec, θ_s = 50 sec). The red dots are the minimums.

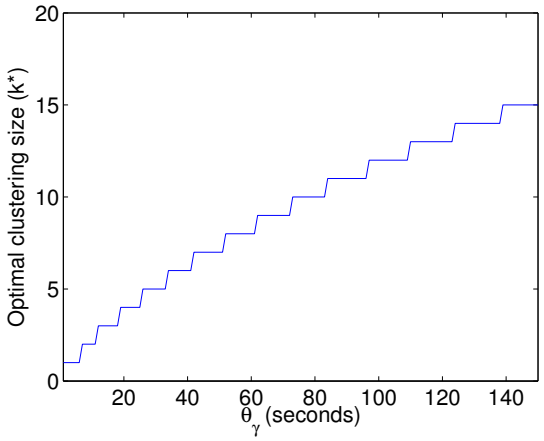

Figure 6.2: Optimal clustering size (k*) with different θ_γ (n = 1000, r = 20, θ_t = 5 sec, θ_s = 50 sec)

From this theoretic analysis, we conclude that (*i*) the longer the inter-arrival time of failures is, the better runtime performance the task clustering has; (*ii*), adjusting the clustering size according to the detected inter-arrival time can improve the runtime performance.

6.3.2 Fault Tolerant Clustering Methods

To improve the fault tolerance from the point of view of clustering, we propose three methods: Dynamic Reclustering (*DR*), Selective Reclustering (*SR*) and Vertical Reclustering (*VR*). In the experiments, we compare the performance of our fault tolerant clustering methods to an existing version of Horizontal Clustering (HC) [79] technique. In this subsection, we first briefly describe these algorithms.

Horizontal Clustering (*HC*). Horizontal Clustering (HC) merges multiple tasks that are at the same horizontal level of the workflow. The clustering granularity (number of tasks within a cluster) of a clustered job is controlled by the user, who defines either the number of tasks per clustered job (*clusters.size*), or the number of clustered jobs per horizontal level of the workflow (*clusters.num*). This algorithm has been implemented and used in Pegasus [79]. For simplicity, we set *clusters.num* to be the same as the number of available resources. In our prior work [20, 21], we have compared the runtime performance with different clustering granularity. The pseudocode of the HC technique is shown in Algorithm 8. The Clustering and Merge Procedures are called in the initial task clustering process while the Reclustering Procedure is called when there is a failed job detected by the monitoring system.

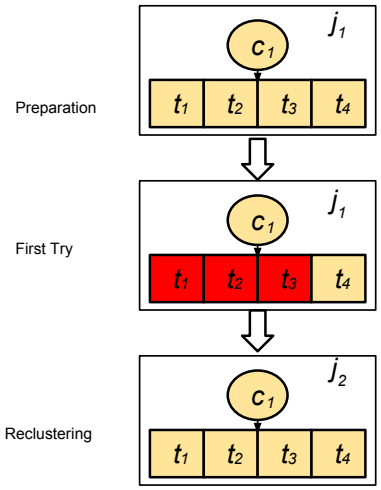

Figure 6.3: Horizontal Clustering (red boxes are failed tasks)

Algorithm 8 Horizontal Clustering algorithm.

Require: W: workflow; C: max number of tasks per job defined by *clusters.size* or *clusters.num*

```
1:  procedure CLUSTERING(W, C)
2:      for level < depth(W) do
3:          TL ← GETTASKSATLEVEL(W, level)        ▷ Partition W based on depth
4:          CL ← MERGE(TL, C)                     ▷ Returns a list of clustered jobs
5:          W ← W − TL + CL                       ▷ Merge dependencies as well
6:      end for
7:  end procedure
8:  procedure MERGE(TL, C)
9:      J ← {}                                    ▷ An empty job
10:     CL ←{}                                    ▷ An empty list of clustered jobs
11:     while TL is not empty do
12:         J.add (TL.pop(C)                      ▷ Pops C tasks that are not merged
13:         CL.add( J)
14:     end while
15:     return CL
16: end procedure
17: procedure RECLUSTERING(J)                     ▷ J is a failed job
18:     J_new ← COPYOF(J)                         ▷ Copy Job J
19:     W ← W + J_new                             ▷ Re-execute it
20: end procedure
```

Figure 6.3 shows an example where the initial clustering size is 4 and as a result there are four tasks in a clustered job. During execution, three out of these tasks (t_1, t_2, t_3) fail. HC will keep retrying all of the four tasks in the following try until all of them succeed. Such a retry mechanism has been implemented and is used in Pegasus [79].

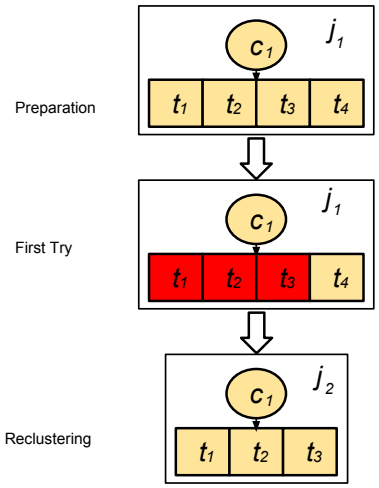

Figure 6.4: Selective Reclustering (red boxes are failed tasks)

Selective Reclustering (*SR*). HC does not adjust the clustering size even when it continuously sees many failures. We further improve the performance of the workflow execution with Selective Reclustering that selects the failed tasks in a clustered job and merges them into a new clustered job. SR is different from HC in that HC retries all tasks of a failed job even though some of the tasks have succeeded.

Figure 6.4 shows an example of SR. In the first try, there are four tasks and three of them (t_1, t_2, t_3) have failed. One task (t_4) succeeds and exits. Only the three failed tasks are merged again into a new clustered job j_2 and the job is retried. This approach does not intend to adjust the clustering size, although the clustering size will become smaller and smaller after each retry since there are fewer and fewer tasks in a clustered job. In this case, the clustering size has decreased from 4 to 3. However, the optimal clustering size may not be 3, which limits the workflow performance if the θ_γ is small and k should be decreased as much as possible. The advantage of SR is that it is simple to implement and can be incorporated into existing workflow management systems without much loss of efficiency as shown in Section 6.4. It also serves as a comparison with the Dynamic Reclustering approach that we propose below. Algorithm 9 shows the pseudocode of SR. The Clustering and Merge procedures are the same as those in HC.

Algorithm 9 Selective Reclustering algorithm.

```
Require: W: workflow; C: max number of tasks per job defined by clusters.size or clusters.num
 1: procedure RECLUSTERING(J)                                   ▷ J is a failed job
 2:     TL ← GETTASKS(J)
 3:     J_new ←{}                                               ▷ An empty job
 4:     for all Task t in TL do
 5:         if t is failed then
 6:             J_new.add (t)
 7:         end if
 8:     end for
 9:     W ← W + J_new                                           ▷ Re-execute it
10: end procedure
```

Dynamic Reclustering (*DR*). Selective Reclustering does not analyze the clustering size, rather it uses a self-adjusted approach to reduce the clustering size if the failure rate is too high. However, it is blind about the optimal clustering size and the actual clustering size may be larger or smaller than the optimal clustering size. We then propose the second method, Dynamic

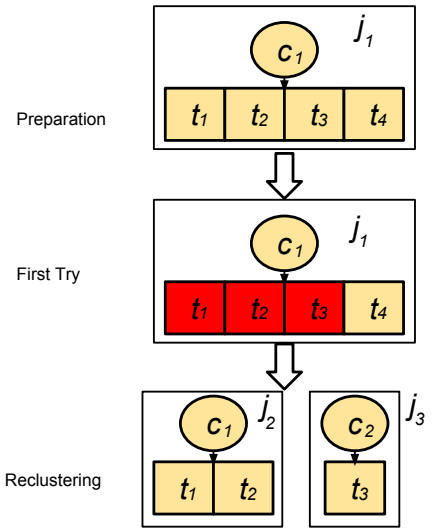

Figure 6.5: Dynamic Reclustering (red boxes are failed tasks)

Reclustering. In DR, only failed tasks are merged into new clustered jobs and the clustering size is set to be k^* according to Equation 6.12.

Figure 6.5 shows an example where the initial clustering size is 4 and thereby there are four tasks in a clustered job at the beginning. At the first try, three tasks within a clustered job have failed. Therefore we have only three tasks to retry and further we need to decrease the clustering size (in this case it is 2) accordingly. We end up with two new jobs j_2 (that has t_1 and t_2) and j_3 that has t_3. Algorithm 10 shows the pseudocode of DR. The Clustering and Merge procedures are the same as those in HC.

Algorithm 10 Dynamic Reclustering algorithm.

Require: W: workflow; C: max number of tasks per job defined by *clusters.size* or *clusters.num*

```
1:  procedure RECLUSTERING(J)                          ▷ J is a failed job
2:      TL ← GETTASKS(J)
3:      J_new ←{}
4:      for all Task t in TL do
5:          if t is failed then
6:              J_new.add (t)
7:          end if
8:          if J_new.size() > k* then
9:              W ← W + J_new
10:             J_new ←{}
11:         end if
12:     end for
13:     W ← W + J_new                                  ▷ Re-execute it
14: end procedure
```

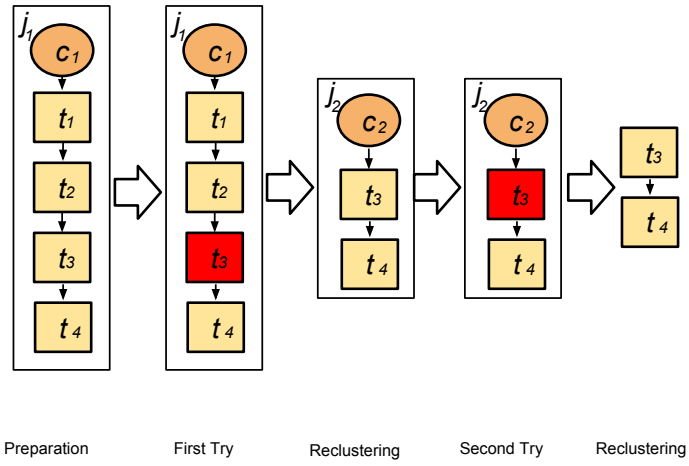

Figure 6.6: Vertical Reclustering (red boxes are failed tasks)

Vertical Reclustering (*VR*). VR is an extension of Vertical Clustering. Similar to Selective Reclustering, Vertical Reclustering only retries tasks that have failed or are not completed. If there is a failure detected, we decrease k by half and recluster the tasks. In Figure 6.6, if there is no assumption of failures initially, we put all the tasks (t_1, t_2, t_3, t_4) from the same pipeline into a clustered job. t_3 fails at the first try assuming it is a failure-prone task (its θ_γ is short). VR retries only the failed tasks (t_3) and tasks that are not completed (t_4) and merges them again into a new job j_2. In the second try, j_2 fails and we divide it into two tasks (t_3 and t_4). Since the clustering size is already 1, VR performs no vertical clustering anymore and would continue retrying t_3 and t_4 (but still following their data dependency) until they succeed. Algorithm 11 shows the pseudocode of VR.

6.4 Results and Discussion

In this section, we evaluate our methods with five workflows, whose runtime information is gathered from real execution traces. The simulation-based approach allows us to control system parameters such as the inter-arrival time of task failures in order to clearly demonstrate the reliability of the algorithms. The five widely used scientific workflow applications are used in

Algorithm 11 Vertical Reclustering algorithm.

Require: W: workflow;
1: **procedure** CLUSTERING(W)
2: **for** $level < depth(W)$ **do**
3: $TL \leftarrow$ GETTASKSATLEVEL($W, level$) ▷ Partition W based on depth
4: $CL, TL_{merged} \leftarrow$ MERGE(TL) ▷ Returns a list of clustered jobs
5: $W \leftarrow W - TL_{merged} + CL$ ▷ Merge dependencies as well
6: **end for**
7: **end procedure**
8: **procedure** MERGE(TL)
9: $TL_{merged} \leftarrow TL$ ▷ All the tasks that have been merged
10: $CL \leftarrow \{\}$ ▷ An empty list of clustered jobs
11: **for all** t in TL **do**
12: $J \leftarrow \{t\}$
13: **while** t has only one child t_{child} and t_{child} has only one parent **do**
14: J.add (t_{child})
15: $TL_{merged} \leftarrow TL_{merged} + t_{child}$
16: $t \leftarrow t_{child}$
17: **end while**
18: CL.add(J)
19: **end for**
20: **return** CL, TL_{merged}
21: **end procedure**
22: **procedure** RECLUSTERING(J) ▷ J is a failed job
23: $TL \leftarrow$ GETTASKS(J)
24: $k^* \leftarrow J.size()/2$ ▷ Reduce the clustering size by half
25: $J_{new} \leftarrow \{\}$
26: **for all** Task t in TL **do**
27: **if** t is failed or not completed **then**
28: J_{new}.add (t)
29: **end if**
30: **if** $J_{new}.size() > k^*$ **then**
31: $W \leftarrow W + J_{new}$
32: $J_{new} \leftarrow \{\}$
33: **end if**
34: **end for**
35: $W \leftarrow W + J_{new}$ ▷ Re-execute it
36: **end procedure**

the experiments: LIGO Inspiral analysis [48], Montage [5], CyberShake [35], Epigenomics [91], and SIPHT [80]. Their main characteristics and structures are shown in Section 2.3.

6.4.1 Experiment conditions

We adopt a trace-based simulation approach, where we extended our WorkflowSim [19] simulator with the fault tolerant clustering methods to simulate a controlled distributed environment. The simulated computing platform is composed by 20 single homogeneous core virtual machines

(worker nodes), which is the quota per user of some typical distributed environments such as Amazon EC2 [1] and FutureGrid [33]. Each simulated virtual machine (VM) has 512MB of memory and the capacity to process 1,000 million instructions per second. The default network bandwidth is 15MB according to the real environment in FutureGrid from where our traces were collected. By default, we merge tasks at the same horizontal level into 20 clustered jobs initially, which is a simple selection of granularity control of the strength of task clustering. The study of granularity size has been done in [21], which shows such selection is acceptable.

We collected workflow execution traces [42, 15] (including overhead and task runtime information) from real runs (executed on FutureGrid and Amazon EC2) of the scientific workflow applications described in Section 2.3. The traces are used to feed the Workflow Generator toolkit [96] to generate synthetic workflows. This allows us to perform simulations with several different configurations under controlled conditions. The toolkit uses the information gathered from actual scientific workflow executions to generate synthetic workflows resembling those used by real world scientific applications. The number of inputs to be processed, the number of tasks in the workflow, and their composition determine the structure of the generated workflow. Such an approach of traced based simulation allows us to utilize real traces and vary the system setting (i.e., the inter-arrival time of failures) and workflow (i.e., avg. task runtime) to fully explore the performance of our fault tolerant clustering algorithms.

Three sets of experiments are conducted. Experiment 1 evaluates the performance of our fault tolerant clustering methods (DR, VR, and SR) over a baseline execution (HC) that is not fault tolerant for the five workflows. The goal of the experiment is to identify conditions where each method works best and worst. In addition, we also evaluate the performance improvement with different θ_γ (the inter-arrival time of task failures). The range of θ_γ is chosen from 10x to 1x of the average task runtime such that the workflows do not run forever and we can visualize the performance difference better.

Experiment 2 evaluates the performance impact of the variation of the average task runtime per level (defined as the average of all the tasks per level) and the average system overheads per level for one scientific workflow application (CyberShake). In particular, we are interested in the

performance of DR based on the results of Experiment 1 and we use $\theta_\gamma = 100$ since it illustrates well the difference between the four methods. The original average task runtime of all the tasks of the CyberShake workflow is about 23 seconds as shown in Table 2.1. In this experiment, we multiply the average task runtime by a multiplier from 0.5 to 1.3. The scale parameter of the system overheads (θ_s) is 50 seconds originally based on our traces and we multiply the system overheads by a multiplier from 0.2 to 1.8.

Experiment 3 evaluates the performance of dynamic estimation and static estimation. In the static estimation process, we only use the prior knowledge to estimate the MLEs of θ_t, θ_s and θ_γ. In the dynamic estimation process, we also leverage the runtime data collected during the execution and update the MLEs respectively.

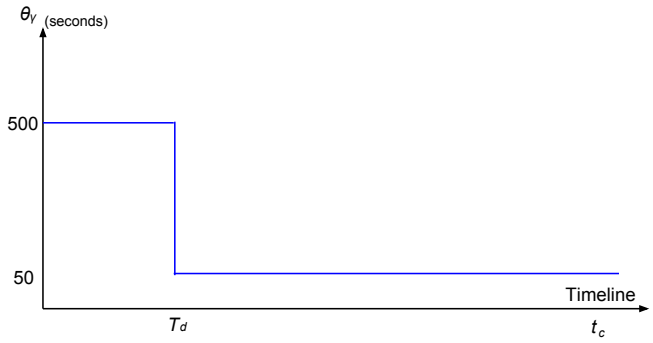

Figure 6.7: A Step Function of θ_γ. t_c is the current time and T_d is the moment θ_γ changes from 500 to 50 seconds

$$\theta_\gamma(t_c) = \begin{cases} 50 & \text{if } t_c \geq T_d \\ 500 & \text{if } 0 < t_c < T_d \end{cases} \tag{6.13}$$

In this experiment, we use two sets of θ_γ function. The first one is a step function, in which we decrease θ_γ from 500 seconds to 50 seconds at time T_d to simulate the scenario where there are more failures coming than expected. We evaluate the performance difference of dynamic estimation and static estimation while $1000 \leq T_d \leq 5000$ based on the estimation of the workflow makespan. The function is shown in Figure 6.7 and Equation 6.13, while t_c is the current time. Theoretically speaking, the later we change θ_γ, the less the reclustering is influenced by

the estimation error and thus the smaller the workflow makespan is. There is one special case when $T_d \rightarrow 0$, which means the prior knowledge is wrong at the very beginning. The second one is a pulse wave function, which the amplitude alternates at a steady frequency between fixed minimum (50 seconds) to maximum (500 seconds) values. The function is shown in Figure 6.8 and Equation 6.14. T_c is the period and τ is the duty cycle of the oscillator. It simulates a scenario where the failures follow a periodic pattern [98] that has been found in many failure traces obtained from production distributed systems. We vary T_c from 1000 seconds to 10000 seconds based on the estimation of workflow makespan and τ from $0.1T_c$ to $0.5T_c$.

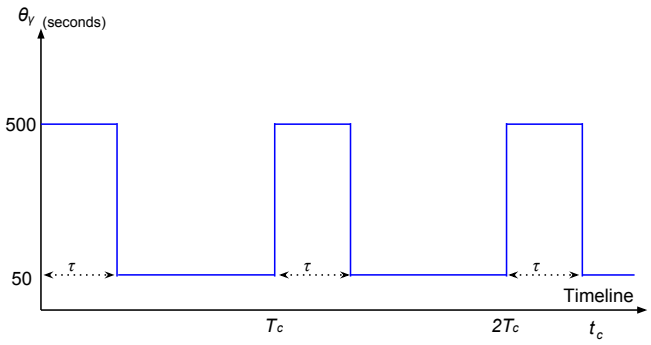

Figure 6.8: A Pulse Function of θ_γ. t_c is the current time and T_c is the period of the wave. τ is the width of the pulse.

$$\theta_\gamma(t_c) = \begin{cases} 500 & \text{if } 0 < t_c \leq \tau \\ 50 & \text{if } \tau < t_c < T_c \end{cases} \tag{6.14}$$

Table 6.1 summarizes the clustering methods to be evaluated in our experiments. In our experiments, our algorithms take less than 10ms to do the reclustering for each job and thereby they are highly efficient even for large-scale workflows.

Abbreviation	Method
DR	Dynamic Reclustering
SR	Selective Reclustering
VR	Vertical Reclustering
HC	Horizontal Clustering

Table 6.1: Methods to Evaluate in Experiements

6.4.2 Results and discussion

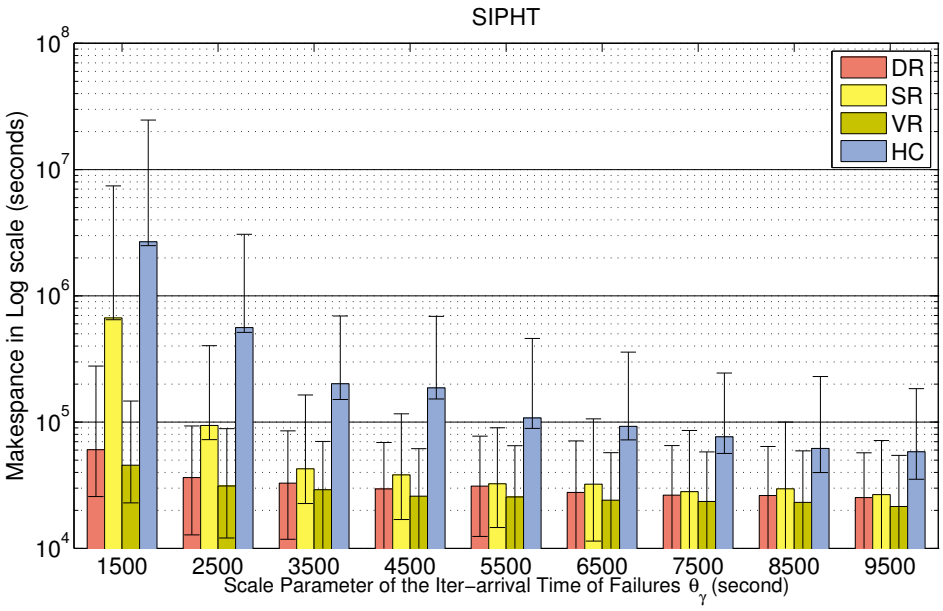

Figure 6.9: Experiment 1: SIPHT Workflow

Experiment 1 Figure 6.9, 6.10, 6.11, 6.12 and 6.13 show the performance of the four reclustering methods (DR, SR, VR and HC) with five workflows respectively. We draw conclusions:

1). DR, SR and VR have significantly improved the makespan compared to HC in a large scale. By decreasing of the inter-arrival time (θ_γ) and consequently generating more task failures, the performance difference becomes more significant.

2). Among the three methods, DR and VR perform consistently better than SR, which fail to improve the makespan when θ_γ is small. The reason is that SR does not adjust k according to the occurrence of failures.

3). The performance of VR is highly related to the workflow structure and the average task runtime. For example, according to Figure 2.11 and Table 2.1, we know that the Epigenomics workflow has a long task runtime (around 50 minutes) and the pipeline length is 4. It means vertical clustering creates really long jobs ($\sim 50 \times 4 = 200$ minutes) and thereby VR is more sensitive to the decrease of γ. As indicated in Figure 6.10, the makespan increases more significantly with the decrease of θ_γ than other workflows. In comparison, the CyberShake workflow

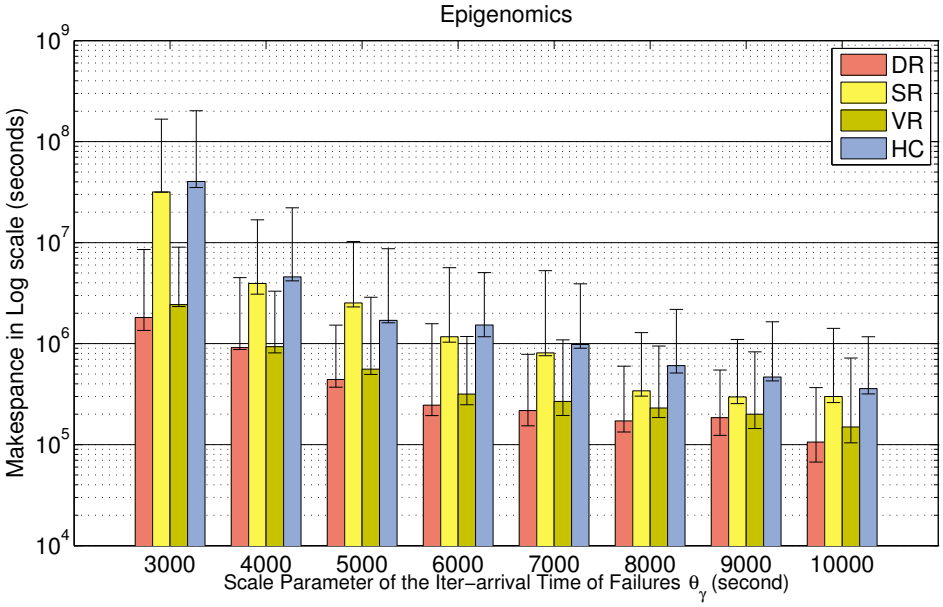

Figure 6.10: Experiment 1: Epigenomics Workflow

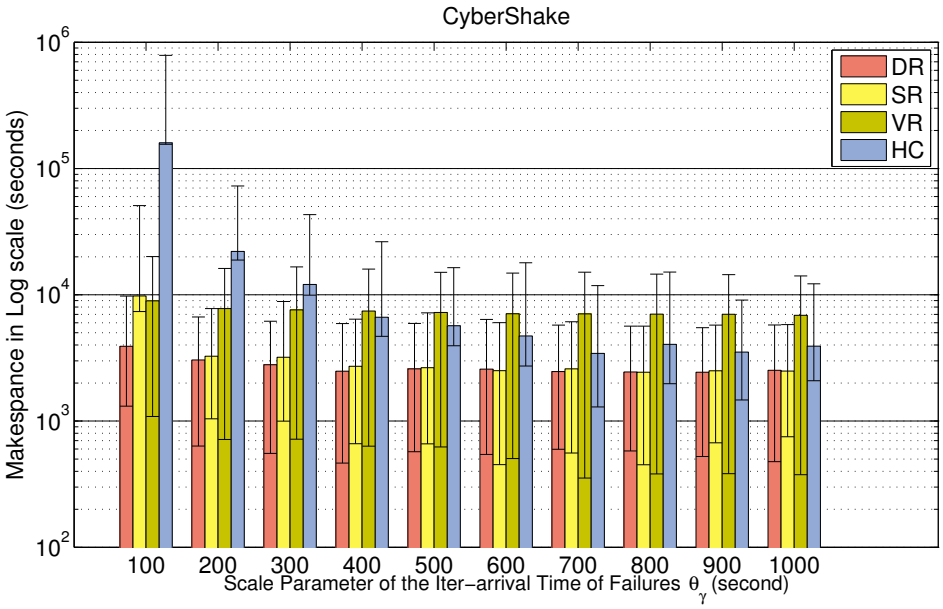

Figure 6.11: Experiment 1: CyberShake Workflow

does not leave much space for vertical clustering methods to improve the makespan since it does not have many pipelines as shown in Figure 2.10. In addition, the average task runtime of the CyberShake workflow is relatively short (around 23 seconds). Compared to horizontal clustering

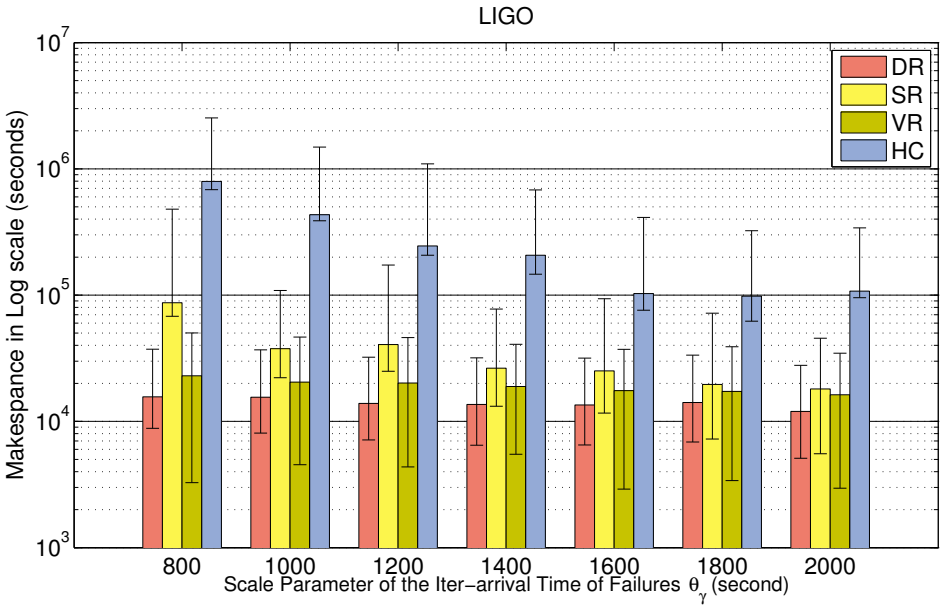

Figure 6.12: Experiment 1: LIGO Workflow

methods such as HC, SR and DR, vertical clustering does not generate long jobs and thus the performance of VR is less sensitive to θ_γ.

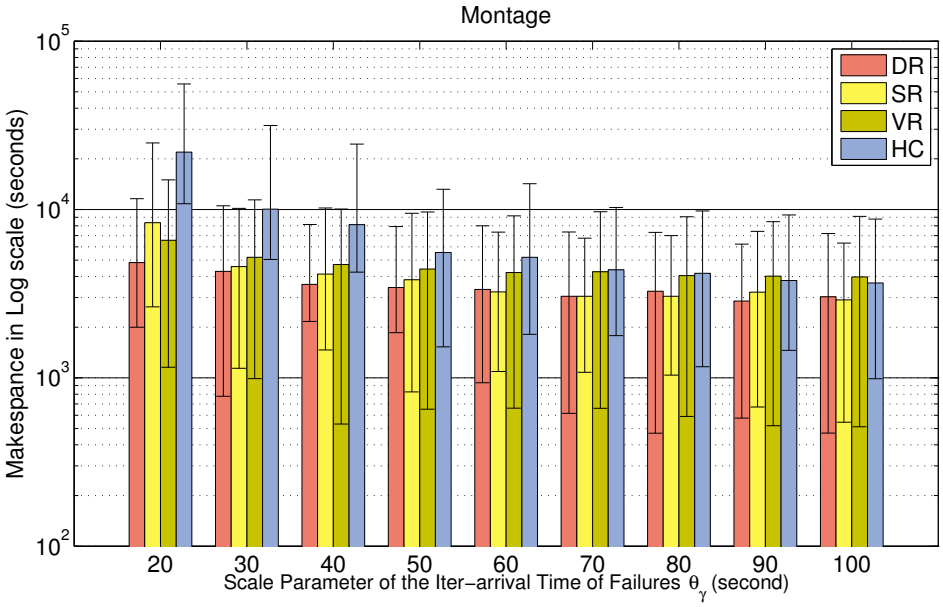

Figure 6.13: Experiment 1: Montage Workflow

Experiment 2

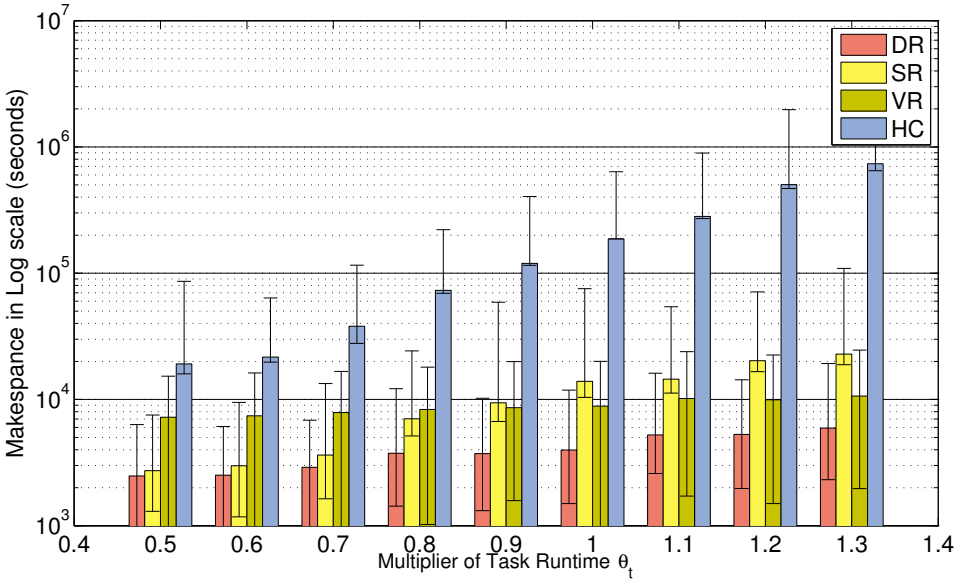

Figure 6.14: Experiment 2: Influence of Varying Task Runtime on Makespan (CyberShake)

Figure 6.14 shows the performance of our methods with different multiplier of θ_t for the CyberShake workflow. We can see that with the increase of the multiplier, the makespan of the workflow increases significantly (increase from a scale of 10^4 to $\sim 10^6$), particularly for HC. The reason is HC is not fault tolerant and it is highly sensitive to the increase of task runtime. While for DR, the reclustering process dynamically adjusts the clustering size based on the estimation of task runtime and thus the performance of DR is more stable.

Figure 6.15 shows the results with different multiplier of θ_s for the CyberShake workflow. Similarly, we can see that with the increase of the multiplier, the makespan increases for all the methods and DR performs best. However, the increase is less significant than that in Figure 6.14. The reason is we may have multiple tasks in a clustered job but only one system overhead per job.

Experiment 3

Figure 6.16 further evaluates the performance of the dynamic estimation and static estimation for the CyberShake workflow with a step function of θ_γ. The reclustering method used in this

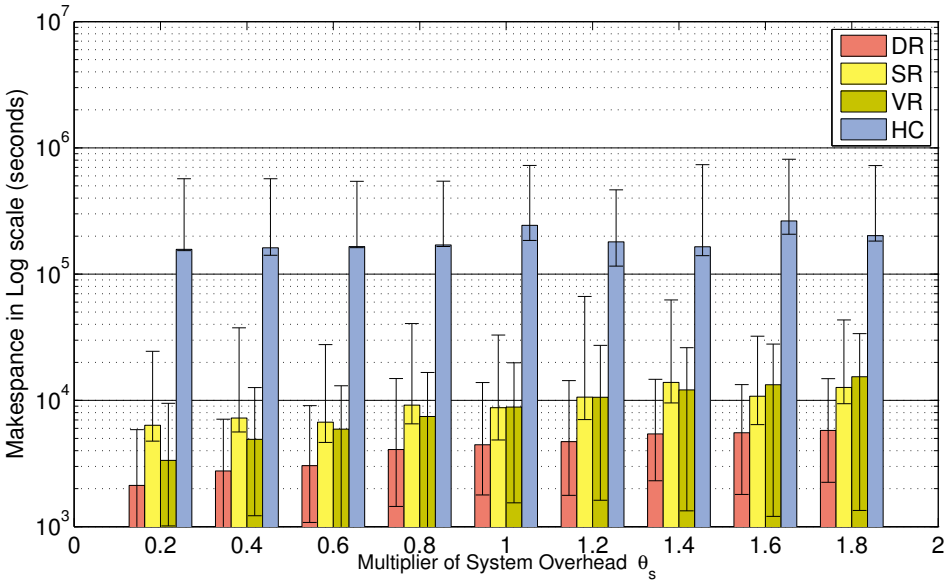

Figure 6.15: Experiment 2: Influence of Varying System Overhead on Makespan (CyberShake)

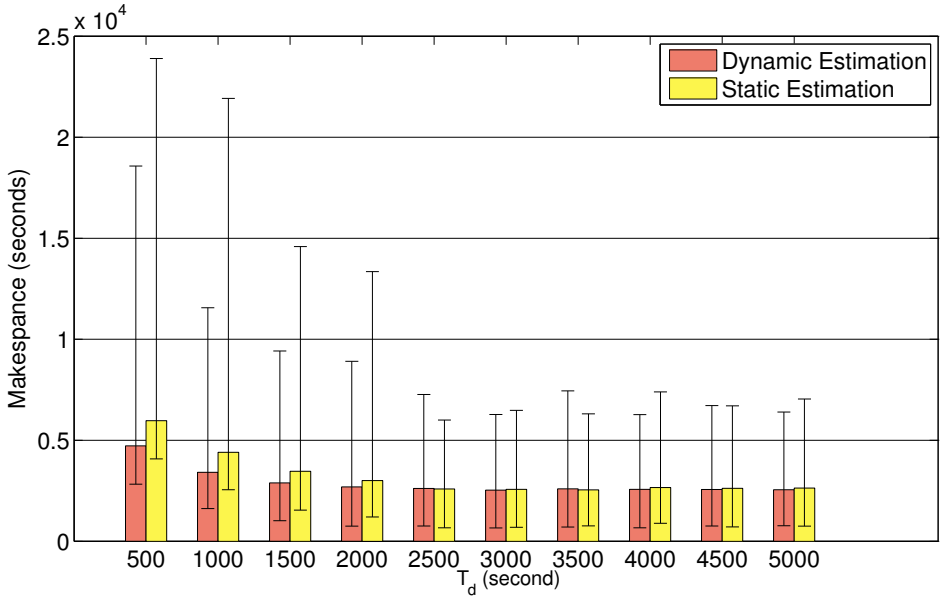

Figure 6.16: Experiment 3: Static Estimation vs. Dynamic Estimation (CyberShake, Step Function)

experiment is DR since it performs best in the last two experiments. In this experiment, we use a step signal and change the inter-arrival time of failures (θ_γ) from 500 seconds to 50 seconds at

T_d. We can see that: 1). with the increase of T_d, both makespan decrease since the change of θ_γ has less influence on the makespan and there is a lower failure rate on average; 2). Dynamic estimation improves the makespan by up to 22.4% compared to the static estimation. The reason is the dynamic estimation process is able to update the MLEs of θ_γ and decrease the clustering size while the static estimation process does not.

For the pulse function of θ_γ, we use $\tau = 0.1T_c, 0.3T_c, 0.5T_c$. Figure 6.17, 6.18 and 6.19 show the performance difference of dynamic estimation and static estimation respectively. When $\tau = 0.1T_c$, DR with dynamic estimation improves the makespan by up to 25.7% compared to case with static estimation. When $\tau = 0.3T_c$, the performance difference between dynamic estimation and static estimation is up to 27.3%. We can also see that when T_c is small (i.e., $T_c = 1000$), the performance difference is not significant since the inter-arrival time of failures changes frequently and the dynamic estimation process is not able to update swiftly. While when T_c is 10000, the performance difference is not significant neither since the period is too long and the workflow has completed successfully. When $\tau = 0.5T_c$, the performance different between dynamic estimation and static estimation is up to 9.1%, since the high θ_γ and the low θ_γ have equal influence on the failure occurrence.

6.5 Summary

In this chapter, we model transient failures in a distributed environment and assess their influence of task clustering. We propose three dynamic clustering methods to improve the fault tolerance of task clustering and apply them to five widely used scientific workflows. From our experiments, we conclude that the three proposed methods improve the makespan significantly compared to an existing algorithm widely used in workflow management systems. In particular, our Dynamic Reclustering method performs best among the three methods since it can adjust the clustering size based on the Maximum Likelihood Estimation of task runtime, system overheads and the inter-arrival time of failures. Our Vertical Reclustering method significantly improves the performance for workflows that have a short task runtime. Our dynamic estimation using

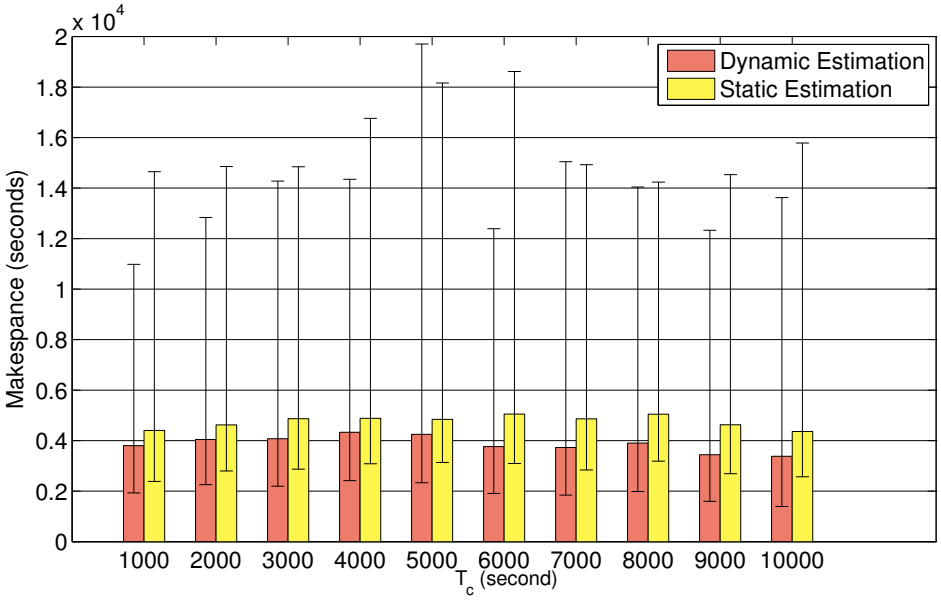

Figure 6.17: Experiment 3: Static Estimation vs. Dynamic Estimation (CyberShake, Pulse Function ($\tau = 0.1T_c$))

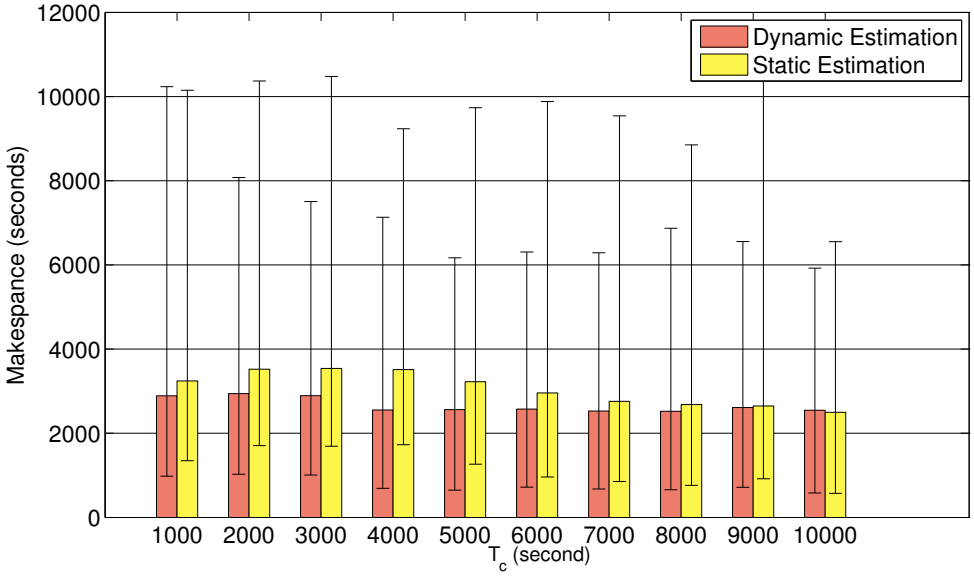

Figure 6.18: Experiment 3: Static Estimation vs. Dynamic Estimation (CyberShake, Pulse Function ($\tau = 0.3T_c$))

on-going data collected from the workflow execution can further improve the fault tolerance in a dynamic environment where the inter-arrival time of failures is fluctuant.

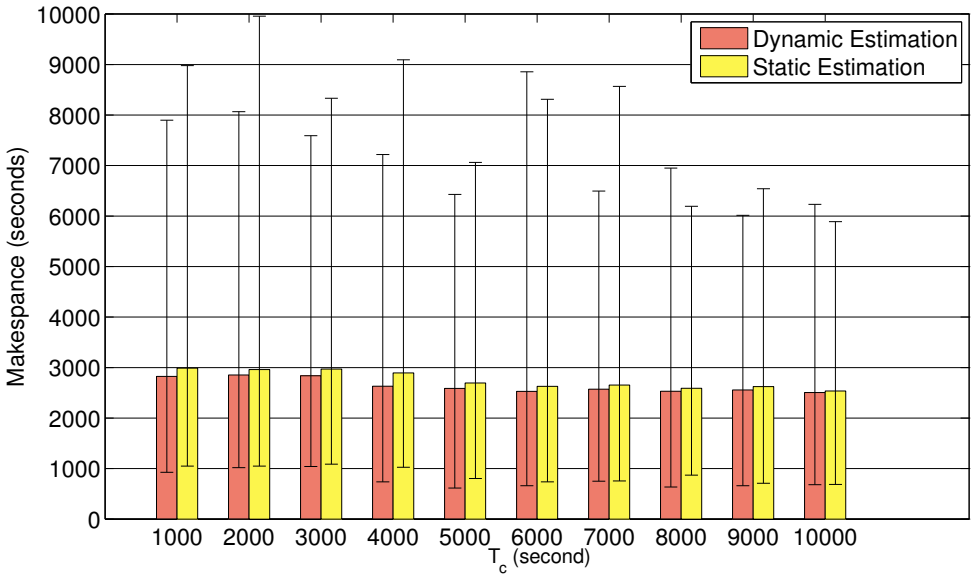

Figure 6.19: Experiment 3: Static Estimation vs. Dynamic Estimation (CyberShake, Pulse Function ($\tau = 0.5T_c$))

Chapter 7

Conclusions

7.1 Summary of Contributions

The main contribution of this thesis is a framework for task clustering and workflow partitioning in distributed systems. We have published 9 papers in peer-reviewed top conferences and 3 submitted journal papers and 1 submitted conference paper.

1. We have developed an overhead-aware workflow model to investigate the performance of task clustering in distributed environments. We presented the overhead characteristics for a wide range of widely used workflows. This overhead analysis and overhead model has been introduced in our published work [20, 15, 21].

2. We have developed partitioning algorithms that use heuristics to divide large-scale workflows into sub-workflows to satisfy resource constraints such as data storage constraint. This work has been published in [18, 17]

3. We have built a statistical model to demonstrate that transient failures can have a significant impact on the runtime performance of scientific workflows. We have developed a Maximum Likelihood Estimation based parameter estimation approach to integrate both prior knowledge and runtime feedback. We have proposed fault tolerant clustering algorithms to dynamically adjust the task granularity and improve the runtime performance. This work has been published in [16].

4. We have examined the reasons that cause runtime imbalance and dependency imbalance in task clustering. We have proposed quantitative metrics to evaluate the severity of the two imbalance problems and a series of balanced clustering methods to address the load

balance problem for five widely used scientific workflows. This work has been published in [20, 21].

5. We have developed an innovative workflow simulator called WorkflowSim with the implementation of popular task scheduling and task clustering algorithms. This simulator is published in [19] and it is available in GitHub [97]. We have built an open source community for the users and developers of WorkflowSim. The community has attracted 50+ researchers from 20+ counties and it is still growing.

7.2 Conclusions and Perspectives

This thesis has demonstrated the benefits of performing workflow restructuring techniques in executing scientific workflows and has proposed and evaluated a series of innovative approaches to substantiate this claim. Our work may inspire other researchers to continue work on this topic.

For example, in Chapter 4, we only consider data storage constraints while researchers may extend our work to further consider other resource constraints such as CPU, memory and network bandwidth. We expect to see similar performance improvement in the context of these resource constraints.

In Chapter 5, one may further analyze the imbalance metrics proposed. For instance, the values of these metrics presented in this chapter are not normalized, and thus their values per level (HIFV, HDV, and HRV) are at different scales. Also, one may analyze more workflow applications, particularly the ones with asymmetric structures, to investigate the relationship between workflow structures and the metric values. We expect that these metrics can perform better with normalization.

Also, as shown in Figure 5.18, *VC-prior* can generate very large clustered jobs vertically and makes it difficult for horizontal methods to improve further. Therefore, researchers can develop imbalance metrics for *VC-prior* to avoid generating large clustered jobs, i.e., based on the accumulated runtime of tasks in a pipeline. We expect to see better performance for *VC-prior* with these new metrics.

As shown in our experiment results, the combination of our balancing methods with vertical clustering have different sensitivity to workflows with distinguished graph structures and runtime distribution. Therefore, a possible future work is the development of a portfolio clustering, which chooses multiple clustering algorithms, and dynamically selects the most suitable one according to the dynamic load.

In Chapter 5, we demonstrate the performance gain of combining horizontal clustering methods and vertical clustering. It may be beneficial to combine multiple algorithms together instead of just two and develop a policy engine that iteratively chooses one algorithm from all of the balancing methods based on the imbalance metrics until the performance gain converges.

Finally, our metrics can be applied to other workflow study areas, such as workflow scheduling where heuristics would either look into the characteristics of the task when it is ready to schedule (local scheduling), or examine the entire workflow (global optimization algorithms). In this thesis, the impact factor metric only uses a family of tasks that are tightly related or similar to each other. This method represents a new approach to solve the existing problems.

In Chapter 6, we only discuss the fault tolerant clustering and apply it to a homogeneous environment. Our work can be combined with fault tolerant scheduling in heterogeneous environments, i.e, a scheduling algorithm that avoids mapping clustered jobs to failure-prone nodes. It is also interesting to combine vertical clustering methods with horizontal clustering methods. For example, we can perform vertical clustering either before or after horizontal clustering, which we believe would bring different performance improvement.

As shown in our experiments, our dynamic estimation does not work well if the inter-arrival time of task failures changes too frequently. It is beneficial to improve its dynamic performance further. For example, researchers may use methods such as the hidden Markov model to provide better prediction results.

Appendix

List of Publications

- Integrating Policy with Scientific Workflow Management for Data-intensive Applications, Ann L. Chervenak, David E. Smith, Weiwei Chen, Ewa Deelman, The 7th Workshop on Workflows in Support of Large-Scale Sciences (WORKS'12), Salt Lake City, Nov 10-16, 2012

- WorkflowSim: A Toolkit for Simulating Scientific Workflows in Distributed Environments, Weiwei Chen, Ewa Deelman, The 8th IEEE International Conference on eScience 2012 (eScience 2012), Chicago, Oct 8-12, 2012

- Integration of Workflow Partitioning and Resource Provisioning, Weiwei Chen, Ewa Deelman, The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012), Doctoral Symposium, Ottawa, Canada, May 13-15, 2012

- Improving Scientific Workflow Performance using Policy Based Data Placement, Muhammad Ali Amer, Ann Chervenak and Weiwei Chen, 2012 IEEE International Symposium on Policies for Distributed Systems and Networks, Chapel Hill, NC, July 2012

- Fault Tolerant Clustering in Scientific Workflows, Weiwei Chen, Ewa Deelman, IEEE International Workshop on Scientific Workflows (SWF), in conjunction with 8th IEEE World Congress on Servicess, Honolulu, Hawaii, Jun 2012

- Workflow Overhead Analysis and Optimizations, Weiwei Chen, Ewa Deelman, The 6th Workshop on Workflows in Support of Large-Scale Science, in conjunction with Supercomputing 2011, Seattle, Nov 2011

- Partitioning and Scheduling Workflows across Multiple Sites with Storage Constraints, Weiwei Chen, Ewa Deelman, 9th International Conference on Parallel Processing and Applied Mathematics (PPAM 2011), Torun, Poland, Sep 2011, Part II, LNCS 7204, pp. 11-12

- Imbalance Optimization in Scientific Workflows, Weiwei Chen, Ewa Deelman, and Rizos Sakellariou, student research competition, the 27th International Conference on Supercomputing (ICS), Eugene, Jun 10-14.

- Balanced Task Clustering in Scientific Workflows, Weiwei Chen, Rafael Ferreira da Silva, Ewa Deelman, Rizos Sakellariou, the 9th IEEE International Conference on e-Science (eScience 2013), Beijing, China, Oct 23-25, 2013

- Imbalance Optimization and Task Clustering in Scientific Workflows, Weiwei Chen, Rafael Ferreira da Silva, Ewa Deelman, Rizos Sakellariou, submitted to the International Journal of Grid Computing and eScience (Future Generation Computer Systems) on Feb 15, 2014

- Pegasus, a Workflow Management System for Large-Scale Science, Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, Kent Wenger, submitted to the International Journal of Grid Computing and eScience (Future Generation Computer Systems) on Feb 15, 2014

- Dynamic and Fault Tolerant Clustering in Scientific Workflows, Weiwei Chen, Rafael Ferreira de Silva, Ewa Deelman, Thomas Fahringer and Rizos Sakellariou, submitted to Scientific Programming Journal

- Community Resources for Enabling Research in Distributed Scientific Workflows, Rafael Ferreira de Silva, Weiwei Chen, Gideon Juve, Karan vahi and Ewa Deelman, submitted to the 10th IEEE International Conference on eScience 2014 (eScience 2014), Guaruja, SP, Brazil, Oct 20-24, 2014

Bibliography

[1] Amazon.com, Inc. Amazon Web Services. `http://aws.amazon.com`.

[2] M. Amer, A. Chervenak, and W. Chen. Improving scientific workflow performance using policy based data placement. In *Policies for Distributed Systems and Networks (POLICY), 2012 IEEE International Symposium on*, pages 86–93, 2012.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[4] Basic local alignment search tools. `http://blast.ncbi.nlm.nih.gov/Blast.cgi`.

[5] G. B. Berriman, E. Deelman, J. C. Good, J. C. Jacob, D. S. Katz, C. Kesselman, A. C. Laity, T. A. Prince, G. Singh, and M. Su. Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand. In *SPIE Conference on Astronomical Telescopes and Instrumentation*, June 2004.

[6] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. Su, and K. Vahi. Characterization of scientific workflows. In *3rd Workshop on Workflows in Support of Large Scale Science (WORKS 08)*, 2008.

[7] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task scheduling strategies for workflow-based applications in grids. In *5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '05)*, 2005.

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

[9] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, Jan. 2011.

[10] S. Callaghan, E. Deelman, D. Gunter, G. Juve, P. Maechling, C. Brooks, K. Vahi, K. Milner, R. Graves, E. Field, et al. Scaling up workflow-based applications. *Journal of Computer and System Sciences*, 76(6):428–446, 2010.

[11] S. Callaghan, P. Maechling, P. Small, K. Milner, G. Juve, T. Jordan, E. Deelman, G. Mehta, K. Vahi, D. Gunter, K. Beattie, and C. X. Brooks. Metrics for heterogeneous scientific workflows: A case study of an earthquake science application. *International Journal of High Performance Computing Applications*, 25(3):274–285, 2011.

[12] Y. Caniou, G. Charrier, and F. Desprez. Evaluation of reallocation heuristics for moldable tasks in computational grids. In *Proceedings of the Ninth Australasian Symposium on Parallel and Distributed Computing - Volume 118*, AusPDC '11, pages 15–24, Darlinghurst, Australia, Australia, 2011. Australian Computer Society, Inc.

[13] J. Cao, S. Jarvis, D. Spooner, J. Turner, D. Kerbyson, and G. Nudd. Performance prediction technology for agent-based resource management in grid environments. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002)*, page 14, April 2002.

[14] J. Celaya and L. Marchal. A fair decentralized scheduler for bag-of-tasks applications on desktop grids. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:538–541, 2010.

[15] W. Chen and E. Deelman. Workflow overhead analysis and optimizations. In *The 6th Workshop on Workflows in Support of Large-Scale Science*, Nov. 2011.

[16] W. Chen and E. Deelman. Fault tolerant clustering in scientific workflows. In *IEEE Eighth World Congress on Services (SERVICES)*, pages 9–16, 2012.

[17] W. Chen and E. Deelman. Integration of workflow partitioning and resource provisioning. In *The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '12)*, May 2012.

[18] W. Chen and E. Deelman. Partitioning and scheduling workflows across multiple sites with storage constraints. In *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Volume Part II*, PPAM'11, pages 11–20, Berlin, Heidelberg, 2012. Springer-Verlag.

[19] W. Chen and E. Deelman. Workflowsim: A toolkit for simulating scientific workflows in distributed environments. In *The 8th IEEE International Conference on eScience*, Oct. 2012.

[20] W. Chen, E. Deelman, and R. Sakellariou. Imbalance optimization in scientific workflows. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 461–462, 2013.

[21] W. Chen, R. Ferreira da Silva, E. Deelman, and R. Sakellariou. Balanced task clustering in scientific workflows. In *2013 IEEE 9th International Conference on eScience*, pages 188–195, 2013.

[22] R. F. da Silva, G. Juve, E. Deelman, T. Glatard, F. Desprez, D. Thain, B. Tovar, and M. Livny. Toward fine-grained online task characteristics estimation in scientific workflows. In *Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science*, pages 58–67. ACM, 2013.

[23] DAGMan: Directed Acyclic Graph Manager. `http://cs.wisc.edu/condor/dagman`.

[24] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In *Across Grid Conference*, 2004.

[25] E. Deelman, D. Gannon, M. Shields, and I. Taylor. Workflows and e-Science: an overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540, May 2009.

[26] E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams, and S. Koranda. GriPhyN and LIGO: building a virtual data grid for gravitational wave scientists. In *11th IEEE International Symposium on High Performance Distributed Computing (HPDC '02)*, 2002.

[27] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger. Pegasus, a workflow management system for large-scale science. *submitted to Future Generation Computer Systems*, 13(3), 2014.

[28] F. Dong and S. G. Akl. Two-phase computation and data scheduling algorithms for workflows in the grid. In *2007 International Conference on Parallel Processing*, page 66, Oct. 2007.

[29] R. Duan, F. Nadeem, J. Wang, Y. Zhang, R. Prodan, and T. Fahringer. A hybrid intelligent method for performance modeling and prediction of workflow activities in grids. In *9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09)*, pages 339–347, 2009.

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

[31] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, et al. Askalon: A development and grid computing environment for scientific workflows. In *Workflows for e-Science*, pages 450–471. Springer, 2007.

[32] R. Ferreira da Silva, T. Glatard, and F. Desprez. On-line, non-clairvoyant optimization of workflow activity granularity on grids. In *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 255–266. Springer Berlin Heidelberg, 2013.

[33] G. C. Fox, G. V. Laszewski, J. Diaz, K. Keahey, R. Figueiredo, S. Smallen, and W. Smith. Futuregrid - a reconfigurable testbed for cloud, hpc and grid computing. In *Contemporary High Performance Computing: From Petascale toward Exascale*, 2013.

[34] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: a computation management agent for Multi-Institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[35] R. Graves, T. Jordan, S. Callaghan, E. Deelman, E. Field, G. Juve, C. Kesselman, P. Maechling, G. Mehta, K. Milner, D. Okaya, P. Small, and K. Vahi. CyberShake: A Physics-Based Seismic Hazard Model for Southern California. *Pure and Applied Geophysics*, 168(3-4):367–381, May 2010.

[36] Z. Guo, M. Pierce, G. Fox, and M. Zhou. Automatic task re-organization in mapreduce. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 335–343, 2011.

[37] M. K. Hedayat, W. Cai, S. J. Turner, and S. Shahand. Distributed execution of workflow using parallel partitioning. In *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on*, pages 106–112. IEEE, 2009.

[38] T. Hey, S. Tansley, and K. Tolle. *The fourth paradigm: data-intensive scientific discovery*. Microsoft Research, 2009.

[39] M. Hussin, Y. C. Lee, and A. Y. Zomaya. Dynamic job-clustering with different computing priorities for computational resource allocation. In *The 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, May 2010.

[40] A. Iosup, O. Sonmez, S. Anoep, and D. Epema. The performance of bags-of-tasks in large-scale distributed systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, pages 97–108, New York, NY, USA, 2008. ACM.

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

[42] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi. Characterizing and profiling scientific workflows. volume 29, pages 682 – 692, 2013. Special Section: Recent Developments in High Performance Computing and Security.

[43] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling. Scientific workflow applications on amazon EC2. In *2009 5th IEEE International Conference on E-Science Workshops*, Dec. 2009.

[44] S. Kalayci, G. Dasgupta, L. Fong, O. Ezenwoye, , and S. Sadjadi. Distributed and adaptive execution of condor dagman workflows. In *Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE'2010)*, July 2010.

[45] T. Kosar and M. Livny. Stork: making data placement a first class citizen in the grid. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 342–349, 2004.

[46] S. Kumar, S. Das, and R. Biswas. Graph partitioning for parallel applications in heterogeneous grid environments. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, 2002.

[47] D. Laforenza, R. Lombardo, M. Scarpellini, M. Serrano, F. Silvestri, and P. Faccioli. Biological experiments on the grid: A novel workflow management platform. In *20th IEEE International Symposium on Computer-Based Medical Systems (CBMS'07)*, pages 489–494. IEEE, 2007.

[48] Laser Interferometer Gravitational Wave Observatory (LIGO). `http://www.ligo.caltech.edu`.

[49] A. Lathers, M. Su, A. Kulungowski, A. Lin, G. Mehta, S. Peltier, E. Deelman, and M. Ellisman. Enabling parallel scientific applications with workflow tools. In *Challenges of Large Applications in Distributed Environments (CLADE 2006)*, 2006.

[50] H. Li, D. Groep, and L. Wolters. Efficient response time predictions by exploiting application and resource state similarities. In *The 6th IEEE/ACM International Workshop on Grid Computing*, page 8, Nov 2005.

[51] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, June 2012.

[52] Q. Liu and Y. Liao. Grouping-based fine-grained job scheduling in grid computing. In *First International Workshop on Education Technology and Computer Science*, Mar. 2009.

[53] P. Maechling, E. Deelman, L. Zhao, R. Graves, G. Mehta, N. Gupta, J. Mehringer, C. Kesselman, S. Callaghan, D. Okaya, H. Francoeur, V. Gupta, Y. Cui, K. Vahi, T. Jordan, and E. Field. SCEC CyberShake WorkflowsAutomating probabilistic seismic hazard analysis calculations. In I. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 143–163. Springer, 2007.

[54] K. Maheshwari, A. Espinosa, D. S. Katz, M. Wilde, Z. Zhang, I. Foster, S. Callaghan, and P. Maechling. Job and data clustering for aggregate use of multiple production cyberinfrastructures. In *Proceedings of the fifth international workshop on Data-Intensive Distributed Computing Date*, DIDC '12, pages 3–12, New York, NY, USA, 2012. ACM.

[55] R. Mats, G. Juve, K. Vahi, S. Callaghan, G. Mehta, and P. J. M. andEwa Deelman. Enabling large-scale scientific workflows on petascale resources using mpi master/worker. In *Proceedings of the 1st conference of the Extreme Science and Engineering Discovery Environment*, July 2012.

[56] S. R. McConnel, D. P. Siewiorek, and M. M. Tsao. The measurement and analysis of transient errors in digital computer systems. In *Proc. 9th Int. Symp. Fault-Tolerant Computing*, pages 67–70, 1979.

[57] N. Muthuvelu, I. Chai, E. Chikkannan, and R. Buyya. On-line task granularity adaptation for dynamic grid applications. In *Algorithms and Architectures for Parallel Processing*, volume 6081 of *Lecture Notes in Computer Science*, pages 266–277. 2010.

[58] N. Muthuvelu, I. Chai, and C. Eswaran. An adaptive and parameterized job grouping algorithm for scheduling grid jobs. In *10th International Conference on Advanced Communication Technology (ICACT 2008)*, volume 2, pages 975 –980, 2008.

[59] N. Muthuvelu, J. Liu, N. L. Soe, S. Venugopal, A. Sulistio, and R. Buyya. A dynamic job grouping-based scheduling for deploying applications with fine-grained tasks on global grids. In *Proceedings of the 2005 Australasian workshop on Grid computing and e-research*, 2005.

[60] N. Muthuvelu, C. Vecchiola, I. Chai, E. Chikkannan, and R. Buyya. Task granularity policies for deploying bag-of-task applications on global grids. *Future Generation Computer Systems*, 29(1):170 – 181, 2013. ¡ce:title¿Including Special section: AIRCC-NetCoM 2009 and Special section: Clouds and Service-Oriented Architectures¡/ce:title¿.

[61] W. K. Ng, T. Ang, T. Ling, and C. Liew. Scheduling framework for bandwidth-aware job grouping-based scheduling in grid computing. *Malaysian Journal of Computer Science*, 19(2):117–126, 2006.

[62] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, Nov. 2004.

[63] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. *Why do internet services fail and what can be done about it?* Computer Science Division, University of California, 2002.

[64] A. Oprescu and T. Kielmann. Bag-of-tasks scheduling under budget constraints. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 351–359, 30 2010-Dec. 3.

[65] S. Ostermann, K. Plankensteiner, R. Prodan, T. Fahringer, and A. Iosup. Workflow monitoring and analysis tool for ASKALON. In *Grid and Services Evolution*. 2009.

[66] S.-M. Park and M. Humphrey. Data throttling for data-intensive workflows. In *IEEE Intl. Symposium on Parallel and Distributed Processing*, Apr. 2008.

[67] K. Plankensteiner, R. Prodan, and T. Fahringer. A new fault tolerance heuristic for scientific workflows in highly distributed environments based on resubmission impact. In *Fifth IEEE International Conference on e-Science (e-Science '09)*, pages 313–320, Dec 2009.

[68] K. Plankensteiner, R. Prodan, T. Fahringer, A. Kertész, and P. Kacsuk. Fault detection, prevention and recovery in current grid workflow systems. In *Grid and Services Evolution*, pages 1–13. Springer, 2009.

[69] R. Prodan. Online analysis and runtime steering of dynamic workflows in the askalon grid environment. In *The Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 389 –400, May 2007.

[70] R. Prodan and T. Fabringer. Overhead analysis of scientific workflows in grid environments. In *IEEE Transactions in Parallel and Distributed System*, volume 19, Mar. 2008.

[71] R. Rodriguez, R. Tolosana-Calasanz, and O. Rana. Automating data-throttling analysis for data-intensive workflows. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '12)*, pages 310–317, 2012.

[72] R. K. Sahoo, A. Sivasubramaniam, M. S. Squillante, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Proceedings of the International Conference on Dependable Systems and Networks*, July 2004.

[73] R. Sakellariou, H. Zhao, and E. Deelman. Mapping Workflows on Grid Resources: Experiments with the Montage Workflow. In F. Desprez, V. Getov, T. Priol, and R. Yahyapour, editors, *Grids P2P and Services Computing*, pages 119–132. 2010.

[74] T. Samak, D. Gunter, M. Goode, E. Deelman, G. Juve, F. Silva, and K. Vahi. Failure analysis of distributed scientific workflows executing in the cloud. In *Network and service management (cnsm), 2012 8th international conference and 2012 workshop on systems virtualiztion management (svm)*, pages 46–54. IEEE, 2012.

[75] T. Samak, D. Gunter, M. Goode, E. Deelman, G. Mehta, F. Silva, and K. Vahi. Failure prediction and localization in large scientific workflows. In *The 6th Workshop on Workflows in Suppporting of Large-Scale Science*, Nov. 2011.

[76] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2006.

[77] B. Schuller, B. Demuth, H. Mix, K. Rasch, M. Romberg, S. Sild, U. Maran, P. Bała, E. Del Grosso, M. Casalegno, et al. Chemomentum-unicore 6 based infrastructure for complex applications in science and technology. In *Euro-Par 2007 Workshops: Parallel Processing*, pages 82–93. Springer, 2008.

[78] S. Shankar and D. J. DeWitt. Data driven workflow planning in cluster management systems. In *Proceedings of the 16th international symposium on High performance distributed computing*, HPDC '07, pages 127–136, New York, NY, USA, 2007. ACM.

[79] G. Singh, M. Su, K. Vahi, E. Deelman, B. Berriman, J. Good, D. S. Katz, and G. Mehta. Workflow task clustering for best effort systems with pegasus. In *15th ACM Mardi Gras Conference*, 2008.

[80] SIPHT. `http://pegasus.isi.edu/applications/sipht`.

[81] O. Sonmez, H. Mohamed, and D. Epema. Communication-aware job placement policies for the koala grid scheduler. In *Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*, pages 79–79. IEEE, 2006.

[82] O. Sonmez, N. Yigitbasi, S. Abrishami, A. Iosup, and D. Epema. Performance analysis of dynamic workflow scheduling in multicluster grids. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 49–60, New York, NY, USA, 2010. ACM.

[83] X.-H. Sun and M. Wu. Grid harvest service: a system for long-term, application-level task scheduling. In *International Parallel and Distributed Processing Symposium (IPDPS)*, page 8, April 2003.

[84] D. Tang, R. K. Iyer, and S. S. Subramani. Failure analysis and modeling of a vaxcluster system. In *Proceedings of the International Symposium on Fault-tolerant computing*, 1990.

[85] I. Taylor, M. Shields, I. Wang, and A. Harrison. The triana workflow environment: Architecture and applications. In *Workflows for e-Science*, pages 320–339. Springer, 2007.

[86] The TeraGrid Project. `http://www.teragrid.org`.

[87] T. L. L. P. T.F. Ang, W.K. Ng and C. Liew. A bandwidth-aware job grouping-based scheduling on grid environment. *Information Technology Journal*, (8):372–377, 2009.

[88] R. Tolosana-Calasanz, O. F. Rana, and J. A. Bañares. Automating performance analysis from taverna workflows. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, CBSE '08, pages 1–15, Berlin, Heidelberg, 2008. Springer-Verlag.

[89] L. Tomas, B. Caminero, and C. Carrion. Improving grid resource usage: Metrics for measuring fragmentation. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '12)*, pages 352–359, 2012.

[90] H. Topcuoglu, S. Hariri, and W. Min-You. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[91] USC Epigenome Center. `http://epigenome.usc.edu`.

[92] Y. Wang, G. Agrawal, G. Ozer, and K. Huang. Removing sequential bottlenecks in analysis of next-generation sequencing data. In *Proceedings of the International Workshop on High Performance Computational Biology (HiCOMB'14)*, Phoenix, Arizona, may 2014.

[93] Y. Wang, W. Jiang, and G. Agrawal. Scimate: A novel mapreduce-like framework for multiple scientific data formats. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 443–450. IEEE, 2012.

[94] Y. Wang, Y. Su, and G. Agrawal. Supporting a light-weight data management layer over hdf5. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 335–342. IEEE, 2013.

[95] M. Wieczorek, R. Prodan, and T. Fahringer. Scheduling of scientific workflows in the askalon grid environment. In *ACM SIGMOD Record*, volume 34, pages 56–62, Sept. 2005.

[96] Workflow Generator. `https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator`.

[97] WorkflowSim in GitHub. http://github.com/WorkflowSim/WorkflowSim-1.0/.

[98] N. Yigitbasi, M. Gallet, D. Kondo, A. Iosup, and D. Epema. Analysis and modeling of time-correlated failures in large-scale distributed systems. In *2010 11th IEEE/ACM International Conference on Grid Computing (GRID)*, pages 65–72. IEEE, 2010.

[99] H. Ying, G. Mingqiang, L. Xiangang, and L. Yong. A webgis load-balancing algorithm based on collaborative task clustering. *Environmental Science and Information Application Technology, International Conference on*, 3:736–739, 2009.

[100] D. Yuan, Y. Yang, X. Liu, and J. Chen. A data placement strategy in scientific cloud workflows. *Future Generation Computer Systems*, 26(8):1200–1214, 2010.

[101] X. Zhang, Y. Qu, and L. Xiao. Improving distributed workload performance by sharing both cpu and memory resources. In *Proceedings of 20th International Conference on Distributed Computing Systems, (ICDCS'2000*, pages 233–241, 2000.

[102] Y. Zhang, A. Mandal, C. Koelbel, and K. Cooper. Combined fault tolerance and scheduling techniques for workflow applications on computational grids. In *9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'09)*, pages 244–251. IEEE, 2009.

[103] Y. Zhang and M. S. Squillante. Performance implications of failures in large-scale cluster scheduling. In *The 10th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2004.

[104] H. Zhao and X. Li. Efficient grid task-bundle allocation using bargaining based self-adaptive auction. In *The 9th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, May 2009.

[105] G. Zheng, A. Bhatele, E. Meneses, and L. V. Kalé. Periodic hierarchical load balancing for large supercomputers. *International Journal of High Performance Computing Applications*, 25(4):371–385, 2011.