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)

May 2014

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 [48], scientific workflows continue to gain their popularity among many science disciplines, including physics [32], astronomy [100], biology [64, 85], chemistry [129], earthquake science [71] 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 [32]. For example, the CyberShake workflow [74] is used by the Southern California Earthquake Center (SCEC) [110] 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 [115]. 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 [105, 50, 139] 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

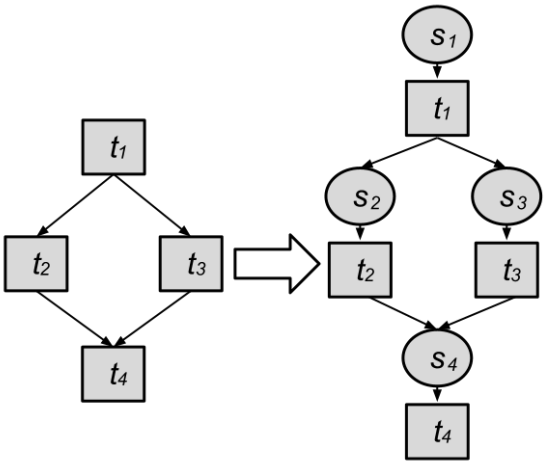

Figure 1.1: Extending DAG to o-DAG

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:

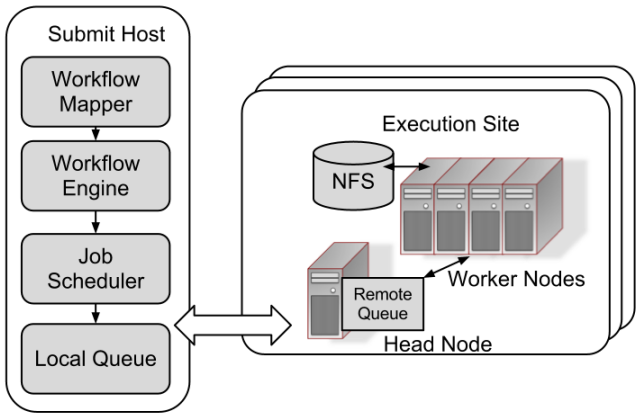

Figure 1.2: System Model

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.

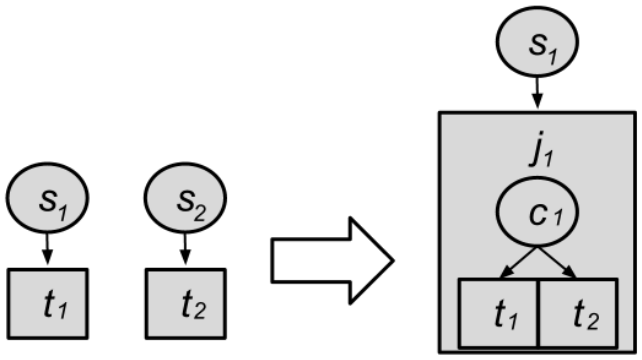

Figure 1.3: Task Clustering

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 [50, 20, 87] dynamically groups tasks together based on the task characteristics but it assumes tasks are independent, which limits its usage in scientific workflows. Singh [105] and Rynge [74] 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 [67]. 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 [108, 52] has taken the communication cost into the scheduling/clustering model and have achieved some significant improvement. The workflow partitioning approach [47, 134, 129, 36] 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 [72, 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 [30, 126]. 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 [138, 113, 101, 98] 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 [138]. 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 [138]. 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 [31]. 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 [138]. Third, tasks can be replicated to different nodes to avoid location-specific failures [137]. 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 [93, 94]. 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 [74, 105, 66, 18] 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 [29], Pegasus [31], Triana [114], DAGuE [12] and GrADS [27] . 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) [29] is a service provided by Condor [43] 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 [31], which stands for Planning for Execution in Grids, was developed at the USC Information Sciences Institute as part of the GriPhyN [32] and SCEC/IT [71] 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 [124]. 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 [127], Grid-Flow [46] and FlowManager [6] are representative examples of this.

ASKALON [39] 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 [94, 93] 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. [111] 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. [94] 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 [35, 118, 121, 123]. The performance method proposed by Duan et al. [35] 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 [81, 82, 83], which reduces the cost of data transfers when grouped tasks share input data [81] 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. [82] 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. [83] and Ang et al. [116] 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. [107] 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 [141] 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 [105] 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 [74, 72, 50, 68]. Li [66] 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. [18] 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 [120], 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

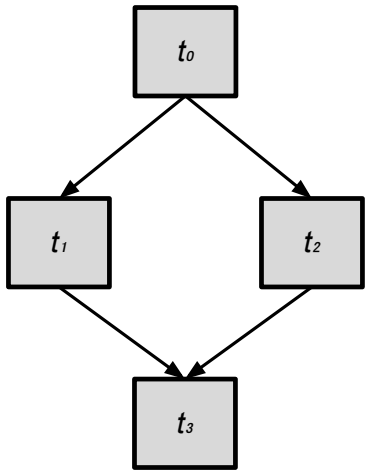

Figure 2.1: A simple DAG with four tasks (t_0, t_1, t_2, t_3). The edges represent the data dependencies between tasks.

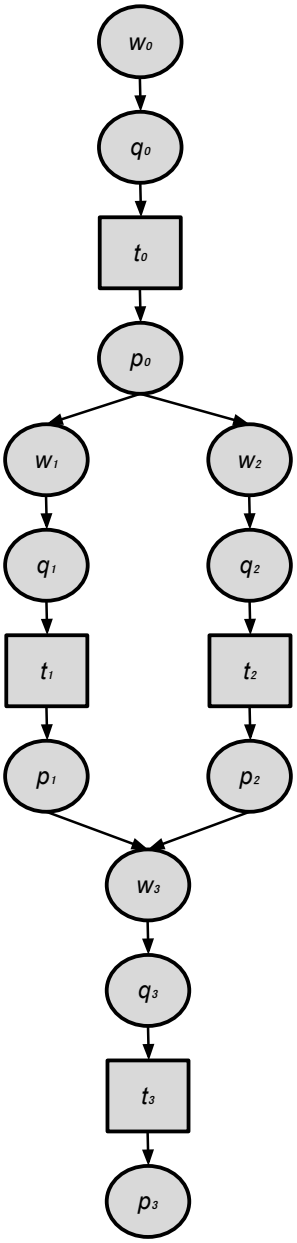

Figure 2.2: An o-DAG with overheads ($w_0 \sim w_3$, $q_0 \sim q_3$, $p_0 \sim p_3$). The edges represent control dependencies or data dependencies

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:

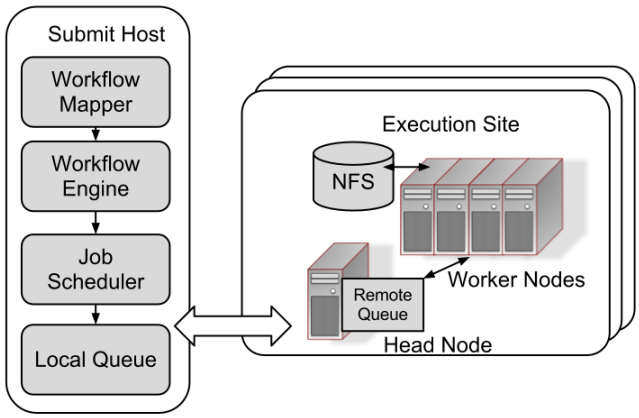

Figure 2.3: System Model

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

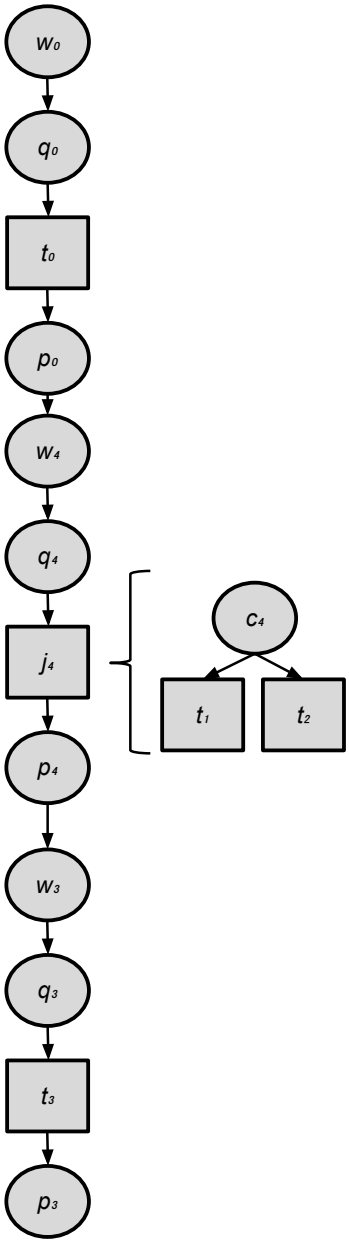

Figure 2.4: A Diamond Workflow after Task Clustering

for resources to be assigned to it. The workflow engine delay reflects the efficiency of a workflow engine (i.e., DAGMan [29]).

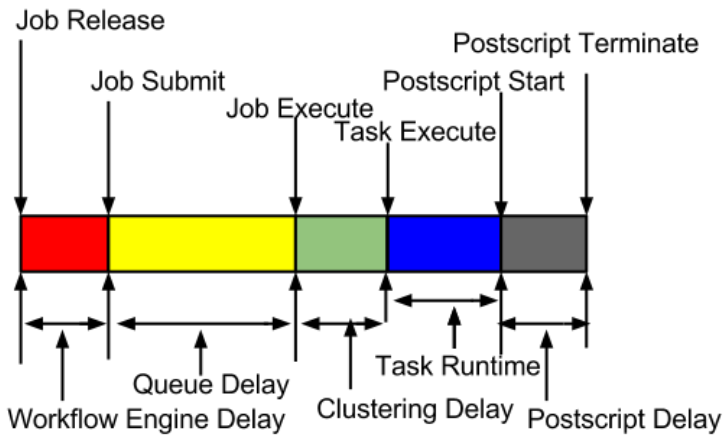

Figure 2.5: Workflow Events

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 [43]) 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 [9] that is used to construct large image mosaics of the sky. Montage was run on FutureGrid [44]. 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.

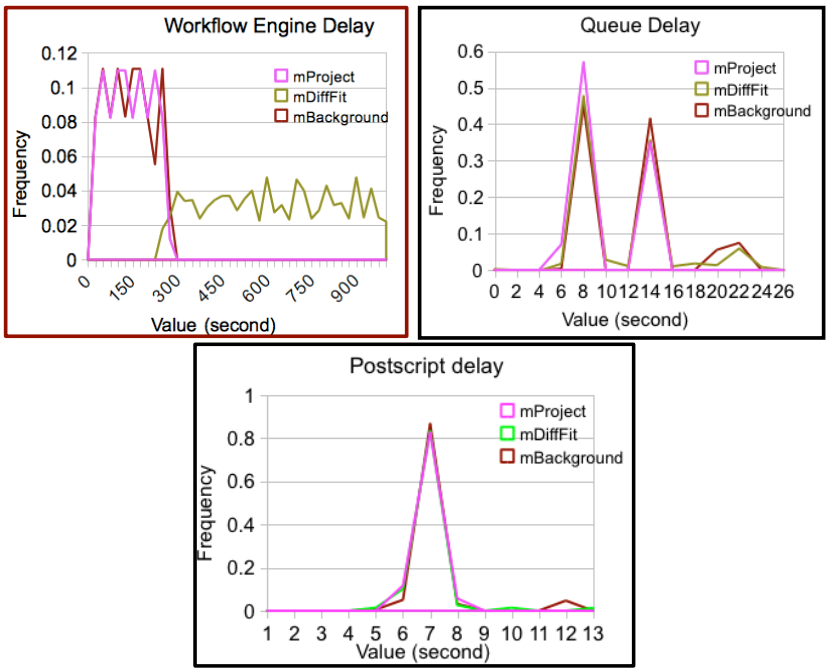

Figure 2.6: Distribution of overheads in the Montage workflow

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.

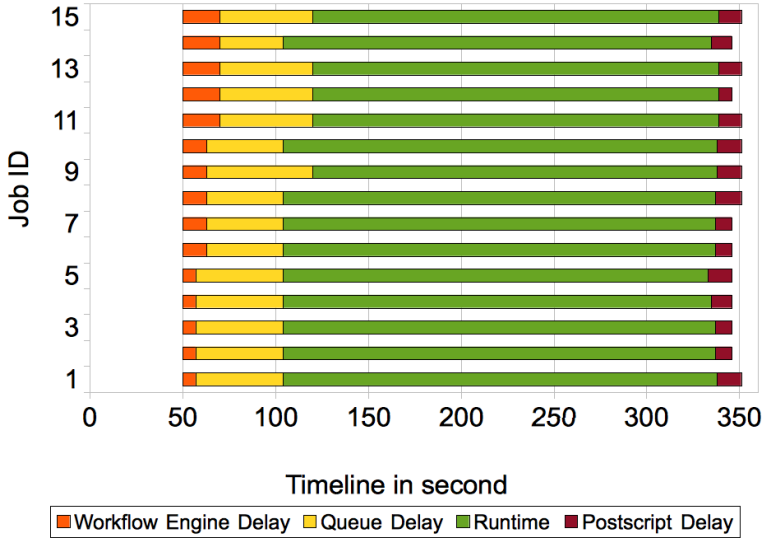

Figure 2.7: Workflow Overhead and Runtime. Clustering delay and data transfer delay are not shown

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.

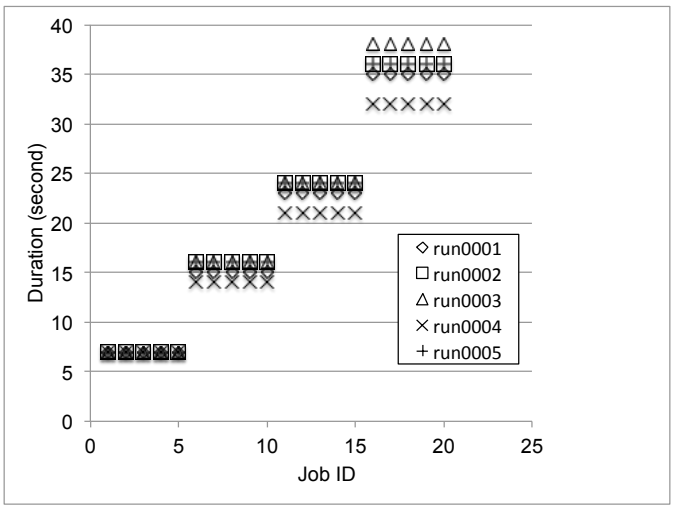

Figure 2.8: Workflow Engine Delay of mProjectPP

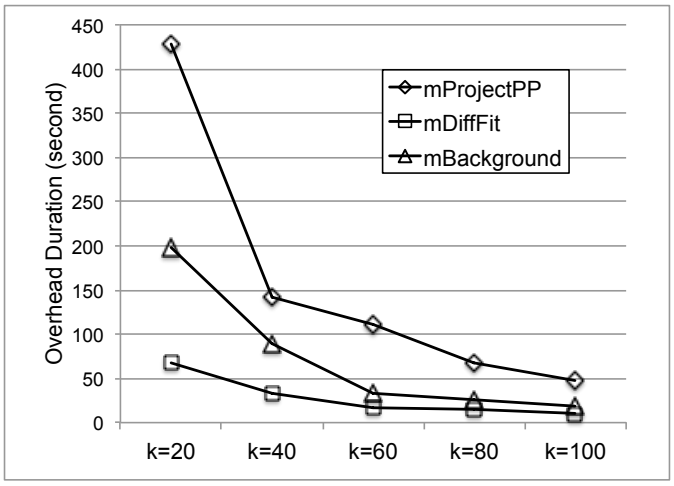

Figure 2.9: Clustering Delay of mProjectPP, mDiffFit, and mBackground

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

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 [89]. 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.

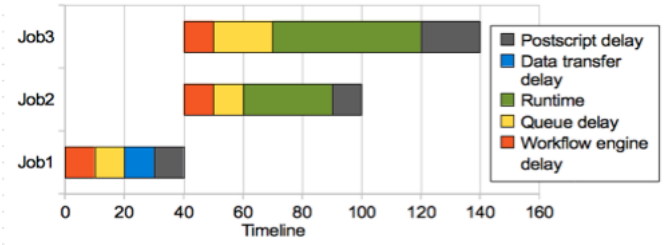

Figure 2.10: The Timeline of an Example Workflow

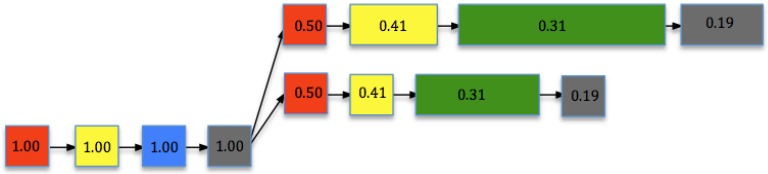

Figure 2.11: Reverse Ranking

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 [120], 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.

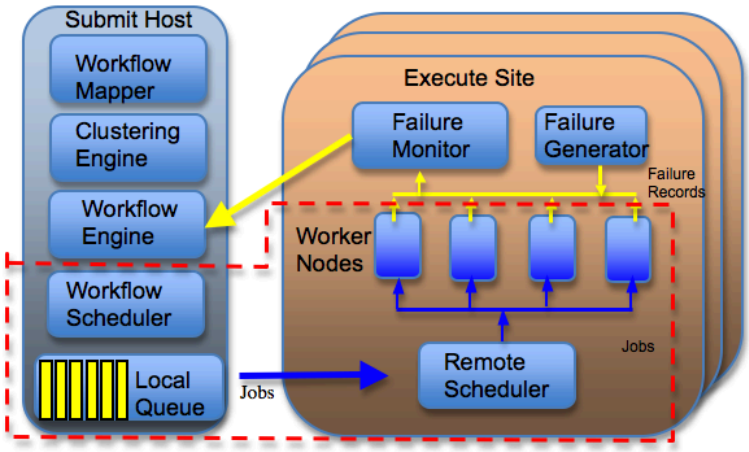

Figure 2.12: WorkflowSim Overview. The area surrounded by red lines is supported by CloudSim

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 [138, 113, 101, 98, 86, 75] 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 [22] 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 [44]. 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

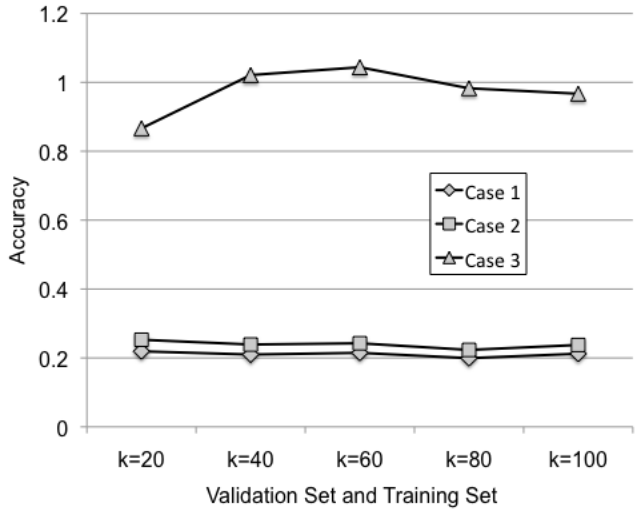

Figure 2.13: Performance of WorkflowSim with different support levels

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

Balanced Clustering

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.

3.1 Motivation

Existing task clustering strategies have demonstrated their effect in some scientific workflows such as CyberShake [74] and LIGO [32]. 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 [25]. A common technique to handle load imbalance is overdecomposition [67]. 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 imbalance problem. It means that the execution of workflows suffers from significant overheads (unavailable data, overloaded resources, or system constraints) due to inappropriate task clustering and job execution. To solve the imbalance problem, we introduce a series of balancing methods to address these two challenges respectively.

However, what makes this problem challenging is that the solutions are usually conflicting. For example, balancing runtime may worsen 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.

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 [23] 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 [81][68] 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.

3.2 Related Work

Overhead analysis [88, 94] is a topic of great interest in the Grid community. Stratan et al. [111] 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. [94] offered a complete Grid workflow overhead classification and a systematic measurement of overheads. In Chen et al. [21], we extended [94] 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. [81] 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 [80] to determine task granularity based on task file size, CPU time, and resource constraints. Recently, they proposed an online scheduling algorithm [79, 82] that groups tasks based on resource network utilization, user's budget, and application deadline. Ng et al. [83] and Ang et al. [116] 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 [68]

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. [105] 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. [41] 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. [67] proposed to use work stealing and a hierarchical persistence-based rebalancing algorithm to address the imbalance problem in scheduling. Zheng et al. [140] 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 [131, 54, 69] 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 [131] 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. [54] 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 [33]. Liu et al. [69] 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. [117] 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. [70] 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.

3.3 Approach

With an o-DAG model, we can explicitly express the process of task clustering. In this paper, we address task clustering horizontally and vertically. **Horizontal Clustering** (HC) merges multiple tasks that are at the same horizontal level of the workflow, in which the horizontal level of a task is defined as the longest distance from the entry task of the DAG to this task. **Vertical Clustering**

(VC) merges tasks within a pipeline of the workflow. Tasks at the same pipeline share a single-parent-single-child relationship, which means a task t_a is the unique parent of a task t_b, which is the unique child of t_a.

Figure 3.1 shows a simple example of how to perform HC, in which two tasks t_2 and t_3, without a data dependency between them, are merged into a clustered job j_1. A job j is a single execution unit composed by one or multiple task(s). Job wrappers are commonly used to execute clustered jobs, but they add an overhead denoted by the clustering delay c. The clustering delay measures the difference between the sum of the actual task runtimes and the job runtime seen by the job scheduler. After horizontal clustering, t_2 and t_3 in j_1 can be executed in sequence or in parallel, if parallelism in one compute node is supported. In this paper, we consider sequential executions only. Given a single resource, the overall runtime for the workflow in Figure 3.1 (left) is $runtime_l = \sum_{i=1}^{4}(s_i + t_i)$, and the overall runtime for the clustered workflow in Figure 3.1 (right) is $runtime_r = s_1 + t_1 + s_2 + c_1 + t_2 + t_3 + s_4 + t_4$. $runtime_l > runtime_r$ as long as $c_1 < s_3$, which is the case in many distributed systems since the clustering delay within a single execution node is usually shorter than the scheduling overhead across different execution nodes.

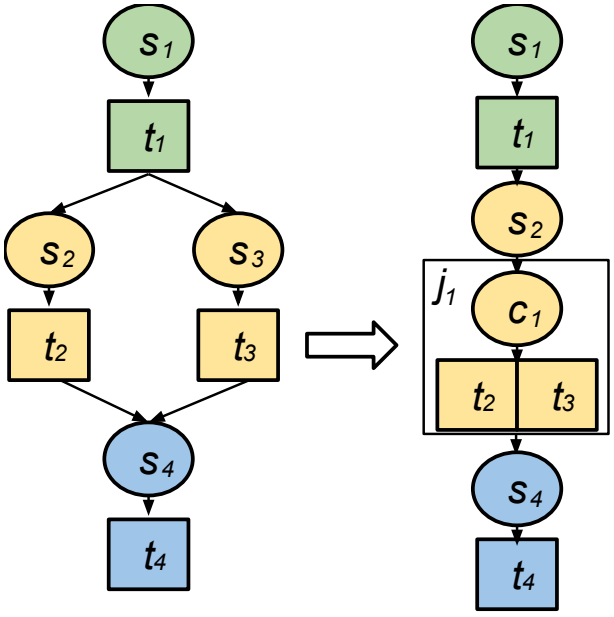

Figure 3.1: An example of horizontal clustering (color indicates the horizontal level of a task).

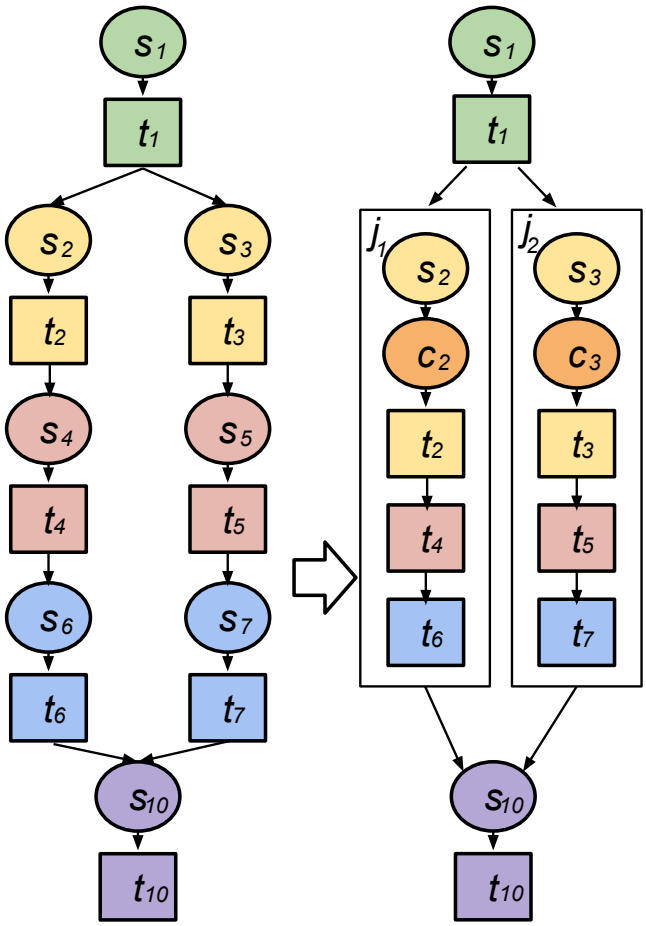

Figure 3.2: An example of vertical clustering.

Figure 3.2 illustrates an example of vertical clustering, in which tasks t_2, t_4, and t_6 are merged into j_1, while tasks t_3, t_5, and t_7 are merged into j_2. Similarly, clustering delays c_2 and c_3 are added to j_1 and j_2 respectively, but system overheads s_4, s_5, s_6, and s_7 are removed.

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.

3.3.1 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 3.3 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 3.3-top). In contrast, a balanced clustering strategy should try its best to evenly distribute task runtime among jobs as shown in Figure 3.3 (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 3.4, 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 3.4) forces t_5 and t_6 to transfer files from two locations and wait for the completion of t_1, t_2, t_3, and t_4. A balanced clustering strategy groups tasks that have the maximum number of child tasks in common. Thus, t_5 can start to execute as soon as t_1 and t_2 are completed, and so can t_6. To measure and quantitatively demonstrate the Dependency

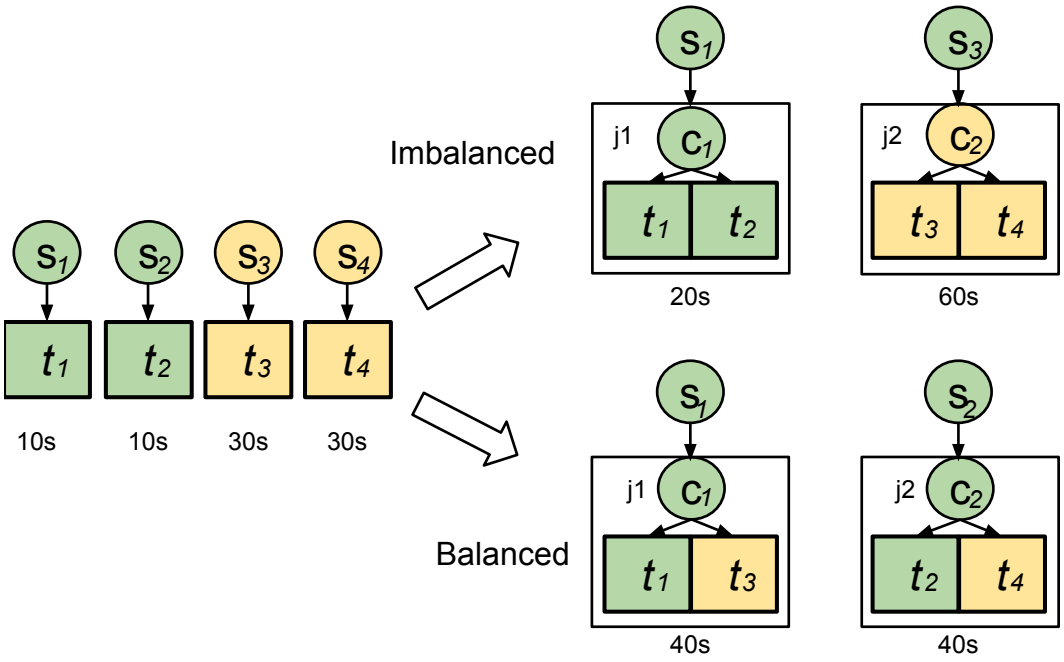

Figure 3.3: An example of Horizontal Runtime Variance.

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:

$$IF(t_u) = \sum_{t_v \in Child(t_u)} \frac{IF(t_v)}{||Parent(t_v)||} \tag{3.1}$$

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 3.4) is 1.0. For

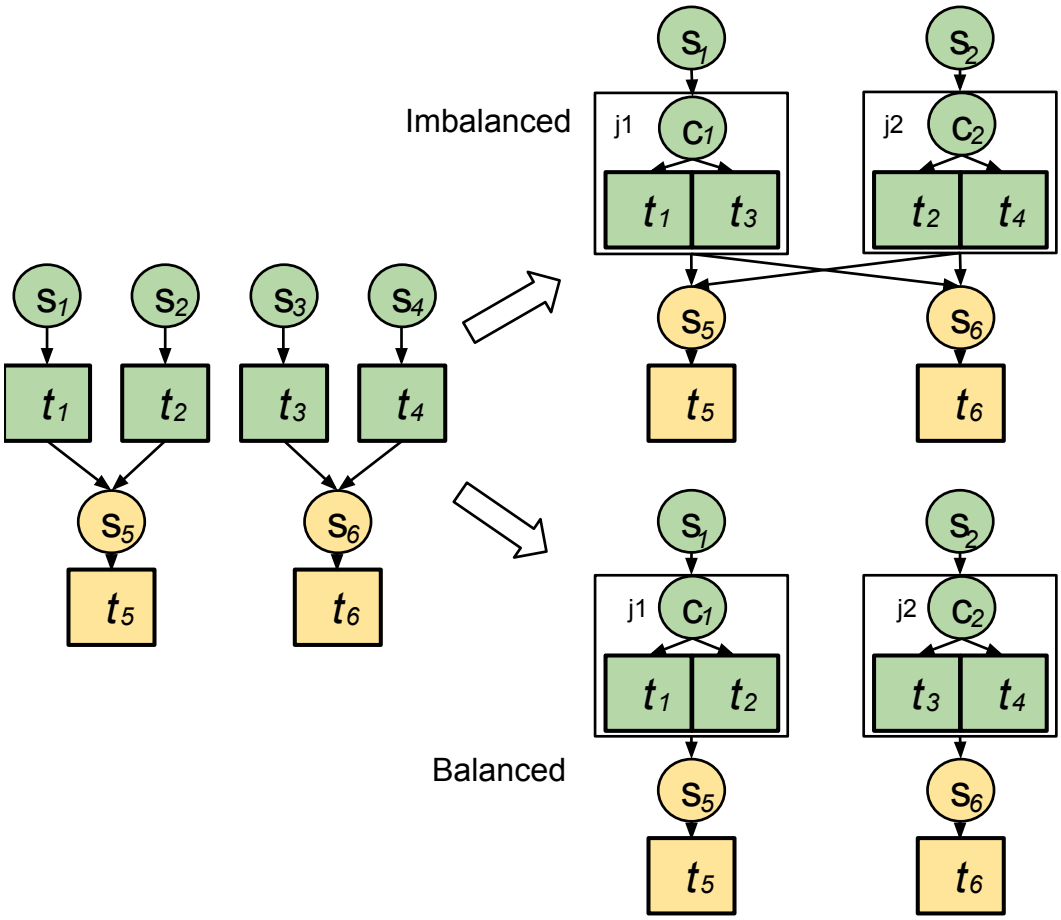

Figure 3.4: An example of Dependency Imbalance.

instance, consider the two workflows presented in Figure 3.5. 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 3.5 (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 $O(n)$.

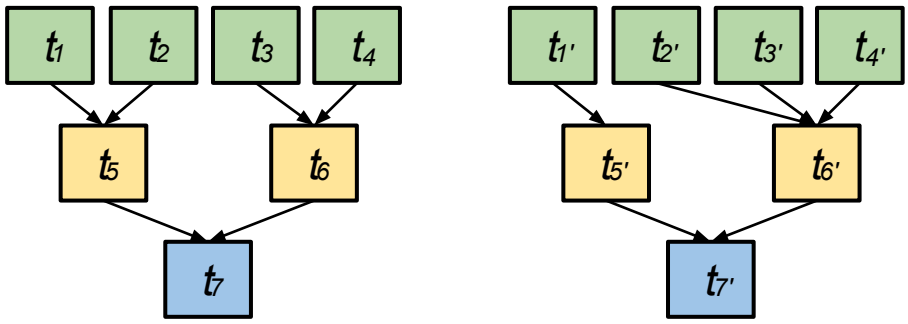

Figure 3.5: Example of workflows with different data dependencies (For better visualization, we do not show system overheads in the rest of the paper).

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

of Figure 3.5 (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 3.1: Distance matrices of tasks from the first level of workflows in Figure 3.5.

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

3.3.2 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 1 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 3.6 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 1 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 2 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

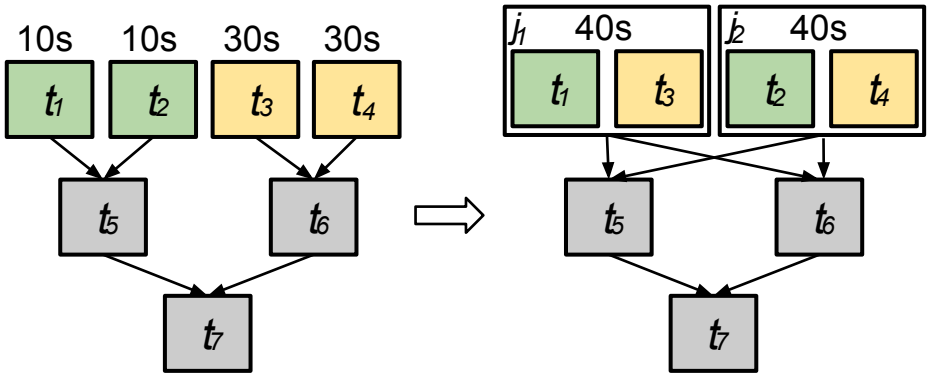

Figure 3.6: An example of the HRB (Horizontal Runtime Balancing) method. By solely addressing runtime variance, data locality problems may arise.

Algorithm 3 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 2 shows the pseudocode of HIFB. For example, in Figure 3.7, 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.

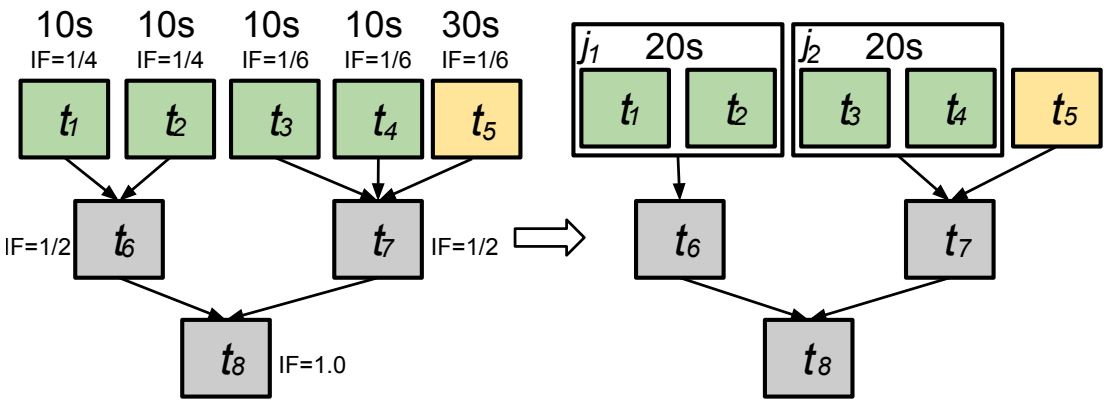

Figure 3.7: An example of the HIFB (Horizontal Impact Factor Balancing) method. Impact factors allow the detection of similarities between tasks.

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 3.6, 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 3 shows the pseudocode of HDB. For instance, in Figure 3.8, 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 3.7), 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 3.9. 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 [63], which is used in the evaluation of this paper (Section 3.4.4).

Table 3.2 summarizes the imbalance metrics and balancing methods presented in this paper.

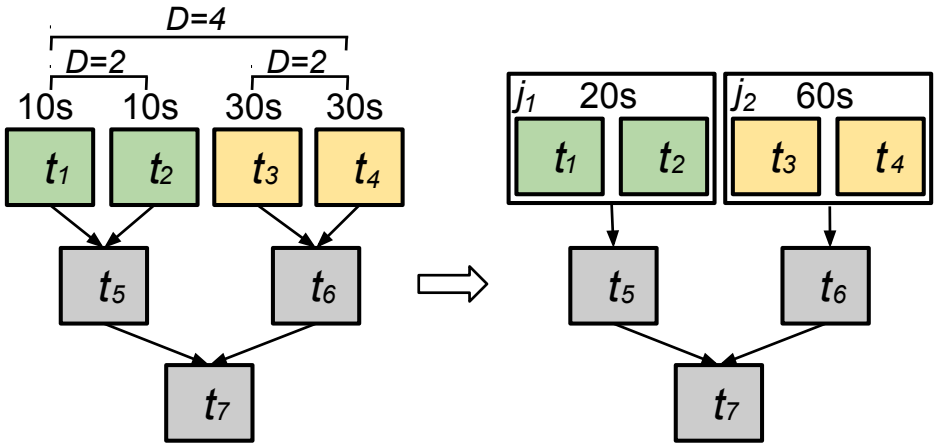

Figure 3.8: An example of the HDB (Horizontal Distance Balancing) method. Measuring the distances between tasks avoids data locality problems.

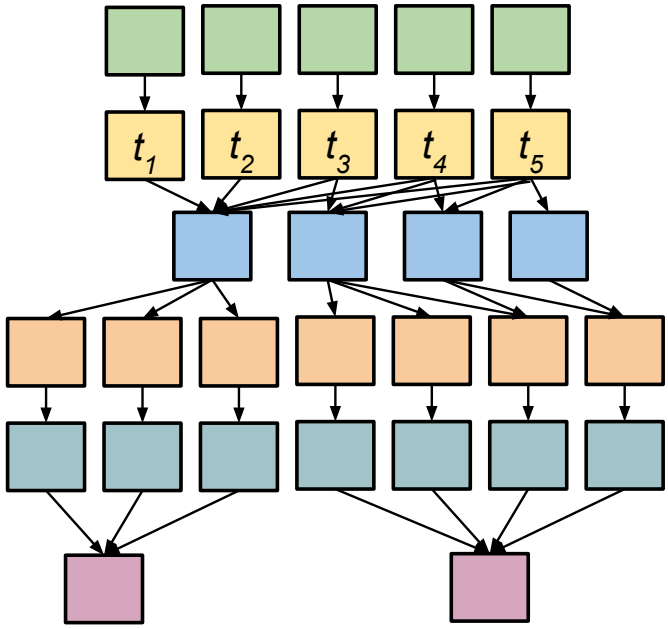

Figure 3.9: A workflow example where HDB yields lower performance than HIFB. HDB does not capture the difference in the number of parents/child tasks, since the distances between tasks (t_1, t_2, t_3, t_4, and t_5) are the same.

3.3.3 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 3.2: 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 3.10 shows an example where pipelined-tasks are merged first, and then the merged pipelines are horizontally clustered based on the runtime variance.

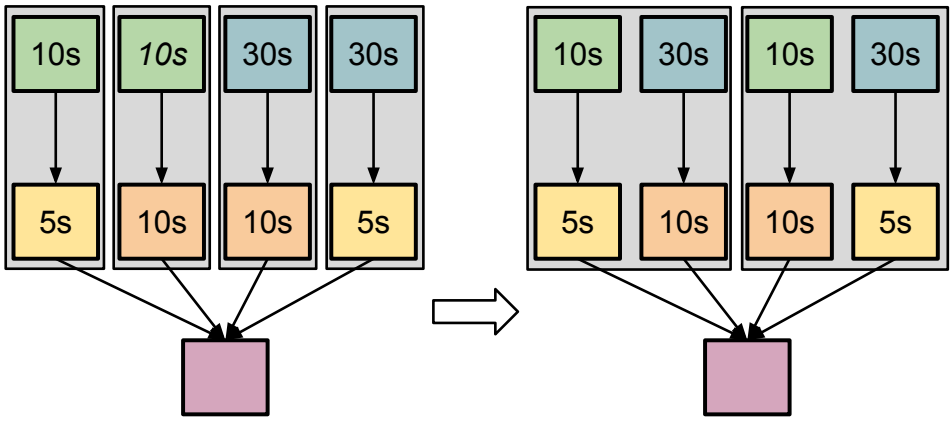

Figure 3.10: *VC-prior*: vertical clustering is performed first, and then the balancing methods.

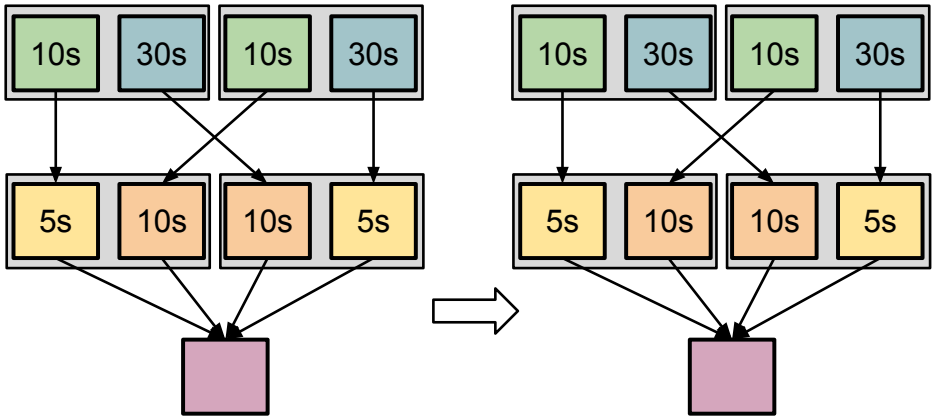

Figure 3.11: *VC-posterior*: horizontal clustering (balancing methods) is performed first, and then vertical clustering (but without changes).

VC-posterior Here, balancing methods are first applied, and then vertical clustering. Figure 3.11 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.

3.4 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) [**?**], which is widely used by workflow management systems such as Pegasus [31]. We also compare our methods with two heuristics described in literature: DFJS [81], and AFJS [68]. 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.

3.4.1 Scientific workflow applications

Five real scientific workflow applications are used in the experiments: LIGO Inspiral analysis [63], Montage [9], CyberShake [45], Epigenomics [122], and SIPHT [106]. In this subsection, we describe each workflow application and present their main characteristics and structures.

LIGO Laser Interferometer Gravitational Wave Observatory (LIGO) [63] 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 3.12 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 3.12.

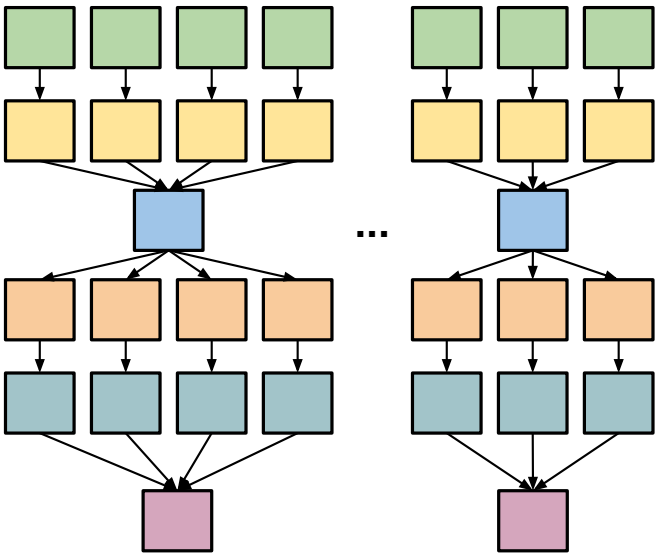

Figure 3.12: A simplified visualization of the LIGO Inspiral workflow.

Montage Montage [9] 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 3.13 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.

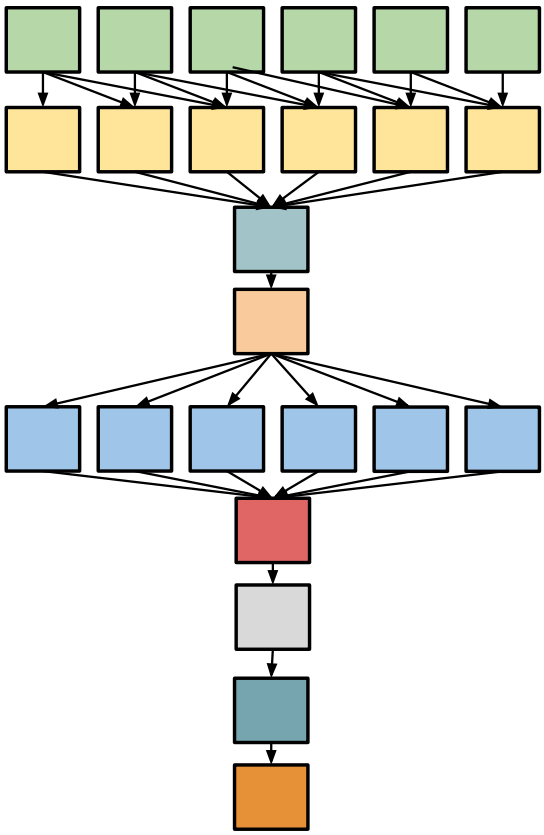

Figure 3.13: A simplified visualization of the Montage workflow.

Cybershake CyberShake [45] 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 3.14 shows an illustration of the Cybershake workflow.

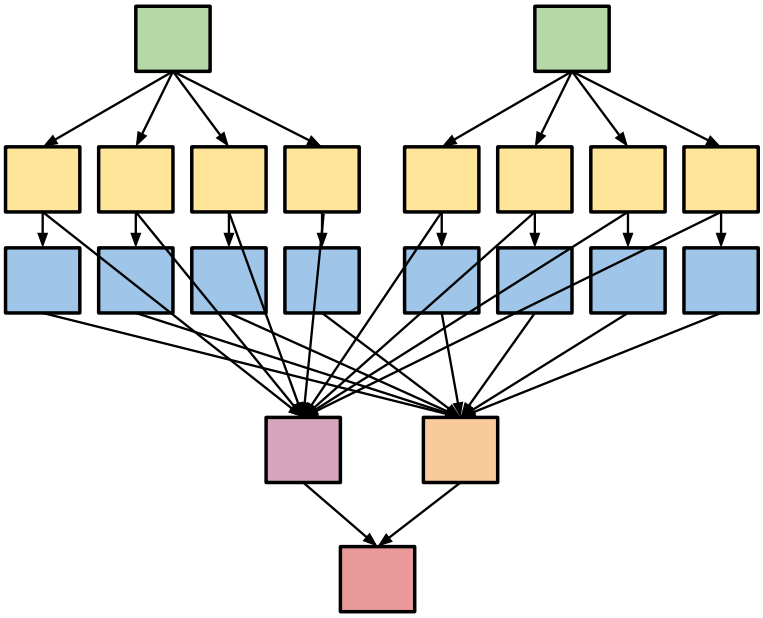

Figure 3.14: A simplified visualization of the CyberShake workflow.

Epigenomics The Epigenomics workflow [122] 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 3.15.

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

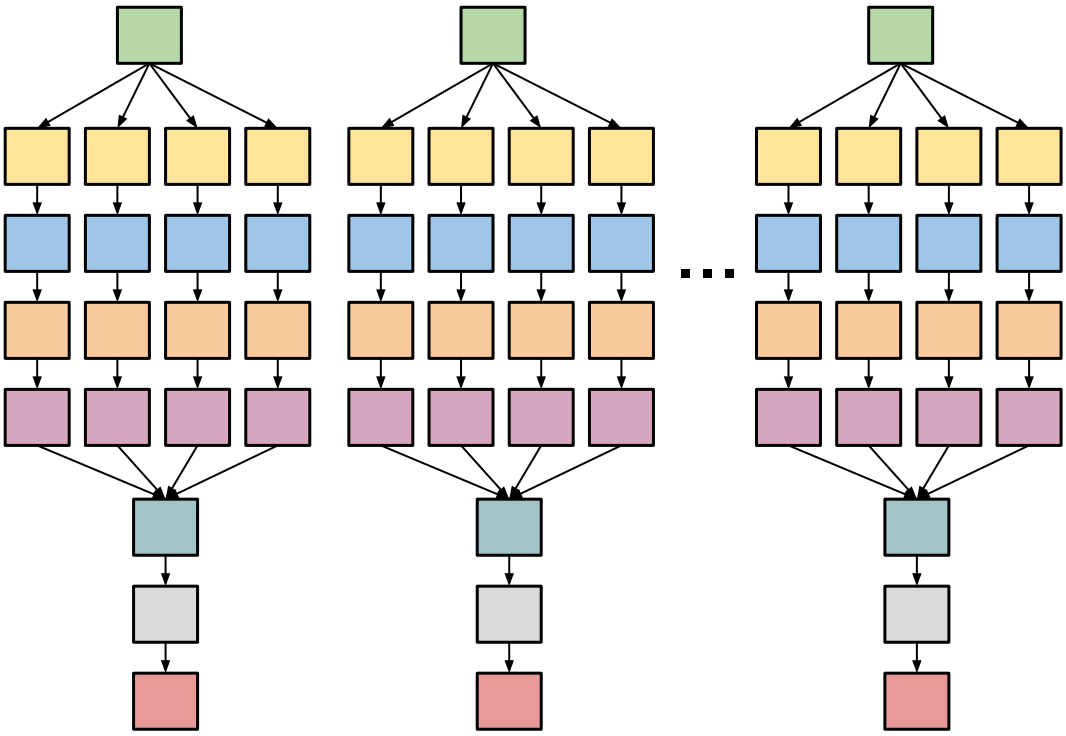

Figure 3.15: A simplified visualization of the Epigenomics workflow with multiple branches.

SIPHT The SIPHT workflow [106] 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 [31]. These involve the prediction of ρ-independent transcriptional terminators, BLAST (Basic Local Alignment Search Tools [106]) 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 3.16.

Table 3.3 shows the summary of the main **workflows characteristics**: number of tasks, average data size, and average task runtimes for the five workflows.

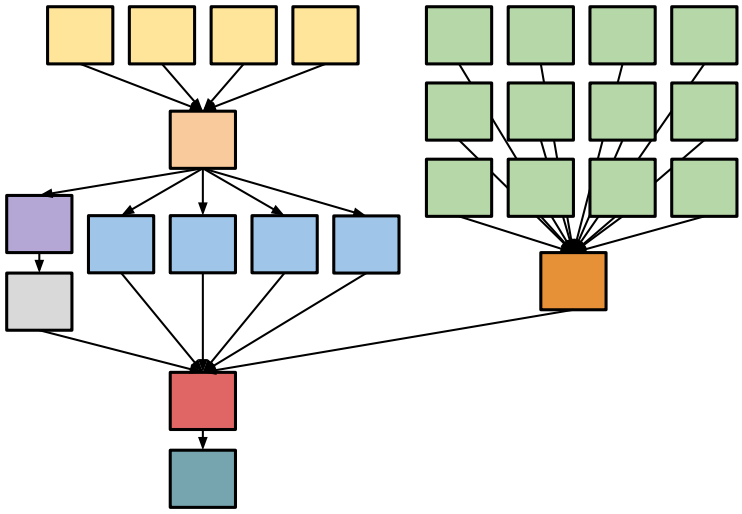

Figure 3.16: A simplified visualization of the SIPHT workflow.

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 3.3: Summary of the scientific workflows characteristics.

3.4.2 Task clustering techniques

In the experiments, we compare the performance of our balancing methods to the Horizontal Clustering (HC) [105] technique, and with two methods well known from the literature, DFJS [81] and AFJS [68]. 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 [105]. For simplicity, we define *clusters.num* as the

number of available resources. In our prior work [26], we have compared the runtime performance with different clustering granularity. The pseudocode of the HC technique is shown in Algorithm 4.

Algorithm 4 Horizontal Clustering algorithm.

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

procedure CLUSTERING(W, C)
  for $level < depth(W)$ **do**
    $TL \leftarrow$ GETTASKSATLEVEL($W, level$) ▷ Partition W based on depth
    $CL \leftarrow$ MERGE(TL, C) ▷ Returns a list of clustered jobs
    $W \leftarrow W - TL + CL$ ▷ Merge dependencies as well
  end for
end procedure
procedure MERGE(TL, C)
  $J \leftarrow \{\}$ ▷ An empty job
  $CL \leftarrow \{\}$ ▷ An empty list of clustered jobs
  while TL is not empty **do**
    J.add (TL.pop(C) ▷ Pops C tasks that are not merged
    CL.add(J)
  end while
  return CL
end procedure

DFJS The dynamic fine-grained job scheduler (DFJS) was proposed by Muthuvelu et al. [81]. 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 5 shows the pseudocode of the heuristic.

AFJS The adaptive fine-grained job scheduler (AFJS) [68] 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 6.

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 5 DFJS algorithm.

Require: W: workflow; $max.runtime$: max runtime of clustered jobs

procedure CLUSTERING($W, max.runtime$)
  for $level <$the depth of W **do**
    $TL \leftarrow$ GETTASKSATLEVEL($W, level$) ▷ Partition W based on depth
    $CL \leftarrow$ MERGE($TL, max.runtime$) ▷ Returns a list of clustered jobs
    $W \leftarrow W - TL + CL$ ▷ Merge dependencies as well
  end for
end procedure
procedure MERGE($TL, max.runtime$)
  $J \leftarrow \{\}$ ▷ An empty job
  $CL \leftarrow\{\}$ ▷ An empty list of clustered jobs
  while TL is not empty **do**
    $t \leftarrow TC$.pop() ▷ Get a task that is not mereged
    if J.runtime + t.runtime $> max.runtime$ **then**
      CL.add(J)
      $J \leftarrow\{\}$
    end if
    J.add(t)
  end while
  return CL
end procedure

Algorithm 6 AFJS algorithm.

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

procedure CLUSTERING($W, max.runtime$)
  for $level <$the depth of W **do**
    $TL \leftarrow$ GETTASKSATLEVEL($W, level$) ▷ Partition W based on depth
    $CL \leftarrow$ MERGE($TL, max.runtime, max.datasize$) ▷ Returns a list of clustered jobs
    $W \leftarrow W - TL + CL$ ▷ Merge dependencies as well
  end for
end procedure
procedure MERGE($TL, max.runtime, max.datasize$)
  $J \leftarrow \{\}$ ▷ An empty job
  $CL \leftarrow\{\}$ ▷ An empty list of clustered jobs
  while TL is not empty **do**
    $t \leftarrow TC$.pop() ▷ Get a task that is not mereged
    if J.runtime + t.runtime $> max.runtime$ OR J.datasize + t.datasize $> max.datasize$ **then**
      CL.add(J)
      $J \leftarrow\{\}$
    end if
    J.add(t)
  end while
  return CL
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 3.4.1. 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 3.17—experiment conditions are presented in Section 3.4.3. 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.

3.4.3 Experiment conditions

We adopt a trace-based simulation approach, where we extended our WorkflowSim [**?**] 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 level. It

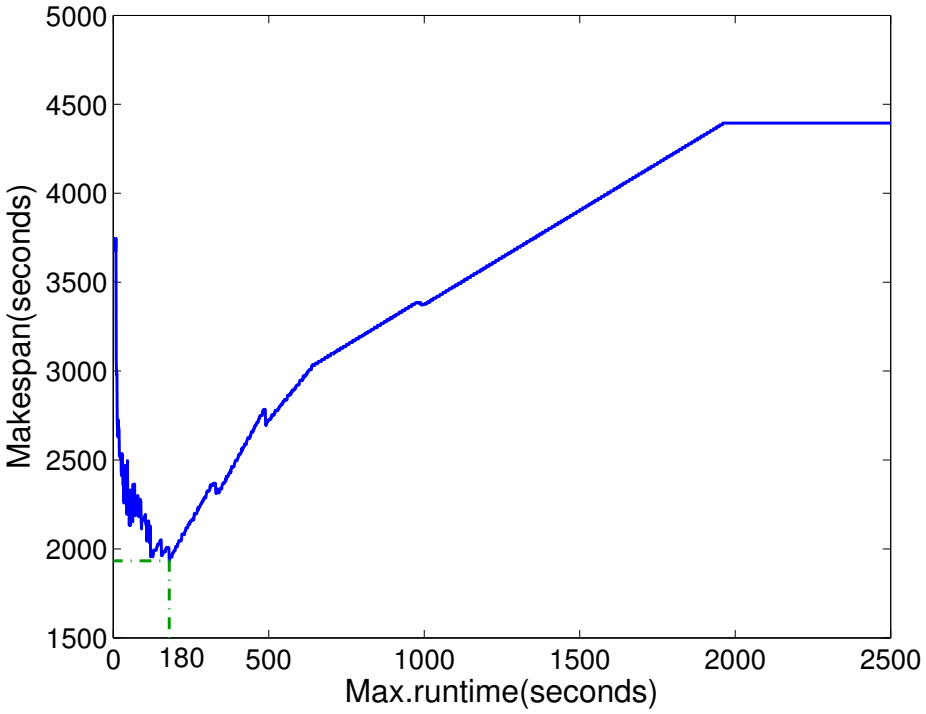

Figure 3.17: Relationship between the makespan of workflow and the specified maximum runtime in DFJS (Montage).

has been recently used in multiple workflow study areas [26, 22, 53] and its correctness has been verified in [**?**].

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 [44]. Amazon EC2 is a commercial, public cloud that has been widely used in distributed computing, in particular for scientific workflows [**?**]. 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 [26], which shows such selection is acceptable.

We collected workflow execution traces [54, 21] (including overhead and task runtime information) from real runs (executed on FutureGrid and Amazon EC2) of the scientific workflow applications described in Section 3.4.1. The traces are used to feed the Workflow Generator toolkit [130] 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 3.3.

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 3.4 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 3.4: Combination Results. '-' indidates the order of performing these algorithms, i.e., VC-HIFB indicates we perform VC first and then HIFB

3.4.4 Results and discussion

Experiment 1 Figure 3.18 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 3.5. 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 3.5. 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 3.15) 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.

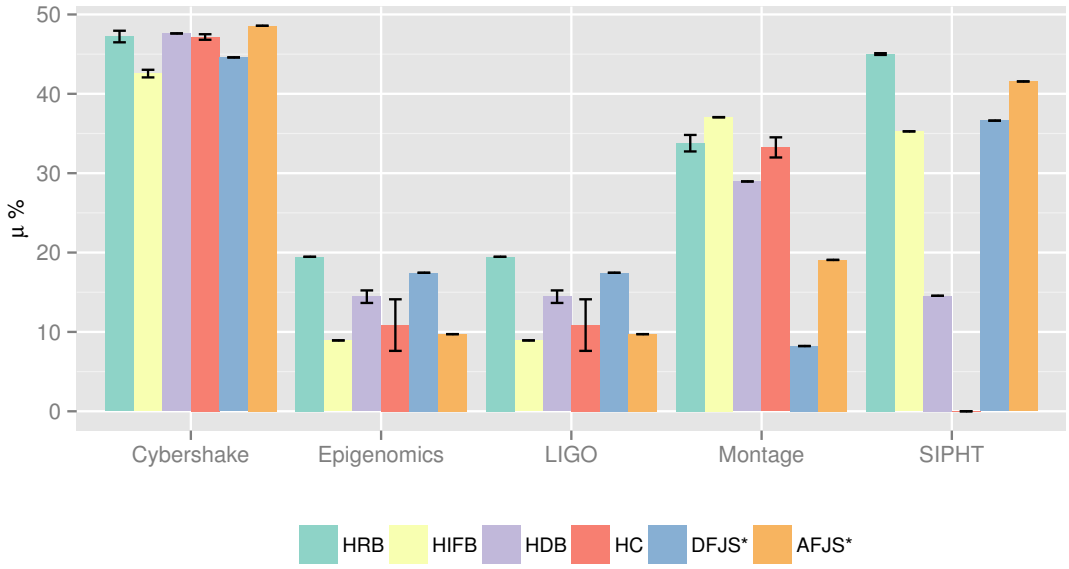

Figure 3.18: Experiment 1: performance gain (μ) over a baseline execution for six algorithms (* indicates the tuned performance of DFJS and AFJS). By default, we have 20 VMs.

Experiment 2 Figure 3.19 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 3.5: 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 3.18. 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 3.12, however, as mentioned in Section 3.3.2, the LIGO workflow has a few branches that depend on each other as shown in Figure 3.9. 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 3.19. 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 3.20 and 3.21 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 3.20), 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 3.21), 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

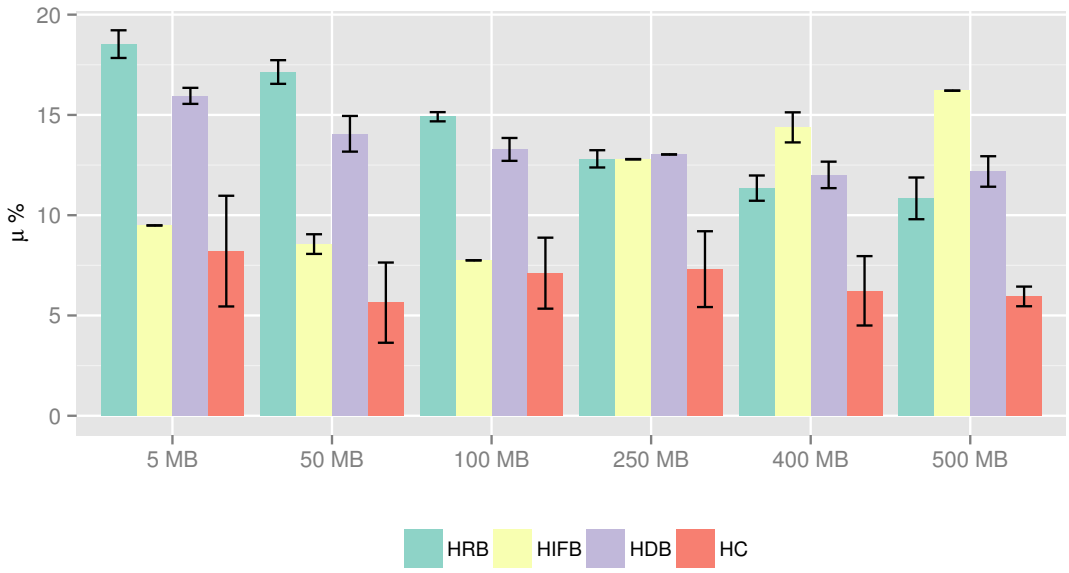

Figure 3.19: Experiment 2: performance gain (μ) over a baseline execution with different average data sizes for the LIGO workflow. The original avg. data size is 5MB.

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 3.22 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 3.14). Similarly, VC does not improve the SIPHT workflow due to the lack of pipelines on its structure (Figure 3.16). Thus, results for this workflow are omitted.

Figure 3.23 shows the performance gain μ for the Montage workflow. In this workflow, vertical clustering is often performed on the two pipelines (Figure 3.13). These pipelines are commonly single-task levels, thereby no horizontal clustering is performed on the pipelines. As

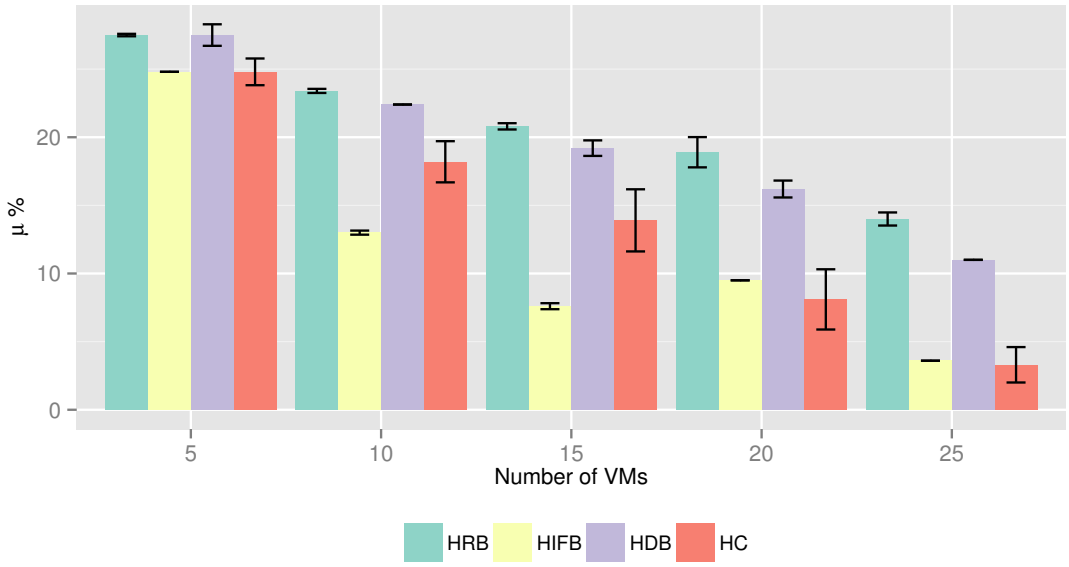

Figure 3.20: Experiment 2: performance gain (μ) over baseline execution with different number of resources for the LIGO workflow (average data size is 5MB).

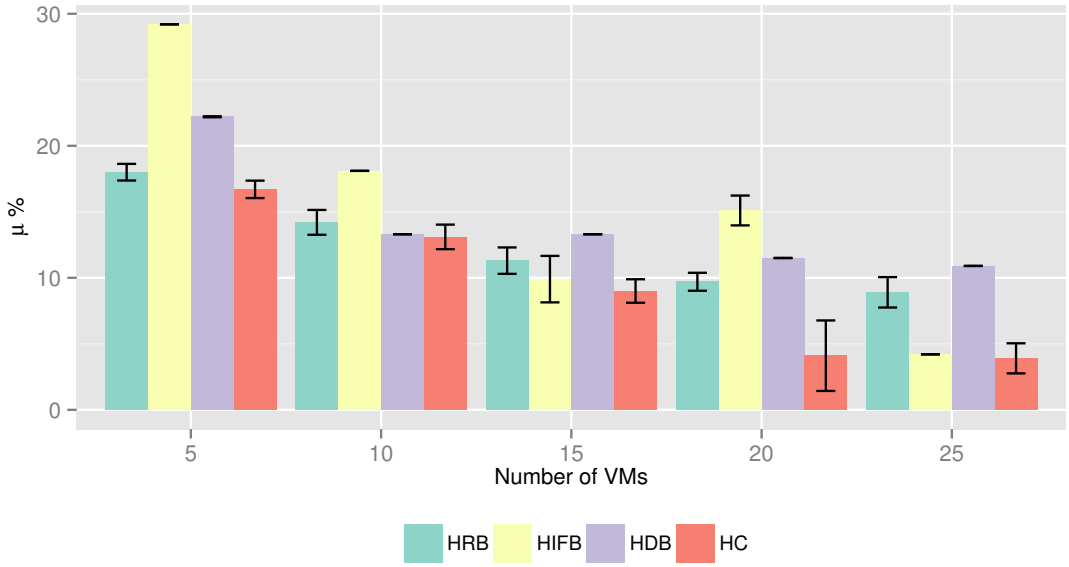

Figure 3.21: Experiment 2: performance gain (μ) over baseline execution with different number of resources for the LIGO workflow (average data size is 500MB).

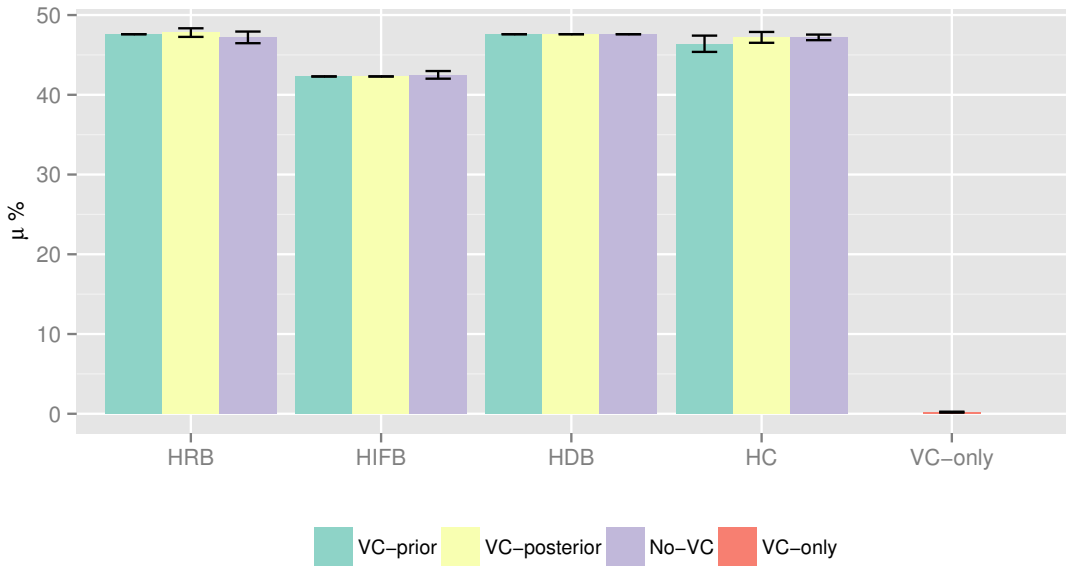

Figure 3.22: Experiment 3: performance gain (μ) for the Cybershake workflow over baseline execution when using vertical clustering (VC).

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 3.24. Vertical clustering yields better performance gain when it is performed prior to horizontal clustering (*VC-prior*). The LIGO workflow structure (Figure 3.12) 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.

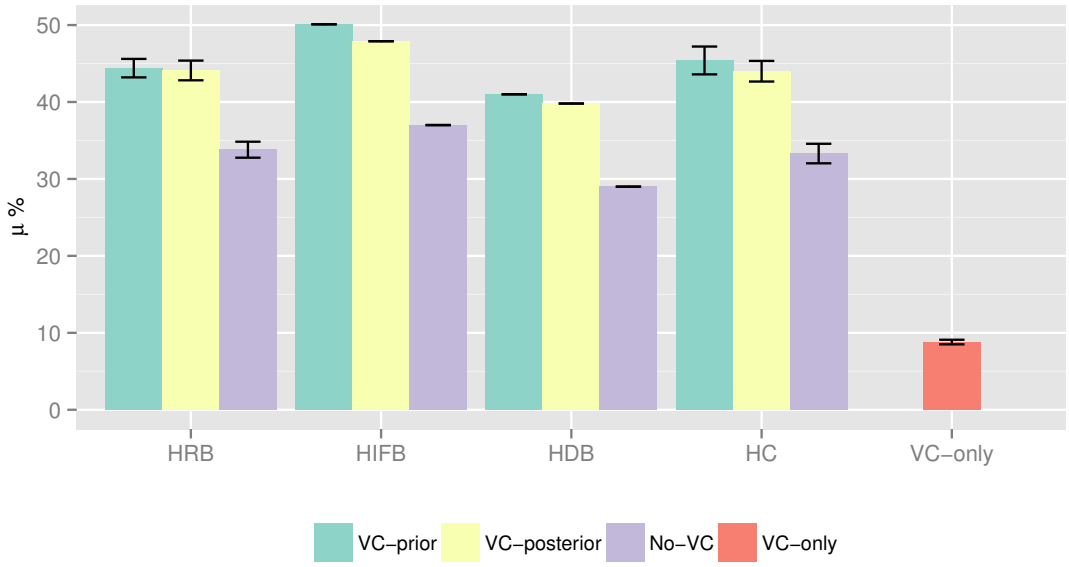

Figure 3.23: Experiment 3: performance gain (μ) for the Montage workflow over baseline execution when using vertical clustering (VC).

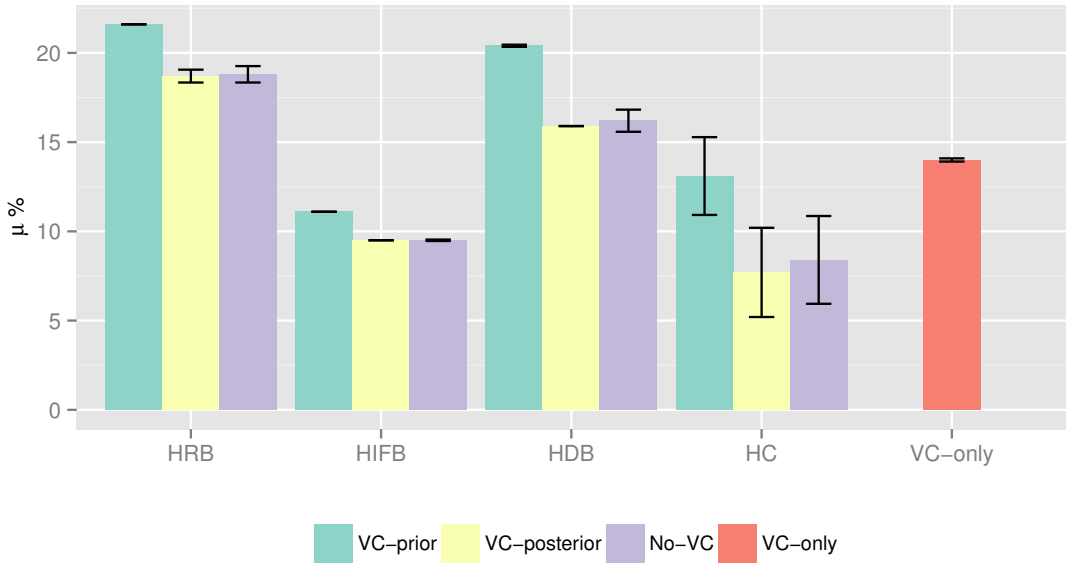

Figure 3.24: Experiment 3: performance gain (μ) for the LIGO workflow over baseline execution when using vertical clustering (VC).

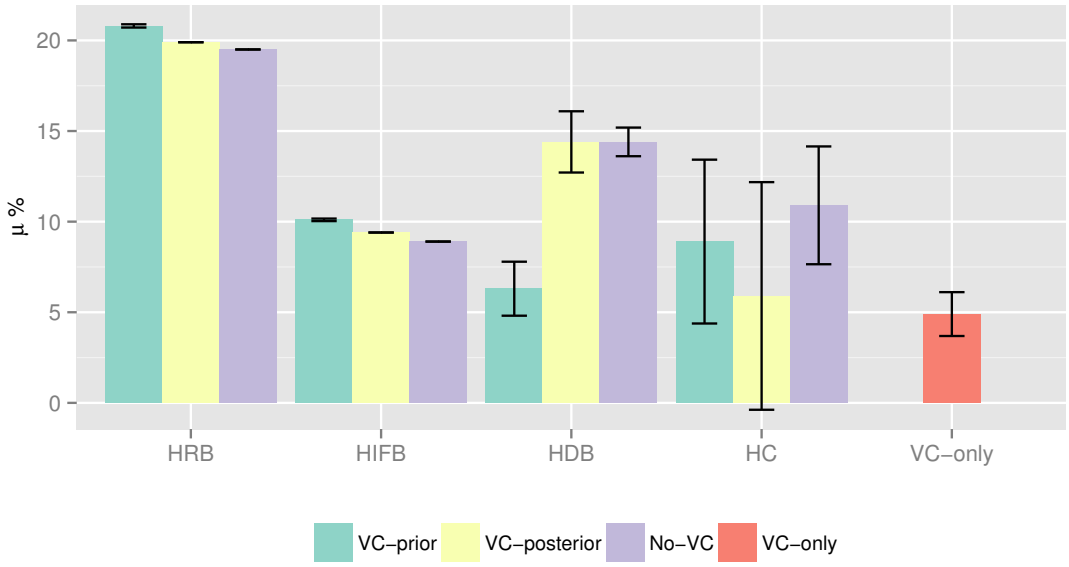

Figure 3.25: Experiment 3: performance gain (μ) for the Epigenomics workflow over baseline execution when using vertical clustering (VC).

Similarly to the LIGO workflow, the performance gain μ values for the Epigenomics workflow (see Figure 3.25) are better when VC is performed *a priori*. This is due to several pipelines inherent to the workflow structure (Figure 3.15). 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. 3.3. 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.

3.5 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 3.25, *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 4

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%.

4.1 Motivation

Data movement between tasks in scientific workflows has received less attention compared to task execution. Often the staging of data between tasks is 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 [44], Teragrid/XSEDE [115] 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 [29] and MPI/DAGMan [74]) while the traditional task clustering technique requires all the tasks can be executed in the same execution environment.

4.2 Related Work

For convenience and cost-related reasons, scientists execute scientific workflows [10, 36] in distributed large-scale computational environments such as multi-cluster grids, that is, grids comprising multiple independent execution sites. Topcuoglu [120] 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 [109] 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 [36] and Wieczorek [129] 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 [36] then developed a distributed service-oriented Enactment Engine with a master-slave architecture for de-centralized coordination of scientific workflows. Kumar [61] 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 [34] and Kalayci [58] 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 [34] and Kalayci [58], 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. [90] 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 [90], 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 [97] 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. [60] 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 [103] 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 [96] 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.

4.3 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).

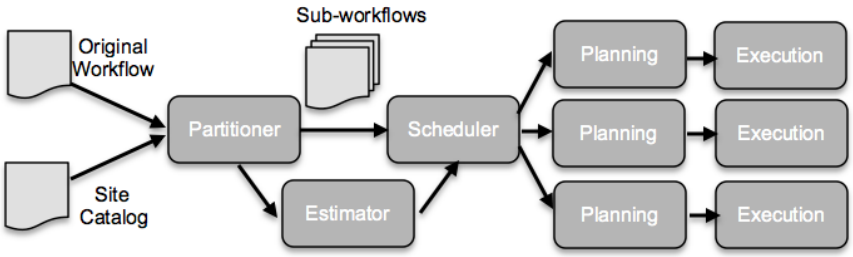

Figure 4.1: The steps to partition and schedule a workflow

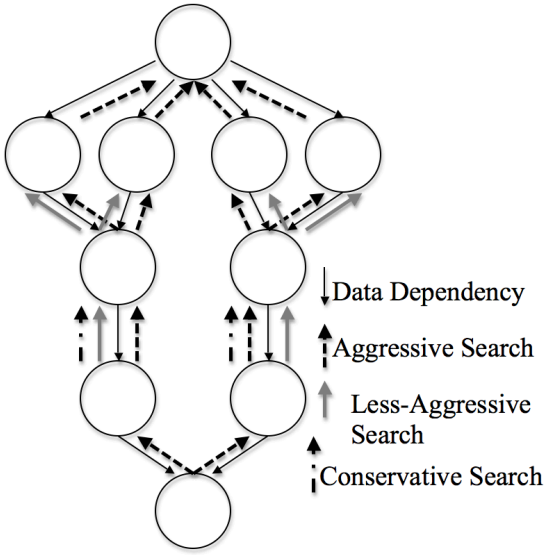

Figure 4.2: Three Steps of Search

Our approach (see Figure 4.1) has three phases: partition, estimate and schedule. The partitioner takes the original workflow and site catalog as input, and outputs various sub-workflows

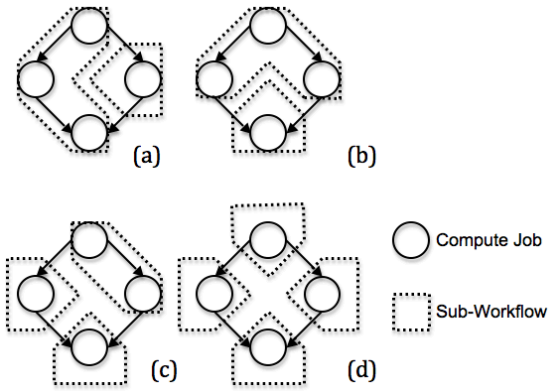

Figure 4.3: Four Partitioning Methods

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 4.3, we show the result of four different partitioning. Partitioning (a) does not work in practice since it has a deadlock loop. Partitioning (c) is valid but not efficient compared to Partitioning (b) or (d) that have more parallelism.

Usually jobs that have parent-child relationships share a lot of data since they have data dependencies. Its reasonable to schedule such jobs into the same partition to avoid extra data transfer and also to reduce the overall runtime. Thus, we propose Heuristic I to find a group of parents and children. Our heuristic only checks three particular types of nodes: the fan-out job,

the fan-in job, and the parents of the fan-in job and 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 [120], 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 4.3(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 4.2 depicts an example of three steps of search while the workflow in it has an average depth of 4. Pseudo-code of Heuristic I is depicted in Algorithm 7 .

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

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

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

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

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

2. Average CPU Time is the quotient of cumulative CPU time of all jobs divided by the number of available resources (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[120] 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.

4.4 Experiments and Discussion

In order to quickly deploy and reconfigure computational resources, we use a private cloud computing resource running Eucalyptus [84]. Eucalyptus is an infrastructure software that provides on-demand access to Virtual Machine (VM) resources. In all the experiments, each VM has

Algorithm 7 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 [29], which provides the workflow execution engine. Each execution site contains a Condor pool and a head node visible to the network.

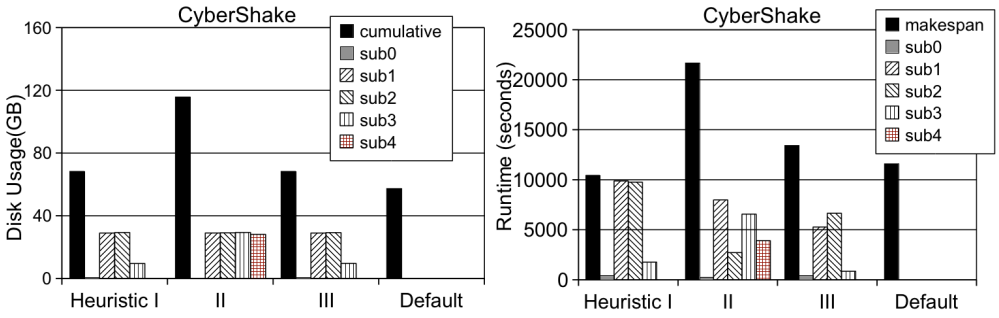

Figure 4.4: Performance of the three heuristics. The default workflow has one execution site with 4 VMs and 8 Condor slots and has no storage constraint.

Table 4.1: 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.

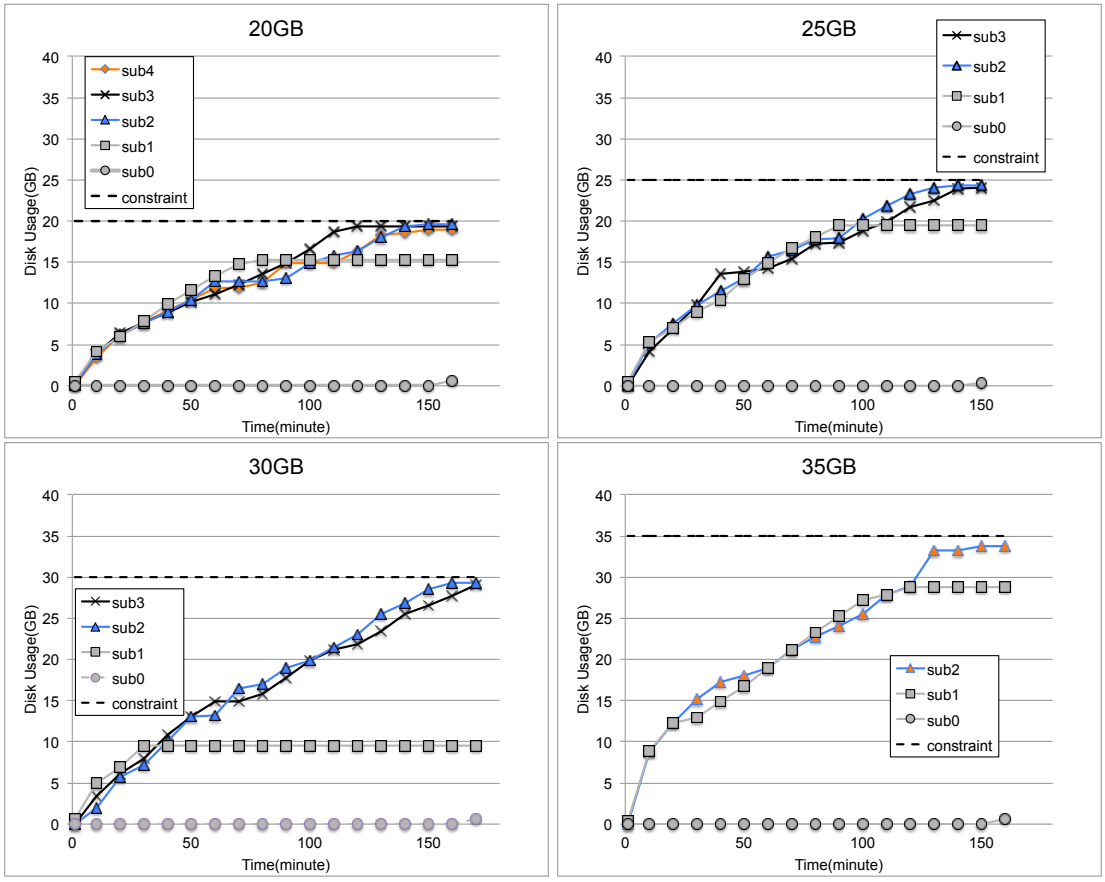

Figure 4.5: CyberShake with storage constraints of 35GB, 30GB, 25GB, and 20GB. They have 3, 4, 4, and 5 sub-workflows and require 2, 3, 3, and 4 sites to run respectively.

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 [57].

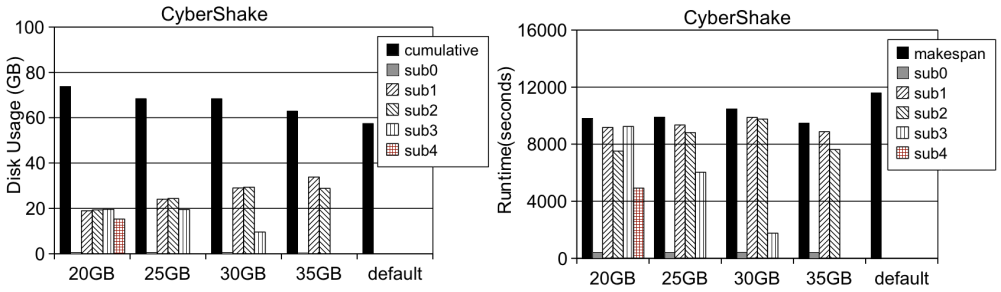

Figure 4.6: Performance of the CyberShake workflow with different storage constraints

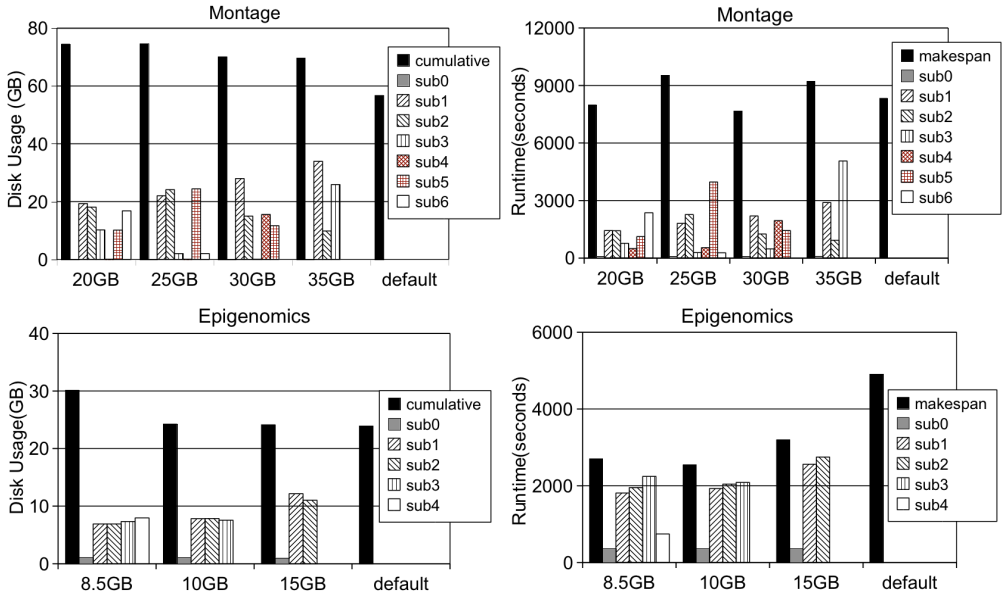

Figure 4.7: Performance of the Montage workflow with different storage constraints

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 4.4 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 4.5 and Table 4.1 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 4.6 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

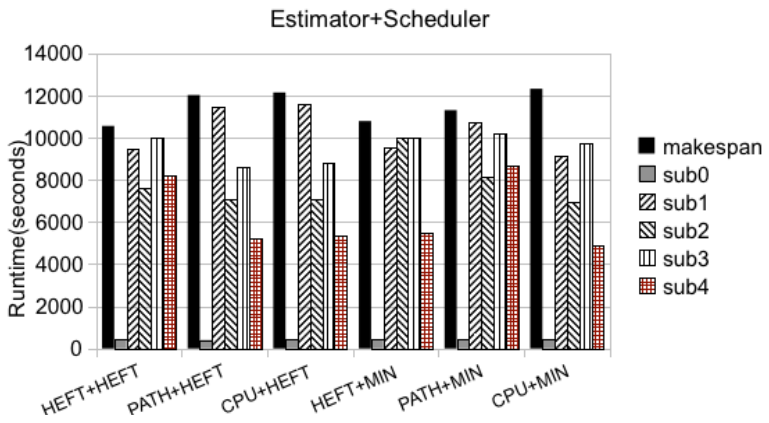

Figure 4.8: Performance of estimators and schedulers

Table 4.2: 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 4.7 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 4.8 and Table 4.2 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 4.8) does not consider the resource availability. The HEFT scheduler is slightly better than MinMin scheduler (or MIN in Figure 4.8). 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 5

Fault Tolerant Clustering

The influence of system failures on task clustering has received little attention. However, system failures play an important role in terms of both system reliability and runtime performance of workflow applications. In this chapter, we analyze the influence of different categories of transient failures and then we propose fault tolerant clustering methods to reduce their influence on the overall workflow performance. Simulation-based experiments show that our dynamic reclustering methods can achieve a speedup of up to 5 in the overall runtime of workflows compared to the horizontal clustering without taking fault tolerance into consideration.

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.

We focus on transient failures because they are expected to be more prevalent than permanent failures [138]. Transient failures are random failures and are recoverable. 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 [138]. Based on their occurrence, we divide the transient failures into two categories: task failure and job failure. If the transient failure occurs to the computation of a task (task failure), other tasks within the job do not necessarily fail. If the transient failure occurs to the clustered job (job failure), all of its tasks fail. Accordingly, we have two models. In the task failure model (TFM), the failure

of a task is a random event that is independent of the workflow characteristics and execution environment (such as which compute node the task is executed on). The task failure rate is the percentage of failed tasks among all executed tasks. Similarly we can define the job failure rate as the percentage of failed jobs among all executed jobs and a job failure model (JFM) in which job failure is a random event. Figure 5.1 shows a simple example of job failure and task failure, in which we have a clustered job (j) with four tasks (t_1, t_2, t_3 and t_4). If all of the four tasks fail (the left figure), it is more likely to have a job failure. If only a portion of tasks within a clustered job fail (for example, in the right figure, only t_1 and t_3 fail), it is more likely to be a task failure.

With such a classification of failures, we then analyzed their distinct influence on the overall runtime performance. Respectively, we present a job failure model and a task failure model. What distinguishes our work is that: 1) we analyzed the optimal size of a clustered job based on the estimation of failure rate; 2) we dynamically retried clustered jobs based on the suggested optimal size of a clustered job; 3) furthermore, we select more reliable resources to run clustered jobs.

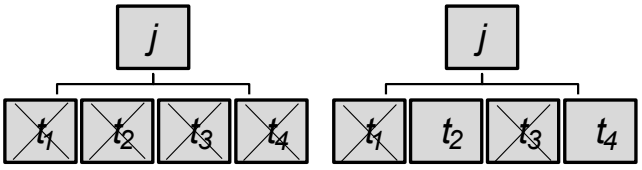

Figure 5.1: Job Failure (Left) and Task Failure(Right). The symbol $\times$ indicates a failure.

5.2 Related Work

Schroeder et al. [101] 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. [98] analyzed the empirical and statistical properties of system errors and failures from a network of heterogeneous servers running a diverse workload. Oppenheimer et al. [86] analyzed the causes of failures from three large-scale Internet services and the effectiveness of various techniques for preventing and mitigating service failure. Benoit [8] et al. analyzed the impact of transient and fail-stop failures

on the complexity of task graph scheduling. Our approach is based on these works. We measure the failure rates in a workflow and then provide fault-aware methods to improve task clustering.

Task retry is a technique that simply retries the failed job until it is successful. 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. The Recovery-Aware Components approach [135] proposed a purely component-based fault tolerant approach to increase the reliability of grid applications, essentially by predicting possible failures with a proactive Markov model and proactively initiating failure aversion. In our work, we also use task retry to resubmit failed tasks to execute. However, the difference is that we also adjust the size of a clustered job (number of tasks within a job) dynamically in order to optimize the runtime performance.

Egwutuoha [37] proposed to use process level redundancy techniques to reduce the runtime of the execution of computational intensive applications and reduced the overhead associated with checkpointing significantly. The application process can be periodically checkpointed so that when a failure occurs, the amount of work to be retried is limited. However, checkpointing increases the runtime of the execution of applications [138]. For a large workflow with many short tasks, the overheads of checkpointing can still limit its benefits. The difference is that we only check the completion status of a job after a job is completed and resubmit it if it failed. This approach avoids frequent checkpointing since most tasks are short and thus we can reduce the overheads of task retry.

Kwok [62] proposed to use a duplication-based approach in scheduling tasks to a heterogeneous cluster of PCs. In duplication-based scheduling, critical tasks are redundantly scheduled to more than one machine in order to reduce the number of inter-task communication operations. The task duplication process is guided by the system heterogeneity in that the critical tasks are scheduled or replicated in faster machines. Meyer et al. [78] presented a generalized approach to planning spatial workflow schedules for Grid execution based on the spatial proximity of files and the spatial range of jobs. They proposed to take advantage of data locality through the use of dynamic replication and schedule jobs in a manner that reduces the number of replicas created and the number of file transfers performed when executing a workflow. 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 task failure rate is low.

Silva et al. [40] presented a self-healing process that quantifies incident degrees of workflow activities from metrics measuring long-tail effect, application efficiency, data transfer issues, and site-specific problems. No strong assumption is made on the task duration or resource characteristics and incident degrees are measured with metrics that can be computed during the runtime. Silva and Rebello [28] described a strategy to endow autonomic MPI applications with the property of self-healing and thus be capable of withstanding multiple simultaneous crash faults of processes and/or processors. In our work, we do not categorize the physical causes of a failure since different causes may share the similar influence on the runtime performance of task clustering. We thus characterize failures in terms of task or job failures since they have totally different influence on the overall runtime.

5.3 Approach

5.3.1 Failure Models

The goal is to reduce the estimated finish time (M) of n tasks in case the failure rate for a clustered job (denoted by β) or a task (denoted by α) is known. M includes the runtime of the clustered job and its subsequent retry jobs if the first try fails. The average time to run a single task once is $\bar{\phi}_t$. k is the cluster size indicating the number of tasks in a clustered job. For a clustered job, let the expectation of retry times 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 retried until it is eventually completed successfully because we assume all the failures are transient. By definition we have, $N = 1/\gamma = 1/(1-\beta)$, while γ is the success rate of a job. Below we show how to estimate M. r is the number of available resources. D is the time delay between jobs and it includes the workflow engine delay, the queue delay and the postscript delay. We assume that $n \gg r$, but n/k is not necessarily much larger than r. Normally at the beginning of workflow

execution, $n/k > r$, which means there are more clustered jobs than available resources. To try all n tasks once irrespective of whether they succeed or fail, one needs approximately $n/(rk)$ execution cycle(s) since at each execution cycle we can execute at most r jobs. Therefore, the time to execute all n tasks once is $n(k\bar{\phi_t} + D + Ck)/(rk)$. And the time to complete them successfully in a faulty environment is $Nn(k\bar{\phi_t} + D + Ck)/(rk) = n(k\bar{\phi_t} + D + Ck)/(rk\gamma)$ since each job requires N retries on average. On the other side, at the end of the workflow execution, since n is decreasing with the process of workflow, it is possible that $n/k < r$, which means there are fewer jobs than the available resources. One needs just one execution cycle to execute these tasks once. The time to complete all n tasks successfully is $N(k\bar{\phi_t} + D + Ck) = (k\bar{\phi_t} + D + Ck)/\gamma$. Below we discuss how we estimate γ in TFM and JFM. In JFM, we have assumed that job failure is an independent event and thereby we only need to collect the failure records of jobs. $\gamma = (1 - \beta)$. In conclusion, in JFM,

$$M = \begin{cases} \dfrac{Nn(k\bar{\phi_t} + D + Ck)}{rk} = \dfrac{n(k\bar{\phi_t} + D + Ck)}{rk\gamma}, & \text{if } \dfrac{n}{k} \geq r \\ N(k\bar{\phi_t} + D + Ck) = \dfrac{k\bar{\phi_t} + D + Ck}{\gamma}, & \text{else} \end{cases} \tag{5.1}$$

In TFM, a clustered job succeeds only if all of its tasks succeed. Therefore the success rate of a clustered job is $\gamma = (1 - \alpha)^k$, and $\beta = (1 - \gamma)$. The task failure is independent and the job failure rate β is dependent on α. In conclusion, in TFM,

$$M = \begin{cases} \dfrac{Nn(k\bar{\phi_t} + D + Ck)}{rk} = \dfrac{n(k\bar{\phi_t} + D + Ck)}{rk(1-\alpha)^k}, & \text{if } \dfrac{n}{k} \geq r \\ N(k\bar{\phi_t} + D + Ck) = \dfrac{k\bar{\phi_t} + D + Ck}{(1-\alpha)^k}, & \text{else} \end{cases} \tag{5.2}$$

To find the minimal value of M and the optimal k in Equation 5.1, we have

$$\begin{aligned} \overset{*}{k} &= \frac{n}{r} \\ \overset{*}{M} &= \frac{k\bar{\phi}_t + D + Ck}{\gamma} \end{aligned} \tag{5.3}$$

$\overset{*}{k}$ is the optimal cluster size and $\overset{*}{M}$ is the minimal runtime of these n tasks. Equation 5.3 shows that in JFM, the job failure rate has no influence on the choice of cluster size. In another word, in JFM we just need to set k to be $\overset{*}{k}$, which is a constant parameter irrespective of the job failure rate. To find the minimal value of M in Equation 5.2, let

$$\frac{dM}{dk} = 0 \tag{5.4}$$

We get

$$\begin{aligned} \overset{*}{k} &= \frac{-D + \sqrt{D^2 - \frac{4D}{\ln(1-\alpha)}}}{2(\bar{\phi}_t + C)} \\ \overset{*}{M} &= \frac{n(\overset{*}{k}\bar{\phi}_t + D + C\overset{*}{k})}{r\overset{*}{k}(1-\alpha)^{\overset{*}{k}}} \end{aligned} \tag{5.5}$$

To discuss the relationship between the variables mentioned above, we show an example workflow with $n = 1000$, $\bar{\phi}_t + C = 5$ sec, $D = 5$ sec, and $r = 20$. Figure 5.2 shows the relationship between the expected runtime $\overset{*}{M}$ and the cluster size $\overset{*}{k}$ in Equation 5.2. We can see that the optimal cluster size (in this example it is 50) does not change with different job failure rates. In comparison with TFM in Figure 5.3, the optimal cluster size (in this example it is about 5) is not equal to n/r. This conclusion is consistent with our previous claim since a clustered job has a higher chance to fail when the task failure rate is higher.

Figure 5.4 shows the relationship between the task failure rate and the optimal cluster size as indicated in Equation 5.5. We can see that when the task failure rate is high ($\alpha > 0.03$), it is

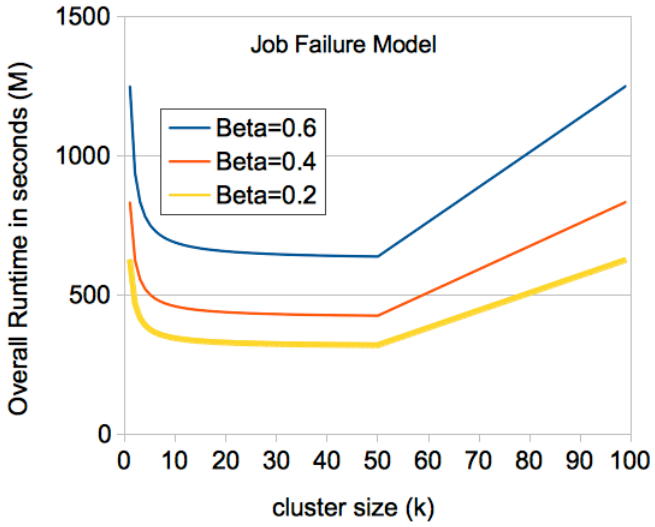

Figure 5.2: Job Failure Model

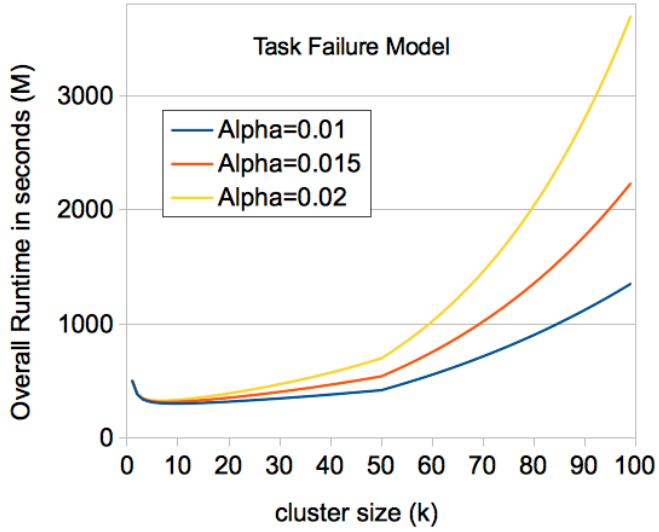

Figure 5.3: Task Failure Model

better to abandon clustering ($\overset{*}{k} < 1$). We can also see that the optimal cluster size $\overset{*}{k}$ decreases with the increase of task failure rate.

Below we introduce the method to tell which model a faulty environment belongs to if one has no knowledge about the cause of failures (transient or permanent, task or job). By analyzing the relationship between expected runtime and cluster size we can tell whether it tends to be

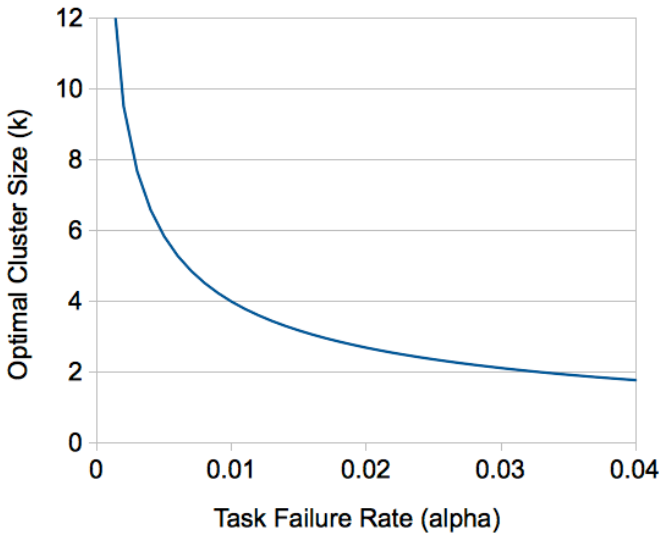

Figure 5.4: Task failure rate and optimal cluster size

TFM or JFM. There are two criteria: 1) whether the overall runtime has an exponential increase (TFM) or a linear increase (JFM) when k is large enough; 2) whether the optimal cluster size is influenced by the failure rates (If yes, TFM; otherwise JFM).

From this theoretic analysis, we conclude that: 1) the lower the task failure rate is, the better runtime performance the task clustering has; 2) in TFM, adjusting cluster size according to the detected task failure rate can improve the runtime performance; 3) in JFM, we just need to set $k = n/r$.

5.3.2 Fault Tolerant Clustering Methods

To improve the fault tolerance of horizontal clustering, we propose three methods: Dynamic Clustering (DC), Selective Reclustering (SR), and Dynamic Reclustering (DR).

1. Dynamic Clustering (DC)

   Dynamic Clustering adjusts the cluster size according to the task failure rate measured from jobs that have already been completed, either successfully or failed.

   Figure 5.5 shows an example where the initial cluster size is 4 and thereby there are four tasks in a clustered job at the beginning. Then three out of these tasks fail. Assume that

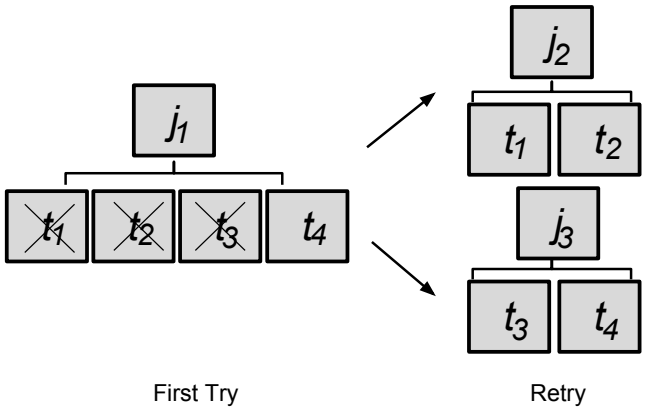

Figure 5.5: Dynamic Clustering

TFM suggests an optimal cluster size to be 2, then this job will be split into two clustered jobs while each has two tasks. The two clustered jobs are then submitted for retry. DC is not aware that there are only three failed tasks.

2. Selective Reclustering (SR)

   In DC, one knows the average task failure rate but the information about which tasks have failed is not available. In practice, it may be hard to identify the failed tasks, but if it is supported, we can 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 the nave job retry in that the latter method retries all tasks of a failed job even though some of the tasks have succeeded. Figure 5.6 shows an example of SR. At the beginning, there are four tasks and three of them have failed. One task succeeds and exits. Only the three failed tasks are merged again into a new clustered job and the job is retried. This approach does not intend to adjust the cluster size, although the cluster size will be smaller and smaller naturally after each retry since there are less and less tasks in a clustered job. In this case, the cluster size has reduced from 4 to 3.

3. Dynamic Reclustering (DR)

   Selective Reclustering does not analyze the failure rate and it uses a natural approach to reduce the cluster size if the failure rate is too high. However, it requires a special ability

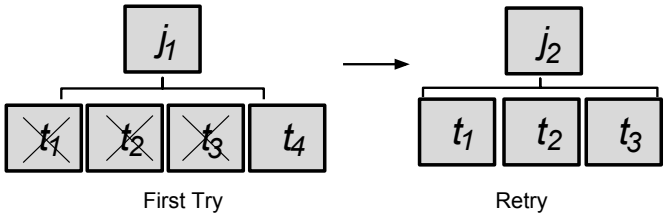

Figure 5.6: Selective Reclustering

to select the failed tasks from all the tasks, while DC does not. We then propose the third method, Dynamic Reclustering, which is a combination of SR and DC to see whether using both strategies can improve the runtime performance of workflows further. In DR, only failed tasks are merged into new clustered jobs and the cluster size is also adjusted according to the detected task failure rate.

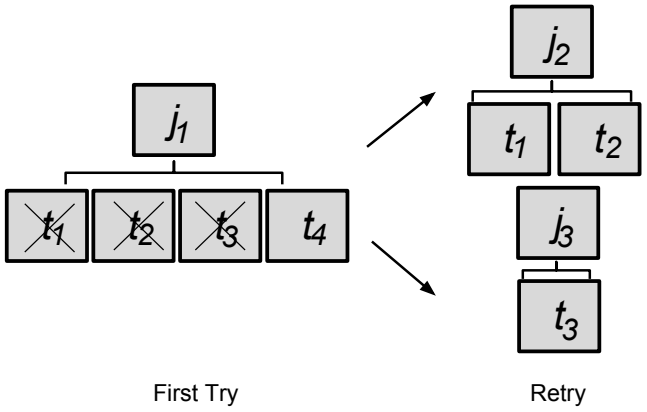

Figure 5.7: Dynamic Reclustering

Figure 5.7 shows the steps of DR. At the last scheduling cycle, three tasks within a clustered job have failed. Therefore we have only three tasks to retry and further we need to adjust the cluster size (in this case it is 2) according to the task failure rate.

5.4 Experiments and Discussion

We use WorkflowSim [24] to simulate these methods. Figure 5.8 compares the performance between DR and NOOP that has no optimization at all. Figure 5.8 shows that without any fault

tolerant optimization, the performance degrades significantly especially when the task failure rate is high. Figure 5.9 compares the three methods that we proposed and it shows that Dynamic Reclustering outperforms the other two because it derives strengths from both. In reality, it is difficult to simulate failures with precise failure rates while WorkflowSim provides a unique platform to evaluate fault tolerant designs.

In conclusion of this section, for horizontal clustering, the first solution dynamically adjusts the cluster size according to the detected task failure rate. The second technique retries the failed tasks within a job. And the last solution is a combination of the first two approaches.

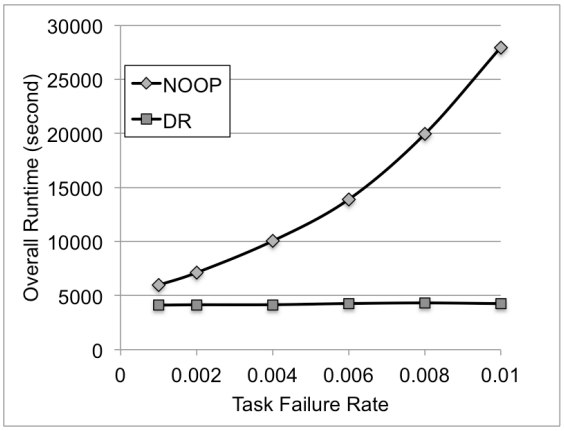

Figure 5.8: Performance between DR and NOOP

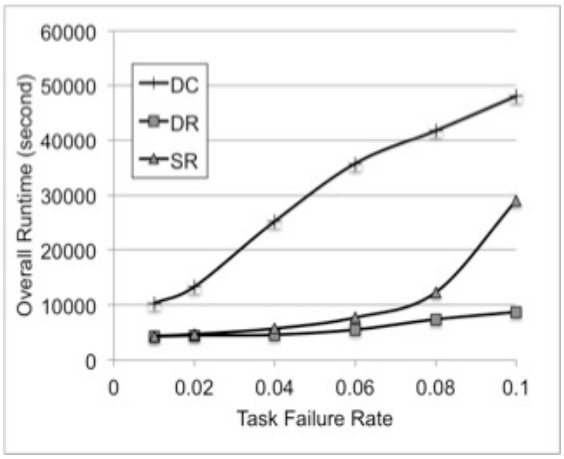

Figure 5.9: Performance between DC, SR, and DR

Chapter 6

Planned Work

The tradeoff between resource cost and runtime performance has been given a lot of attention in the past few years. However, how task clustering influences the resource cost and runtime performance respectively has not been discussed throughly. In this chapter, we propose to develop resource-aware clustering methods that adjust the strength of task clustering (size of a clustered job, i.e.) in order to improve both the resource cost and overall runtime of executing scientific workflows on clouds.

6.1 Motivation

The resource requirement of large scale scientific workflows has been increasing with the scale of computation and data transfer. Recently, the development of cloud computing [49, 4] has made it possible for anyone to easily lease large scale computing infrastructure from commercial resource providers. However, the resource usage of clouds is associated with monetary cost and the studies in reducing resource cost have been concluded ever since the emergence of cloud computing. Juve [55] et al. has utilized clouds to execute large scale scientific workflows and provided a series of provisioning and scheduling algorithms to improve the runtime and resource cost together. At the same time, there are still scheduling overheads for workflows running on a cloud and task clustering still applies in that case. The challenge is, on one hand, the amount of resources provisioned should not exceed the maximum parallelism degree of a workflow, otherwise resources are wasted and resource cost is increased without any performance gain. On the other hand, the amount of resources should be large enough so that the runtime performance of workflows is not influenced by the lack of resources. What makes this problem even more

challenging is that these two goals (performance vs cost) are usually conflicting and it is a typical multi-objective optimization problem [112, 128].

In our work, we specifically focus on balancing the resource cost and runtime performance through task clustering, which we call resource aware clustering. We plan to model the relationship between overall runtime and the resources used (and thus the associated resource cost). The approach we plan to use is to adjust the strength of task clustering (for example, size of a clustered job). Each job is coarse-grained enough to enable efficient execution and reduce scheduling overheads, while being fine-grained enough to fully utilize the resources offered by the hardware.

There is a number of work on estimating the overall runtime of workflows [35, 118, 121, 123]. For example, the performance method by Duan et al. [35] 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. However, it requires many parameters and a lot of parameter tuning, which is not feasible in practice. What distinguishes our work is: 1) We aim to provide a general but simple estimation model of the relationship between the overall runtime of a workflow and the resource cost. We assure that the model is useful in practice with less parameter training; 2) System overheads are included in our model to make it more precise and flexible in practice; 3) We especially focus on the task clustering problem and we use a hierarchical framework of computational activities.

6.2 Related Work

Resource provisioning techniques have recently been developed to acquire resources for dedicated usage before execution starts and thus scheduling overheads can be reduced. Juve [56] has developed a resource provisioning tool called Wrangler to allocate resources for the exclusive use of a single user for a given period of time. This minimizes queuing delays because a

user's jobs no longer compete with other jobs for access to resources. Villegas [125] presented a comprehensive and empirical performance-cost analysis of provisioning and allocation policies in IaaS clouds. They introduced a taxonomy of both types of policies, based on the type of information used in the decision process, and mapped to this taxonomy eight provisioning and four allocation policies. However, the problem with resource provisioning is that many grid sites do not allow users to take advantage of the feature [104]. Our approach further improves the overall runtime performance of workflows and reduces resource cost of workflow execution by dynamically adjusting the strategies of task clustering to ensure that a job is coarse-grained enough to enable efficient execution, while being fine-grained enough to fully utilize the available resources.

There are two ways to achieve resource provisioning traditionally. In advanced reservation [19, 42, 76, 77], a resource is configured to run only a single user's jobs on a given set of resources for a limited amount of time. Elmroth [38] proposed an algorithm to perform resource selection based on performance predictions and provided support for advance reservations. The algorithms proposed in their implementation perform resource selection based on performance predictions, and provide support for advance reservations as well as coallocation of multiple resources for coordinated use. However, advanced resource reservations is not always possible in all the Grid resources [104], for instance, in the desktops resources where the local users has more priority to run their jobs. Our work has assumed resources are provisioned before the execution ever starts and we do not discuss the implementation of the specific resource provisioning tools. However, advanced reservation techniques can be introduced to our model as our future work.

The other way to do resource provisioning is to use pilot job [102, 56], in which resources are provisioned by submitting ordinary jobs (called pilot jobs) to the execution site. Instead of running an application task, the pilot jobs work as a remote resource manager configured to manage application tasks from an application-level scheduler hosted outside of this execution site. This is also called multi-level scheduling [95] or placeholder scheduling [91]. However, users are required to estimate the resource usage before they ever start the execution, which may

not be feasible in practice. Also, similar to advanced reservation, pilot jobs may not be supported in many execution sites. Compared to our work, similar to the case of advanced reservation, we do not differentiate the implementation of resource provisioning and both implementation can be used in our work.

With the advance in resource provisioning, researchers have been trying to improve the efficiency of task scheduling [59, 2, 51, 92, 132]. Adamuthe et al. [2] proposed to use Genetic Algorithms to maximize the resource utilization and minimize job completion time under resource and time constraints. Khajevand [59] proposed a Minimum First-fit Cost-Makespan Trade-off (MinFCMT) heuristic algorithm in order to effectively schedule an application in Utility Grids so that the application makespan and allocation-cost can be minimized. Huu [51] designed virtual infrastructures allocation strategies for cloud computing platforms, which size and topology are optimized according to some user-controlled metric, using the application expertise captured by the workflow representation. Pop [92] presented a decentralized dynamic resource scheduling solution of large scale workflows onto distributed, heterogeneous Grid environments. Their work aims to optimize the scheduling performance and fault management. Yu [132] aimed to improve the performance of scheduling workflows with rigid time constraints based on the estimation of resource state reliability. However, task scheduling is still a NP-hard problem and the performance gain through task scheduling is limited by the complexity of scheduling algorithms. In our case, the runtime performance of task clustering depends on the ratio between computation, data transfer and system overheads and thus its performance gain is more significant than task scheduling in many distributed environments.

Another approach besides task scheduling that attracts much attention is task reallocation. With the aim of dynamically balancing the computational load among resources, some jobs have to be moved from one resource to another and/or from one period of time to another, which is called task reallocation [119]. Caniou [17] presented a reallocation mechanism that tunes parallel jobs each time they are submitted to the local resource manager (which implies also each time a job is migrated). They only needed to query batch schedulers with simple submission or cancellation requests. Its authors also presented different reallocation algorithms and studied

their behaviors in the case of a multi-cluster Grid environment. In [136], a pre-emptive process migration method was proposed to dynamically migrate processes from overloaded nodes to lightly-loaded nodes. However, it can achieve a good balance only when there are some idle compute nodes (e.g. when the number of task processes is less than that of compute nodes). In our case with large scale scientific workflows, usually we have more tasks than available compute nodes.

Adaptive execution is the third approach used to achieve the balance between resource cost and runtime performance. Due to the inherently dynamic characteristics of the resources and the occasional unpredictable behavior of the workflow tasks, the workflow execution plan may require variations to meet the desired Quality of Service objectives. Previous research [65, 99, 133] has considered enabling workflows to adapt to changing resource availability by re-planning portions of workflow. The problem considered by this approach is the inverse of the one addressed by resource provisioning. In their model, the workflows must adapt to changing resource availability rather than the resource availability adapting to changing workflows. This approach has to halt the execution of the original plan and enact a new execution plan, which is very costly to perform. Kalayci [58] proposed a new adaption approach that operates at the sub-workflow level and has lower overheads. In contrast, our work adjust the strategies of task clustering and resource provisioning together. Such a hybrid approach provides us a better scalability and adaption to the dynamic environments.

6.3 Planned Work

We plan to develop resource-aware task clustering methods and discuss their runtime performance and overall resource utilization.

The first goal is to determine if we can predict the runtime of real workflows with enough accuracy for it to be useful based on only predictions of the total CPU hour, critical path (the longest path from the root task to the exit task), max width (the maximum parallelism degree of

a workflow), and number of resources for workflow execution. The disadvantage of many traditional workflow estimation models is that they involve many assumptions and require tuning a lot of parameters, which is not feasible in practice. In our work, we aim to use as few parameters as possible and assure that the model is simple but effective in practice.

Once we have answered these questions, we can use the model we developed to create a task clustering algorithm for resource provisioning tools, which involves partitioning the workflow and predicting the number of resources, overall runtime and data storage required for each partition. We also aim to examine a workflow and develop an estimate that can be used to provision VMs on a cloud (i.e., Amazon EC2) and predict the cost of running workflows. We can also improve the provisioning algorithms with deadline and budget constraints proposed in [73].

Another concern we will focus on is the probability of under/over-predicting the workflow wall time. We aim to have an estimate of the overall runtime that is roughly closed to the actual overall runtime (i.e., 90%). It would be easy to achieve 100% accuracy by just doubling or tripling a crude estimate, but that is not feasible in practice. We would like to know how much the overall runtime of a workflow varies over repeated runs of the same workflow.

One further problem with the experiment we are about to undertake is that we initially assume that we have perfect estimates. In other words, we plan to use the actual runtime of the tasks from experiments to build a model that predicts the overall runtime of a workflow. In doing that we are basically assuming we have a perfect estimate. It would be interesting to take our perfect estimate and make it imperfect by increasing the variance of task runtime to see how our prediction changes. We thus can answer a question: How accurate do our task runtimes need to be in order to get a workflow overall runtime prediction with a given accuracy?

6.4 Research Plan

We have presented the current status of our work and outlined the tasks that are required to address the research questions proposed in this dissertation proposal. We next introduce a tentative thesis plan in three phases.

Phase I (May-Oct, 2013)

In this period, we build the theoretic model of resource provisioning and build a system platform to evaluate our results.

1. Develop the theoretical model used in resource-aware task clustering.

   In this period, analyze the theoretic relationship between overall runtime of workflows and the number of resources used.

2. Use Amazon EC2 to build a system and test the environment.

   In this period, build/use a cloud-provisioning system on top of Amazon EC2.

3. Collect traces of applications

   In this period, collect the runtime traces of different applications including Montage and Periodograms. Then use simulators to vary the parameters and evaluate the accuracy of the prediction based on workflow characteristics such as overall runtime, critical path and resource characteristics such as the number of resources to use.

Phase II (Nov 2013 - Jan 2014)

1. Implement algorithms and techniques used in resource-aware task clustering.

   Develop resource-aware task clustering algorithms to partition a workflow and predict the number of resources and overall runtime required for each partition.

2. Develop an estimation method to be used to provision VMs on a cloud and to predict the cost of running workflows. Furthermore, improve provisioning algorithms with deadline and budget constraints.

3. Integrate the algorithms with an existing resource provisioning tool

   Extend the DAG workflow to MPI workflow and integrate the algorithms used in theoretic analysis into a resource provisioning tool and then compare its performance with different workflow instances.

4. Data processing and analysis

   Present and visualize the results and conclude the performance.

Phase III (Feb-Apr, 2014)

1. Thesis write-up.

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] A. Benoit, L.-C. Canon, E. Jeannot, Y. Robert, et al. On the complexity of task graph scheduling with transient and fail-stop failures. 2010.

[9] 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.

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

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

[19] C. Castillo, G. Rouskas, and K. Harfoush. Efficient resource management using advance reservations for heterogeneous grids. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, 2008.

[20] 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.

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

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

[23] W. Chen and E. Deelman. Integration of workflow partitioning and resource provisioning. May 2012.

[24] 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.

[25] 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.

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

[27] 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.

[28] J. A. da Silva and V. E. F. Rebello. A hybrid fault tolerance scheme for easygrid mpi applications. In *Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '11, pages 4:1–4:6, New York, NY, USA, 2011. ACM.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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 *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, pages 339–347, 2009.

[36] 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.

[37] I. Egwutuoha, S. Chen, D. Levy, and B. Selic. A fault tolerance framework for high performance computing in cloud. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 709–710, 2012.

[38] E. Elmroth and J. Tordsson. A standards-based grid resource brokering service supporting advance reservations, coallocation, and cross-grid interoperability. *Concurr. Comput. : Pract. Exper.*, 21(18):2298–2335, Dec. 2009.

[39] 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.

[40] R. Ferreira da Silva, T. Glatard, and F. Desprez. Self-healing of operational workflow incidents on distributed computing infrastructures. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 318–325, 2012.

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

[42] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Quality of Service, 1999. IWQoS '99. 1999 Seventh International Workshop on*, pages 27–36, 1999.

[43] 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.

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

[45] 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.

[46] 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.

[47] 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.

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

[49] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good. On the use of cloud computing for scientific workflows. In *3rd International Workshop on Scientific Workflows and Business Workflow Standards in e-Science (SWBES '08)*, 2008.

[50] 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.

[51] T. Huu and J. Montagnat. Virtual resources allocation for workflow-based applications distribution on a cloud infrastructure. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 612–617, 2010.

[52] 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.

[53] 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.

[54] 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.

[55] G. Juve and E. Deelman. Resource provisioning options for Large-Scale scientific workflows. In *4th IEEE International Conference on eScience (eScience '08)*, Dec. 2008.

[56] G. Juve, E. Deelman, K. Vahi, and G. Mehta. Experiences with resource provisioning for scientific workflows using corral. *Scientific Programming*, 18(2), Apr. 2010.

[57] 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.

[58] 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.

[59] V. Khajevand, H. Pedram, and M. Zandieh. Provisioning-based resource management for effective workflow scheduling on utility grids. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:719–720, 2012.

[60] 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.

[61] 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.

[62] Y.-K. Kwok. Parallel program execution on a heterogeneous pc cluster using task duplication. In *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*, pages 364–374, 2000.

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

[64] 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.

[65] K. Lee, N. W. Paton, R. Sakellariou, E. Deelman, A. A. A. Fernandes, and G. Mehta. Adaptive workflow processing and execution in pegasus. In *Proceedings of the 3rd International Conference on Grid and Pervasive Computing Workshops*, 2008.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. Cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds.

[74] 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.

[75] 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.

[76] A. S. Mcgough, A. Afzal, J. Darlington, N. Furmento, A. Mayer, and L. Young. Making the grid predictable through reservations and performance modelling. *Comput. J.*, 48(3):358–368, May 2005.

[77] T. Meinl. Advance reservation of grid resources via real options. In *E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services, 2008 10th IEEE Conference on*, pages 3–10, 2008.

[78] L. Meyer, J. Annis, M. Wilde, M. Mattoso, and I. Foster. Planning spatial workflows to optimize grid performance. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 786–790, New York, NY, USA, 2006. ACM.

[79] 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.

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

[81] 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.

[82] 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¿.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

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

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

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

[91] C. Pinchak, P. Lu, and M. Goldenberg. Practical heterogeneous placeholder scheduling in overlay metacomputers: Early experiences. In *Job Scheduling Strategies for Parallel Processing*, pages 205–228. 2002.

[92] F. Pop, C. Dobre, and V. Cristea. Decentralized dynamic resource allocation for workflows in grid environments. In *Symbolic and Numeric Algorithms for Scientific Computing, 2008. SYNASC '08. 10th International Symposium on*, pages 557–563, 2008.

[93] 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.

[94] 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.

[95] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a fast and light-weight tasK executiON framework. In *IEEE/ACM Conference on Supercomputing*, 2007.

[96] 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.

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

[98] 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.

[99] R. Sakellariou and H. Zhao. A low-cost rescheduling policy for efficient mapping of workflows on grid systems. *Sci. Program.*, 12(4):253–262, Dec. 2004.

[100] 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.

[101] 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.

[102] I. Sfiligoi. glideinWMS—a generic pilot-based workload management system. *Journal of Physics: Conference Series*, 119(6):062044, 2008.

[103] 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.

[104] G. Singh, C. Kesselman, and E. Deelman. Performance impact of resource provisioning on workflows. Technical Report 05-850, University of Southern California, 2005.

[105] 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.

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

[107] 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.

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

[109] 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.

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

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

[112] A. K. M. K. A. Talukder, M. Kirley, and R. Buyya. Multiobjective differential evolution for scheduling workflow applications on global Grids. *Concurrency Computation: Practice and Experience*, 21(13):1742–1756, 2009.

[113] 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.

[114] 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.

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

[116] 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.

[117] 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.

[118] 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.

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

[120] 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.

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

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

[123] 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.

[124] 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.

[125] D. Villegas, A. Antoniou, S. Sadjadi, and A. Iosup. An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 612–619, 2012.

[126] 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.

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

[128] M. Wieczorek, A. Hoheisel, and R. Prodan. Towards a general model of the multi-criteria workflow scheduling on the grid. *Future Generation Computer Systems*, 25(3):237–256, Mar. 2009.

[129] 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.

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

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

[132] J. Yu, G.-Z. Tian, and L. Cheng. Allocating resource in grid workflow based on state prediction. In *Embedded and Ubiquitous Computing, 2008. EUC '08. IEEE/IFIP International Conference on*, volume 2, pages 417–422, 2008.

[133] Z. Yu and W. Shi. An adaptive rescheduling strategy for grid workflow applications. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 2007.

[134] 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.

[135] I. I. Yusuf, H. W. Schmidt, and I. D. Peake. Evaluating recovery aware components for grid reliability. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 277–280, New York, NY, USA, 2009. ACM.

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

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

[138] 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.

[139] 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.

[140] 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.

[141] 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.