TASK CLUSTERING FOR LARGE-SCALE SCIENTIFIC WORKFLOWS

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

Copyright 2014 Weiwei Chen

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 and simulation. Task clustering is a runtime optimization technique that merges multiple short workflow tasks into a single job such that the scheduling overheads and communication cost are reduced and the overall runtime performance is significantly improved. However, the recent emergence of large-scale scientific workflows executing on modern distributed environments, such as grids and clouds, requires a new methodology that considers task clustering in a comprehensive way. Our work investigates the key concern of workflow studies and proposes a series of dynamic methods to improve the overall workflow performance, including runtime performance, fault tolerance, and resource utilization. Simulation-based and experiment-based evaluation verifies the effectiveness of our methods.

Chapter 1

Introduction

Over the years, with the emerging of the fourth paradigm of science discovery [44], scientific workflows continue to gain their popularity among many science disciplines, including physics [31], astronomy [86], biology [56, 73], chemistry [113], earthquake science [63] and many more. Scientific workflows increasingly require tremendous amounts of data processing and workflows with up to a few million tasks are not uncommon [16]. 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 [31]. For example, the CyberShake workflow [65] is used by the Southern California Earthquake Center (SCEC) [96] to characterize earthquake hazards using the Probabilistic Seismic Hazard Analysis (PSHA) technique. CyberShake workflows composed of more than 800,000 jobs have been executed on the TeraGrid [102]. When executing these applications on a multi-machine parallel environment, such as the Grid or the Cloud, significant system overheads may exist. An overhead is defined as the time of performing miscellaneous work other than executing the user's computational activities. Overheads may adversely influence runtime performance [**?**]. In order to minimize the impact of such overheads, task clustering [91, 45, 120] has been developed to merge small tasks into larger jobs so that the number of computational activities is reduced and their computational granularity is increased thereby reducing the (mostly scheduling related) system overheads.

Traditionally a workflow is modeled as a Directed Acyclic Graph (DAG). 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. A job (j) is a single execution unit and it contains one or

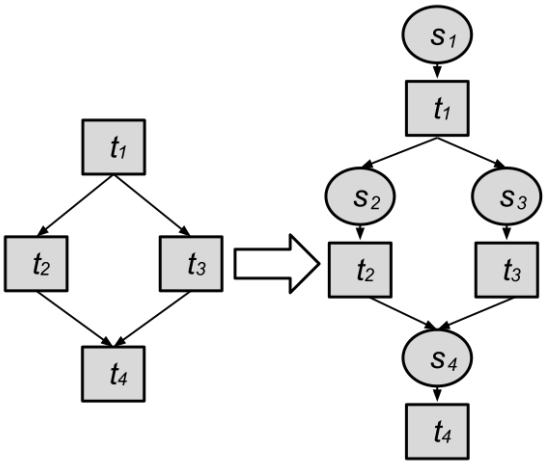

Figure 1.1: Extending DAG to o-DAG

multiple task(s). 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. In this work, we extend the DAG model to be overhead aware (o-DAG). The reason is that system overheads play an important role in workflow execution and they constitute a major part of the overall runtime when tasks are poorly clustered. Fig 1.1 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 classification of overheads is based on the model of a typical workflow management system shown in Fig 1.2. The components in this WMS are listed below:

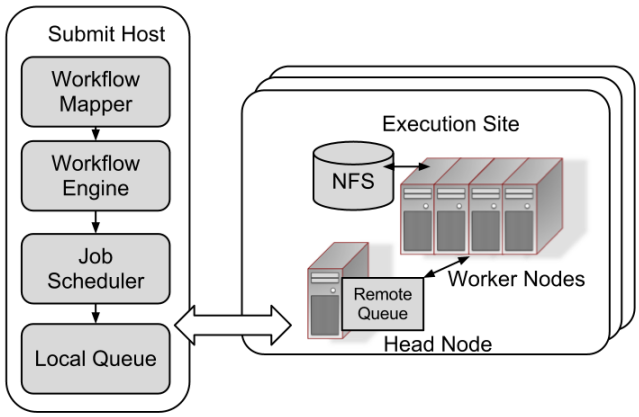

Figure 1.2: System Model

Workflow Mapper generates an executable workflow based on an abstract workflow provided by the user or 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.

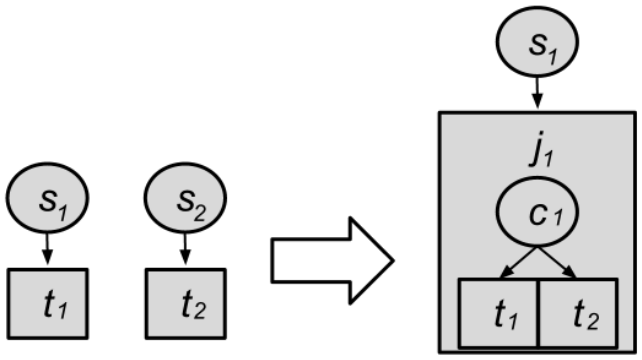

Figure 1.3: Task Clustering

With o-DAG model, we can explicitly express the process of task clustering. For example, in Fig 1.3, two tasks t_1 and t_2 without data dependency between them are merged into a clustered job j_1. Scheduling overheads (s_2) are reduced but clustering delay (c_1) is added.

In this work, we identify the new challenges when executing complex scientific applications:

The first challenge users face when merging workflow tasks is the **imbalance of computation**. 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 [45, 19, 75] dynamically groups tasks together based on the task characteristics but it assumes tasks are independent, which limits its usage in scientific workflows. Singh [91] and Rynge [65] examine the workflow structure and groups tasks together into jobs in a static way for best effort systems. However, this work ignores the computational requirement of tasks and may end up with an imbalanced load on the resources. A popular technique in workload studies to address the load balancing challenge is over-decomposition [59]. This method decomposes

computational work into medium-grained 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 that is offered by the hardware.

The second challenge has to do with the **data management within a workflow**. Scientific applications are often data intensive and usually need collaborations of scientists from different institutions, hence application data in scientific workflows are usually distributed. Particularly with the emergence of grids and clouds, scientists can upload their data and launch their applications on scientific cloud workflow systems from anywhere in the world via the Internet. However, merging tasks that have input data from or have output data to different data centers is time-consuming and inefficient. 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. Communication-aware scheduling [94, 47] has taken the communication cost into the scheduling/clustering model and have achieved some significant improvement. The workflow partitioning approach [43, 117, 113, 35] represents the clustering 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 [64, 15] are used to select the right parameters and achieve better runtime performance but the approach is not automatic and requires much experience in distributed systems.

Resource management is the third challenge that is brought by the recent emergence of cloud computing and resource provisioning techniques. Along with the increase of the scale of workflows, the number and the variety of computational resources to use has been increasing consistently. Infrastructure-as-a-Service (IaaS) clouds offer the ability to provision resources on-demand according to a pay-per-use model and adjust resource capacity according to the changing demands of the application [1]. Task clustering can still be applied to this cloud scenario [29, 111]. However, the decisions required in cloud scenarios not only have to take into account performance-related metrics such as workow makespan, but must also consider the resource utilization, since the resources from commercial clouds usually have monetary costs associated

with them. Therefore, the adoption of task clustering on cloud computing requires the development of new methods for the integration of task clustering and resource provisioning.

The fourth challenge has to do with **fault tolerance**. Existing clustering strategies ignore or underestimate the impact of the occurrence of failures on system behavior, despite the increasing impact of failures in large-scale distributed systems. Many researchers [119, 99, 88, 85] have emphasized the importance of fault tolerance 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 [119]. 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 [119]. In a faulty environment, there are usually three approaches for managing workflow failures. First, one can simply retry the entire job when its computation is not successful as in the Pegasus Workflow Management System [30]. 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 [119]. Third, tasks can be replicated to different nodes to avoid location-specific failures [118]. However, inappropriate clustering (and replication) parameters may cause severe performance degradation if they create long-running clustered jobs.

After examining the major challenges in executing large-scale scientific workflows, we contribute to the studies of workflow performance improvement through task clustering in the following aspects:

First of all, we **extend the existing DAG model to be overhead aware** and quantitatively 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 [81, 82]. In contrast, we investigate the distributions and patterns of different overheads and discuss how the system environment (system configuration, etc.) influences the distribution of overheads. Furthermore, we present quantitative metrics to measure and

evaluate the characters (robustness, sensitivity, balance, etc.) of workflows. Finally, we analyze the relationship between these metrics and the workflow performance with different optimization methods.

Second, to **solve the computation imbalance problem**, we introduce a series of balanced clustering methods. The computation imbalance problem means that the execution of workflows suffers from significant overheads (unavailable data, overloaded resources, or system constraints) due to inefficient task clustering and job execution. 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 four metrics to reflect the internal structure (in terms of runtime and dependency) of the workflow.

Third, we introduce **data aware workflow partitioning** to reduce the data transfer between clustered jobs. 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 and experiment-based evaluation is performed to validate its effectiveness.

Fourth, we propose **fault tolerant clustering** that dynamically adjusts the clustering strategy based on the current trend of failures. We classify transient failures into two categories, task failure and job failure and furthermore indicate their distinct influence on the overall performance. During the runtime, this approach estimates the failure distribution among all the resources and dynamically merges tasks into jobs of moderate size and recluster failed jobs to avoid failures.

Finally, to achieve a resource aware clustering in scientific workflows, we propose to **integrate resource provisioning with task clustering**. We first analyze the relationship between the overall runtime of a clustered job and the allocated resources (CPU, memory, and storage).

Then we develop resource aware clustering algorithms to partition a workflow and predict the number of cores and wall time required for each partition. Furthermore, we integrate the clustering algorithms with resource provisioning and develop an estimation method to be used to provision VMs on a cloud and predict the cost of running the workflow. We also investigate the scenario when resources are constrained (such as storage) and propose algorithms and heuristics to improve the workflow performance.

Overall, this thesis aims to **improve the overall performance of task clustering in large-scale scientific workflows**. We present both experiment-based and simulation-based evaluation of a wide range of scientific workflows.

Chapter 2

Workflow and System Model

In this chapter, we first introduce how we extend the existing DAG model to be overhead aware and we also describe the system model that we use in this work. We then analyze the overhead characteristics and distributions across different distributed platforms. Finally we introduce 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 overhead into consideration. Using our model, the accuracy of runtime estimation can be improved by up to 5 times.

2.1 Motivation

Traditionally the optimization of workflow performance has been focusing on reducing overall runtime of computational activities through techniques such as task scheduling that aims to adjust the mapping from tasks to resources. However, these approaches have over-simplified the characteristics of real distributed environments and under-estimated the complexity of large scale workflows. In practice, due to the distributed nature of these resources, the large number of tasks in a workflow, and the complex dependencies among the tasks, significant system overheads can occur during workflow execution. For example, a Montage workflow has around 10,000 tasks, which is a significant load for workflow management tools to schedule or maintain. On one hand, the duration of these tasks is usually around a few seconds, but the system overheads in distributed systems such as Grids can reach up to a few minutes. Merging these short workflow tasks into a larger group of tasks and executing them together can reduce the number of operations (such as job submission) and thus reduce the system overheads significantly. The process of merging tasks into a single job (group of tasks) is called task clustering.

Task clustering has been widely used in optimizing scientific workflows and can achieve significant improvement in the overall runtime performance [65, 91, 58, 17] of workflows. However, there is a lack of a generic and systematic analysis and modeling of task clustering to improve the overall workflow performance including runtime, fault tolerance, data movement and resource utilization etc. To address this challenge, this work extends 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 task clustering and provide a series of novel optimization methods to further improve the overall workflow performance.

In this chapter, we introduce our o-DAG model and present our overhead analysis on a series of widely used workflows, which is a base of our optimization methods that will be introduced in the rest of this proposal.

2.2 Related Work

The Directed Acyclic Graph (DAG) has been widely used in many workflow management systems such as DAGMan [28], Pegasus [30], Triana [100], DAGuE [12] and GrADS [26] . Each node in the DAG represents a workflow task, and the edges represent dependencies between the tasks that constrain the order in which the tasks are executed. The Directed Acyclic Graph Manager (DAG-Man) [28] is a service provided by Condor [39] for executing multiple jobs with dependencies. The DAGMan meta-scheduler processes the DAG dynamically, by sending to the Condor scheduler the jobs as soon as their dependencies are satisfied and they become ready to execute. DAGMan can also help with the resubmission of uncompleted portions of a DAG when one or more nodes resulted in failure, which is called job retry. Pegasus [30], which stands for Planning for Execution in Grids, was developed at the USC Information Sciences Institute as part of the GriPhyN [31] and SCEC/IT [63] projects. Pegasus receives an abstract workflow description in a XML format from users, produces a concrete or executable workflow, and submits it to DAGMan for execution.

A Petri net is a directed bipartite graph, in which the nodes represent transitions and places. The directed arcs describe which places are pre- and/or postconditions for which transitions occurs. Ordinary Petri nets and their extensions have been widely used for the specification, analysis and implementation of workflows [110]. Petri nets also enable powerful analysis techniques, which can be used to verify the correctness of workow procedures. In the scientific workflow community, Petri nets have also been utilized and GWorkflowDL [112], Grid-Flow [42] and FlowManager [6] are representative examples of this.

ASKALON [36] is a Grid environment for composition and execution of scientific workflow applications and uses the standard Web Services Description Language (WSDL) to model workflows. WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. In ASKALON, the scheduler optimizes the workflow performance using the execution time as the most important goal function. The scheduler interacts with the enactment engine, which is a service that supervises the reliable and fault tolerant execution of the tasks and the transfer of files.

Compared to these approaches, we extend the original DAG model to be overhead aware so as to analyze the performance of task clustering. An overhead [82, 81] is defined as the time of performing miscellaneous work other than executing the users computational activities. In our overhead-aware DAG model (o-DAG), a node can represent either a computational task/job or system overhead during the runtime. A directed edge can represent either a data dependencies between computational tasks/jobs or a control dependency between overheads and computations. Such an extension of the DAG model provides us the ability to model the process of task clustering and analyze the runtime performance of different task clustering strategies.

Overheads play an important role in distributed systems. Stratan et al. [97] evaluates workflow engines including DAGMan/Condor and Karajan/Globus in a real-world grid environment. Their methodology focuses on five system characteristics: the overhead, the raw performance, the stability, the scalability and the reliability. They have pointed out that head node consumption

should not be negligible and the main bottleneck in a busy system is often the head node. Prodan et al. [82] offers a complete grid workflow overhead classification and a systematic measurement of overheads. Ostberg et al. [**?**] used the Grid Job Management Framework (GJMF) as a testbed for characterization of Grid Service-Oriented Architecture overhead, and evaluate the efficiency of a set of design patterns for overhead mediation mechanisms featured in the framework. In comparison with their work, (1) we focus on measuring the overlap of major overheads imposed by workflow management systems and execution environments; (2) we present a study of the distribution of overheads instead of just overall numbers; (3) we compare workflows running in different platforms (dedicated clusters, clouds, grids) and different environments (resource availability, file systems), explaining how they influence the resulting overheads; and (4) we analyze how existing optimization techniques improve the workflow runtime by reducing or overlapping overheads.

Performance Analysis of scientific workflows has also been studied in [34, 105, 107, 109]. The performance method proposed by Duan et al. [34] is based on a hybrid Bayesian-neural network for predicting the execution time of workflow tasks. Bayesian network is a graphical modeling approach that we use to model the effects of different factors affecting the execution time (referred as factors or variables), and the interdependence of the factors among each other. The important attributes are dynamically selected by the Bayesian network and fed into a radial basis function neural network to make further predictions. In comparison, our work in overhead analysis focuses on the relationship between system overheads and the performance of different optimization methods. We investigated a wide range of scientific workflows and analyzed how system overheads influence the performance of optimization of these workflows.

The low performance of lightweight (a.k.a. fine-grained) tasks is a common problem on widely distributed platforms where the communication overheads and scheduling overheads are high, such as grid systems. To address this issue, fine-grained tasks are commonly merged into coarse-grained tasks [69, 70, 71], which reduces the cost of data transfers when grouped tasks share input data [69] and saves scheduling overheads such as queueing time when resources are limited. However, task grouping also limits parallelism and therefore should be used carefully.

Muthuvelu et al. [70] proposed an algorithm to group bag 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. Then, Keat et al. [71] and Ang et al. [103] extended the work of Muthuvelu et al. by introducing bandwidth to enhance the performance of task clustering. Resources are sorted in descending order of bandwidth, then assigned to grouped tasks downward ordered by processing requirement length. Afterwards, Soni et al. [93] proposed an algorithm to group lightweight tasks into coarse-grained tasks (GBJS) based on processing capability, bandwidth, and memory-size of the available resources. Tasks are sorted into ascending order of required computational power, then, selected in first come first serve order to be grouped according to the capability of the resources. Zomaya and Chan [122] studied limitations and ideal control parameters of task clustering by using genetic algorithm. Their algorithm performs task selection based on the earliest task start time and task communication costs; it converges to an optimal solution of the number of clustered jobs and tasks per clustered job. In contrast, our work has discussed the influence of data dependencies across different levels while they only focus on computational activities (bag-of-tasks).

Singh [91] proposed to use horizontal clustering and label based clustering in scientific workflows to reduce the scheduling overheads in a best-effort approach. The horizontal clustering merges tasks at the same level, while level of a task refers to the distance from a root task to this task using Breadth-First-Search. Label based clustering uses labels set by the users manually and merges tasks with the same labels together. Task clustering strategies have demonstrated their effect in some scientific workflows [65, 64, 45, 60]. Li [58] developed algorithm that uses horizontal clustering to group tasks that can be scheduled to run simultaneously. Tasks with the same scheduling priority (determined by the scheduling level) are merged and scheduled to run simultaneously. Cao et al. [17] proposed a static scheduling heuristic, called DAGMap that consists of three phases, namely prioritizing, grouping, and independent task scheduling. Task grouping is based on dependency relationships and task upward priority (the longest distance form this task to the exit task). Compared to their work, we propose a generic workflow

model that takes system overheads into consideration and provide a series of overhead aware task clustering strategies to optimize the overall runtime performance of workflows.

Task clustering is one typical category of task scheduling, which maps resources to tasks based on different criteria. List Heuristics assign a priority to a task and the scheduling algorithms attempt to execute the higher priority nodes first. Most scheduling algorithms for Grid systems are based on this approach. For example, scheduling algorithms, such as HEFT [106], MaxMin [13], MinMin [11], etc., have been widely used in optimizing the runtime performance of many scientific workflows. Genetic Algorithms [2] and Neural Network [7] are also proposed to address the scheduling problem. Compared to them, we aim to generate a group of tasks that are suitable for scheduling and execution while the resource selection is not our major challenge since we already have many mature scheduling algorithms.

2.3 Approach

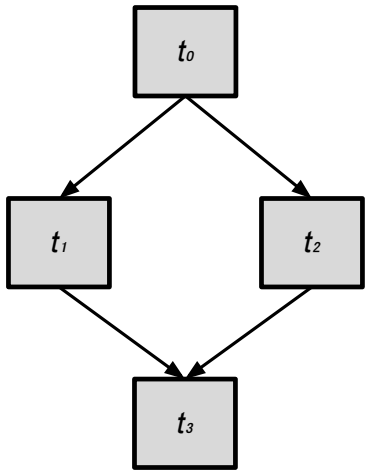

Figure 2.1: A simple DAG with four tasks (t_0, t_1, t_2, t_3). The edges represent the data dependencies between tasks.

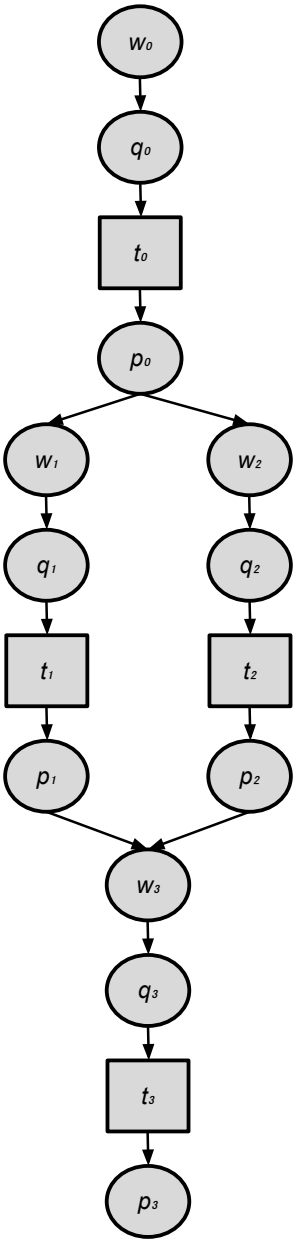

Figure 2.2: An o-DAG with overheads ($w_0 \sim w_3$, $q_0 \sim q_3$, $p_0 \sim p_3$). The edges represent control dependencies or data dependencies

Traditionally a workflow is modeled as a Directed Acyclic Graph (DAG). 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. Fig 2.1 shows a simple workflow with four tasks. A job (j) is a single execution unit and it contains one or multiple task(s). 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. Fig 2.2 shows how we augment a DAG in Fig 2.1 to be an o-DAG with the capability to represent scheduling overheads (s) such as workflow engine delay (w), queue delay (q), and postscript delay (q). Fig 2.4 further shows how we perform task clustering in this simple workflow, in which we merge t_1 and t_2 into a new job j_4. The scheduling overheads associated with t_1 and t_2 are removed and the overheads including the clustering delay (c_4) of j_4 are added.

The classification of overheads is based on the model of a typical workflow management system (WMS) shown in Fig 2.3. The components in this WMS are listed below:

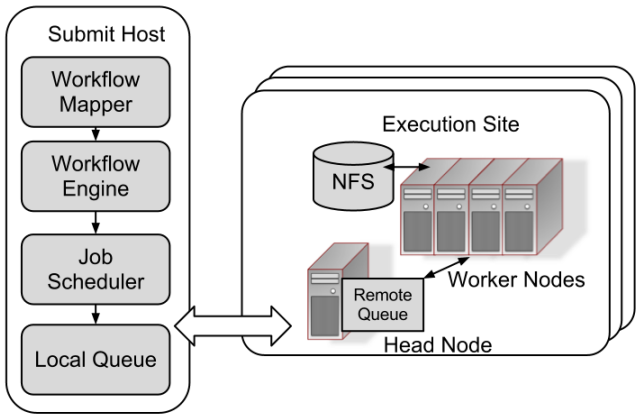

Figure 2.3: System Model

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

Workflow Engine executes the jobs 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.

2.3.1 Overhead Classification

The execution of a job is comprised of a series of events as shown in Figure 2.5 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 2.5 shows a typical timeline of overheads and runtime in a compute 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

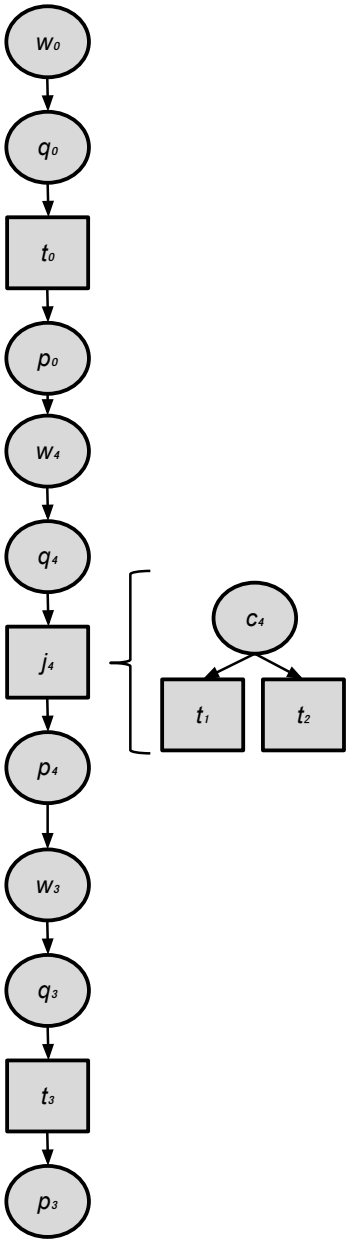

Figure 2.4: A Diamond Workflow after Task Clustering

for resources to be assigned to it. The workflow engine delay reflects the efficiency of a workflow engine (i.e., DAGMan [28]).

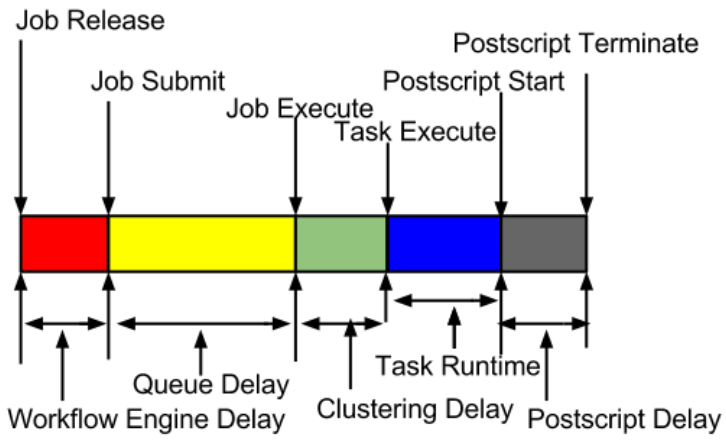

Figure 2.5: Workflow Events

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 [39]) 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. 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.

2.3.2 Overhead Distribution

We examined the overhead distributions of a widely used astronomy workflow called Montage [8] that is used to construct large image mosaics of the sky. Montage was run on FutureGrid [40]. FutureGrid is a distributed, high-performance testbed that provides scientists with a set of computing resources to develop parallel, grid, and cloud applications.

Figure 2.6 shows 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 2.6) 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.

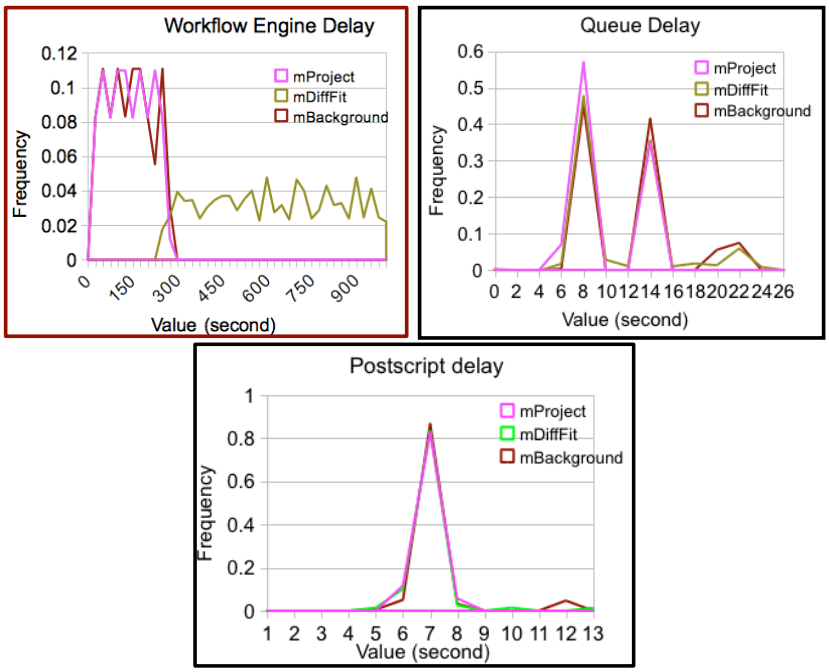

Figure 2.6: Distribution of overheads in the Montage workflow

We use Workflow Engine Delay as an example to show the necessity to model overheads appropriately. Figure 2.7 shows a real trace of overheads and runtime in the Montage 8 degree workflow (for visibility issues, we only show the first 15 jobs at the mProjectPP level). We can see that Workflow Engine Delay increases steadily after every five jobs. For example, the Workflow Engine Delay of jobs with ID from 6 to 10 is approximately twice of that of jobs

ranging from ID1 to ID5. Figure 2.8 further shows the distribution of Workflow Engine Delay at the mProjectPP level in the Montage workflow that was run five times. After every five jobs, the Workflow Engine Delay increases by 8 seconds approximately. We call this special nature of workflow overhead as cyclic increase. The reason is that Workflow Engine (in this trace it is DAGMan) releases five jobs by default in every working cycle. Therefore, simply adding a constant delay after every job execution has ignored its potential influence on the performance.

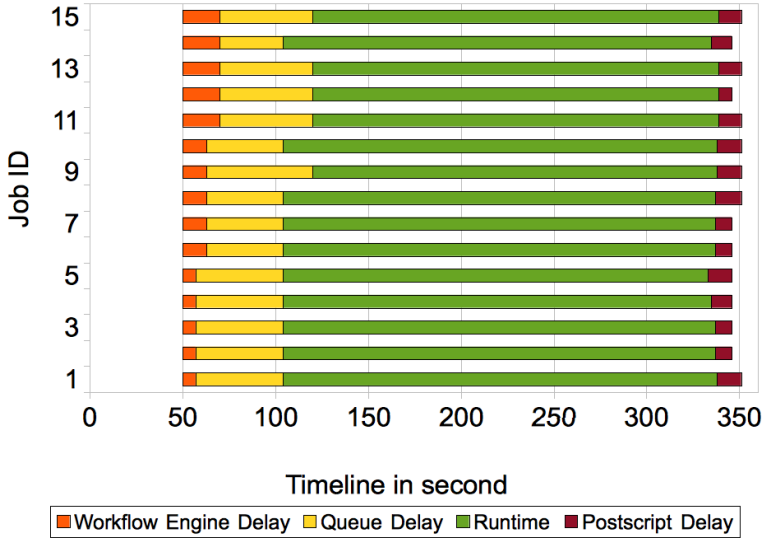

Figure 2.7: Workflow Overhead and Runtime. Clustering delay and data transfer delay are not shown

Figure 2.9 shows the average value of Clustering Delay of mProjectPP, mDiffFit, and mBackground. It is clear that with the increase of k (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 k. 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 $k = i$ directly if we have known the delay when $k = j$. Therefore we can predict all the clustering cases as long as we have gathered one clustering case.

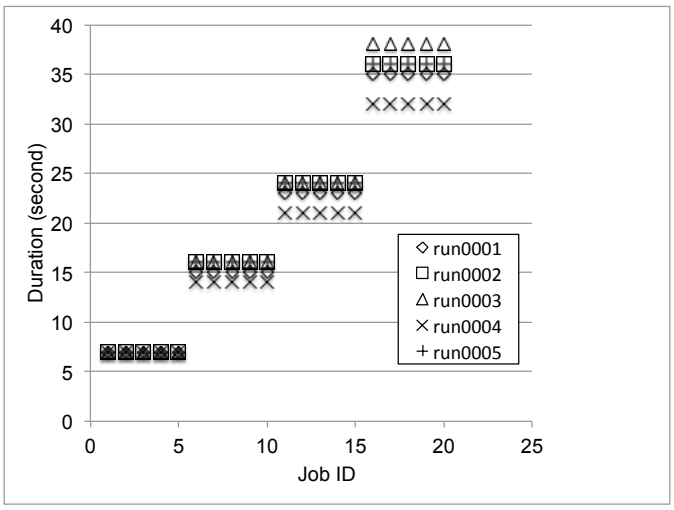

Figure 2.8: Workflow Engine Delay of mProjectPP

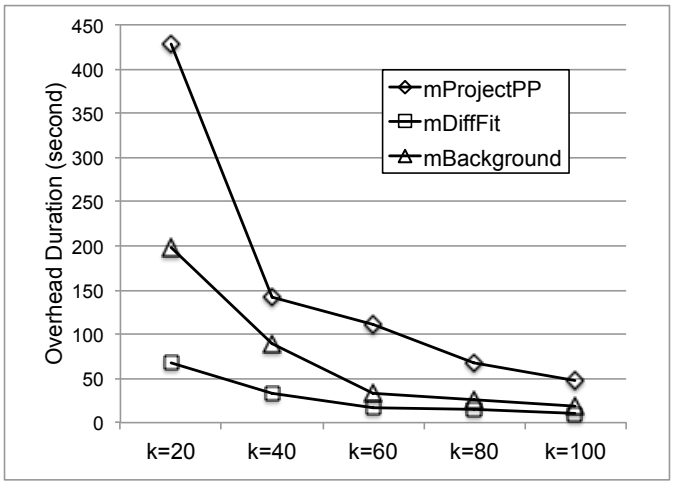

Figure 2.9: Clustering Delay of mProjectPP, mDiffFit, and mBackground

$$\frac{ClusteringDelay|_{k=i}}{ClusteringDelay|_{k=j}} = \frac{n/i}{n/j} = \frac{j}{i} \tag{2.1}$$

2.3.3 Metrics to Evaluate Cumulative Overheads

After identifying the major overheads in workflows and describe how they are measured based on workflow events, we provide an integrated and comprehensive quantitative analysis of workflow overheads. The observation on overhead distribution and characteristics enable researchers

to build a more realistic model for simulations of real applications. Our analysis also offers guidelines for developing further optimization methods.

In this section, we define four metrics to calculate cumulative overheads of workflows, which are Sum, $Projection(PJ)$, $Exclusive\ Projection(EP)$ and $Reverse\ Ranking(RR)$. Sum simply adds up the overheads of all jobs without considering their overlap. PJ subtracts from Sum all overlaps of the same type of overhead. It is equal to the projection of all overheads to the timeline. EP subtracts the overlap of all types of overheads from PJ. It is equal to the projection of overheads of a particular type excluding all other types of overheads to the timeline. RR uses a reverse ranking algorithm to index overheads and then calculates the cumulative overhead weighted by the ranks. The idea is brought by web page indexing algorithms such as PageRank [77]. Figure 2.11 shows how to calculate the reverse ranking value (RR) of the same workflow graph in Figure 2.10.

$$RR(j_u) = d + (1 - d) \times \sum_{j_v \in Child(j_u)} \frac{RR(j_v)}{L(j_v)} \tag{2.2}$$

Equation 2.2 means that the RR of a node (overhead or job) is determined by the RR of its child nodes. d is the damping factor, which usually is 0.15 as in PageRank. $L(j_v)$ is the number of parents that node j_v has. Intuitively speaking, a node is more important if it has more child nodes and its child nodes are more important. In terms of workflows, it means an overhead has more power to control the release of other overheads and computational activities. There are two differences compared to the original PageRank:

1. We use output link pointing to child nodes while PageRank uses input link from parent nodes, which is why we call it reverse ranking algorithm.

2. Since a workflow is a DAG, we do not need to calculate RR iteratively. For simplicity, we assign the RR of the root node to be 1. And then we calculate the RR of a workflow (G) based on the equation below:

$$RR(G) = \sum RR(j_u) \times \phi_{j_u} \tag{2.3}$$

ϕ_{j_u} indicates the duration of job j_u. RR evaluates the importance of an overhead and represents the cumulative overhead weighted by this importance. The reason we have four metrics of calculating cumulative overheads is to present a comprehensive overview of the impact of overlaps between the various overheads and runtime. Many optimization methods such as Data Placement Services [5] try to overlap overheads and runtime to improve the overall performance. By analyzing these four types of cumulative overheads, researchers have a clearer view of whether their optimization methods have overlapped the overheads of a same type (if $PJ < Sum$) or other types (if $EP < PJ$). RR shows the connectivity within the workflow, the larger the denser. We use a simple example workflow with three jobs to show how to calculate the overlap and cumulative overheads. Figure 2.10 shows the timeline of our example workflow. Job1 is a parent job of Job 2 and Job 3.

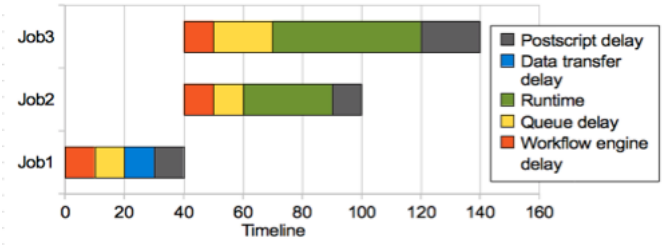

Figure 2.10: The Timeline of an Example Workflow

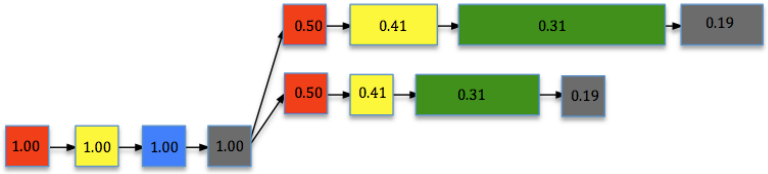

Figure 2.11: Reverse Ranking

At $t = 0$, job 1, a stage-in job, is released: $queue\ delay = 10$, $workflow\ engine\ delay = 10$, $runtime = 10$, and $postscript\ delay = 10$. At $t = 40$, job 3 is released: $workflow\ engine\ delay = 10$, $queue\ delay = 20$, $runtime = 50$, and $postscript\ delay =$

20. At $t = 40$, job 2 is released: $workflow\ engine\ delay = 10$, $queue\ delay = 10$, $runtime = 30$, $postscript\ delay = 10$.

In calculating the cumulative runtime, we do not include the runtime of stage-in jobs because we have already classified it as data transfer delay. The overall makespan for this example workflow is 140. Table 2.1 shows the percentage of overheads and job runtime over makespan.

Table 2.1: Percentage of Overheads and Runtime

Percentage	Sum	PJ	EP	RR
runtime	57.14%	42.86%	28.57%	17.71%
queue delay	28.57%	21.43%	14.29%	13.00%
workflow engine delay	21.43%	14.29%	14.29%	14.29%
postscript delay	28.57%	28.57%	21.43%	11.21%
data transfer delay	7.14%	7.14%	7.14%	7.14%
sum	142.86%	114.29%	85.71%	63.36%

In Table 2.1, we can conclude that the sum of Sum is larger than makespan and smaller than makespan$\times$(number of resources) because it does not count the overlap at all. PJ is larger than makespan since the overlap between more than two types of overheads may be counted twice or more. EP is smaller than makespan since some overlap between more than two types of overheads may not be counted. RR shows how intensively these overheads and computational activities are connected to each other.

2.4 Experiments and Discussion

In this section, we introduce our workflow simulator called WorkflowSim that utilizes the o-DAG model to simulate large scale workflows. We verify its effectiveness through a series of experiments. The evaluation of the performance of workflow optimization techniques in real infrastructures is complex and time consuming. As a result, simulation-based studies have become a widely accepted way to evaluate workflow systems. For example, scheduling algorithms, such as HEFT [106], MaxMin [13], MinMin [11], etc., have used simulators to evaluate their effectiveness. A simulation-based approach reduces the complexity of the experimental setup and saves

much effort in workflow execution by enabling the testing of their applications in a repeatable and controlled environment.

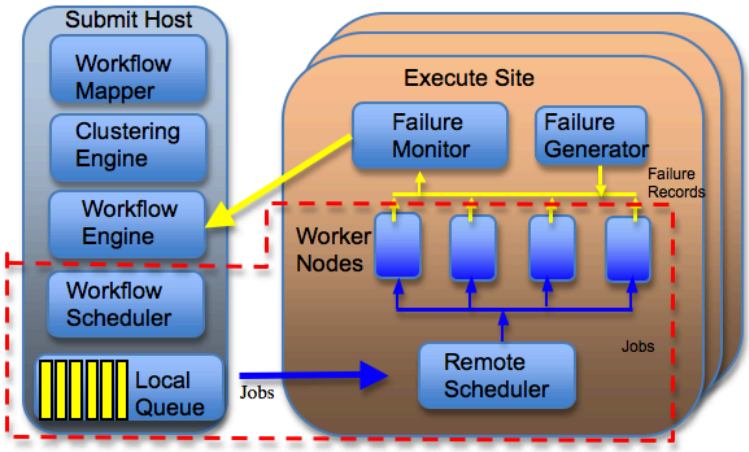

Figure 2.12: WorkflowSim Overview. The area surrounded by red lines is supported by CloudSim

However, an accurate simulation framework for scientific workflows is required to generate reasonable results, particularly considering that the overall system overhead [**?**] plays a significant role in the workflows runtime. 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 [119, 99, 88, 85, 74, **?**] have emphasized the importance of fault tolerant design and concluded that the failure rates in modern distributed systems should not be neglected. A simulation with support for randomization and layered failures is supported 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 [14] fail to provide fine granularity simulations of workflows. For example, they lack the support of task clustering, which is a popular technique that merges small tasks into a large job to reduce task execution overheads. 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.

2.4.1 Components of WorkflowSim

As Fig 2.12 shows, there are multiple layers of components involved in preparing and executing a workflow. Among them, Workflow Mapper, Workflow Engine, Workflow Scheduler and Job Execution have been introduced in last section. Below we introduce three components that have not been introduced.

1. Clustering Engine

   The Clustering Engine merges tasks into jobs so as to reduce the scheduling overheads.

2. Failure Generator component is introduced to inject task/job failures at each execution site. After the execution of each job, Failure Generator randomly generates task/job failures based on the distribution and average failure rate that a user has specified.

3. Failure Monitor collects failure records (e.g., resource id, job id, task id) and returns them to the workflow management system so that it can adjust the scheduling strategies dynamically.

We also modified other components to support fault tolerant optimization. In a failure-prone environment, there are several options to improve workflow performance. First, one can simply retry the entire job or only the failed part of this job when its computation is not successful. This functionality is added to the Workflow Scheduler, which checks the status of a job and takes action based on the strategies that a user selects. Furthermore, Reclustering is a technique that we have proposed [21] that aims to adjust the task clustering strategy based on the detected failure rate. This functionality is added to the Workflow Engine.

2.4.2 Experiments and Validation

We use task clustering as an example to illustrate the necessity of introducing 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 [40]. We built an execution site with 20 worker nodes and we executed the Montage workflow five times in every single configuration of k, which is the maximum number of clustered jobs in a horizontal level. These five traces of workflow execution with the same k 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 k 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:

$$Accuracy = \frac{Predicted\,Overall\ Runtime}{Real\ Overall\ Runtime} \tag{2.4}$$

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. Here, we do not discuss the randomization or distribution of job runtime since we rely on CloudSim to provide a convincing model of job execution.

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 overheads (Case 2), and

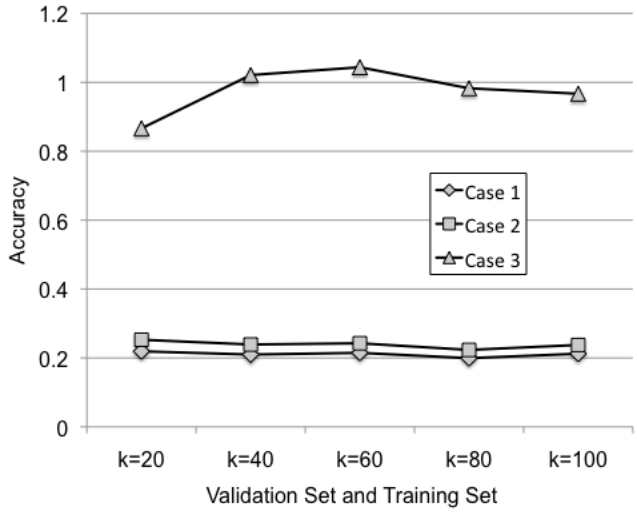

Figure 2.13: Performance of WorkflowSim with different support levels

WorkflowSim, that has covered both aspects (Case 3). Intuitively speaking, we expect that the order of the accuracy of them should be Case 3 $>$ Case 2 $>$ Case 1.

Fig 2.13 shows 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 since the traces within the same training set may have much variance. Fig 2.13 also shows that the accuracy of both Case 1 and Case 2 are much lower than Case 3. The reason why Case 1 does not give an exact result 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.

Chapter 3

Data Aware Workflow Partitioning

When mapping data-intensive tasks to compute resources, scheduling mechanisms need to take into account not only the execution time of the tasks, but also the overheads of staging the dataset. This also applies to the task clustering problem since merging tasks usually means data transfers associated with these tasks have to be merged as well. 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. 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%.

3.1 Motivation

Data movement between tasks in scientific workflows has received less attention compared to task execution. Often the staging of data between tasks is either assumed or the time delay in data transfer is considered to be negligible 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 [40], Teragrid/XSEDE [102] and Amazon EC2 [4] 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.

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 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. A sub-workflow is a workflow and also a job of a higher-level workflow. 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 [28] and MPI/DAGMan [65]) while the traditional task clustering technique requires all the tasks can be executed in the same execution environment.

3.2 Related Work

For convenience and cost-related reasons, scientists execute scientific workflows [10, 35] in distributed large-scale computational environments such as multi-cluster grids, that is, grids comprising multiple independent execution sites. Topcuoglu [106] 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 a hierarchical view of data transfer is necessary. Second, they do not consider the resource availability experienced in grids, which also makes accurate predictions of computation and communication costs difficult. Sonmez [95] 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 [35] and Wieczorek [113] have discussed the scheduling and partitioning scientific workflows in dynamic grids with challenges such as a broad set of unpredictable overheads and possible failures. Duan [35] then developed a distributed service-oriented Enactment Engine with a master-slave architecture for de-centralized coordination of scientific workflows. Kumar [53] proposed the use of graph partitioning for partition the resources of a distributed system, but not the workflow DAG, which means the resources are provisioned into different execution sites but the workflows are not partitioned at all. Dong [33] and Kalayci [51] have discussed the use of graph partitioning algorithms for the workflow DAG according to features of the workflow itself and the status of selected available resource clusters. Our work focuses on the workflow partitioning problem with resource constraints. Compared to Dong [33] and Kalayci [51], we extend their work to estimate the overall runtime of sub-workflows and then schedule these sub-workflows based on the estimates.

Park et al. [78] 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 [78], 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 [84] proposed an automated and trace-based workflow structural analysis method for DAGs. Files transfers are accomplished 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.

Data Placement techniques try to strategically manage placement of data before or during the execution of a workflow. Kosar et al. [52] 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. [5] 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 placement of data based on information collected during task executions and data transfers. Shankar [90] 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 problem but our work can be extended to consider the data placement strategies they have proposed in the future.

Data replication is a common way to increase the availability of data and implicit replication also occurs when scientists download and share the data for experimental purposes, in contrast to explicit replications done by workflow systems. Ranganathan [83] conducted extensive studies for identifying dynamic data or task replication strategies, asynchronous data placement and job and data scheduling algorithms for Data Grids. They concluded through simulations of independent jobs that scheduling jobs to locations that contain the data they need and asynchronously replicating popular data sets to remote sites can improve the performance. We do not address data replication in our current work but it is worthy of further investigation of how data replication can improve the performance of workflow partitioning.

3.3 Approach

To efficiently partition workflows, we proposed 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 (MinMin and HEFT).

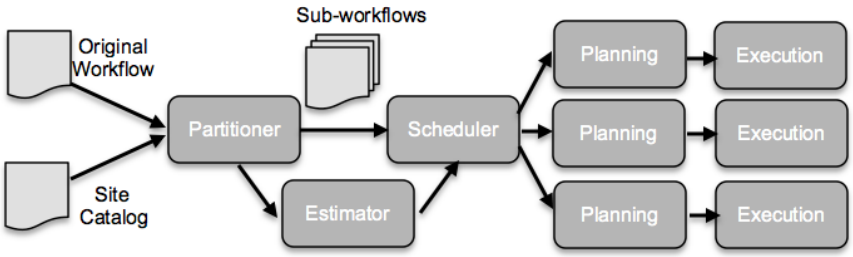

Figure 3.1: The steps to partition and schedule a workflow

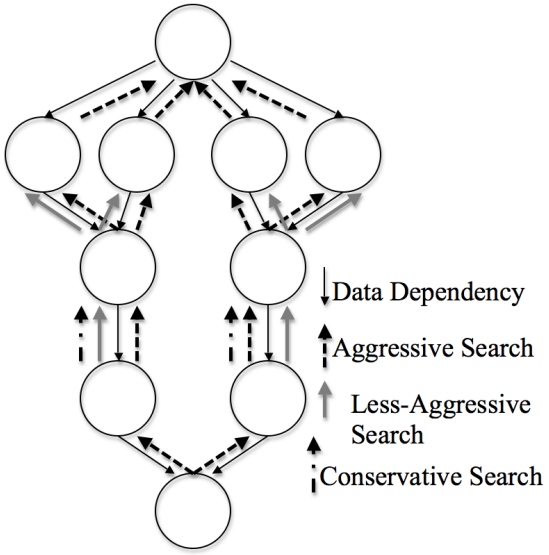

Figure 3.2: Three Steps of Search

Our approach (see Figure 3.1) has three phases: partition, estimate and schedule. The partitioner takes the original workflow and site catalog as input, and outputs various sub-workflows

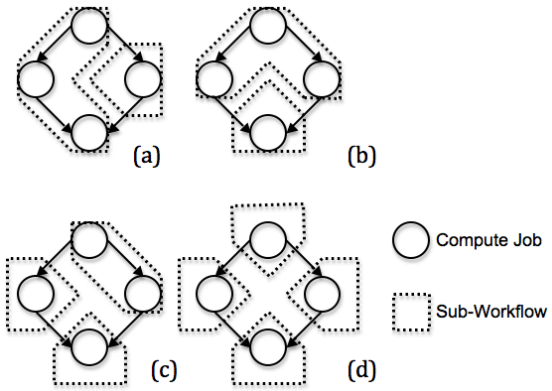

Figure 3.3: Four Partitioning Methods

that respect the storage constraintsthis 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 guarantee to find a valid mapping of sub-workflows satisfying storage constraints. Then we 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 3.3, 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 search 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 [106], 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 its possible to add the whole fan structure into the sub-workflow (aggressive search). If not, similar to Figure 3.3(d), a cut is issued between this fan-in job and its parents to avoid cross dependencies and increase parallelism. Then a less aggressive 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 3.2 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 3.3 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 (its the number of Condor slots in our experiments, which is also the maximum number of Condor jobs that can be run on one machine).

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[106] and MinMin[11], which represent global and local optimizations respectively. But 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 its the major factor influencing the performance. This work can be easily extended to considering more factors. Although some more comprehensive algorithms can be adopted, HEFT or MinMin are able to find an optimal 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.

3.4 Experiments and Discussion

In order to quickly deploy and reconfigure computational resources, we use a private cloud computing resource running Eucalyptus [72]. Eucalyptus is an infrastructure software that provides on-demand access to Virtual Machine (VM) resources. In all the experiments, each VM has

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
```

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 which 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 [28], which provides the workflow execution engine. Each execution site contains a Condor pool and a head node visible to the network.

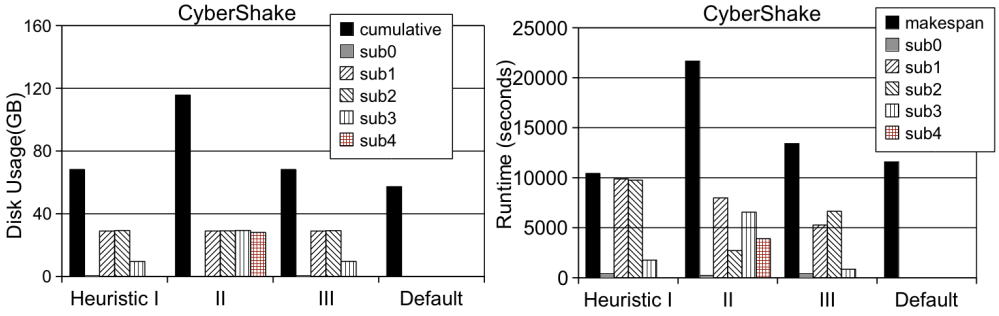

Figure 3.4: Performance of the three heuristics. The default workflow has one execution site with 4 VMs and 8 Condor slots and has no storage constraint.

Table 3.1: 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 that is concerned with in order to minimize the overall makespan. We compare the results of different partitioners, estimators and schedulers.

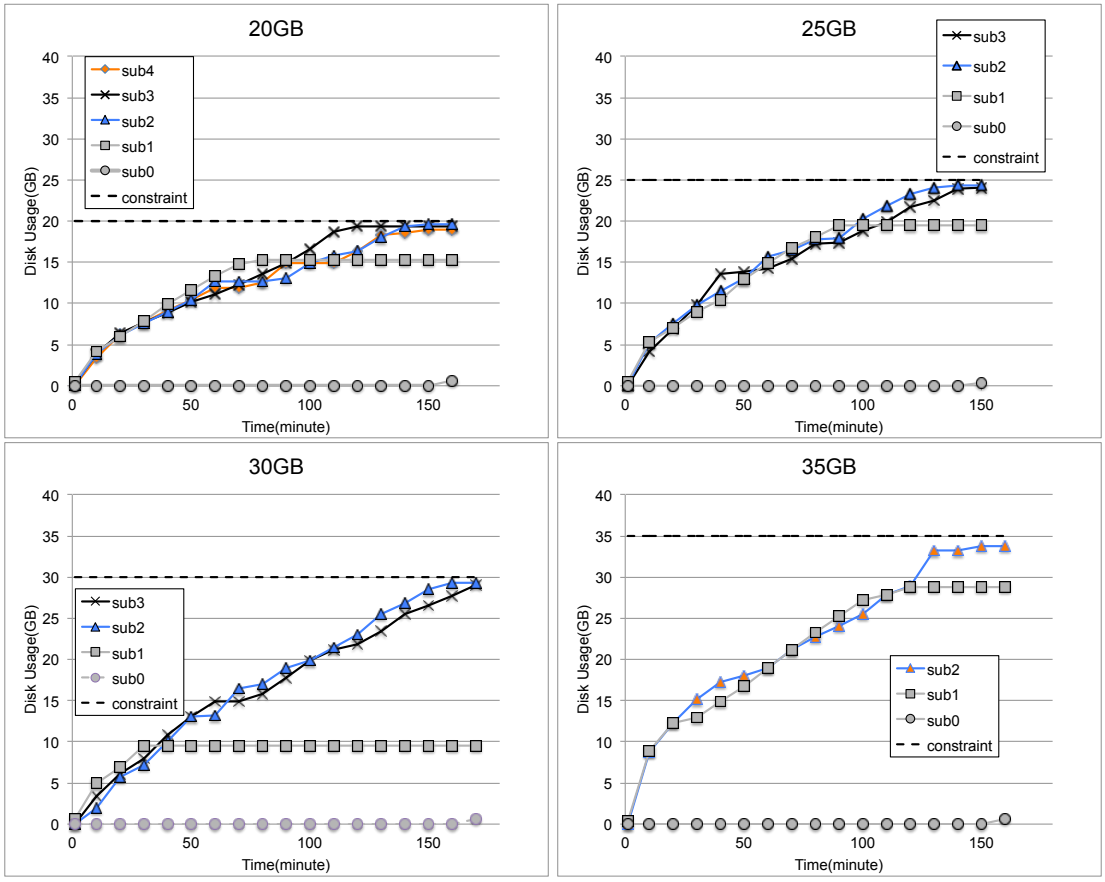

Figure 3.5: CyberShake with storage constraints of 35GB, 30GB, 25GB, and 20GB. They have 3, 4, 4, and 5 sub-workflows and require 2, 3, 3, and 4 sites to run respectively.

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 [50].

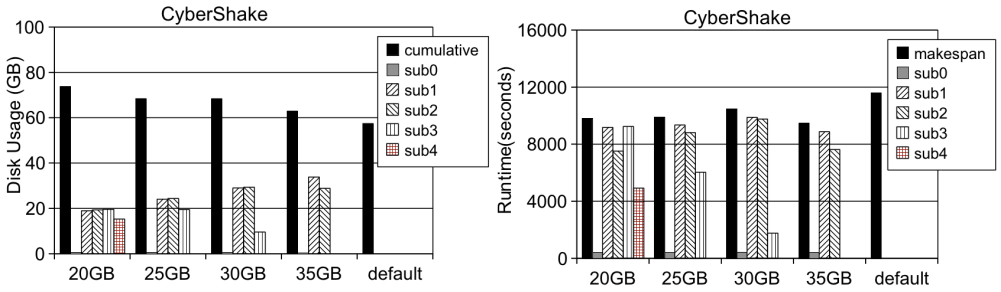

Figure 3.6: Performance of the CyberShake workflow with different storage constraints

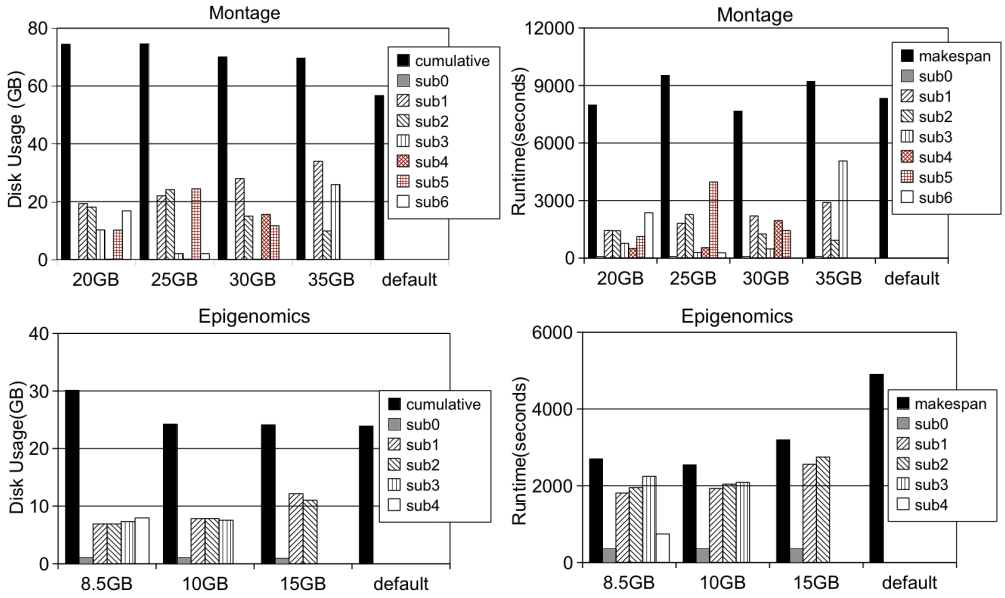

Figure 3.7: Performance of the Montage workflow with different storage constraints

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 3.4 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 3.5 and Table 3.1 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 3.6 depicts the performance of both disk usage and runtime. Storage constraints for 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

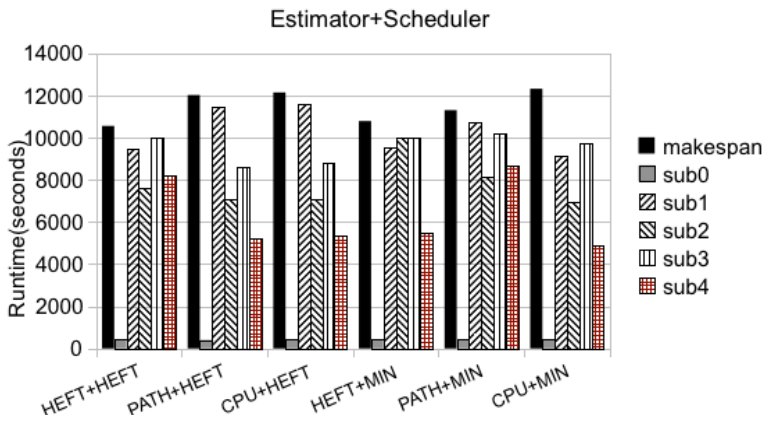

Figure 3.8: Performance of estimators and schedulers

Table 3.2: 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

sites with a storage constraint of 35GB performs best. The makespan (overall completion time) improves by 18.38% and the cumulative disk usage increases by 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 3.7 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 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 3.8 and Table 3.2 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 Average CPU Time (or CPU in Figure 4.7) does not take the dependencies into consideration and the Critical Path (or PATH in Figure 3.8) does not consider the resource availability. The HEFT scheduler is slightly better than MinMin scheduler (or MIN in Figure 3.8). Although HEFT scheduler uses a global optimization algorithm compared to MinMins local optimization, 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.

In conclusion, we provide a solution to address the problem of scheduling large workflows across multiple sites with storage constraints. 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.

Chapter 4

Balanced Clustering

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. In this chapter, we examine the reasons that cause load imbalance in task clustering. Furthermore, we propose a series of task balancing methods to address these imbalance problems. A trace-based simulation shows our methods can significantly improve the runtime performance (the speedup is up to 1.4) of a widely used physics workflow compared to the naive implementation of task clustering.

4.1 Motivation

Existing task clustering strategies have demonstrated their effect in some scientific workflows such as CyberShake [65] and LIGO [31]. 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 [59]. 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 has to do 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 [22] that represents the 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 [69][60] that only examines free tasks to be merged and optimizes the 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 extend the previous work 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 number of resources; and (*iv*) the performance impact of combining our balancing methods with vertical clustering.

4.2 Related Work

Overhead analysis [76, 82] is a topic of great interest in the Grid community. Stratan et al. [97] evaluate in a real-world environment Grid workflow engines including DAGMan/Condor and Karajan/Globus. Their methodology focuses on five system characteristics: overhead, raw performance, stability, scalability, and reliability. They pointed out that head node consumption should not be negligible and the main bottleneck in a busy system is often the head node. Prodan et al. [82] offered a complete Grid workflow overhead classification and a systematic measurement of overheads. In Chen et al. [20], we extended [82] by providing a measurement of major overheads imposed by workflow management systems and execution environments and analyzed how existing optimization techniques improve runtime by reducing or overlapping overheads. The prevalent existence of system overheads is an important reason why task clustering provides significant performance improvement for workflow-based applications. In this paper, we aim to further improve the performance of task clustering under imbalanced load.

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. [69] 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 [68] to determine task granularity based on task file size, CPU time, and resource constraints. Recently, they proposed an online scheduling algorithm [67, 70] that groups tasks based on resource network utilization, user's budget, and application deadline. Ng et al. [71] and Ang

et al. [103] 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 [60] 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.

Task granularity control has also been addressed in scientific workflows. For instance, Singh et al. [91] proposed a level- and label-based clustering. In level-based clustering, tasks at the same level can be clustered together. The number of clusters or tasks per cluster are 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, which is prone to errors. Recently, Ferreira da Silva et al. [38] 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.

A plethora of balanced scheduling algorithms have been developed in the networking and operating system domains. Many of these schedulers have been extended to the hierarchical setting. Lifflander et al. [59] proposed to use work stealing and a hierarchical persistence-based rebalancing algorithm to address the imbalance problem in scheduling. Zheng et al. [121] 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 [13] (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.

Workflow patterns [116, 49, 61] are used to capture and abstract the common structure within a workflow and they give insights on designing new workflows and optimization methods. Yu and Buyya [116] proposed a taxonomy that characterizes and classifies various approaches for building and executing workflows on Grids. They also provided a survey of several representative Grid workflow systems developed by various projects world-wide to demonstrate the comprehensiveness of the taxonomy. Juve et al. [49] provided a characterization of workflow from 6 scientific applications and obtained task-level performance metrics (I/O, CPU, and memory consumption). They also presented an execution profile for each workflow running at a typical scale and managed by the Pegasus workflow management system [32]. Liu et al. [61] proposed a novel pattern based time-series forecasting strategy which utilizes a periodical sampling plan to build representative duration series. We illustrate the relationship between the workflow patterns (asymmetric or symmetric workflows) and the performance of our balancing algorithms.

Some work in the literature has further attempted to define and model workflow characteristics with quantitative metrics. In [3], the authors proposed a robustness metric for resource allocation in parallel and distributed systems and accordingly customized the definition of robustness. Tolosana et al. [104] defined a metric called Quality of Resilience to assess how resilient workflow enactment is likely to be in the presence of failures. Ma et al. [62] proposed a graph distance based metric for measuring the similarity between data oriented workflows with variable time constraints, where a formal structure called time dependency graph (TDG) is proposed and further used as representation model of workflows. Similarity comparison between two workflows can be reduced to computing the similarity between TDGs. In this paper, we focus on novel quantitative metrics that are able to demonstrate the imbalance problem in scientific workflows.

4.3 Approach

Task clustering has been widely used to address the low performance of very short running tasks on platforms where the system overhead is high, such as distributed computing infrastructures.

However, up to now, techniques do not consider the load balance problem. In particular, merging tasks within a workflow level without considering the runtime variance may cause load imbalance (Runtime Imbalance), or merging tasks without considering their data dependencies may lead to data locality problems (Dependency Imbalance). In this section, we introduce metrics that quantitatively capture workflow characteristics to measure runtime and dependence imbalances. We then present methods to handle the load balance problem.

4.3.1 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 4.1 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 4.1-top). In contrast, a balanced clustering strategy should try its best to evenly distribute task runtime among jobs as shown in Figure 4.1 (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.

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 4.2, 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 4.2) forces t_5 and t_6 to transfer files from two locations and wait for

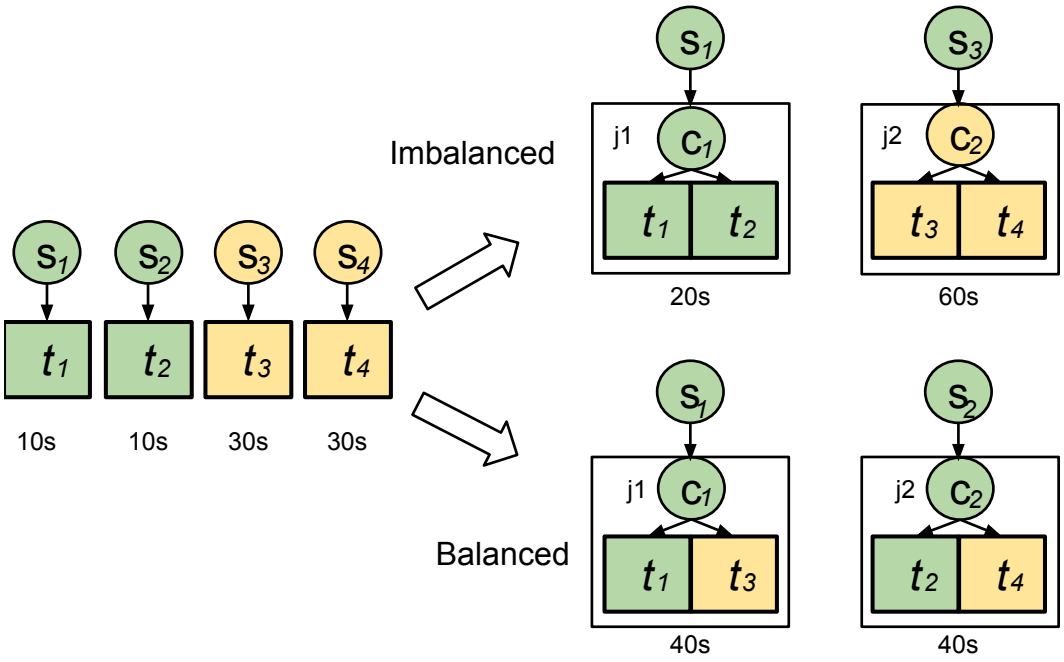

Figure 4.1: An example of Horizontal Runtime Variance.

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:

$$IF(t_u) = \sum_{t_v \in Child(t_u)} \frac{IF(t_v)}{||Parent(t_v)||} \tag{4.1}$$

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 4.2) is 1.0. For

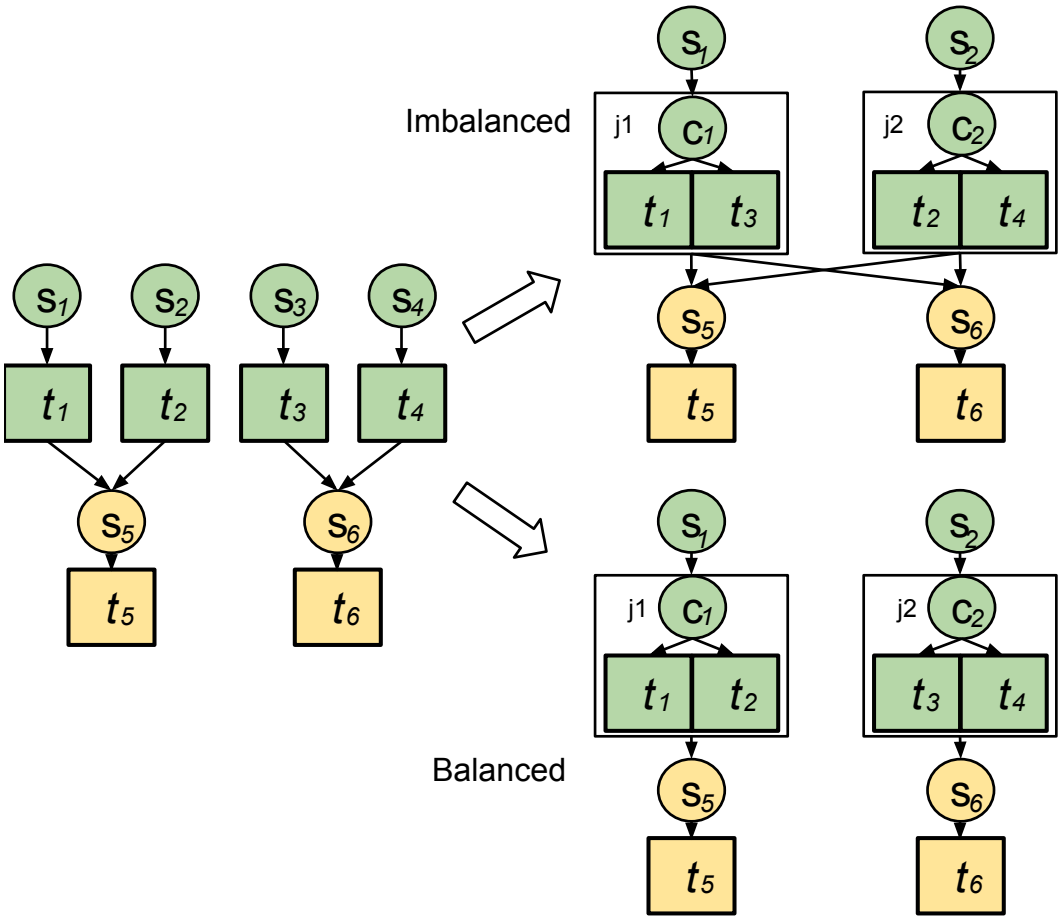

Figure 4.2: An example of Dependency Imbalance.

instance, consider the two workflows presented in Figure 4.3. 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 4.3 (left). In this paper, 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 $O(n)$.

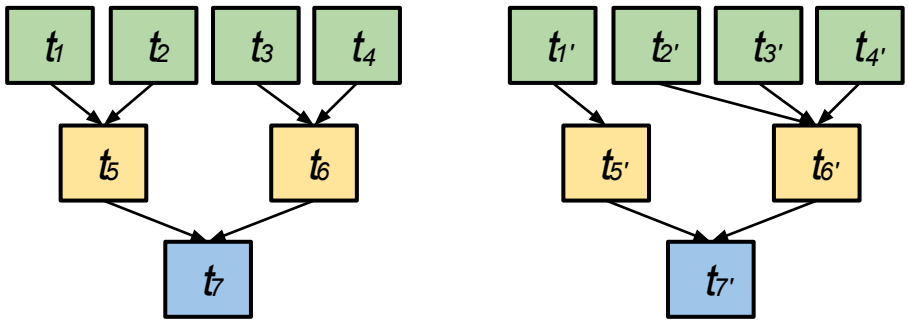

Figure 4.3: Example of workflows with different data dependencies (For better visualization, we do not show system overheads in the rest of the paper).

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 4.1 shows the distance matrices of tasks from the first level for both workflows

of Figure 4.3 (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 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 4.1: Distance matrices of tasks from the first level of workflows in Figure 4.3.

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

4.3.2 Balanced clustering methods

In this subsection, we introduce our balanced clustering methods used to improve the runtime and dependency balances 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 4.4 shows an example of HRB where tasks in the first level have different runtimes and should be grouped into two 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.

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 2 Horizontal Runtime 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**
    $J \leftarrow$ the job with shortest runtime and less than C tasks
    J.add (t) ▷ Adds the task to the shortest job
  end for
  for $i < R$ **do**
    CL.add(J_i)
  end for
  return CL
end procedure

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

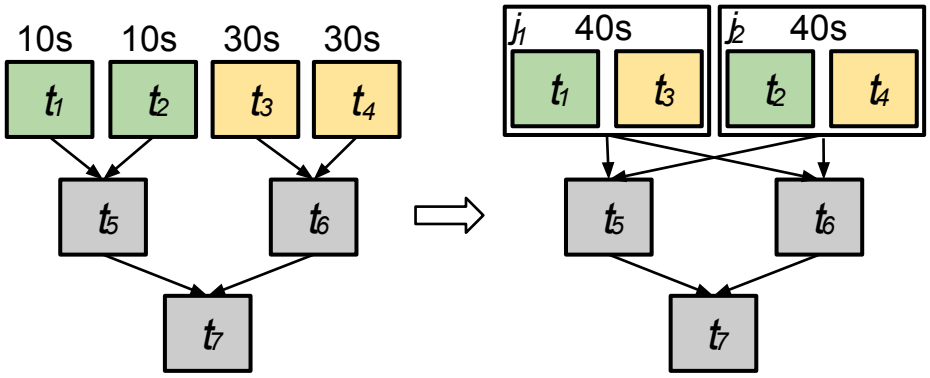

Figure 4.4: An example of the HRB (Horizontal Runtime Balancing) method. By solely addressing runtime variance, data locality problems may arise.

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 4.5, 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.

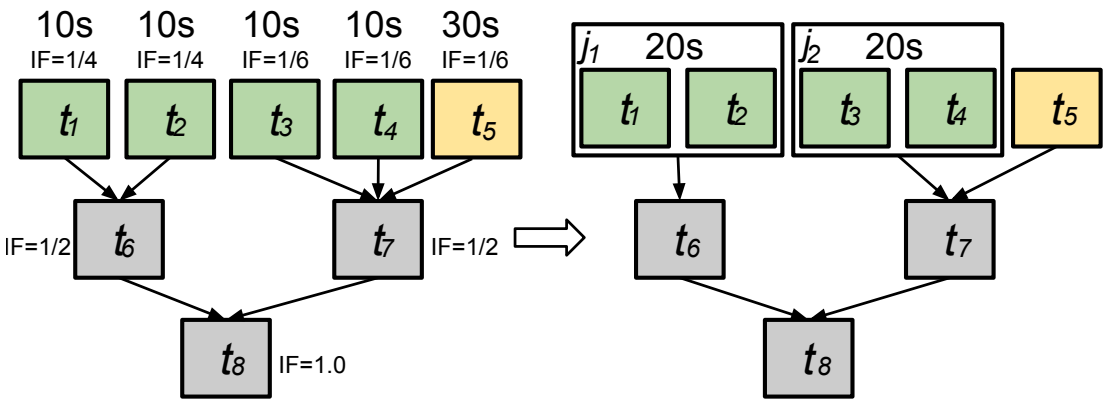

Figure 4.5: An example of the HIFB (Horizontal Impact Factor Balancing) method. Impact factors allow the detection of similarities between tasks.

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 4.4, 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 4.6, 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 4.5), 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 4.7. 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 impact factors are slightly different. Example of such scientific workflows include the LIGO Inspiral workflow [55], which is used in the evaluation of this paper (Section 4.4.4).

Table 4.2 summarizes the imbalance metrics and balancing methods presented in this paper.

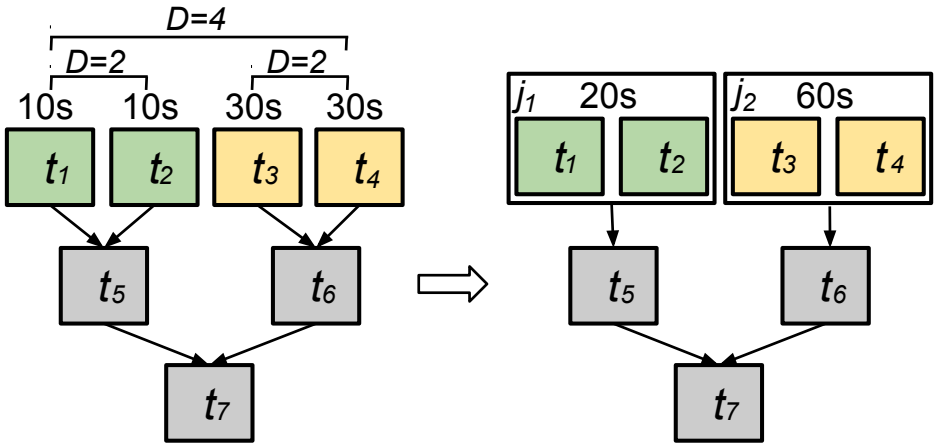

Figure 4.6: An example of the HDB (Horizontal Distance Balancing) method. Measuring the distances between tasks avoids data locality problems.

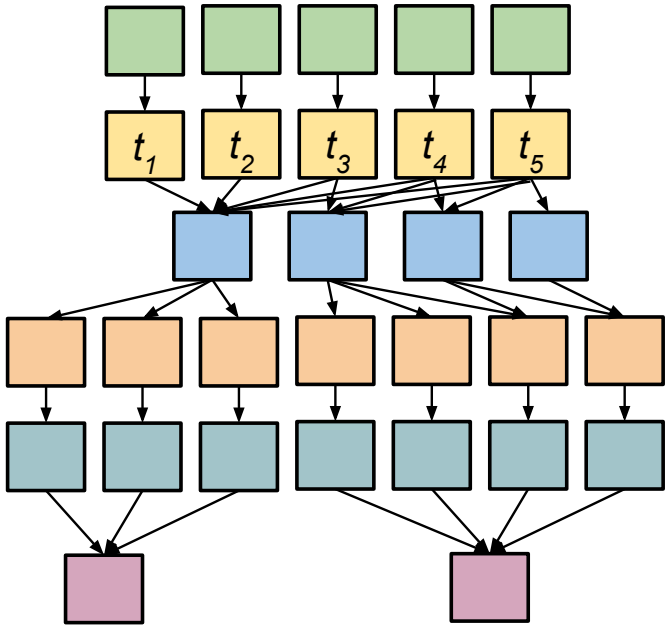

Figure 4.7: A workflow example where HDB yields lower performance than HIFB. HDB does not capture the difference in the number of parents/child tasks, since the distances between tasks (t_1, t_2, t_3, t_4, and t_5) are the same.

4.3.3 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

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 4.2: Summary of imbalance metrics and balancing methods.

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 4.8 shows an example where pipelined-tasks are merged first, and then the merged pipelines are horizontally clustered based on the runtime variance.

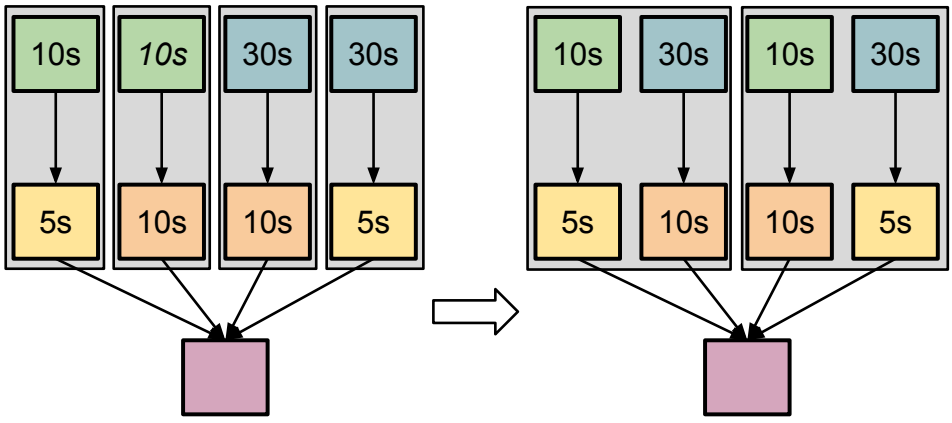

Figure 4.8: *VC-prior*: vertical clustering is performed first, and then the balancing methods.

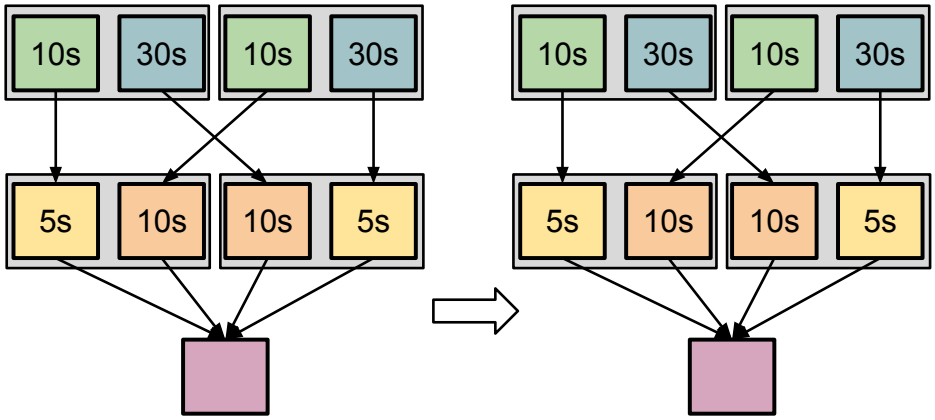

Figure 4.9: *VC-posterior*: horizontal clustering (balancing methods) is performed first, and then vertical clustering (but without changes).

VC-posterior Here, balancing methods are first applied, and then vertical clustering. Figure 4.9 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.

4.4 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) [91], which is widely used by workflow management systems such as Pegasus [30]. We also compare our methods with two heuristics described in literature: DFJS [69], and AFJS [60]. 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.

4.4.1 Scientific workflow applications

Five real scientific workflow applications are used in the experiments: LIGO Inspiral analysis [55], Montage [8], CyberShake [41], Epigenomics [108], and SIPHT [92]. In this subsection, we describe each workflow application and present their main characteristics and structures.

LIGO Laser Interferometer Gravitational Wave Observatory (LIGO) [55] workflows are used to search for gravitational wave signatures in data collected by large-scale interferometers. The 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 4.10 shows a simplified version of this workflow. The LIGO Inspiral workflow is separated into multiple groups of interconnected tasks, which we call branches in the rest of our paper. However, each branch may have a different number of pipelines as shown in Figure 4.10.

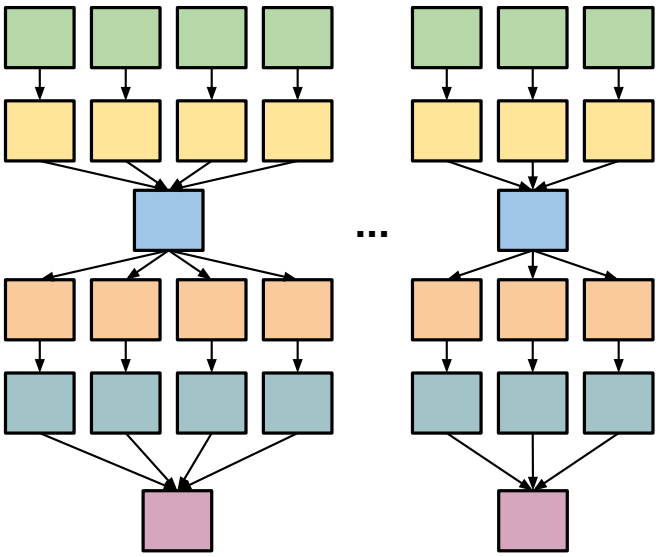

Figure 4.10: A simplified visualization of the LIGO Inspiral workflow.

Montage Montage [8] 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 re-projects 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 4.11 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. The structure of the workflow changes to accommodate increases in the number of inputs, which corresponds to an increase in the number of computational tasks.

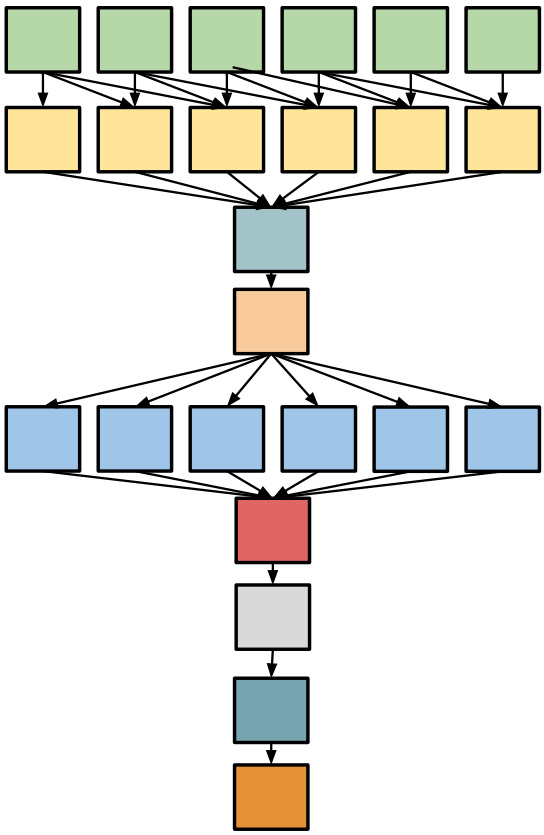

Figure 4.11: A simplified visualization of the Montage workflow.

Cybershake CyberShake [41] 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 4.12 shows an illustration of the Cybershake workflow.

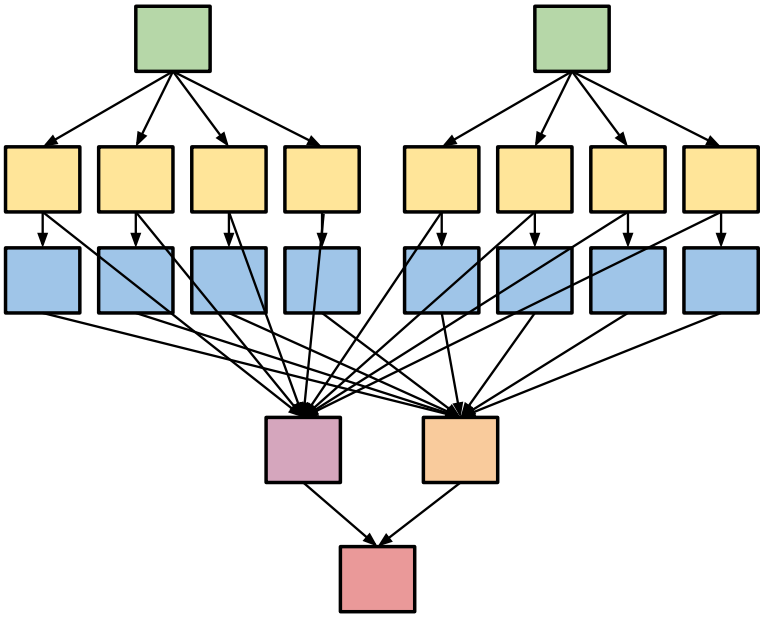

Figure 4.12: A simplified visualization of the CyberShake workflow.

Epigenomics The Epigenomics workflow [108] is a data-parallel workflow. Initial data are acquired from the Illumina-Solexa Genetic Analyzer in the form of DNA sequence lanes. Each Solexa machine can generate multiple lanes of DNA sequences. These data are converted into a format that can be used by sequence mapping software. The mapping software can do one of two major tasks. It either maps short DNA reads from the sequence data onto a reference genome, or it takes all the short reads, treats them as small pieces in a puzzle and then tries to assemble an entire genome. In our experiments, 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. Epigenomics is a CPU-intensive application and its simplified structure is shown in Figure 4.13.

Different to the LIGO Inspiral workflow, each branch in Epigenomics has exactly the same number of pipelines, which makes it more symmetric.

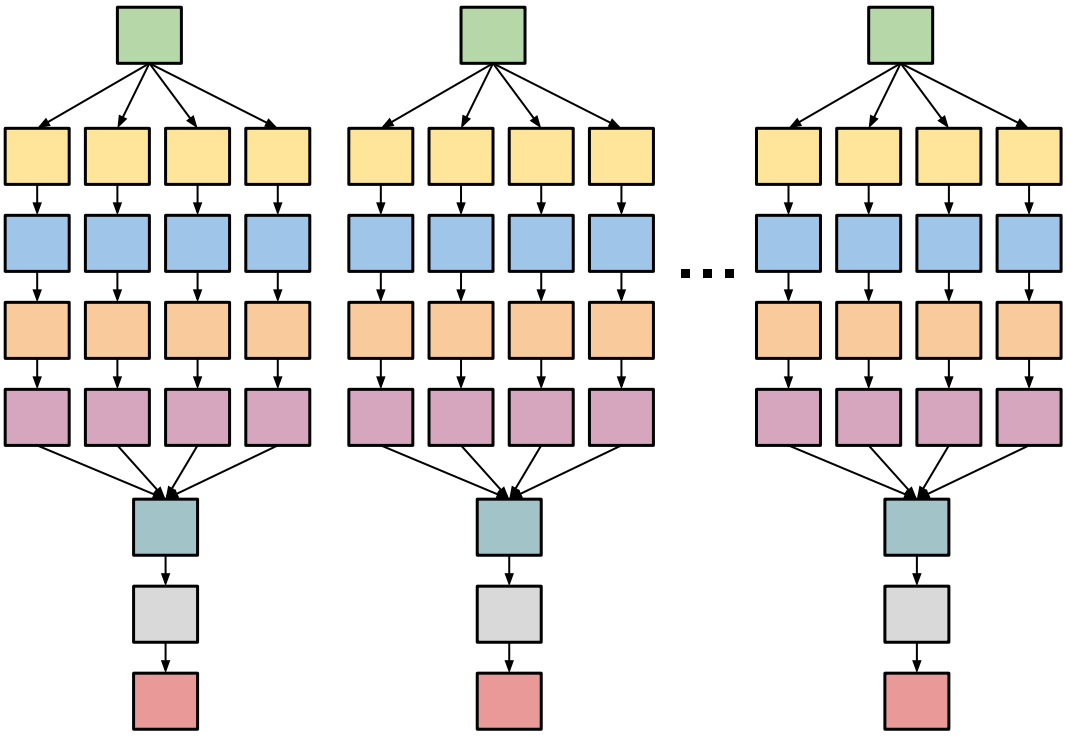

Figure 4.13: A simplified visualization of the Epigenomics workflow with multiple branches.

SIPHT The SIPHT workflow [92] 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 [30]. These involve the prediction of ρ-independent transcriptional terminators, BLAST (Basic Local Alignment Search Tools [92]) 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 4.14.

Table 5.1 shows the summary of the main **workflows characteristics**: number of tasks, average data size, and average task runtimes for the five workflows.

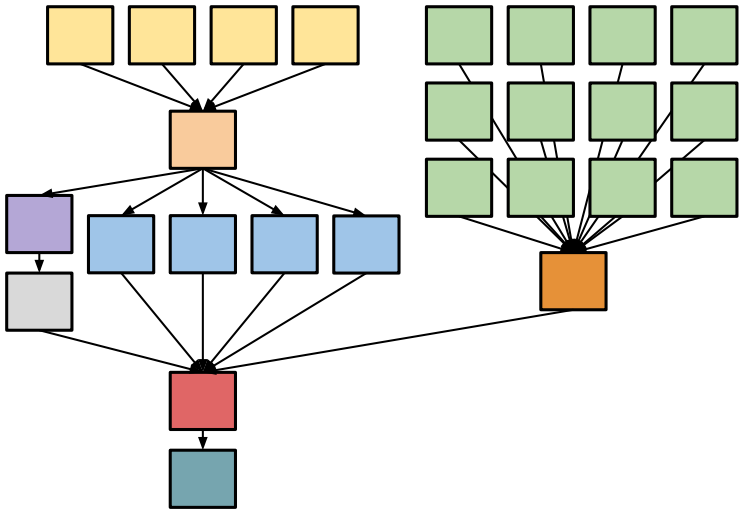

Figure 4.14: A simplified visualization of the SIPHT workflow.

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 4.3: Summary of the scientific workflows characteristics.

4.4.2 Task clustering techniques

In the experiments, we compare the performance of our balancing methods to the Horizontal Clustering (HC) [91] technique, and with two methods well known from the literature, DFJS [69] and AFJS [60]. 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 [91]. For simplicity, we define *clusters.num* as the

number of available resources. In our prior work [25], we have compared the runtime performance with different clustering granularity. The pseudocode of the HC technique is shown in Algorithm 8.

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. [69]. 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.

AFJS The adaptive fine-grained job scheduler (AFJS) [60] 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,

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

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: $TL \leftarrow$ GETTASKSATLEVEL($W, level$) ▷ Partition W based on depth
4: $CL \leftarrow$ MERGE($TL, max.runtime, max.datasize$) ▷ Returns a list of clustered jobs
5: $W \leftarrow W - TL + CL$ ▷ Merge dependencies as well
6: **end for**
7: **end procedure**
8: **procedure** MERGE($TL, max.runtime, max.datasize$)
9: $J \leftarrow \{\}$ ▷ An empty job
10: $CL \leftarrow\{\}$ ▷ An empty list of clustered jobs
11: **while** TL is not empty **do**
12: $t \leftarrow TC$.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: $J \leftarrow\{\}$
16: **end if**
17: J.add(t)
18: **end while**
19: **return** CL
20: **end procedure**

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 4.4.1. 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 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 4.15—experiment conditions are presented in Section 4.4.3. 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 negligible.

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

4.4.3 Experiment conditions

We adopt a trace-based simulation approach, where we extended our WorkflowSim [23] simulator with the balanced clustering methods and imbalance metrics to simulate a controlled distributed environment. WorkflowSim is a workflow simulator that extends CloudSim [14] by providing support for task clustering, task scheduling, and resource provisioning at the workflow

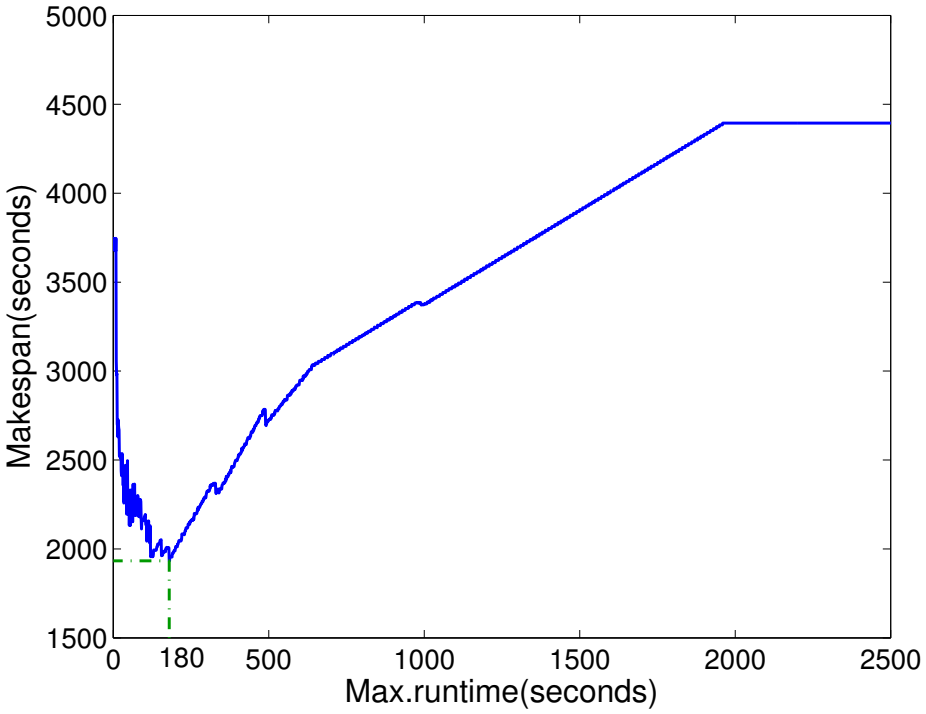

Figure 4.15: Relationship between the makespan of workflow and the specified maximum runtime in DFJS (Montage).

level. It has been recently used in multiple workflow study areas [23, 21, 48] and its correctness has been verified in [23].

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 [4] and FutureGrid [40]. Amazon EC2 is a commercial, public cloud that has been widely used in distributed computing, in particular for scientific workflows [9]. FutureGrid is a distributed, high-performance testbed that provides scientists with a set of computing resources to develop parallel, grid, and cloud applications. 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 [25], which shows such selection is acceptable.

We collected workflow execution traces [49, 20] (including overhead and task runtime information) from real runs (executed on FutureGrid and Amazon EC2) of the scientific workflow applications described in Section 4.4.1. The traces are used to feed the Workflow Generator toolkit [114] 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 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 5.1.

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 4.4 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 4.4: Combination Results. '-' indidates the order of performing these algorithms, i.e., VC-HIFB indicates we perform VC first and then HIFB

4.4.4 Results and discussion

Experiment 1 Figure 4.16 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 4.5. 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 4.5. 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 4.13) 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.

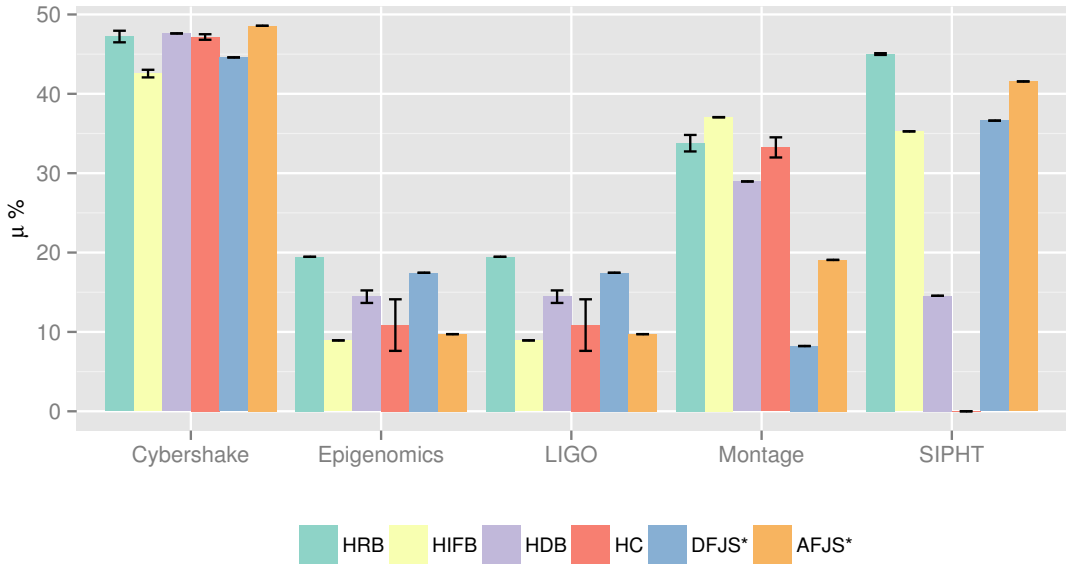

Figure 4.16: Experiment 1: performance gain (μ) over a baseline execution for six algorithms (* indicates the tuned performance of DFJS and AFJS). By default, we have 20 VMs.

Experiment 2 Figure 4.17 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

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

improvement among these balancing methods is significantly different for LIGO compared to other workflows as shown in Figure 4.16. For small data sizes (up to 100 MB), the application is 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 4.10, however, as mentioned in Section 4.3.2, the LIGO workflow has a few branches that depend on each other as shown in Figure 4.7. 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 and more constrained by the interdependent branches, which is shown in Figure 4.17. 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.

Figures 4.18 and 4.19 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 4.18), 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 4.19), data dependencies have more impact on the performance than the runtime variation. In particular, in the high contention scenario HDB performs poor clustering leading the system to data

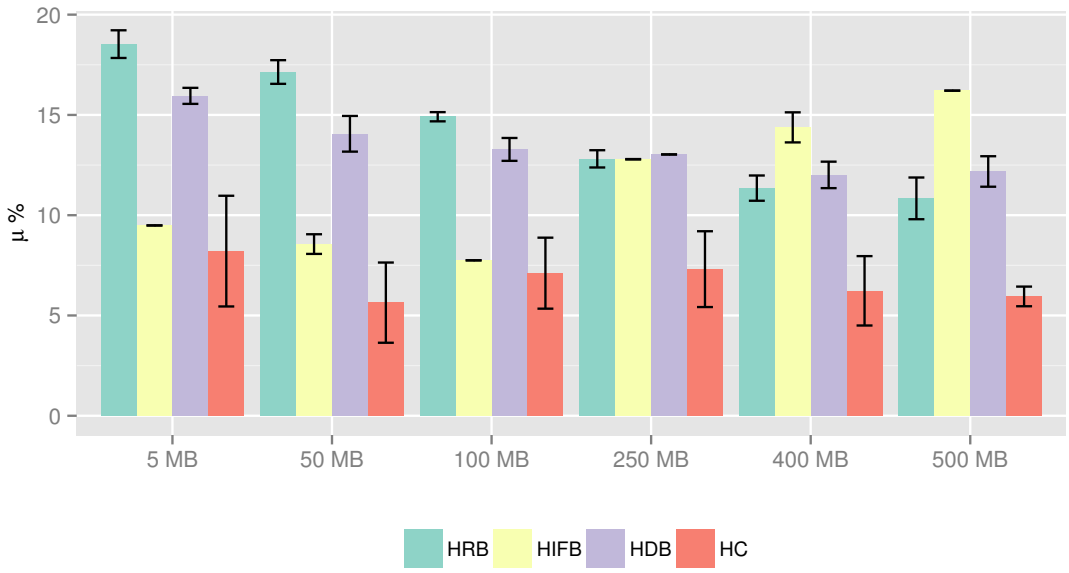

Figure 4.17: Experiment 2: performance gain (μ) over a baseline execution with different average data sizes for the LIGO workflow. The original avg. data size is 5MB.

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. Similarly to the CPU-intensive case, under low contention, runtime variance increases its importance and then HRB performs better.

Experiment 3 Figure 4.20 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 ($\mu(\textit{VC-only}) \approx 0.2\%$), because the workflow structure has no explicit pipeline (see Figure 4.12). Similarly, VC does not improve the SIPHT workflow due to the lack of pipelines on its structure (Figure 4.14). Thus, results for this workflow are omitted.

Figure 4.21 shows the performance gain μ for the Montage workflow. In this workflow, vertical clustering is often performed on the two pipelines (Figure 4.11). These pipelines are commonly single-task levels, thereby no horizontal clustering is performed on the pipelines. As

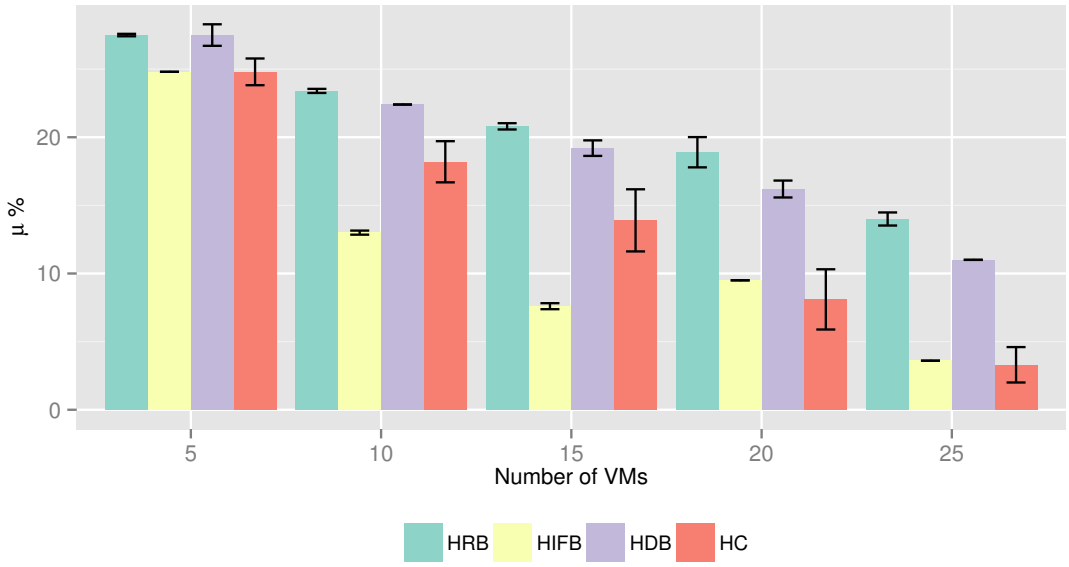

Figure 4.18: Experiment 2: performance gain (μ) over baseline execution with different number of resources for the LIGO workflow (average data size is 5MB).

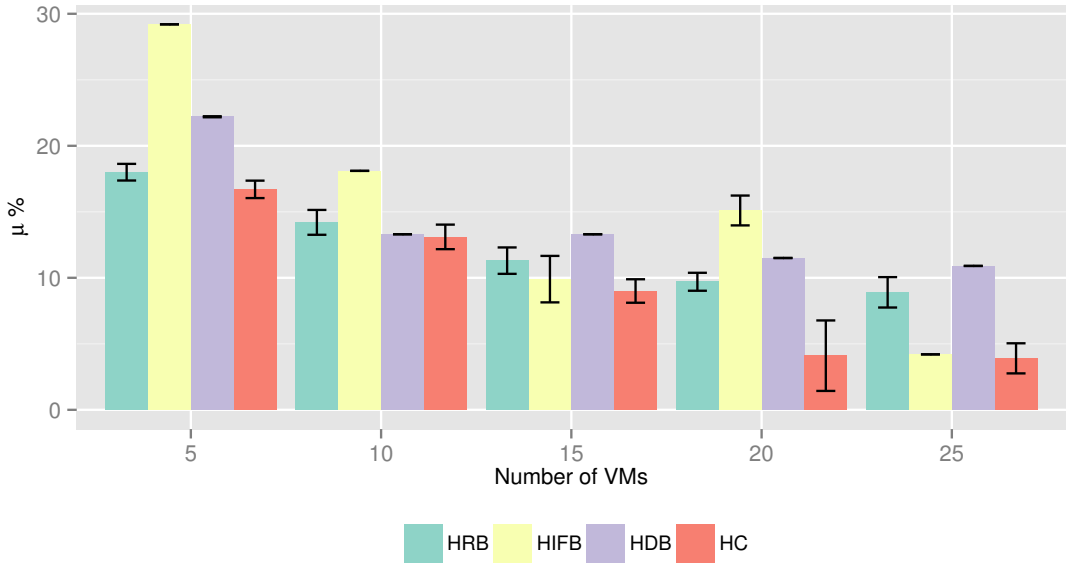

Figure 4.19: Experiment 2: performance gain (μ) over baseline execution with different number of resources for the LIGO workflow (average data size is 500MB).

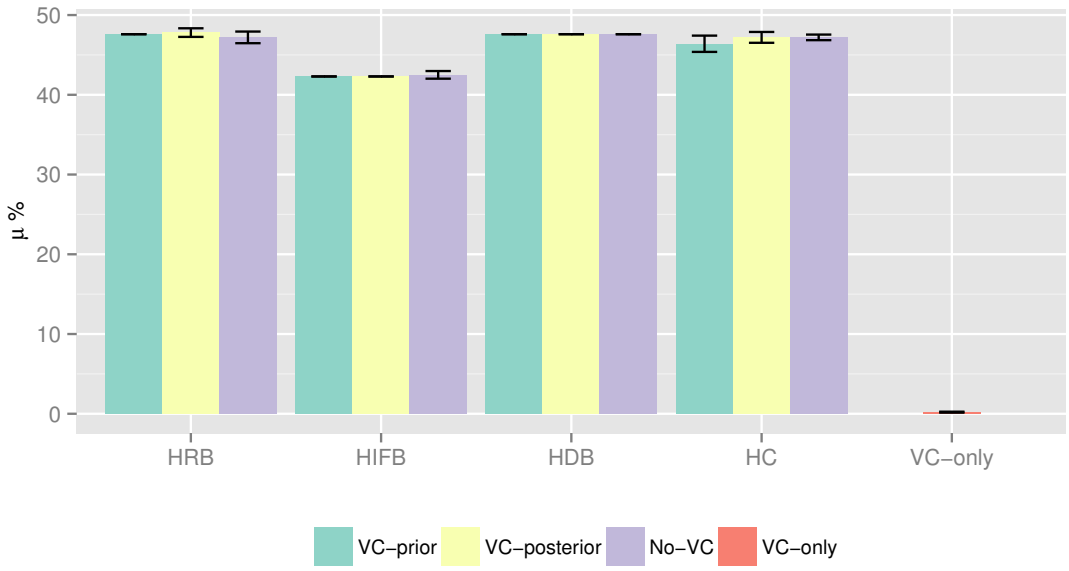

Figure 4.20: Experiment 3: performance gain (μ) for the Cybershake workflow over baseline execution when using vertical clustering (VC).

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.

The performance gain μ for the LIGO workflow is shown in Figure 4.22. Vertical clustering yields better performance gain when it is performed prior to horizontal clustering (*VC-prior*). The LIGO workflow structure (Figure 4.10) has several pipelines that when primarily clustered vertically reduce system overheads (e.g. queuing and scheduling times). 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.

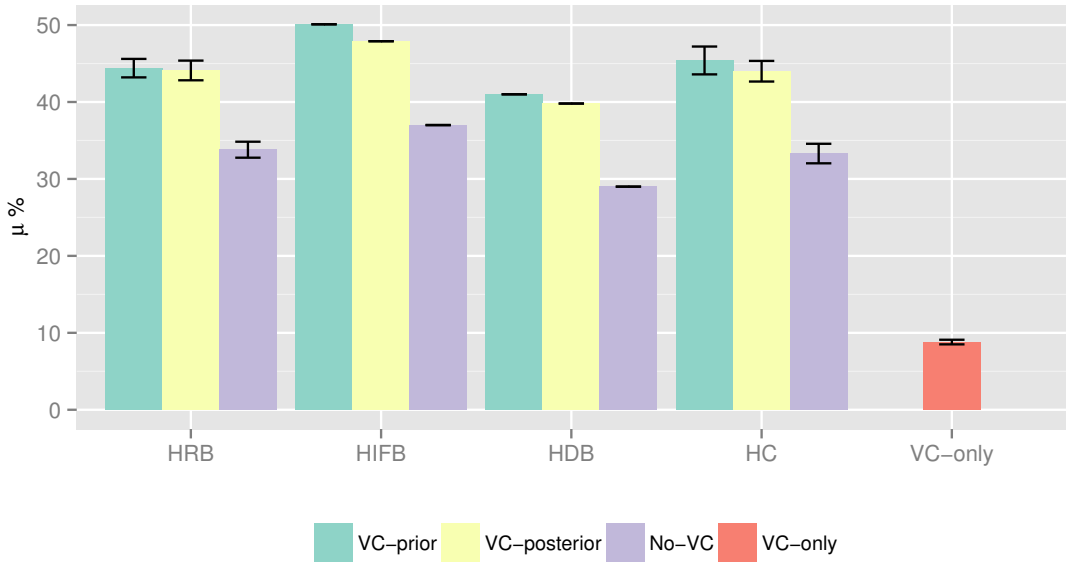

Figure 4.21: Experiment 3: performance gain (μ) for the Montage workflow over baseline execution when using vertical clustering (VC).

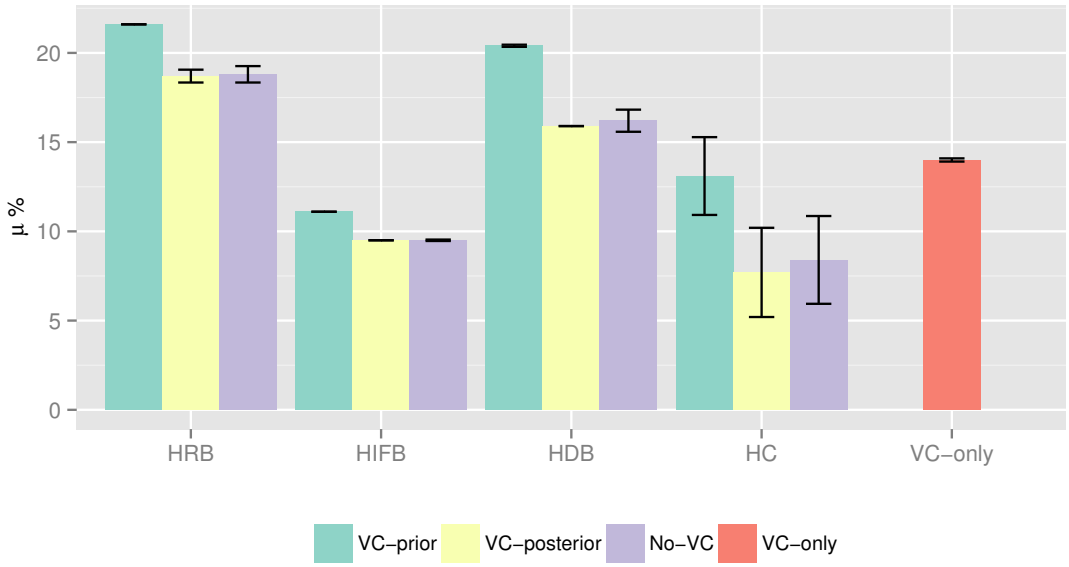

Figure 4.22: Experiment 3: performance gain (μ) for the LIGO workflow over baseline execution when using vertical clustering (VC).

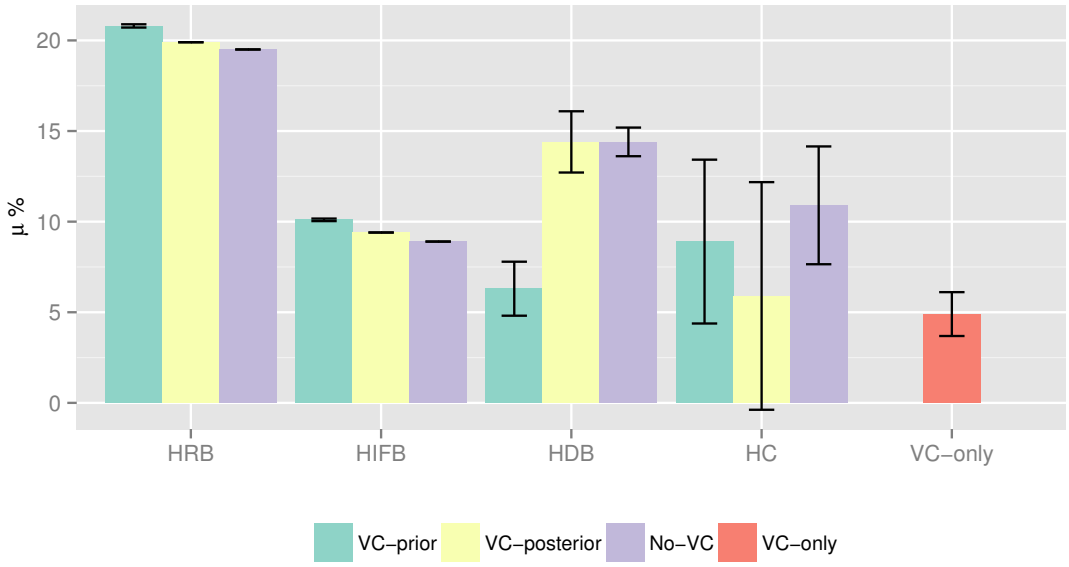

Figure 4.23: Experiment 3: performance gain (μ) for the Epigenomics workflow over baseline execution when using vertical clustering (VC).

Similarly to the LIGO workflow, the performance gain μ values for the Epigenomics workflow (see Figure 4.23) are better when VC is performed *a priori*. This is due to several pipelines inherent to the workflow structure (Figure 4.13). However, vertical clustering has poorer performance if it is performed prior to the HDB algorithm. The reason is the average task runtime of Epigenomics is much larger than other workflows as shown in Table. 5.1. 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.

4.5 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, but 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.

In the future, we plan to further analyze the imbalance metrics proposed. For instance, the values of these metrics presented in this paper are not normalized, and thus their values per level (HIFV, HDV, and HRV) are in different scales. Also, we plan to analyze more workflow applications, particularly the ones with asymmetric structures, to investigate the relationship between workflow structures and the metric values.

Also, as shown in Figure 4.23, *VC-prior* can generate very large clustered jobs vertically and makes it difficult for horizontal methods to improve further. Therefore, we aim to develop imbalance metrics for *VC-prior* to avoid generating large clustered jobs, i.e., based on the accumulated runtime of tasks in a pipeline.

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 most suitable one according to the dynamic load.

In this paper, we demonstrate the performance gain of combining horizontal clustering methods and vertical clustering. We plan to combine multiple algorithms together instead of just two. We will 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, we aim at applying our metrics 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 work, 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.

Chapter 5

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 environments. A trace based simulation is performed and it shows that our methods improve the workflow makespan significantly for five important applications.

5.1 Motivation

Task clustering is an effective method to reduce scheduling overhead and increase the computational granularity of tasks executing on distributed resources. 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 indicate that such failures can have a significant impact on the runtime performance of workflows under existing clustering strategies that ignore failures.

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 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 [30], ASKALON [37] and Chemomentum [89]. 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 [119]. Third, tasks can be replicated to different nodes to avoid failures that are related to one specific worker node [79]. However, inappropriate 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 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 level (defined as the longest distance from the entry task of the workflow) of the workflows has the same type. The characteristics of tasks such as the task runtime, memory peak and disk usage are highly related to the task type [27, 49]. Task type related failure is a type of failure that only occurs to some specific types of tasks. Samak [87] 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 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 fail or not. The job-level fault tolerant clustering has been discussed in our prior work [21].

Compared to our prior work in [21], 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 Maximum Likelihood Estimation (MLE) that has been recently used in machine learning. Prior knowledge about the parameters are modeled as a distribution with known parameters. Posterior knowledge about the execution information are also modeled 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 are in the same family if the likelihood distribution follows some specific distribution and they are called *conjugate distributions*. 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.

The two horizontal methods were introduced and evaluated in [21] on two workflows. We complement this previous paper by studying (*i*) the performance gain of using two horizontal methods and one vertical method over a baseline execution on a larger set of workflows (five widely used scientific applications); (*ii*) the performance impact of the variance of the distribution of the task runtime, the system overheads and the inter-arrival time of failures; (*iii*) the performance difference of dynamic estimation and static estimation with variation of inter-arrival time of failures.

5.2 Related Work

Failure analysis and modeling [99] presents system characteristics such as error and failure distribution and hazard rates. Schroeder et al. [88] has studied the statistics of the data, including the root cause of failures, the mean time between failures, and the mean time to repair. Sahoo et al. [85] analyzed the empirical and statistical properties of system errors and failures from a network of heterogeneous servers running a diverse workload. Oppenheimer et al. [74] analyzed the causes of failures from three large-scale Internet services and the effectiveness of various techniques for preventing and mitigating service failure. McConnel [66] analyzed the transient errors in computer systems and showed that transient errors follow a Weibull distribution. In [98, 46] Weibull distribution is one of the best fit for the workflow traces they used. Based on these work, we measure the inter-arrival time of failures in a workflow and then provide methods to improve task clustering.

More and more workflow management systems are taking fault tolerance into consideration. The Pegasus workflow management system [30] has incorporated a task-level monitoring system and used job retry to address the issue of task failures. They also used provenance data to track the failure records and analyzed the causes of failures [87]. Plankensteiner et. al. [80] have surveyed the fault detection, prevention and recovery techniques in current grid workflow management systems such as ASKALON [37], Chemomentum [89], Escogitare [54] and Triana [101]. Recovery techniques such as replication, checkpointing, task resubmission and task migration etc. have been provided. We are specifically joining the work of failure analysis and the optimization in task clustering. To be best of our knowledge, none of existing workflow management systems have provided such features.

Overhead analysis [76, 82] is a topic of great interest in the Grid community. Stratan et al. [97] evaluate in a real-world environment Grid workflow engines including DAGMan/Condor and Karajan/Globus. Their methodology focuses on five system characteristics: overhead, raw performance, stability, scalability, and reliability. They pointed out that head node consumption should not be negligible and the main bottleneck in a busy system is often the head node. Prodan

et al. [82] offered a complete Grid workflow overhead classification and a systematic measurement of overheads. In Chen et al. [20], we extended [82] by providing a measurement of major overheads imposed by workflow management systems and execution environments and analyzed how existing optimization techniques improve runtime by reducing or overlapping overheads. The prevalent existence of system overheads is an important reason why task clustering provides significant performance improvement for workflow-based applications. In this work, we aim to further improve the performance of task clustering in a faulty environment.

Machine learning has been used to predict execution time [34, 18, 57, 27] and system overheads [20], and develop probability distributions for transient failure characteristics. Duan et.al. [34] 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. [27] used regression trees to dynamically estimate task behavior including process I/O, runtime, memory peak and disk usage. We reuse the knowledge gained from prior work on failure analysis, overhead analysis and task runtime analysis. We then use prior knowledge based Maximum Likelihood Estimation to integrate both the knowledge and runtime feedbacks and adjust the estimation accordingly.

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 bag of tasks. For instance, Muthuvelu et al. [69] proposed a clustering algorithm that groups bag of tasks based on their runtime—tasks are grouped up to the resource capacity. Later, they extended their work [68] to determine task granularity based on task file size, CPU time, and resource constraints. Recently, they proposed an online scheduling algorithm [67] that groups tasks based on resource network utilization, user's budget, and application deadline. Ng et al. [71] and Ang et al. [103] 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 [60]

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.

Task granularity control has also been addressed in scientific workflows. For instance, Singh et al. [91] proposed a level- and label-based clustering. In level-based clustering, tasks at the same level can be clustered together. The number of clusters or tasks per cluster are 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, which is prone to errors. Recently, Ferreira da Silva et al. [38] 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 fault tolerance.

5.3 Approach

In this chapter, the *goal* is to reduce the workflow makespan in a faulty environment by adjusting the clustering size (k). In task clustering, the clustering size (k) is an important parameter to influence the performance. We define it as the number of tasks in a clustered job. The reason why task clustering can help improve the performance is that it can reduce the scheduling cycles that workflow tasks go through since the number of jobs has decreased. The result is a reduction in the scheduling overhead and possibly other overheads as well [20]. Additionally, in the ideal case without any failures, the clustering size is usually equal to the number of all the parallel tasks divided by the number of available resources. Such a naive 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 failure rates especially the task failure rate. Intuitively speaking, if the task

failure rate is high, 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 $\boldsymbol{\gamma}$.

5.3.1 Task Failure Model

In our prior work [20], we have verified that system overheads s fits Gamma or Weibull distribution better than the other two distributions (Exponential and Normal). Schroeder et. al. [88] 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 [98, 46] Weibull, Gamma and Lognormal distribution are among the best fit for the workflow traces they used. Without loss of generality, we choose 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 $p(\theta|D, a, b)$ are in the same family as the prior distribution $p(\theta|a, b)$, the prior and the posterior are then called conjugate distributions, and the prior is called a conjugate prior for the likelihood function. For example, the Inverse-Gamma family 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 [88, 46, 98, 20] on the 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 [88] 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 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:

$$\theta_\gamma \sim \Gamma^{-1}(a_\gamma + n, b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}) \tag{5.1}$$

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

$$MLE(\theta_\gamma) = \frac{b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}}{a_\gamma + n + 1} \tag{5.2}$$

The understanding of the MLE has two folds: 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 [98, 46] 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 deepest depth from the entry task 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 the task failures are independent in each worker node (but with the same distribution) without

considering the failures that brings 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 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 shape parameter is still a Gamma distribution. We also assume the system overhead is independent of all the task runtimes and it has the same shape parameter ($\phi_{ts} = \phi_t = \phi_s$) with the task runtime. The job runtime irrespective of whether it succeeds of fails is:

$$\boldsymbol{d} \sim \Gamma(\phi_{ts}, k\theta_t + \theta_s) \tag{5.3}$$

$$MLE(\boldsymbol{d}) = (k\theta_t + \theta_s)(\phi_{ts} - 1) \tag{5.4}$$

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:

$$N_i = \frac{1}{1 - F(d_i)} = \frac{1}{e^{-(\frac{d_i}{\theta_\gamma})^{\phi_\gamma}}} = e^{(\frac{d_i}{\theta_\gamma})^{\phi_\gamma}} \tag{5.5}$$

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

$$M_i = d_i N_i = d_i e^{(\frac{d_i}{\theta_\gamma})^{\phi_\gamma}} \tag{5.6}$$

Equation 5.6 has involved two distributions $\boldsymbol{d}$ and θ_γ (ϕ_γ is known). From Equation 5.1, we have:

$$\frac{1}{\theta_\gamma} \sim \Gamma(a_\gamma + n, \frac{1}{b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}}) \tag{5.7}$$

$$MLE(\frac{1}{\theta_\gamma}) = \frac{a_\gamma + n - 1}{b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}} \tag{5.8}$$

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:

$$MLE(M_i) = MLE(d_i) e^{(MLE(d_i) MLE(\frac{1}{\theta_\gamma}))^{\phi_\gamma}} \tag{5.9}$$

Equation 5.9 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:

$$\begin{aligned} \boldsymbol{M} &= \frac{1}{r} \sum_{i=1}^{\frac{n}{k}} M_i \sim \Gamma \\ MLE(\boldsymbol{M}) &= \frac{n}{rk} MLE(M_i) \qquad (5.10) \\ &= \frac{n}{rk} MLE(d_i) e^{(MLE(d_i) MLE(\frac{1}{\theta_\gamma}))^{\phi_\gamma}} \qquad (5.11) \end{aligned}$$

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 5.10. $\arg min$ stands for the argument (k) of the minimum[1], that is to say, the k such that $MLE(\boldsymbol{M})$ attains its minimum value.

$$k^* = \arg min\{MLE(\boldsymbol{M})\} \tag{5.12}$$

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 minimal. Methods such as Newton's method can be used to find the minimal $MLE(\boldsymbol{M})$ and the corresponding k. Figure 5.1 shows an example of $MLE(\boldsymbol{M})$ using static estimation with a low task failure rate ($\theta_\gamma = 40$s), a medium task failure rate ($\theta_\gamma = 30$s) and a high task failure rate ($\theta_\gamma = 20$s). Other parameters are $n = 50$, $\theta_t = 5$ sec, and $\theta_s = 50$ sec and all the shape parameters are 2 for simplicity. These parameters are close to the level of mProjectPP in the Montage workflow that we simulate in Section 5.4. Figure 5.2 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 5.2) when $\theta_\gamma = 20, 30, 40$ is 4,5 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 many times.

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.

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

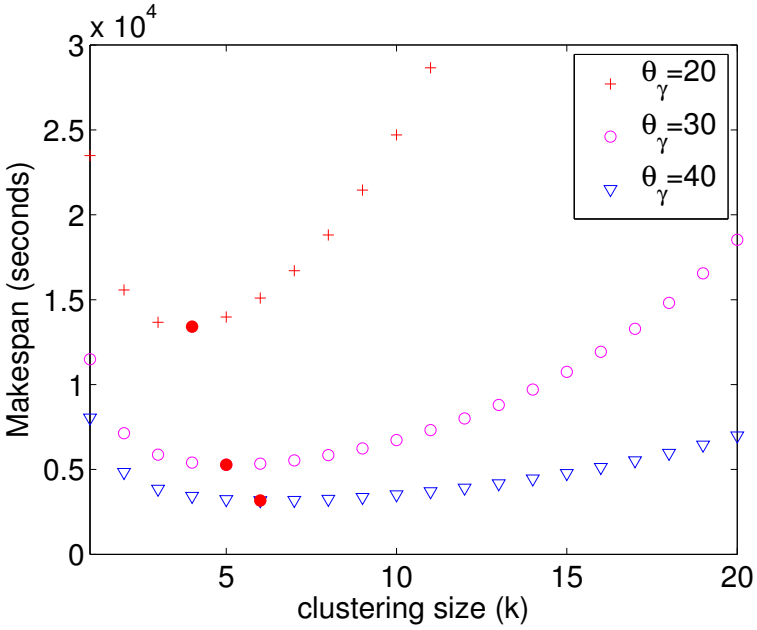

Figure 5.1: Makespan with different clustering size and θ_γ. ($n = 1000$, $r = 20$, $\theta_t = 5$ sec, $\theta_s = 50$ sec). The red dots are the minimums.

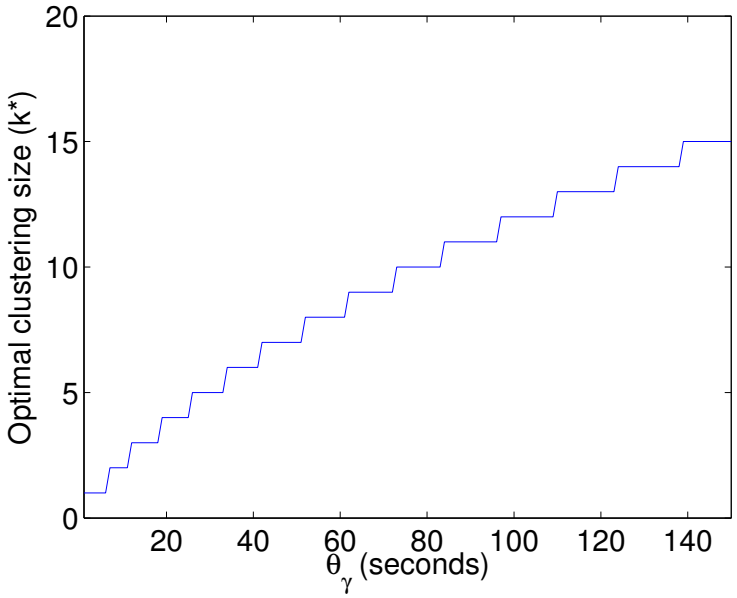

Figure 5.2: Optimal clustering size (k*) with different θ_γ ($n = 1000$, $r = 20$, $\theta_t = 5$ sec, $\theta_s = 50$ sec)

5.3.2 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) [91] 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 [91]. For simplicity, we set *clusters.num* to be the same as the number of available resources. In our prior work [24, 25], 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 Procedure are called in the initial task clustering process while the Reclustering Procedure is called when there is a failed job returned.

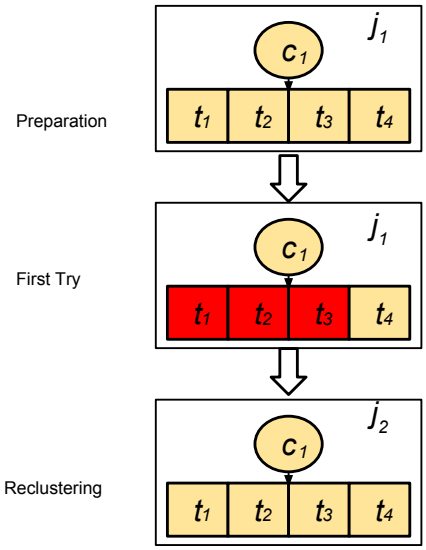

Figure 5.3: Horizontal Clustering (red boxes are failed tasks)

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 5.3 shows an example where the initial clustering size is 4 and thereby there are four tasks in a clustered job at the beginning. During execution, three out of these tasks (t_1, t_2, t_3) fail. HC will keep retrying all of the four tasks in next try until all of them succeed. Such a retry mechanism has been implemented and used in Pegasus [91].

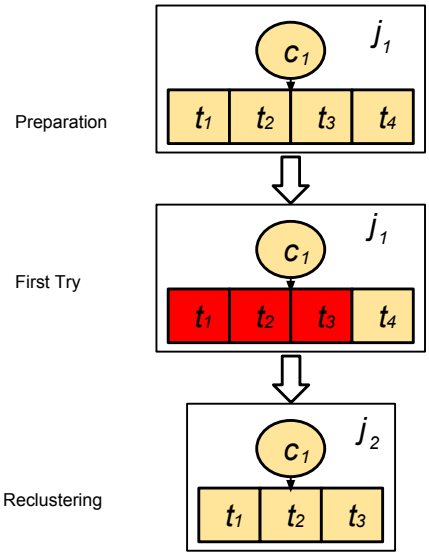

Figure 5.4: Selective Reclustering (red boxes are failed tasks)

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

Figure 5.4 shows an example of SR. At 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 be smaller and smaller spontaneously after each retry since there are less and less 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 its 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 be incorporated into existing workflow management systems without loss of much efficiency as shown in Section 5.4. 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 Reclustering. In DR, only failed tasks are merged into new clustered jobs and the clustering size is set to be k^* according to Equation 5.12.

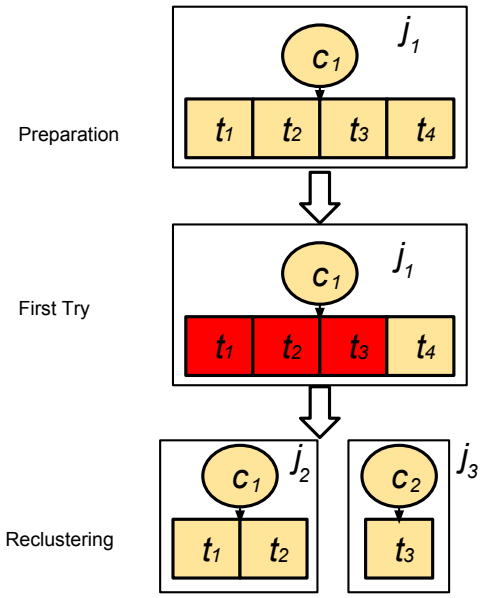

Figure 5.5: Dynamic Reclustering (red boxes are failed tasks)

Figure 5.5 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
```

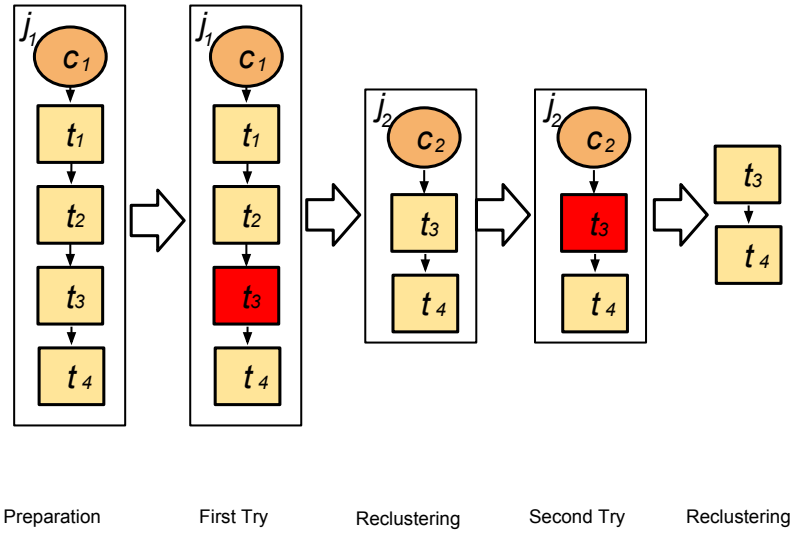

Figure 5.6: Vertical Reclustering (red boxes are failed tasks)

Vertical Reclustering (*VR*). VR is an extension of Vertical Clustering. Similar to Selective Reclustering, Vertical Reclustering only retries tasks that are failed or not completed. If there is a failure detected, we decrease k by half and recluster them. In Figure 5.6, 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 unfortunately 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.

5.4 Experiments 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

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 reliability of the algorithms. Our methods can also be applied to real workflow management systems as long as they support task-level failure monitoring. Five widely used scientific workflow applications are used in the experiments: LIGO Inspiral analysis [55], Montage [8], CyberShake [41], Epigenomics [108], and SIPHT [92]. Their main characteristics and structures are shown in section 4.4.1.

Table 5.1 shows the summary of the main **workflow characteristics**: number of tasks, average data size, and average task runtimes for the five workflows.

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 5.1: Summary of the scientific workflows characteristics.

5.4.1 Experiment conditions

We adopt a trace-based simulation approach, where we extended our WorkflowSim [23] simulator with the fault tolerant clustering methods to simulate a controlled distributed environment. WorkflowSim is an open source workflow simulator that extends CloudSim [14] by providing support for task clustering, task scheduling, and resource provisioning at the workflow level. It has been recently used in multiple workflow study areas [23, 21, 48] and its correctness has been verified in [23].

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 [4] and FutureGrid [40]. Amazon EC2 is a commercial, public cloud that has been widely used in distributed computing, in particular for scientific workflows [9]. FutureGrid is a distributed, high-performance testbed that provides scientists with a set of computing resources to develop parallel, grid, and cloud applications. 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 [25], which shows such selection is acceptable.

We collected workflow execution traces [49, 20] (including overhead and task runtime information) from real runs (executed on FutureGrid and Amazon EC2) of the scientific workflow

applications described in Section 4.4.1. The traces are used to feed the Workflow Generator toolkit [114] 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 under 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 has the maximum 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 5.1. 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.

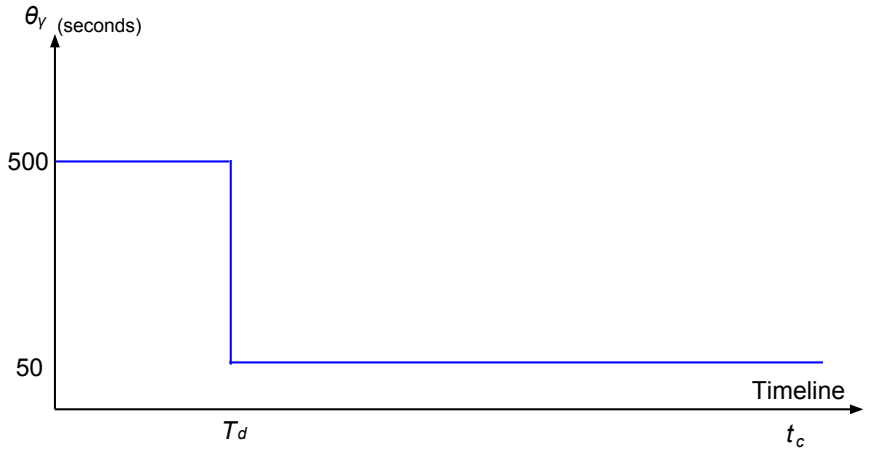

Figure 5.7: A Step Function of θ_γ. t_c is the current time and T_d is the moment θ_γ changes from 500 to 50 seconds

$$\theta_\gamma(t_c) = \begin{cases} 50 & \text{if } t_c \geq T_d \\ 500 & \text{if } 0 < t_c < T_d \end{cases} \tag{5.13}$$

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 5.7 and Equation 5.13, 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 less the makespan is. There is one special case when $T_d \to 0$, 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 5.8 and Equation 5.14. 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 [115] 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 $0.5T_c$.

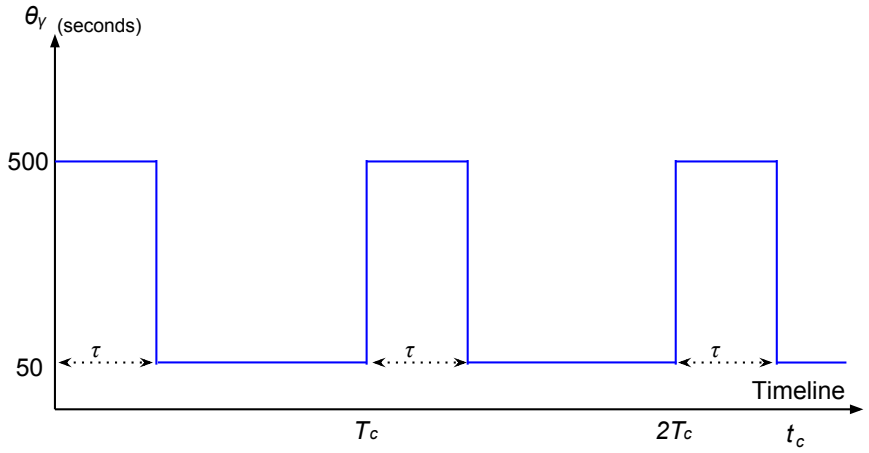

Figure 5.8: A Pulse Function of θ_γ. t_c is the current time and T_c is the period of the wave. τ is the width of the pulse.

$$\theta_\gamma(t_c) = \begin{cases} 500 & \text{if } 0 < t_c \leq \tau \\ 50 & \text{if } \tau < t_c < T_c \end{cases} \tag{5.14}$$

Table 5.2 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 5.2: Methods to Evaluate in Experiements

5.4.2 Results and discussion

Experiment 1 Figure 5.9, 5.10, 5.11, 5.12 and 5.13 show the performance of the four reclustering methods (DR, SR, VR and HC) with five workflows respectively. We draw conclusions:

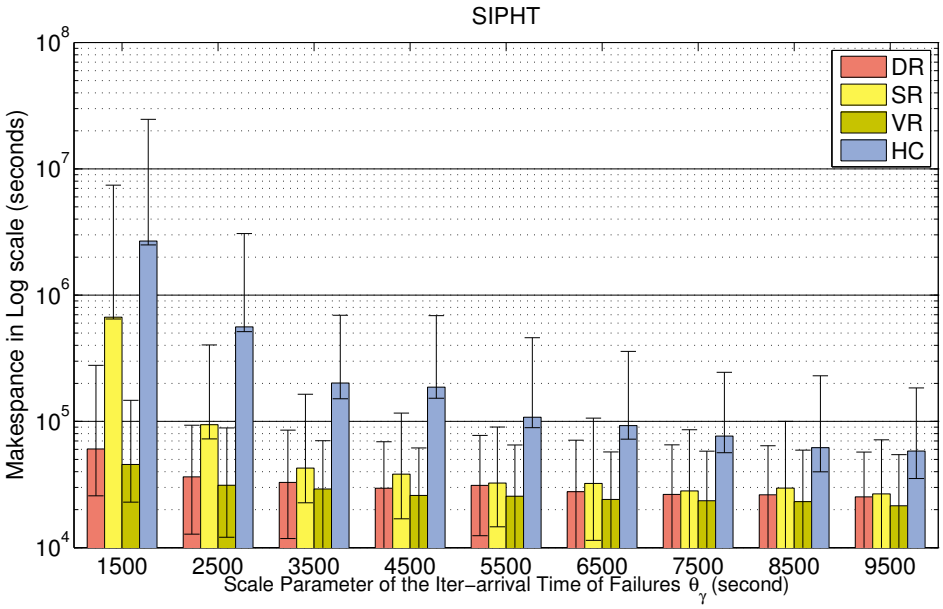

Figure 5.9: Experiment 1: SIPHT Workflow

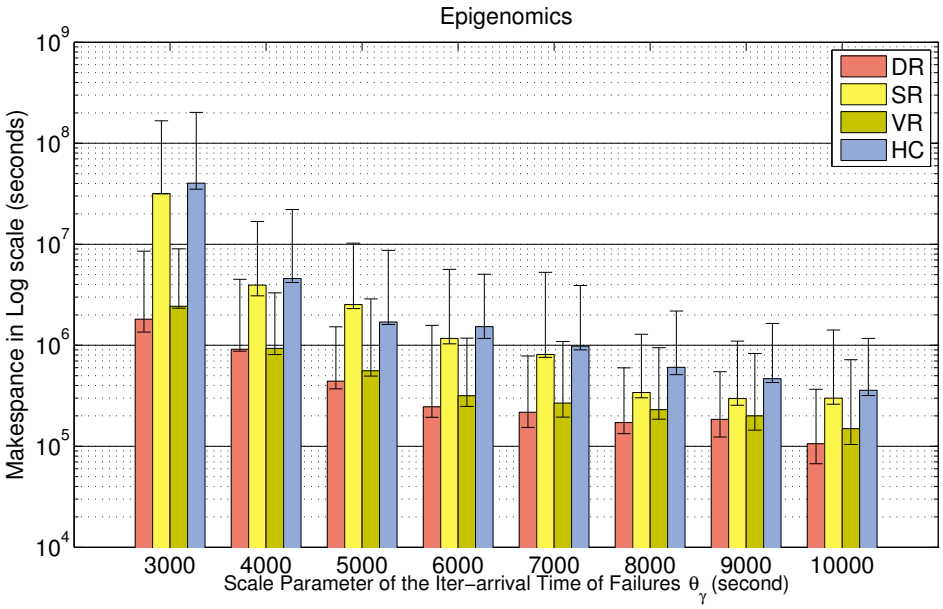

Figure 5.10: Experiment 1: Epigenomics Workflow

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 more failures are generated, the performance difference becomes more significant.

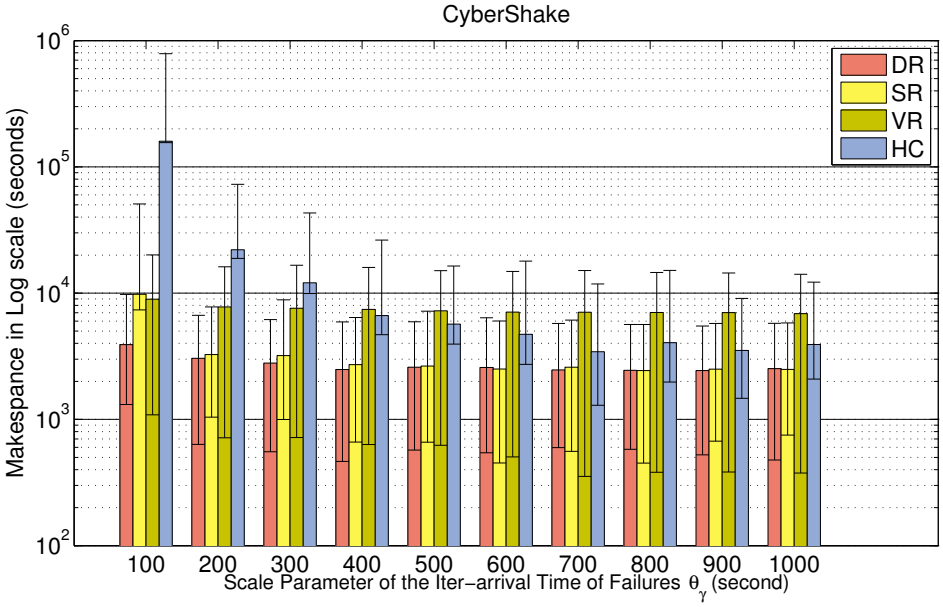

Figure 5.11: Experiment 1: CyberShake Workflow

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 the SR does not adjust k according to the occurrence of failures.

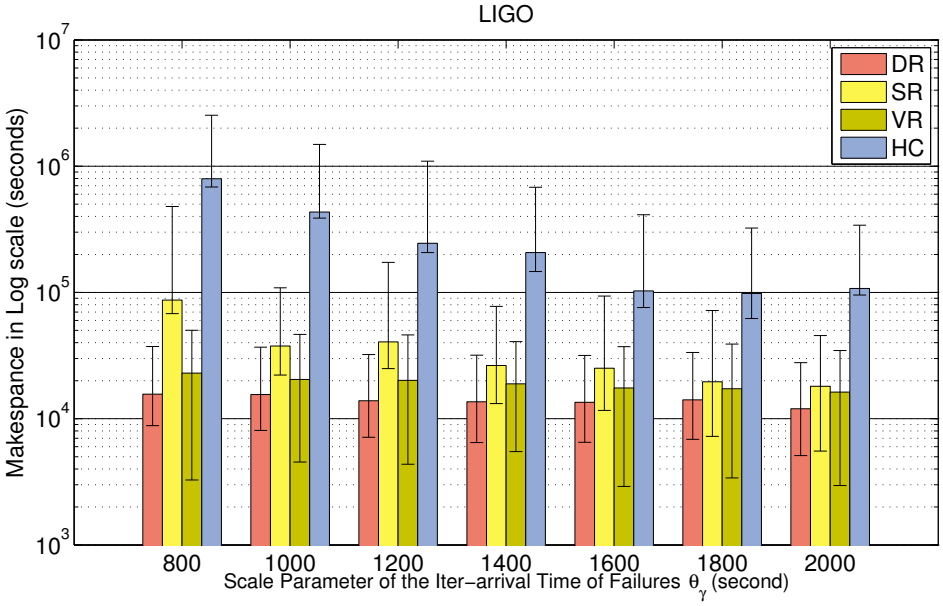

Figure 5.12: Experiment 1: LIGO Workflow

3). The performance of VR is highly related to the workflow structure and the average task runtime. For example, according to Figure 4.13 and Table 5.1, 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 5.10, the makespan increases more significantly with the decrease of θ_γ than other workflows. In comparison, the CyberShake workflow does not leave much space for vertical clustering methods to improve since it does not have many pipelines as shown in Figure 4.12. In addition, the average task runtime of the CyberShake workflow is relatively short (around 23 seconds). Compared to horizontal 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.

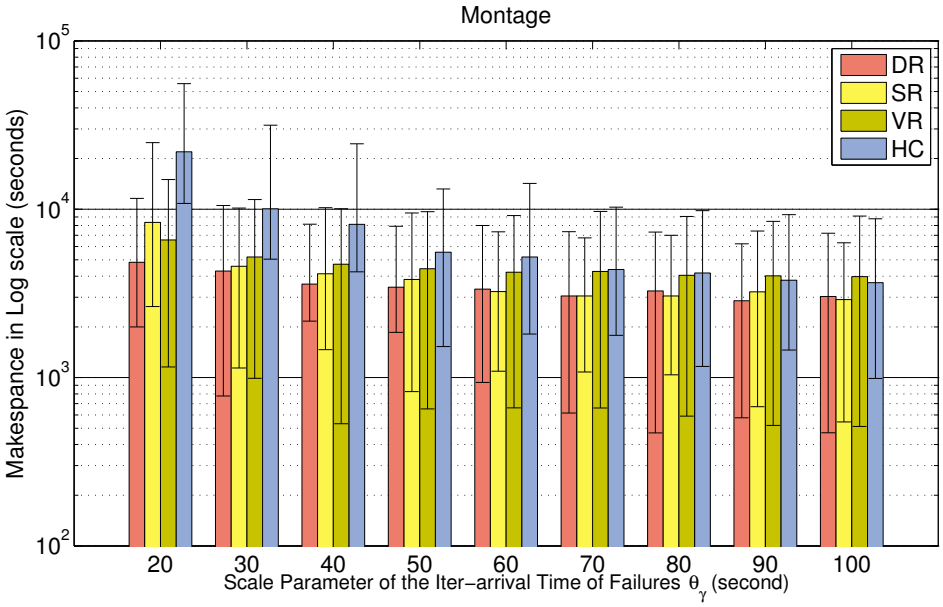

Figure 5.13: Experiment 1: Montage Workflow

Experiment 2 Figure 5.14 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 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.

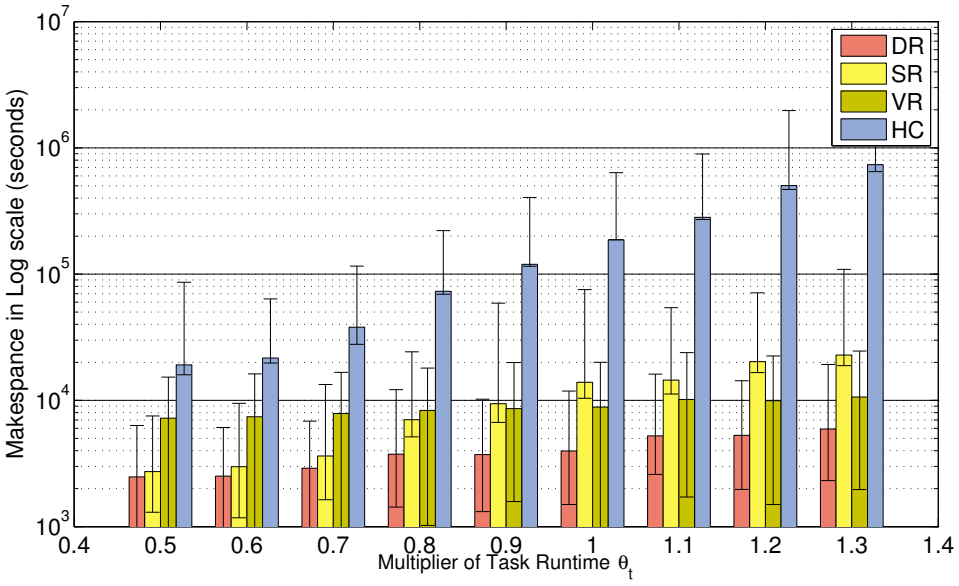

Figure 5.14: Experiment 2: Influence of Varying Task Runtime on Makespan (CyberShake)

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.

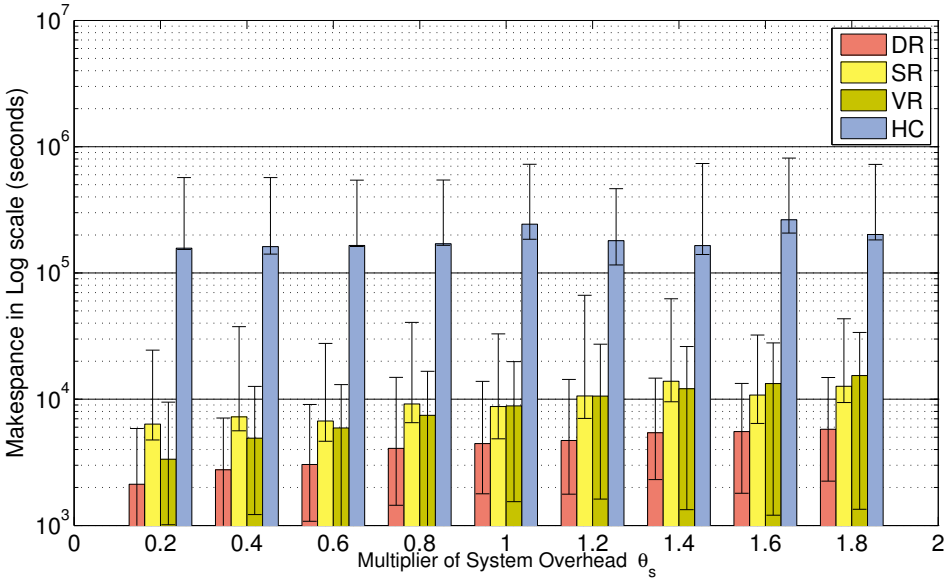

Figure 5.15: Experiment 2: Influence of Varying System Overhead on Makespan (CyberShake)

Figure 5.15 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 5.14. The reason is we may have multiple tasks in a clustered job but only one system overhead per job.

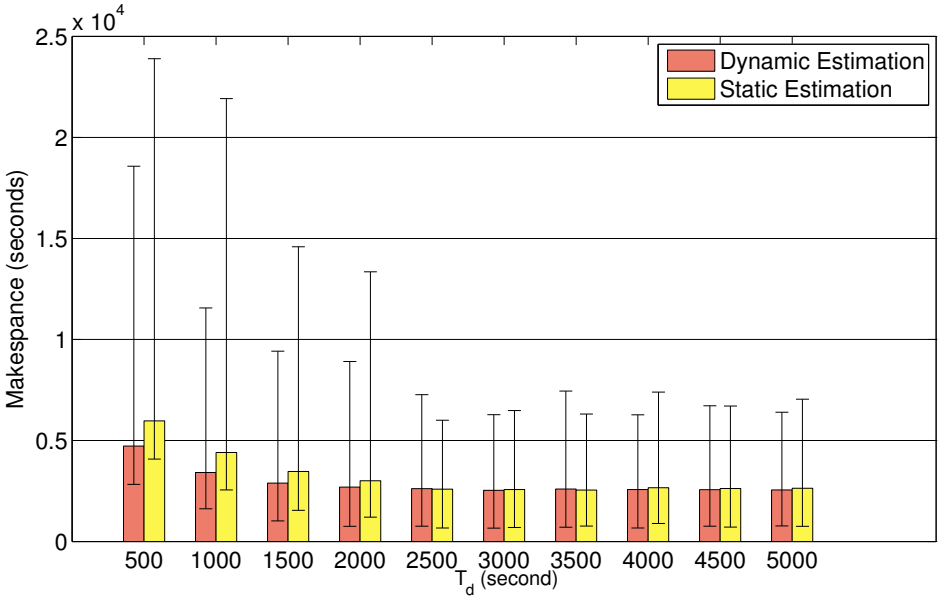

Figure 5.16: Experiment 3: Static Estimation vs. Dynamic Estimation (CyberShake, Step Function)

Experiment 3 Figure 5.16 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 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 5.17, 5.18 and 5.19 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.

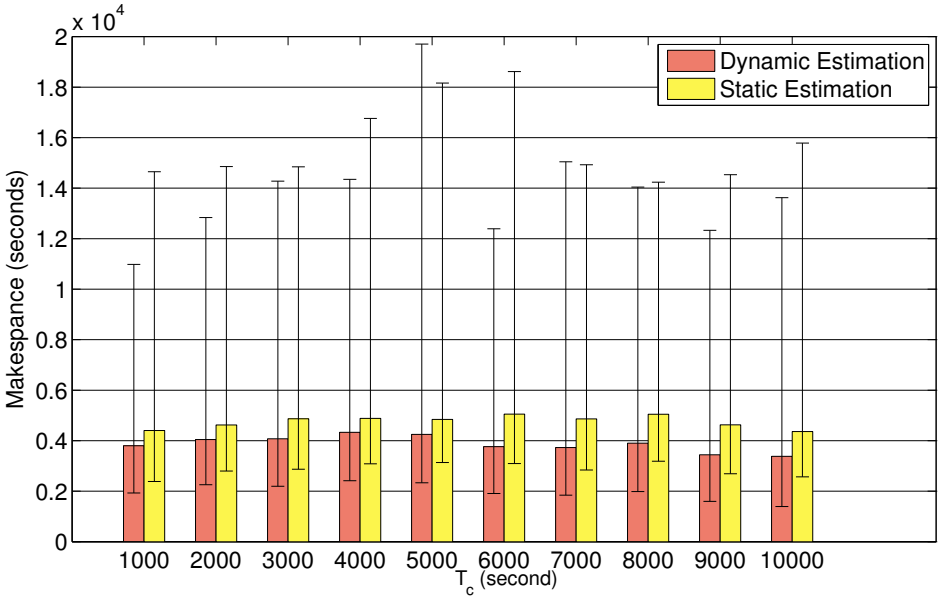

Figure 5.17: Experiment 3: Static Estimation vs. Dynamic Estimation (CyberShake, Pulse Function ($\tau = 0.1T_c$))

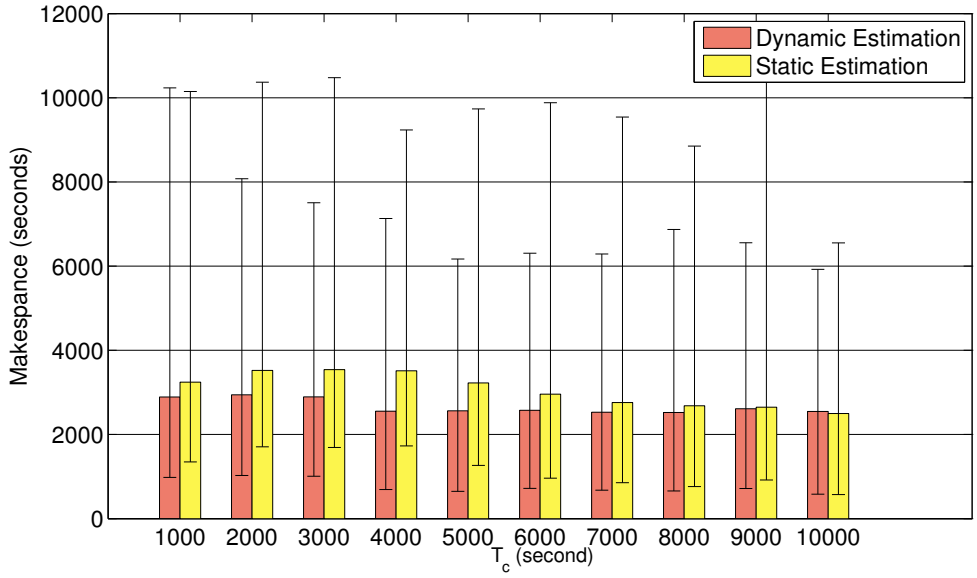

Figure 5.18: Experiment 3: Static Estimation vs. Dynamic Estimation (CyberShake, Pulse Function ($\tau = 0.3T_c$))

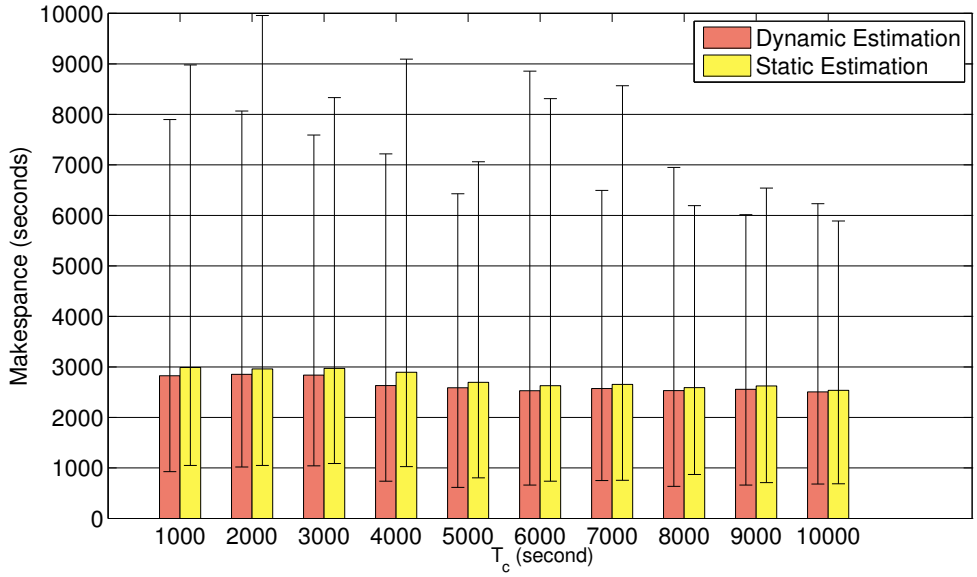

Figure 5.19: Experiment 3: Static Estimation vs. Dynamic Estimation (CyberShake, Pulse Function ($\tau = 0.5T_c$))

5.5 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 improves the performance significantly for workflows that have a short task runtime. Our dynamic estimation using 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.

In this chapter, we only discuss the fault tolerant clustering and apply it to a homogeneous environment. In the future, we aim to combine our work with fault tolerant scheduling in heterogeneous environments, i.e, a scheduling algorithm that avoids mapping clustered jobs to failure-prone nodes. We are also interested 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 works well under some constraints, which encourages us to improve its performance further. For example, we may weight data based on the time it was collected, the closer the more weight on it.

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, 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

Bibliography

[1] S. Abrishami, M. Naghibzadeh, and D. Epema. Deadline-constrained workflow scheduling algorithms for iaas clouds. *Future Generation Computer Systems*, 2012.

[2] A. C. Adamuthe and R. S. Bichkar. Minimizing job completion time in grid scheduling with resource and timing constraints using genetic algorithm. In *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*, ICWET '11, pages 338–343, New York, NY, USA, 2011. ACM.

[3] S. Ali, A. Maciejewski, H. Siegel, and J.-K. Kim. Measuring the robustness of a resource allocation. *Parallel and Distributed Systems, IEEE Transactions on*, 15(7):630–641, 2004.

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

[5] 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.

[6] L. Aversano, A. Cimitile, P. Gallucci, and M. Villani. Flowmanager: a workflow management system based on petri nets. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, pages 1054–1059, 2002.

[7] K. R. R. Babu, P. Mathiyalagan, and S. N. Sivanandam. Task scheduling using aco-bp neural network in computational grids. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, ICACCI '12, pages 428–432, New York, NY, USA, 2012. ACM.

[8] 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.

[9] G. B. Berriman, G. Juve, E. Deelman, M. Regelson, and P. Plavchan. The application of cloud computing to astronomy: A study of cost and performance. In *Workshop on e-Science challenges in Astronomy and Astrophysics*, 2010.

[10] 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.

[11] 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.

[12] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed dag engine for high performance computing. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1151–1158, 2011.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] H. Cao, H. Jin, X. Wu, S. Wu, and X. Shi. Dagmap: Efficient scheduling for dag grid workflow job. In *9th IEEE/ACM International Conference on Grid Computing*, pages 17–24, 2008.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

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

[29] E. Deelman. Grids and clouds: Making workflow applications work in heterogeneous distributed environments. *International Journal of High Performance Computing Applications*, 24(3):284–298, Aug. 2010.

[30] 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.

[31] 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.

[32] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[33] 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.

[34] 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.

[35] 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.

[36] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. Seragiotto, and H. Truong. ASKALON: a tool set for cluster and grid computing. *Concurrency and Computation: Practice & Experience*, 17(2-4):143–169, 2005.

[37] 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.

[38] 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.

[39] 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.

[40] FutureGrid. `http://futuregrid.org/`.

[41] 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.

[42] Z. Guan, F. Hern, P. Bangalore, J. Gray, A. Skjellum, V. Velusamy, and Y. Liu. Grid-flow: A grid-enabled scientific workflow system with a petri net-based interface. 2004.

[43] 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.

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

[45] 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.

[46] 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.

[47] 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.

[48] F. Jrad, J. Tao, and A. Streit. A broker-based framework for multi-cloud workflows. In *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, pages 61–68. ACM, 2013.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

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

[56] 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.

[57] 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.

[58] Y. Li, A. YarKhan, J. Dongarra, K. Seymour, and A. Hurault. Enabling workflows in gridsolve: request sequencing and service trading. *The Journal of Supercomputing*, pages 1–20, 2011.

[59] 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.

[60] 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.

[61] X. Liu, J. Chen, K. Liu, and Y. Yang. Forecasting duration intervals of scientific workflow activities based on time-series patterns. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 23–30, 2008.

[62] Y. Ma, X. Zhang, and K. Lu. A graph distance based metric for data oriented workflow retrieval with variable time constraints. *Expert Syst. Appl.*, 41(4):1377–1388, Mar. 2014.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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¿.

[71] 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.

[72] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. Eucalyptus: A technical report on an elastic utility computing architecture linking your programs to useful systems. UCSB Computer Science Technical Report 2008-10, 2008.

[73] 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.

[74] 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.

[75] 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.

[76] P.-O. Ostberg and E. Elmroth. Mediation of service overhead in service-oriented grid architectures. In *12th IEEE/ACM International Conference on Grid Computing (GRID)*, pages 9–18, 2011.

[77] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. 1999.

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

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] K. Ranganathan and I. Foster. Identifying dynamic replication strategies for a high-performance data grid. In *In Proc. of the International Grid Computing Workshop*, pages 75–86, 2001.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

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

[93] V. K. Soni, R. Sharma, and M. K. Mishra. Grouping-based job scheduling model in grid computing. *WorldAcademy of Science, Engineering and Technology*, 65:781, 2005.

[94] 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.

[95] 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.

[96] Southern California Earthquake Center (SCEC). `http://www.scec.org`.

[97] C. Stratan, A. Iosup, and D. H. Epema. A performance study of grid workflow engines. In *9th IEEE/ACM International Conference on Grid Computing*, pages 25–32. IEEE, 2008.

[98] 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.

[99] 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.

[100] I. Taylor, M. Shields, I. Wang, and A. Harrison. Visual grid workflow in triana. *Journal of Grid Computing*, 3(3-4):153–169, Jan. 2006.

[101] 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.

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

[103] 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.

[104] R. Tolosana-Calasanz, M. Lackovic, O. F Rana, J. Á. Bañares, and D. Talia. Characterizing quality of resilience in scientific workflows. In *Proceedings of the 6th workshop on Workflows in support of large-scale science*, pages 117–126. ACM, 2011.

[105] 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.

[106] 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.

[107] H. Truong and T. Fahringer. SCALEA-G: a unified monitoring and performance analysis system for the grid. *Scientific Programming*, 12(4):225–237, 2004.

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

[109] M. Uysal, T. M. Kurc, A. Sussman, and J. H. Saltz. A performance prediction framework for data intensive applications on large scale parallel machines. In *4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, 1998.

[110] W. M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[111] J. Vöckler, G. Juve, E. Deelman, M. Rynge, and G. B. Berriman. Experiences using cloud computing for a scientific workflow application. In *2nd Workshop on Scientific Cloud Computing (ScienceCloud '11)*, 2011.

[112] M. Vossberg, A. Hoheisel, T. Tolxdorff, and D. Krefting. A reliable dicom transfer grid service based on petri net workflows. In *8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID '08)*, pages 441–448, 2008.

[113] 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.

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

[115] 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.

[116] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3–4), 2005.

[117] 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.

[118] 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.

[119] 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.

[120] 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.

[121] 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.

[122] A. Zomaya and G. Chan. Efficient clustering for parallel tasks execution in distributed systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 167–, 2004.