Workflow Analysis with Quantitative and Structural Metrics

Weiwei Chen[a,*], Rafael Ferreira da Silva[b], Ewa Deelman[a], Rizos Sakellariou[c]

[a] *University of Southern California, Information Sciences Institute, Marina del Rey, CA, USA*
[b] *University of Lyon, CNRS, Villeurbanne, France*
[c] *University of Manchester, School of Computer Science, Manchester, U.K.*

Abstract

As scientific workflows grow in complexity and importance, the designers of scientific workflows need a deeper and broader understanding of the characteristics and features of workflows and how they behave in order to improve the overall performance. This paper provides a quantitative, structural and overhead aware analysis of workflows of diverse scientific applications, including astronomy, bioinformatics, earthquake science, and gravitational-wave physics. The workflow analysis is based on novel workflow structural metrics that provide substantial information about the characteristics of workflow structures and concretely show how they influence the overall performance. This information includes density, overlap, and sensitivity etc.. This paper further applies these metrics to three popular workflow research scenarios including workflow profiling, task clustering and overhead aware task scheduling. Trace based analysis and simulations show their strong capabilities in revealing hidden information and driving improvements on workflow performance.

Keywords: Scientific workflows, Log analysis, Workflow performance data, Workflow Simulation

1. Introduction

Many computational scientists develop and use complex, data-intensive simulations and analyses [1] that are often structured as scientific workflows, which consist of many computational tasks with complex data dependencies between them. Scientific workflows continue to gain their popularity among many science disciplines, including physics [2], astronomy [3], biology [4, 5], chemistry [6], earthquake science [7] and many more. As scientific workflows grow in complexity and importance, the designers of scientific workflows need a deeper and broader understanding of the characteristics and features of workflows and how they behave in order to drive improvements in algorithms for provisioning resources, scheduling computational jobs, and managing data. Research [8, 9, 10, 11, 12, 13, 14, 15, 16] has been conducted to elaborate the characterization of a wide variety of scientific workflows. In characterizing the execution profile for each workflow, Juve [8] and Callaghan [11] have recently present such metrics as the number of jobs of each type and the I/O, memory, and CPU requirements of each job type from diverse application domains such as astronomy, biology, gravitational physics and earthquake science. Tolosana [10] has proposed 18 metrics to characterize workflow resilience from the perspectives of user, workflow enactor and resource manager. Stampede [12] captures application level logs and resource information, normalizes these to standard representations, and stores these logs in a centralized general purpose schema.

However, there are still challenges that are not yet solved.

First of all, many of these workflow analysis tools use non-quantitative tools. By quantitative, we first mean the metrics are numerical. Yildiz [13] discussed the appropriateness of standard modeling notations to scientific workflow modeling and present the basic scientific workflow structures. Garijo [17] further introduced detection of common scientific workow fragments using templates and execution provenance. However, these workflow structures themselves cannot serve as a criterion to evaluate the specific performance of a workflow since they are not numerical. Second, we mean these metrics must be comparable across different platforms and workflows. For example, the average task runtime [8] and the average task failure rate [11] are highly related to the characteristics of execution platforms (system load and resource availability etc.) and thus they do not serve as a reliable metric to guide the design of new optimization methods or a new workflow management system.

Next, there is a lack of utilizing structural information in these workflow analysis tools. Scientific workflows are typically a graph constructed by data dependencies between computational tasks. A data dependency means there is a data transfer between two tasks (output data for one and input data for the other). Data dependencies between workow tasks play an important role particularly with the emergence of data intensive workflows [11]. Existing metrics [8, 11, 15] (average failure rate, average task runtime, and average resource requirement etc.) treat workflows as no difference to workloads that have no data dependencies between them. Tolosana [10] has touched some basic structural information such as the average

*Corresponding address: USC Information Sciences Institute, 4676 Admiralty Way Ste 1001, Marina del Rey, CA, USA, 90292, Tel: +1 310 448-8408
Email addresses: `weiweich@acm.org` (Weiwei Chen), `rafael.silva@creatis.insa-lyon.fr` (Rafael Ferreira da Silva), `deelman@isi.edu` (Ewa Deelman), `rizos@cs.man.ac.uk` (Rizos Sakellariou)

number of joins in a workflow. But such metrics cannot tell us a global picture of workflow graph such as whether this workflow is dense or sparse, whether this workflow is more sensitive to one particular part of it and whether this workflow is relatively parallel or sequential.

Furthermore, most of these metrics ignore or underestimate the influence of system overheads that play a significant role in the workflow's runtime [18, 19, 20]. In heterogeneous distributed systems, workflows may experience different types of overheads, which are defined as the time of performing miscellaneous work other than executing users computational activities. Since the causes of overheads differ, the overheads have diverse distributions and behaviors. For example, the time to run a post-script that checks the return status of a computation is usually a constant. However, queue delays incurred while tasks are waiting in a batch scheduling systems can vary widely. In our work, we add overhead as a constitution of the extended workflow graph and this approach allows to apply graph manipulations and graph-theoretic algorithms to the corresponding workflow graphs.

Finally, the community has lacked a deep understanding of these data and how they are related to the overall performance improvement of workflows. In this paper, we propose a series of quantitative and structural metrics that are able to reflect the structure of workflow graphs. Generally speaking, we believe such metrics can (i) guide the formulation of a workflow from the perspective of designers and (ii) help workflow management systems (WMS) refine their orchestration for workflow execution (task clustering and task scheduling etc.). Specifically, in this paper we analyze three usage scenarios including workflow profiling, task clustering, and task scheduling that our metrics can be applied to as discussed below.

Workflow Profiling is an effective dynamic analysis approach to investigate complex applications in practice. The realistic characteristics of data-intensive workflows are critical to optimal workflow orchestration and profiling is an effective approach to investigate the behaviors of such complex applications. Traditionally, the common and straightforward approach to address issues of performance optimization is using makespan-centric, DAG scheduling algorithms [21]. While scheduling remains an NP-hard problem, overhead analysis tools can take an important role in giving new insight to solutions that have not been previously considered and that can be further understood. Task clustering [22], job throttling [23], pre-staging [24] and many other solutions have been proposed to reduce the impact of overheads through overlapping overheads and runtimes without requiring runtime improvement of computational jobs. Instead of giving the average task runtime, in Section 5, we propose four metrics to reflect the projection of cumulative overheads on timeline and show how they are related to the overall performance of different workflow optimization methods.

Task clustering techniques [20, 22, 25, 26] have been developed to group ne-grained tasks into coarse-grained tasks so that the number of computational activities is reduced and their computational granularity is increased thereby reducing the system overheads. Grouping tasks without considering these dependencies may lead to data locality problems where output data produced by parent tasks are poorly distributed [27]. Particularly, there is a tradeoff between runtime and data dependency balancing and thus a quantitative measurement of workow characteristics is required to serve as a criterion to select and balance these solutions. To achieve this goal, in Session 6, we propose a series of metrics based on the concept of impact factor and dependency distance that reect the internal structure (in terms of both runtime and dependency) of the workow and use these metrics to guide the task clustering during the runtime.

Task scheduling [21, 28, 29] has long ignored or underestimate the influence of system overheads. When executing these applications on a multi-machine distributed environment, such as the Grid or the Cloud, significant system overheads may exist [18, 19, 30, 31, 32] and the problem of choosing robust schedules becomes more and more important. Traditionally, a carefully crafted schedule is based on deterministic or statistic estimates for the execution time of computational activities that compose a workflow. However, in such an environment, this approach may prove to be grossly inefficient [32], as a result of various unpredictable overheads that may occur at runtime. Thus, to mitigate the impact of uncertain overheads, it is necessary to choose a schedule that guarantees overhead robustness, that is, a schedule that is affected as little as possible by various overhead changes. In Session refsec:sensitivity, we propose a series of structural metrics to evaluate the sensitivity of workflow performance over the overheads and further use them to guide the selection of different scheduling heuristics.

Together, we provide a broader overview of workflow characteristics along with the structural, quantitative and overhead aware information. These characterizations can be widely used by the research community to develop synthetic workflows, benchmarks, and simulations for evaluating workflow management systems.

The next Section gives an overview of the related work, Section 3 presents our workflow and execution environment models, Section 9 details our heuristics and algorithms, Section 7 reports experiments and results, and the paper closes with a discussion and conclusions.

2. Related Work

Some work in the literature has attempted to define and model robustness. In [33], the authors propose a general method to define a metric for robustness. First, a performance metric is chosen. In our case, this performance metric is the overall runtime including overhead duration as we want the execution time of an application to be as stable as possible. Second, one has to identify the parameters that make the performance metric uncertain. In our case, it is the duration of the individual overheads. Third, one needs to find how a modification of these parameters changes the value of the performance metric. In our case, the answer is, as an increase of the overhead generally implies an increase of the overall runtime. A schedule A is said to be more robust than another schedule B if the variation for A is larger than that for B. Following this approach, Canon [34] analyzed the robustness of 20 static DAG scheduling heuristics using a metric for robustness the standard deviation of the makespan over a large number of measurement. Braun et al. [35] evaluated 11 heuristics examined and for the cases studied there, the relatively simple Min-min heuristic performs well in comparison to the other techniques. In comparison, we focus on varying the parameters related to overhead instead of computational tasks.

A plethora of studies on task scheduling [36, 30, 31, 28] have been developed in the distributed and parallel computing domains. Many of these schedulers have been extended to consider both the computational cost and communication cost. A static or statistic estimation of communication cost or data transfer delay [30, 31] has been considered in the scheduling problem. In contrast, we focus on the scheduling overheads that have been ignored or underestimated for long and we demonstrate how their unique timeline patterns influence the overhead robustness.

Workflow patterns [37, 8, 38] are used to capture and abstract the common structure within a workflow and they give insights on designing new workflows and optimization methods. Yu [37] 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 worldwide to demonstrate the comprehensiveness of the taxonomy. Juve [8] 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 [39]. Liu [38] proposed a novel pattern based time-series forecasting strategy which ulitilizes a periodical sampling plan to build representative duration series. Compared to them, we discover a common pattern of intervals existing in system overheads while executing scientific workflows. We also leverage this knowledge to evaluate the overhead robustness of existing heuristics and develop new heuristics.

Overhead analysis [19, 18] is a topic of great interest in the grid community. Stratan [40] evaluates workflow engines including DAGMan/Condor and Karajan/Globus in a real-world grid environment. Sonmez [41] investigated the prediction of the queue delay in grids and assessed the performance and benefit of predicting queue delays based on traces gathered from various resource and production grid environments. Prodan [19] offers a grid workflow overhead classification and a systematic measurement of overheads. Our prior work [18] further investigated the major overheads and their relationship with different optimization techniques. In this paper, we leverage these knowledge to enhance the existing scheduling heuristics and provide insights on designing new algorithms.

The low performance of *fine-grained* tasks is a common problem in widely distributed platforms where the scheduling overhead and queuing times at resources are high, such as Grid and Cloud systems. Several works have addressed the control of task granularity of bag of tasks. For instance, Muthuvelu et al. [42] proposed a clustering algorithm that groups bag of tasks based on their runtime—tasks are grouped up to the resource capacity. Later, they extended their work [43] to determine task granularity based on task file size, CPU time, and resource constraints. Recently, they proposed an online scheduling algorithm [44, 45] that groups tasks based on resource network utilization, user's budget, and application deadline. Ng et al. [46] and Ang et al. [47] 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 [48] 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 are not applicable to scientific workflows, since data dependencies are not considered.

3. Model and Design

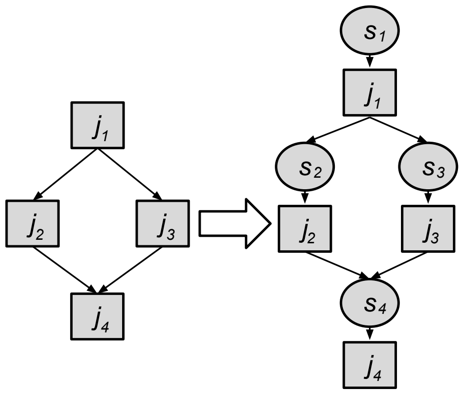

Figure 1: Extending DAG to o-DAG.

A workflow is modeled as a Directed Acyclic Graph (DAG) as shown in 1. Each node in the DAG often represents a workflow job (j), and the edges represent dependencies between the jobs that constrain the order in which the jobs are executed. Dependencies typically represent data-flow dependencies in the application, where the output files produced by one job are used as inputs of another job. Each job is a single execution unit and it may contains one or multiple tasks, which is a program and a set of parameters that need to be executed. Fig. 1 (left) shows an illustration of a DAG composed by four jobs. This model fits several workflow management systems such as Pegasus [39], Askalon [49], and Taverna [50].

Fig. 2 shows a typical workflow execution environment. The submit host prepares a workflow for execution (clustering, mapping, etc.), and worker nodes, at an execution site, execute jobs individually. The main components are introduced below:

Workflow Mapper. generates an executable workflow based on an abstract workflow provided by the user or work- flow composition system. It also restructures the workflow to optimize performance and adds tasks for data management and provenance information generation. In this work, workflow mapper

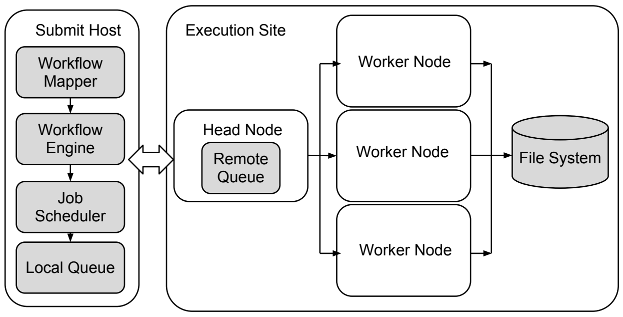

Figure 2: A workflow system model.

is particularly used to merge small tasks together into a job such that system overheads are reduced, which is called Task Clustering. A job is a single execution unit in the workflow execution systems and it may contain one or more tasks.

Workflow Engine. executes jobs defined by the workflow in order of their dependencies. Only jobs that have all their parent jobs completed are submitted to the Job Scheduler. Workflow Engine relies on the resources (compute, storage, and network) defined in the executable workflow to perform the necessary actions. The time period when a job is free (all of its parents have completed successfully) to when it is submitted to the job scheduler is denoted the workflow engine delay. The workflow engine delay is usually configured by users to assure that the entire workflow scheduling and execution system is not overloaded.

Job Wrapper. extracts tasks from clustered jobs and executes them at the worker nodes. The clustering delay is the elapsed time on the extraction process.

Job Scheduler and Local Queue. manage individual workflow jobs and supervise their execution on local and remote resources. The time period when a job is submitted to the job scheduler to when the job starts its execution in a worker node is denoted the queue delay. It reflects both the efficiency of the job scheduler and the resource availability.

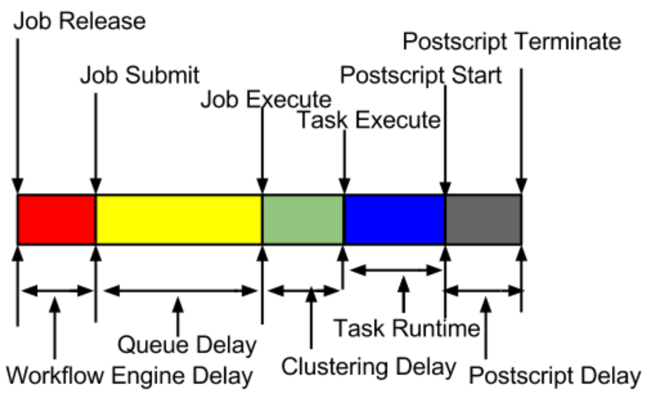

Figure 3: Workflow Events

The execution of a job is comprised of a series of events as shown in Figure 3 and they are defined as:

1. Job Release is defined as the time when the workflow engine identifies that a job is ready to be submitted (when its parents have successfully completed).
2. Job Submit is defined as the time when the workflow engine submits a job to the local queue.
3. Job Execute is defined as the time when the workflow engine sees a job is being executed.
4. Task Execute is defined as the time when the job wrapper sees a task is being executed.
5. Pre/Postscript Start is defined as the time when the workflow engine starts to execute a pre/postscript.
6. Pre/Postscript Terminate is defined as the time when the pre/postscript returns a status code (success or failure).

Figure **??** 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).

As shown in our prior work [**?**], we have classified workflow overheads into three categories as follows.

1. Workflow Engine Delay measures the time between when the last parent job of a job completes and the time when the job gets submitted to the local queue. The completion time of the last parent job means this job is released to the ready queue and is waiting for resources to be assigned to it. The workflow engine delay reflects the efficiency of a workflow engine (i.e., DAGMan [51]).
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 [52]) to execute a job and the availability of resources for the execution of this job.
3. Pre/Postscript Delay is the time taken to execute a lightweight script under some execution systems before/after the execution of a job. For example, prescripts prepare working environment before the execution of a job starts and 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.

The overhead aware DAG model (o-DAG) we use in this work is an extension of the traditional DAG model. System overheads play an important role in workflow execution and constitute a major part of the overall runtime when jobs are poorly mapped to resources. Fig. 1 shows how we augment a DAG to be an o-DAG with the capability to represent system overheads (s) such as workflow engine delay and queue delay.

In summary, an o-DAG representation allows the specification of high level system overhead details, which is more suitable for the study of overhead aware scheduling.

With an o-DAG model, we can explicitly express the process of task clustering. For instance, in Fig. 3, two tasks t1 and t2, without data dependency between them, are merged into a clustered job j1. 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 a overhead denoted the clustering delay c. 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, t1 and t2 in j1 can be executed in sequence or in parallel, if supported. In this paper, we consider sequential executions only. Given a single resource, the overall runtime for the workflow in Fig. 3 (left) is runtime1 = s1 +t1 +s2 +t2 , and the overall runtime for the clustered workflow in Fig. 3 (right) is runtime2 = s1 + c1 + t1 + t2. runtime1 ¿ runtime2 as long as c1 ¡ s2, which is the case of many distributed systems since the clustering delay within an execution node is usually shorter than the scheduling overhead across different execution nodes. Fig 3 shows a typical example of a Horizontal Clustering

Fig. 3: An Example of Task Clustering.

(HC) technique that groups tasks at the same horizontal level. In our work, we define the level of a task as the longest depth from the root task to this task (Depth First Search) because the longest depth controls the final release of this task. In summary, an o-DAG representation allows the specifi- cation of high level system overhead details, which is more suitable than DAG models when clustering tasks.

4. Workflow Profiling Metrics

Profiling is an effective dynamic analysis approach to investigate complex applications in practice. The realistic characteristics of data-intensive workflows are critical to optimal workflow orchestration and profiling is an effective approach to investigate the behaviors of such complex applications. Second, ParaTrac automatically exploits fine-grained data-processes interactions in workflow to help users intuitively and quantitatively investigate realistic execution of data-intensive workflows. We use low-level I/O profiles and informative workflow DAGs to illustrate the vantage of fine-grained profiling by helping users comprehensively understand the application behaviors and refine the scheduling for complex workflows.

As shown in the overhead paper, we define four metrics to calculate overlapped 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. Usage of Overlapping Overheads Metrics: With cumulative overhead metrics, we can tell whether a workflow optimization method fully utilizes the overlap between overheads and computational activities.

4.1. Metrics to Evaluate Cumulative Overheads and Runtimes

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 [53]. Figure 7 shows how to calculate the reverse ranking value (RR) of the same workflow graph in Figure 6.

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

Equation 3 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:

$$RR(G) = \sum RR(j_u) \times \phi_{j_u} \tag{2}$$

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 [24] 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 6 shows the timeline of our example workflow. Job1 is a parent job of Job 2 and Job 3.

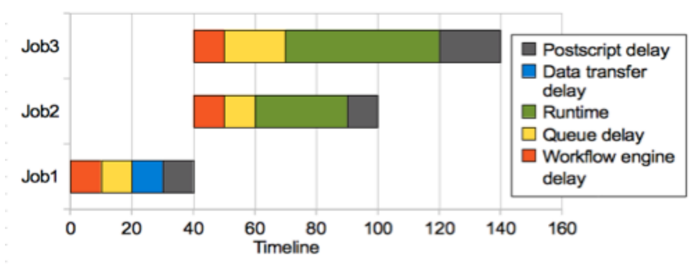

Figure 4: The Timeline of an Example Workflow

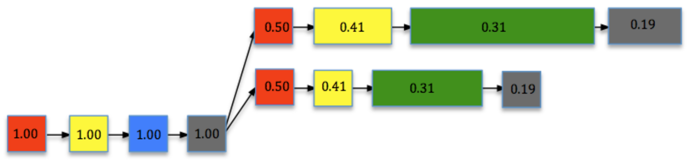

Figure 5: Reverse Ranking

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 shows the percentage of overheads and job runtime over makespan.

Table 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, we can conclude that the sum of Sum is larger than makespan and smaller than makespan×(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.

4.2. *Experiments and Evaluations*

We examined the overhead distributions of a wide range of workflows in our experiments . These workflows were run on distributed platforms including clouds, grids and dedicated clusters. On clouds, virtual machines were provisioned and then the required services (such as file transfer services) were deployed. We examined two clouds: Amazon EC2 [54] and FutureGrid [55]. Amazon EC2 is a commercial, public cloud that is been widely used in distributed computing. We examined the overhead distributions of a widely used astronomy workflow called Montage [56] that is used to construct large image mosaics of the sky. Montage was run on FutureGrid [55]. FutureGrid is a distributed, high-performance testbed that provides scientists with a set of computing resources to develop parallel, grid, and cloud applications.

4.3. *Relationship between Overhead Metrics and Overall Performance*

In this section, we aim to investigate the relationship between the overhead metrics that we proposed and the overall performance of popular workflow restructuring techniques. Among them, task clustering [57] is a technique that increases the computational granularity of tasks by merging small tasks together into a clustered job, reducing the impact of the queue wait time and also the makespan of the workflow. Data or job throttling [23] limits the amount of parallel data transfer to avoid overloading supporting services such as data servers. Throttling is especially useful for unbalanced workflows in which one task might be idle while waiting for data to arrive. The aim of throttling is to appropriately regulate the rate of data transfers between the workflow tasks via data transfer servers by ways of restricting the data connections, data threads or data transfer jobs. Provisioning tools often deploy pilot jobs as placeholders for the execution of application jobs. Since a placeholder can allow multiple application jobs to execute during its lifetime, some job scheduling overheads can be reduced.

How Task Clustering Reduces Overheads

In the following sections, we use a Montage workflow to show how different optimization methods improve overall performance. Many workflows are composed of thousands of fine computational granularity tasks. Task clustering is a technique that increases the computational granularity of tasks by merging small jobs together into a clustered job, reducing the impact of the queue wait time and minimizing the makespan of the workflow. Table 4.2 compares the overheads and runtime of the Montage workflow. We can conclude that with clustering, although the average overheads do not change much, the cumulative overheads decrease greatly due to the decreased number of jobs. With clustering, the makespan has been reduced by

53.3% by reducing the number of all jobs from 3461 to 104 in this example. Figure 4.5 shows the percentage of workflow overheads and runtime. The percentage is calculated by the cumulative overhead (PJ, or EP) divided by the makespan of workflows. With clustering, the portion of runtime is increased significantly. Figure 4.6 profiles the number of active jobs during execution and it also shows that with clustering the resource utilization is improved significantly.

How Job Throttling Reduces Overheads

Data or job throttling [13] limits the amount of parallel data transfer to avoid overloading supporting services such as data servers. Throttling is especially useful for unbalanced workflows in which one task might be idle while waiting for data to arrive. The aim of throttling is to appropriately regulate the rate of data transfers between the workflow tasks via data transfer servers by ways of restricting the data connections, data threads or data transfer jobs. Especially on cloud platforms, I/O requests need to go through more layers than a physical cluster; and thereby workflows may suffer a higher overhead from data servers.

In our experiments, the data transfer service is deployed on a virtual machine that is similar to a worker node. In this section, we evaluate a simple static throttling strategy where the Condor scheduler limits the number of concurrent jobs to be run and thereby restricts the number of parallel I/O requests. There are 32 resources available and we evaluate the cases with throttling parameters that are equal to 24, 16 and 12 in Table 4.3. In the case of 24, the resources are better utilized but the data server is heavily loaded. In the case of 12, the resources are underutilized even though the data server has more capabilities. In this experiment, both PJ and EP reflect the variation trend of overheads and makespan better than Sum.

Figure 4.7 shows the percentage of workflow overheads and runtime. Figure 4.8 profiles the number of active jobs during execution. Montage is an unbalanced workflow because the three major types of jobs (mProjectPP, mDiffFit, and mBackground) impose a heavy load on the data server while the other jobs in the workflow do not. Figure 4.8 shows that with throttling the maximum number of active jobs is restricted. With limited throttling (reducing threshold from 24 to 16), the data transfer requests are distributed in the timeline more evenly and, as a result, their overhead is reduced. However, with over throttling (reducing threshold from 16 to 12), resources are not fully utilized and thus the makespan is increased.

How Pre-staging Reduces Overheads

Scientific workflows often consume and produce a large amount of data during execution. Data pre-staging [14] transfers input data before the computational activities are started or even before the workflow is mapped onto resources. Data placement policies distribute data in advance by placing data sets where they may be requested or by replicating data sets to improve runtime performance. In our experiments, because data is already pre-staged, the implementation of the stage-in job is to create a soft link to the data from the workflows working directory, making it available to the workflow jobs. Table 4.4 and Figure 4.9 show the cumulative overheads and runtime of the Montage workflows running with and without pre-staging. Looking at the rows for PJ in Table 4.4, we can conclude that pre-staging improves the overall runtime by reducing the data transfer delay. For the case without pre-staging the EP for data transfer delay is zero because it overlaps with the workflow engine delay of another job. Therefore, in this experiment, PJ reflects the variation trend of the makespan more consistently.

How Provisioning Reduces Overheads

Many of the scientific applications presented here consist of a large number of short-duration tasks whose runtimes are greatly influenced by overheads present in distributed environments. Most of these environments have an execution mode based on batch scheduling where jobs are held in a queue until resources become available to execute them. Such a best-effort model normally imposes heavy overheads in scheduling and queuing. For example, Condor-G [23] uses Globus GRAM [37] to submit jobs to remote clusters. The Globus Toolkit normally has a significant overhead compared to running Condor directly as an intra domain resource and job management system. Provisioning tools often deploy pilot jobs as placeholders for the execution of application jobs. Since a placeholder can allow multiple application jobs to execute during its lifetime, some job scheduling overheads can be reduced. In our experiments, we compared the performance of Condor-G (without provisioning) and Corral (with provisioning).

Table 4.5 and Figure 4.10 show the percentage of workflow overheads and runtime. The percentage is calculated by the cumulative overhead (Sum, PJ, or EP) divided by the makespan of workflows. Comparing Sum, PJ and EP, we can conclude that the overheads with provisioning have been reduced significantly because the local scheduler has direct control over the resources without going through Globus.

5. Workflow Profiling Metrics

Profiling is an effective dynamic analysis approach to investigate complex applications in practice. The realistic characteristics of data-intensive workflows are critical to optimal workflow orchestration and profiling is an effective approach to investigate the behaviors of such complex applications. Second, ParaTrac automatically exploits fine-grained data-processes interactions in workflow to help users intuitively and quantitatively investigate realistic execution of data-intensive workflows. We use low-level I/O profiles and informative workflow DAGs to illustrate the vantage of fine-grained profiling by helping users comprehensively understand the application behaviors and refine the scheduling for complex workflows.

As shown in the overhead paper, we define four metrics to calculate overlapped 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. Usage of Overlapping Overheads Metrics: With cumulative overhead metrics, we can tell whether a workflow optimization method fully utilizes the overlap between overheads and computational activities.

5.1. Metrics to Evaluate Cumulative Overheads and Runtimes

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 [53]. Figure 7 shows how to calculate the reverse ranking value (RR) of the same workflow graph in Figure 6.

$$RR(j_u) = d + (1-d) \times \sum_{j_v \in Child(j_u)} \frac{RR(j_v)}{L(j_v)} \tag{3}$$

Equation 3 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:

$$RR(G) = \sum RR(j_u) \times \phi_{j_u} \tag{4}$$

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 [24] 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 6 shows the timeline of our example workflow. Job1 is a parent job of Job 2 and Job 3.

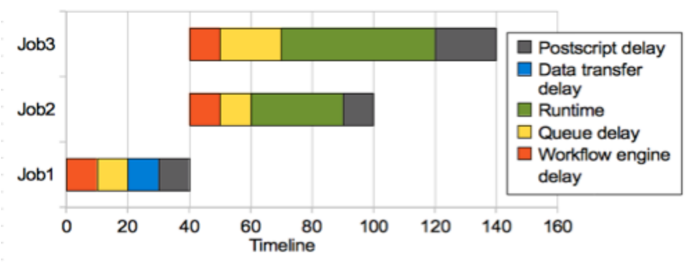

Figure 6: The Timeline of an Example Workflow

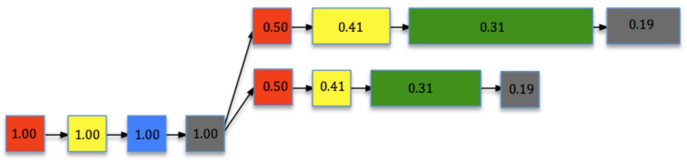

Figure 7: Reverse Ranking

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 shows the percentage of overheads and job runtime over makespan.

Table 2: 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, we can conclude that the sum of Sum is larger than makespan and smaller than makespan×(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.

5.2. Experiments and Evaluations

We examined the overhead distributions of a wide range of workflows in our experiments . These workflows were run on distributed platforms including clouds, grids and dedicated clusters. On clouds, virtual machines were provisioned and then the required services (such as file transfer services) were deployed. We examined two clouds: Amazon EC2 [54] and FutureGrid [55]. Amazon EC2 is a commercial, public cloud that is been widely used in distributed computing. We examined the overhead distributions of a widely used astronomy workflow called Montage [56] that is used to construct large image mosaics of the sky. Montage was run on FutureGrid [55]. FutureGrid is a distributed, high-performance testbed that provides scientists with a set of computing resources to develop parallel, grid, and cloud applications.

5.3. Relationship between Overhead Metrics and Overall Performance

In this section, we aim to investigate the relationship between the overhead metrics that we proposed and the overall performance of popular workflow restructuring techniques. Among them, task clustering [57] is a technique that increases the computational granularity of tasks by merging small tasks together into a clustered job, reducing the impact of the queue wait time and also the makespan of the workflow. Data or job throttling [23] limits the amount of parallel data transfer to avoid overloading supporting services such as data servers. Throttling is especially useful for unbalanced workflows in which one task might be idle while waiting for data to arrive. The aim of throttling is to appropriately regulate the rate of data transfers between the workflow tasks via data transfer servers by ways of restricting the data connections, data threads or data transfer jobs. Provisioning tools often deploy pilot jobs as placeholders for the execution of application jobs. Since a placeholder can allow multiple application jobs to execute during its lifetime, some job scheduling overheads can be reduced.

How Task Clustering Reduces Overheads

In the following sections, we use a Montage workflow to show how different optimization methods improve overall performance. Many workflows are composed of thousands of fine computational granularity tasks. Task clustering is a technique that increases the computational granularity of tasks by merging small jobs together into a clustered job, reducing the impact of the queue wait time and minimizing the makespan of the workflow. Table 4.2 compares the overheads and runtime of the Montage workflow. We can conclude that with clustering, although the average overheads do not change much, the cumulative overheads decrease greatly due to the decreased number of jobs. With clustering, the makespan has been reduced by 53.3% by reducing the number of all jobs from 3461 to 104 in this example. Figure 4.5 shows the percentage of workflow overheads and runtime. The percentage is calculated by the cumulative overhead (*PJ*, or *EP*) divided by the makespan of workflows. With clustering, the portion of runtime is increased significantly. Figure 4.6 profiles the number of active jobs during execution and it also shows that with clustering the resource utilization is improved significantly.

How Job Throttling Reduces Overheads

Data or job throttling [13] limits the amount of parallel data transfer to avoid overloading supporting services such as data servers. Throttling is especially useful for unbalanced workflows in which one task might be idle while waiting for data to arrive. The aim of throttling is to appropriately regulate the rate of data transfers between the workflow tasks via data transfer servers by ways of restricting the data connections, data threads or data transfer jobs. Especially on cloud platforms, I/O requests need to go through more layers than a physical cluster; and thereby workflows may suffer a higher overhead from data servers.

In our experiments, the data transfer service is deployed on a virtual machine that is similar to a worker node. In this section, we evaluate a simple static throttling strategy where the Condor scheduler limits the number of concurrent jobs to be run and thereby restricts the number of parallel I/O requests. There are 32 resources available and we evaluate the cases with throttling parameters that are equal to 24, 16 and 12 in Table 4.3. In the case of 24, the resources are better utilized but the data server is heavily loaded. In the case of 12, the resources are underutilized even though the data server has more capabilities. In this experiment, both *PJ* and *EP* reflect the variation trend of overheads and makespan better than Sum.

Figure 4.7 shows the percentage of workflow overheads and runtime. Figure 4.8 profiles the number of active jobs during execution. Montage is an unbalanced workflow because the three major types of jobs (mProjectPP, mDiffFit, and mBackground) impose a heavy load on the data server while the other jobs in the workflow do not. Figure 4.8 shows that with throttling the maximum number of active jobs is restricted. With limited throttling (reducing threshold from 24 to 16), the data transfer requests are distributed in the timeline more evenly and, as a result, their overhead is reduced. However, with over throttling (reducing threshold from 16 to 12), resources are not fully utilized and thus the makespan is increased.

How Pre-staging Reduces Overheads

Scientific workflows often consume and produce a large amount of data during execution. Data pre-staging [14] transfers input data before the computational activities are started or even before the workflow is mapped onto resources. Data placement policies distribute data in advance by placing data sets where they may be requested or by replicating data sets to improve runtime performance. In our experiments, because data is already pre-staged, the implementation of the stage-in job is to create a soft link to the data from the workflows working directory, making it available to the workflow jobs. Table 4.4 and Figure 4.9 show the cumulative overheads and runtime of the Montage workflows running with and without pre-staging. Looking at the rows for PJ in Table 4.4, we can conclude that pre-staging improves the overall runtime by reducing the data transfer delay. For the case without pre-staging the EP for data transfer delay is zero because it overlaps with the workflow engine delay of another job. Therefore, in this experiment, PJ reflects the variation trend of the makespan more consistently.

How Provisioning Reduces Overheads

Many of the scientific applications presented here consist of a large number of short-duration tasks whose runtimes are greatly influenced by overheads present in distributed environments. Most of these environments have an execution mode based on batch scheduling where jobs are held in a queue until resources become available to execute them. Such a best-effort model normally imposes heavy overheads in scheduling and queuing. For example, Condor-G [23] uses Globus GRAM [37] to submit jobs to remote clusters. The Globus Toolkit normally has a significant overhead compared to running Condor directly as an intra domain resource and job management system. Provisioning tools often deploy pilot jobs as placeholders for the execution of application jobs. Since a placeholder can allow multiple application jobs to execute during its lifetime, some job scheduling overheads can be reduced. In our experiments, we compared the performance of Condor-G (without provisioning) and Corral (with provisioning).

Table 4.5 and Figure 4.10 show the percentage of workflow overheads and runtime. The percentage is calculated by the cumulative overhead (Sum, PJ, or EP) divided by the makespan of workflows. Comparing Sum, PJ and EP, we can conclude that the overheads with provisioning have been reduced significantly because the local scheduler has direct control over the resources without going through Globus.

6. Imbalance Metrics

6.1. Introduction

Although the majority of the tasks within these applications are often relatively short running (from a few seconds to a few minutes), in aggregate they represent a significant amount of computation and data [58]. When executing these applications on a multi-machine distributed environment, such as the Grid or the Cloud, significant system overheads may exist and may adversely slowdown the application performance [**?**]. To minimize the impact of such overheads, task clustering techniques [42, 43, 44, 45, 46, 47, 48, 59] have been developed to group *fine-grained* tasks into *coarse-grained* tasks so that the number of computational activities is reduced and their computational granularity is increased thereby reducing the (mostly scheduling related) system overheads [**?**].

However, there are several challenges that have not yet been addressed.

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 [60]. A common technique to handle load imbalance is overdecomposition [61].

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 that is offered by the hardware.

Data dependencies between workflow tasks play an important role when clustering tasks within a level. A data dependency means that there is a data transfer between two tasks (output data for one and input data for the other). Grouping tasks without considering these dependencies may lead to data locality problems where output data produced by parent tasks are poorly distributed. Thus, data transfer times and failures probability increase. Therefore, we claim that data dependencies of subsequent tasks should be considered.

In this work, we generalize these two challenges (Runtime Imbalance and Dependency Imbalance) to the generalized load balance problem. We introduce a series of balancing methods to address these challenges as our first contribution. A performance evaluation study shows that the methods can significantly reduce the imbalance problem. However, there is a tradeoff between runtime and data dependency balancing. For instance, balancing runtime may aggravate the Dependency Imbalance problem, and vice versa. A quantitative measurement of workflow characteristics is required to serve as a criterion to select and balance these solutions. To achieve this goal, we propose a series of metrics that reflect the internal structure (in terms of task runtimes and dependencies) of the workflow as our second contribution.

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 [62] that represents the clustering problem as a global optimization problem and aims to minimize the overall workflow execution time. However, the complexity of solving such an optimization problem does not scale well since most methods use genetic algorithms. The second one is a bottom-up approach [42, 48] that only examines free tasks to be merged and optimizes the clustering results locally. In contrast, our work extends these approaches to consider the neighboring tasks including siblings, parents, and children because such a family of tasks has strong connections between them.

Our third contribution is an analysis of the quantitative metrics and balancing methods. These metrics characterize the workflow imbalance problem. A balancing method, or a combination of those, is selected through the comparison of the relative values of these metrics.

6.2. 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 is meaningful to reduce the standard deviation of job runtime. Fig. 8 shows an example of four independent tasks t_1, t_2, t_3 and t_4 where task runtime of t_1 and t_2 is half of that of t_3 and t_4. 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., $HRV > 0$ (Fig. 8-top). In contrast, a balanced clustering strategy should try its best to evenly distribute task runtime among jobs as shown in Fig. 8 (bottom). Generally speaking, a smaller *HRV* means that the runtime of tasks at the same horizontal level is more evenly distributed and therefore it is less necessary to balance the runtime distribution. However, runtime variance is not able to describe how regular is the structure of the dependencies between the tasks.

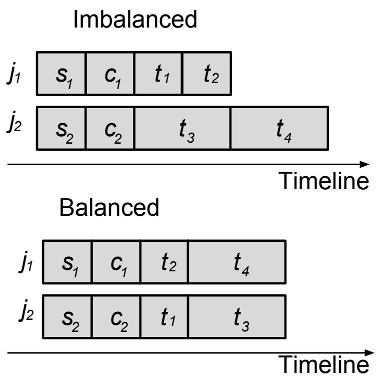

Figure 8: An Example of Runtime Variance.

Dependency Imbalance means that the task clustering at one horizontal level forces the tasks at the next level (or even subsequent levels) to have severe data locality problem and thus loss of parallelism. For example, in Fig. 9, we show a two-level workflow composed of four tasks in the first level and two in the second. Merging t_1 with t_2 and t_3 with t_4 (imbalanced workflow in Fig. 9) 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_3 are completed, and so can t_6. To measure and quantitatively demonstrate the Dependency Imbalance of a workflow, we propose two metrics: (*i*) Impact Factor Variance, and (*ii*) Distance Variance.

We define the **Impact Factor Variance** (*IFV*) of tasks as the standard deviation of their impact factor. The intuition behind the Impact Factor is that we aim to capture the similarity of tasks/jobs in a graph by measuring their relative impact factor or importance to the entire graph. Intuitively speaking, tasks with similar impact factors should be merged together compared to tasks with different impact factors. Also, if all the tasks have

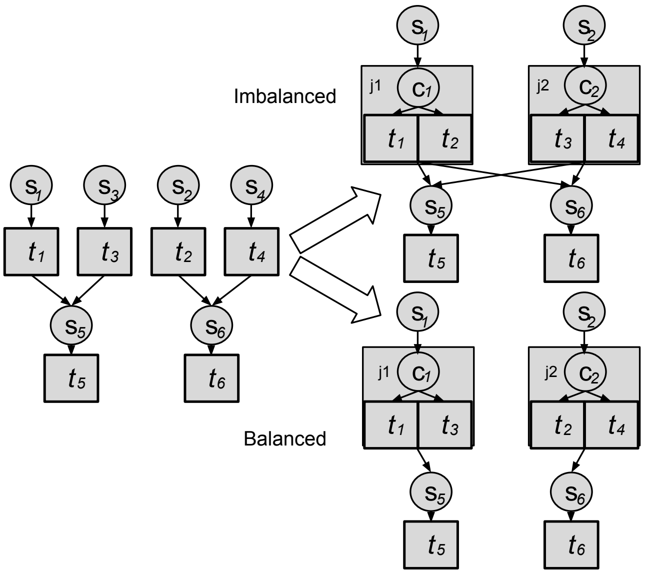

Figure 9: An Example of Dependency Variance.

similar impact factors, the workflow structure tends to be more 'even' or 'regular'. The **Impact Factor** (*IF*) of a task t_u is defined as follows:

$$IF(t_u) = \sum_{t_v \in Child(t_u)} \frac{IF(t_v)}{L(t_v)} \tag{5}$$

where $Child(t_u)$ denotes the set of child tasks of t_u, and $L(t_v)$ the number of parent tasks of t_v. For simplicity, we assume the *IF* of a workflow exit task (e.g. t_5 in Fig. 9) as 1.0. For instance, consider the two workflows presented in Fig. 10. *IF* for t_1, t_2, t_3, and t_4 are computed as follows:

$$\begin{aligned} IF(t_7) = 1.0, IF(t_6) = IF(t_5) = IF(t_7)/2 = 0.5 \\ IF(t_1) = IF(t_2) = IF(t_5)/2 = 0.25 \\ IF(t_3) = IF(t_4) = IF(t_6)/2 = 0.25 \end{aligned}$$

Thus, IFV(t_1, t_2, t_3, t_4) = 0. In contrast, *IF* for t'_1, t'_2, t'_3, and t'_4 are:

$$\begin{aligned} IF(t'_7) = 1.0, IF(t'_6) = IF(t'_5) = IF(t'_1) = IF(t'_7)/2 = 0.5 \\ IF(t'_2) = IF(t'_3) = IF(t'_4) = IF(t'_6)/3 = 0.17 \end{aligned}$$

Therefore, the *IFV* value for t'_1, t'_2, t'_3, t'_4 is 0.17, which means it is less regular than the workflow in Fig. 10 (left). In this work, we use **HIFV** (Horizontal IFV) to indicate the *IFV* of tasks at the same horizontal level. The time complexity of calculating all the *IF* of a workflow with n tasks is $O(n)$.

Distance Variance (*DV*) describes how 'closely' 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

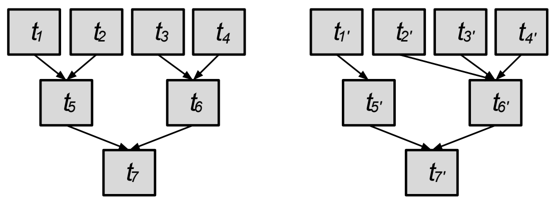

Figure 10: Example of workflows with different data dependencies.

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 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 shows the distance matrices of tasks from the first level for both workflows of Fig. 10 (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. Intuitively speaking, a smaller HDV means the tasks at the same horizontal level are more equally 'distant' to each other and thus the workflow structure tends to be more 'evenly' and 'regular'.

In conclusion, Runtime Variance and Dependency Variance offer a quantitative and comparable tool to measure and evaluate the internal structure of a workflow.

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: Distance matrices of tasks from the first level of workflows in Fig. 10.

6.3. Balanced Clustering Methods

In this subsection, we introduce our balanced clustering methods used to improve the runtime balance and dependency balance in task clustering. We first introduce the basic runtime-based clustering method and then two other balancing methods that address the Dependency Imbalance problem. We use the metrics presented in the previous subsection to evaluate a given workflow to decide which balancing method(s) is(are) more appropriate.

Algorithm 1 shows the pseudocode of our balanced clustering algorithm that uses a combination of these balancing methods and metrics. The maximum number of clustered jobs (size of CL) is equal to the number of available resources multiplied by a *clustering factor*. We compare the performance of using different *clustering factor* in Section 5.

We examine tasks in a level-by-level approach starting from the level with the largest width (number of tasks at the same level, `line 2`). The intuition behind this breadth favored approach is that we believe it should improve the performance

Algorithm 1 Balanced Clustering algorithm

```
Require: W: workflow; CL: list of clustered jobs; C: the required size of CL;
Ensure: The job runtime of CL are as even as possible
 1: procedure CLUSTERING(W, D, C)
 2:     Sort W in decreasing order of the size of each level
 3:     for level <the depth of W do
 4:         TL ← GETTASKSATLEVEL(w, level)        ▹ Partition W based on depth
 5:         CL ← MERGE(TL, C)                     ▹ Form a list of clustered jobs
 6:         W ← W − TL + CL                       ▹ Merge dependencies as well
 7:     end for
 8: end procedure
 9: procedure MERGE(TL, C)
10:     Sort TL in decreasing order of task runtime
11:     for t in TL do
12:         J ← GETCANDIDATEJOB(CL, t)            ▹ Get a candidate task
13:         J ← J + t                             ▹ Merge it with the clustered job
14:     end for
15:     return CL
16: end procedure
17: procedure GETCANDIDATEJOB(CL, t)
18:     Selects a job based on balanced clustering methods
19: end procedure
```

most. Then, we determine which type of imbalance problem a workflow experiences based on the balanced clustering metrics presented previously (HRV, $HIFV$, and HDV), and accordingly, we select a combination of balancing methods. GETCANDIDATEJOB selects a job (`line 12`) from a list of potential candidate jobs (CL) to be merged with the targeting task (t). Below we introduce the three balancing methods proposed in this work.

Horizontal Runtime Balancing (HRB) aims to evenly distribute task runtime among jobs. Tasks with the longest runtime are added to the job with the shortest runtime. This greedy method is used to address the imbalance problem caused by runtime variance at the same horizontal level. Fig. 11 shows how HRB works in an example of four jobs with different job runtime (assuming the height of a job is its runtime). For the given task (t_0), HRB sorts the potential jobs (j_1, j_2, j_3, and j_4) based on their runtime and selects the shortest job (in this case j_1 or j_2).

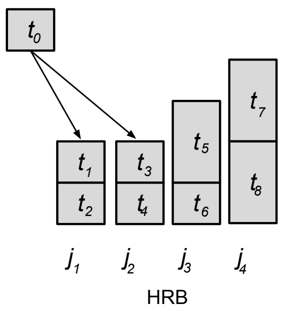

Figure 11: An example of HRB.

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.

In HRB, candidate jobs are sorted by their runtime, while in

HIFB jobs are first sorted based on their similarity of IF, then on runtime. For example, in Fig. 12, assuming 0.2, 0.2, 0.1, and 0.1 IF values of j_1, j_2, j_3, and j_4 respectively, HIFB selects a list of candidate jobs with the same IF value, i.e. j_3 and j_4. Then, HRB is performed to select the shortest job (j_3).

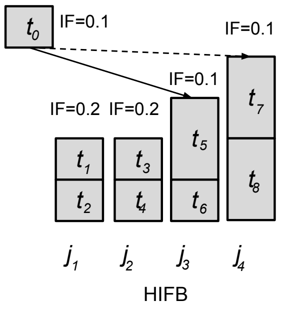

Figure 12: An example of HIFB.

Similarly, in HDB jobs are sorted based on the distance between them and the targeted task t_0, then on their runtimes. For instance, in Fig. 13, assuming 2, 4, 4, and 2 the distances to j_1, j_2, j_3, and j_4 respectively, HDB selects a list of candidate jobs with the minimal distance (j_1 and j_4). Then, HRB is performed to select the shortest job (j_1).

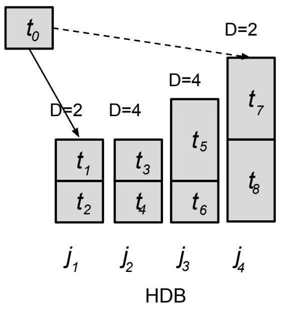

Figure 13: An example of HDB.

In conclusion, these balancing methods have different preference on the selection of a candidate job to be merged with the targeting task. HIFB tends to group tasks that share similar position/importance to the workflow structure. HDB tends to group tasks that are closed to each other to reduce data transfers. Table 4 summarizes the imbalance metrics and balancing methods presented in this work.

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

Table 4: Imbalance metrics and balancing methods.

7. Experiment and Evaluation

The experiments presented hereafter evaluate the performance of our balancing methods in comparison with an existing and effective task clustering strategy named Horizontal Clustering (HC) [63], which is widely used by workflow management systems such as Pegasus.

7.1. Experiment Conditions

We extended the WorkflowSim [**?**] simulator with the balanced clustering methods and imbalance metrics to simulate a distributed environment where we could evaluate the performance of our methods when varying the average data size and task runtime. 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 [64] and FutureGrid [55]. Each machine has 512MB of memory and the capacity to process 1,000 million instructions per second. Task scheduling is data-aware, i.e. tasks are scheduled to resources which have the most input data available.

Two workflows are used in the experiments: LIGO [58] inspiral analysis, and Epigenomics [65]. Both workflows are generated and varied using the WorkflowGenerator[1]. LIGO is composed by 400 tasks and its workflow structure is presented in Fig. 23 (top); Epigenomics has about 500 tasks and is structured as showed in Fig. 23 (bottom). Runtime (average and task runtime distribution) and overhead (workflow engine delay, queue delay, and network bandwidth) information were collected from real traces production environments [18, 8], then used as input parameters for the simulations.

Three sets of experiments are conducted. Experiment 1 aims at determining an appropriate *clustering factor* such that both the workflow runtime performance and the reliability over the dynamic system variation are improved. We randomly select 20% from LIGO workflow tasks and increase their task runtime by a factor of *Ratio* to simulate the system variation in a production environment.

Experiment 2 evaluates the reliability and the influence of average data size in our balancing methods, since data has becoming more and more intensive in scientific workflows [8]. In this experiment set, there is no runtime variance ($HRV = 0$). The original average data size (both input and output data) of the LIGO workflow is about 5MB, and of the Epigenomics workflows is about 45MB. We increase the average data size up to 5GB.

Experiment 3 evaluates the influence of the runtime variation (HRV) in our balancing methods. We assume a normal distribution to vary task runtimes based on average and standard deviation. In this experiment set, there is no variation on the data size.

Simulation results present a confidence level of 95%. We define the performance gain over HC (μ) as the performance of the balancing methods related to the performance of Horizontal Clustering (HC). Thus, for values of $\mu > 0$ our balancing

[1]https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator

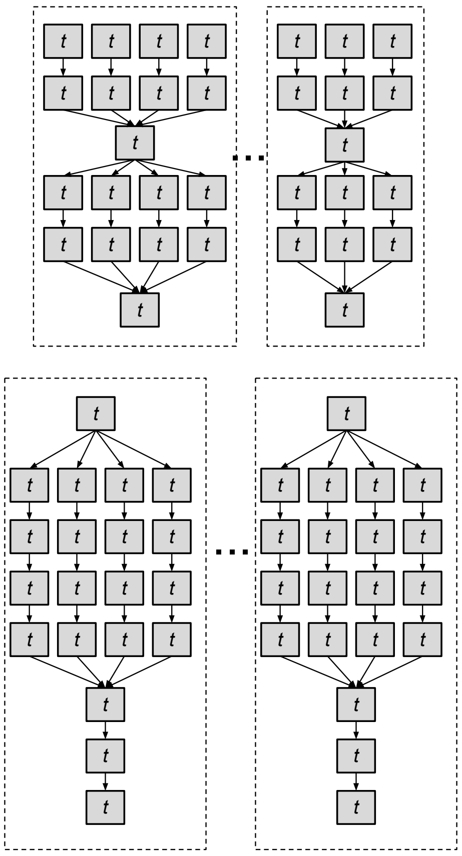

Figure 14: A simplified visualization of the LIGO Inspiral workflow (top) and Epigenomics workflow (bottom).

methods perform better than the HC method. Otherwise, the balancing methods perform poorer.

7.2. Results and Discussion

As we have mentioned in Subsection 6.3 , the maximum number of clustered jobs is equal to the number of available resources multiplied by a *clustering factor*. Experiment 1: Fig. 15 shows the speedup of Horizontal Clustering (HC) for different *Ratio* and *clustering factors*. We use the speedup of HC in overall runtime compared to the original overall runtime without clustering. The speedup decreases with the increase of the *clustering factor*. However, a smaller *clustering factor* performs worse when the *Ratio* is high. For simplicity, we use *clustering factor* = 2 in the experiments conducted in this work.

Experiment 2: Fig. 16 (top) shows the performance gain over HC μ of the balancing methods compared to the HC method for the LIGO workflow. HIFB and HDB significantly increase the performance of the workflow execution. Both strategies capture the structural and runtime information, reducing data transfers between tasks, while HRB focuses on runtime distribution, which in this case is none. Fig. 16 (bottom) shows the performance of the balancing methods for the Epigenomics workflow. When increasing the average data size, only HDB demonstrates significantly improvement related to HC. Investigating

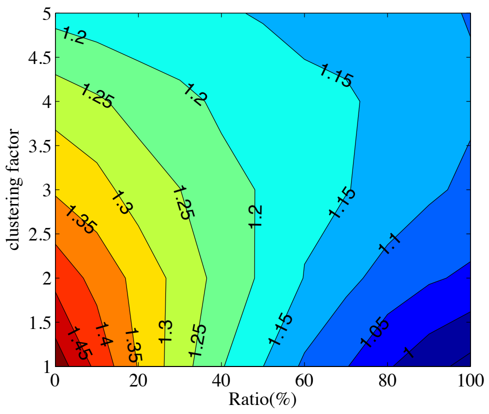

Figure 15: Experiment 1: Speedup of Horizontal Clustering (HC).

the structure of the Epigenomics workflow (Fig. 23-bottom), we can see that all tasks at the same horizontal level share the same IFs ($HIFV = 0$), because each branch (surrounded by dash lines) happen to have the same amount of pipelines. Thus, HIFB has no performance improvement when compared to HC. However, for LIGO (Fig. 23-top), $HIFV \neq 0$, thus HIFB improves the workflow runtime performance. HDB captures the strong connections between tasks (data dependencies) and HIFB captures the weak connections (similarity in terms of structure). In both workflows, HDV is not zero thus HDB performs better than HC.

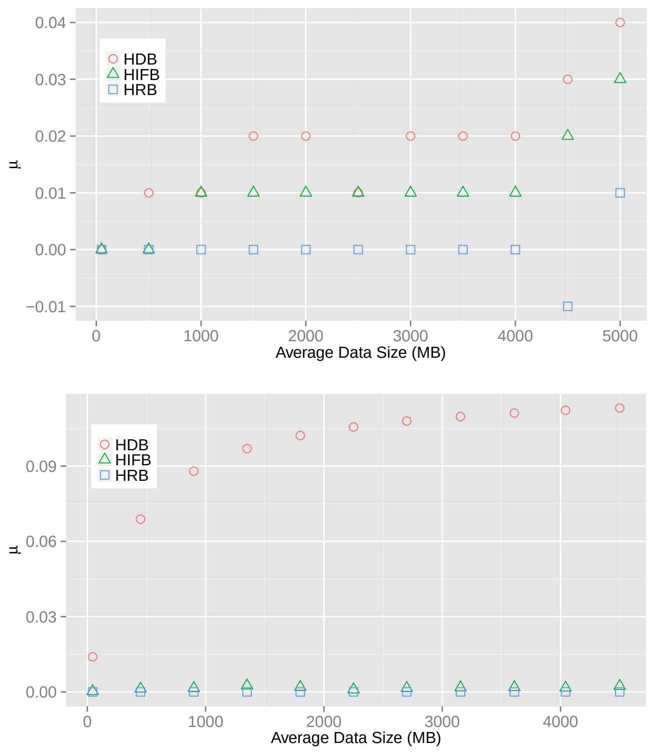

Figure 16: Experiment 2: Performance of the LIGO workflow (top) and the Epigenomics workflow (bottom).

Experiment 3: Fig. 17 shows the performance gain μ when varying task runtimes for the LIGO workflow. As expected, when *HRV* increases HRB over performs HC. However, HDB and HIFB demonstrate poor performance because they merge tasks based on data dependencies first, and then, they balance the runtime distribution. For high values of *HRV*, we just simply need to use HRB. Otherwise, we can use either HDB or HIFB while in some cases HIFB fails to capture the structural information.

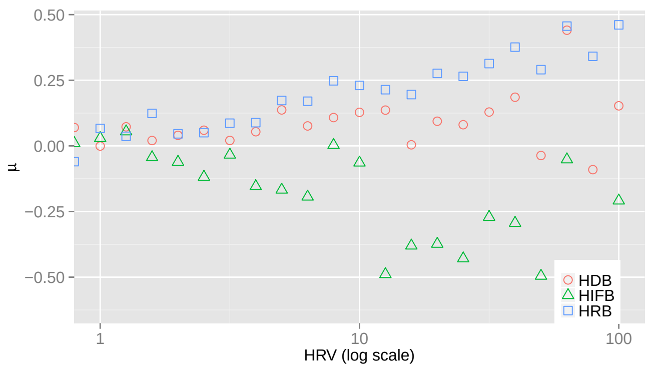

Figure 17: Experiment 3: Influence of *HRV* (LIGO workflow).

8. Sensitivity Metrics

8.1. Introduction

Many computational scientists develop and use large-scale, loosely-coupled applications that are often structured as scientific workflows, which consist of many computational tasks with data dependencies between them. When executing these applications on a multi-machine distributed environment, such as the Grid or the Cloud, significant system overheads may exist [18**?** , 30, 31**?**] and the problem of choosing robust schedules becomes more and more important. Traditionally, a carefully crafted schedule is based on deterministic or statistic estimates for the execution time of computational activities that compose a workflow. However, in such an environment, this approach may prove to be grossly inefficient [**?**], as a result of various unpredictable overheads that may occur at runtime. Thus, to mitigate the impact of uncertain overheads, it is necessary to choose a schedule that guarantees overhead robustness, that is, a schedule that is affected as little as possible by various overhead changes.

There are several ways to achieve overhead robustness. A first approach is to integrate the overhead estimation into the job scheduling problem. A static or statistic estimation of communication cost or data transfer delay [30, 31] has been considered in the scheduling problem. Once we have the deterministic or statistic information of overheads, we can treat the system overhead as computational activities and the goal is to minimize the overall runtime including overhead duration. However, this approach only applies to the estimation of data transfer delay since the highly unpredictable variability and variety of other overheads make it a challenging work and not efficient in practice. Our prior work [**?**] has shown the variation of overheads may be comparable to the job runtime and thus makes it unrealistic in a real environment.

A significant amount of work [**?** 36, 30, 31] in the literature has focused on proposing algorithms that are aware of the dynamic changes of runtime environments. Task rescheduling [66**? ?**] is a typical approach that dynamically allocates tasks to an idle processor in order to take into account information that has been made available during the execution. Specifically, resource load [30] can be used to estimate the variance. However, rescheduling a task is costly as it implies some extra communication and synchronization costs. Relevant studies [66] indicate that it is important to have a static schedule with good properties before the start of the execution. Therefore, even if a dynamic strategy is used, a good initial placement would reduce the possibility of making a bad decision.

Another approach is to overestimate the execution time of individual jobs. Delay scheduling [**?**] waits for a small amount of time, letting other MapReduce jobs launch tasks instead and this method can achieve a better tradeoff of locality and fairness. However, this method only applies to workload scheduling and particularly MapReduce jobs since the duration of them is short and thus it is not difficult to estimate the scheduling delay. Also, this results in a waste of resources as it induces a lot of idle time during the execution, if the overhead is shorter than the estimation. Second, overheads do not simply work as an attachment to the job runtime and it involves a lot more complicated patterns such as periodicity [**?**].

In this paper, we first present our work on evaluating the overhead robustness of scheduling heuristics and we indicate a list of heuristics that are overhead robust even without an estimate of the overhead duration. Second, since the estimate of overhead duration is difficult, we develop new heuristics that leverage the pattern information of workflow overheads, which represents a new approach to design overhead robust algorithms. To the best of our knowledge, so far, no study has systematically tried to evaluate the scheduling heuristics with respect to the overhead robustness.

8.2. Overhead Patterns

In this section, we introduce the common overhead patterns in workflow execution. In scientific workflow systems, time related functionalities such as workflow scheduling normally requires effective forecasting of activity patterns. In this work, we mainly focus on the overhead pattern that refers to a representative time series of overhead activities that occurs repeatedly and regularly in workflow execution. An overhead pattern is composed of ordered overhead activities obtained from scientific workflow system logs or other forms of historical data. Pattern discovery [38] usually starts from a periodical sampling plan to build representative duration series (Job Release, etc.) and then conducts time-series segmentation to discover the pattern sets and predicts the activity duration intervals with pattern matching results.

The motivation for pattern based analysis comes from the observation that for those duration-series segments where the number of concurrent activity instances is similar, these activity durations reveal comparable statistical features. Due to the dynamic nature of underlying resources, pattern based forecasting of overheads can improve the effectiveness of scheduling heuristics.

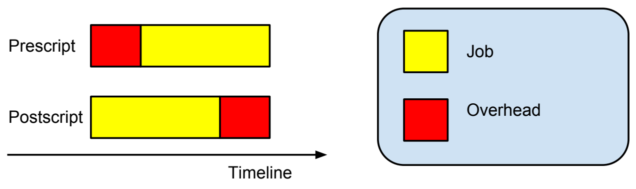

Figure 18: Adherence Pattern

Most of the work [30, 31] view the overhead duration as an attachment to runtime in workflow timeline. For example, the Pre/Post-script Delay is usually constant, which we call it Adherence Pattern as shown in Fig. 18. Scheduling algorithms can just add the delay to the job runtime without significant change to the algorithms. For this pattern, we have shown in our prior work [**?**] that it does not have much influence on the overhead robustness. However, we observe that the Workflow Engine Delay and the Queue Delay increases periodically and steadily. For example, Fig. 19 shows the Gantt chart of part of a real trace[1]. The Workflow Engine Delay (red) of the first 16 jobs is 5 seconds and then it increases to 10 seconds. We call this common overhead pattern the Incremental and Periodical Pattern (IPP). We observe the Queue Delay also increases periodically but the period is interrupted by the resource availability and thus it has a more complicated IPP. In the rest of this paper, we focus on the IPP of the Workflow Engine Delay and we will cover the Queue Delay in our future work. The reason why IPP prevalently exists is that many workflow management components are queue based systems. They repeatedly check their queues to find whether there are idle jobs, if yes they will process and submit these jobs, otherwise it will wait for a interval and continue. In Fig. 20 we abstract the IPP from the trace, which shows a repeatedly increase by a interval. We define throughput of a workflow management component as the maximum number of allowed jobs in queue. For example, the interval and the throughput of the Workflow Engine in Fig. 19 are around 5 seconds and 16 respectively.

These overhead patterns reappear frequently during a duration series and they represent unique behavior of the workflow management components. After we define and discover these typical patterns, the intervals and throughputs of these patterns can be estimated and statistically captured from historical traces.

9. Overhead Robust Heuristics

In this section, we introduce the overhead robustness of four pairs of scheduling heuristics. Heuristics are widely used in workflow scheduling since the scheduling problem is a NP-hard problem and the time complicity of a global optimization of the overall runtime is not affordable. Usually, heuristics utilize

[1]Details: http://www.isi.edu/˜wchen/fgrid/run

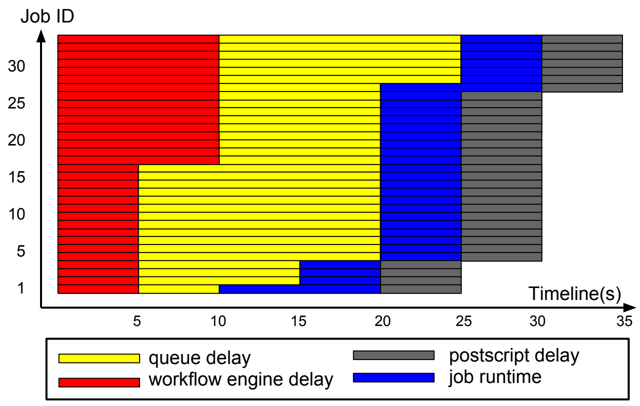

Figure 19: Workflow Execution Gantt Chart of a Real Trace

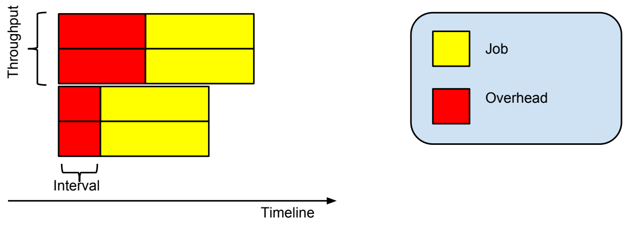

Figure 20: Incremental and Periodical Pattern

unique features of workflows or resources to guide the mapping of jobs to resources. For example, both of the MINMIN [28] algorithm and the MAXMIN [35] algorithm utilize the runtime of a job as the feature. However, they conduct the mapping in an opposite way, that is, MINMIN chooses the job with the shortest runtime while MAXMIN chooses the job with the longest runtime. Intuitively speaking, there must be either of them (we call it overhead robust) that performs better than the other one (we call it overhead unrobust) since they operates oppositely. Inspired by these coupling heuristics, we use a relative overhead robustness approach to evaluate these scheduling heuristics. For a pair of heuristics that use the same feature, we define the relative overhead robustness as the performance gain of the overhead robust heuristic against the other one. The more performance gain we have, the more significant this feature has on the overhead robustness. Also, it is not fair to compare scheduling heuristics that use different features because they may have vastly different performance even without overheads. Furthermore, in practice, we can use scheduling heuristics with different features at the same time but not those with the same features. Below we introduce the four pairs of scheduling heuristics including one that we propose.

MINMIN and *MAXMIN*: The MINMIN heuristic begins with a set of all unmapped jobs. Then, the set of minimum completion times for each job, M namely, is found. Next, the job with the overall minimum completion time from M is selected and assigned to the corresponding resource (hence the name MINMIN). Last, the newly mapped job is removed from the unmapped jobs, and the process repeats until all jobs are mapped. The MAXMIN heuristic is very similar to MINMIN. The MAXMIN heuristic also begins with the set of all unmapped jobs. Then, the set of minimum completion times, M, is found. Next, the job with the overall maximum completion time from M is selected and assigned to the corresponding resource (hence the name MAXMIN). Last, the newly mapped job is removed from the unmapped jobs, and the process repeats until all jobs are mapped. Intuitively speaking, we believe MAXMIN has better overhead robustness than MINMIN. As shown in Fig. 21, assuming we have two jobs and the interval of the overhead is 1. MAXMIN releases a job with longer runtime first, which can overlap with the increment of the overheads when MAXMIN releases the other job. However, the performance gain depends on the values of overhead interval, job runtime, overhead duration and the resource availability.

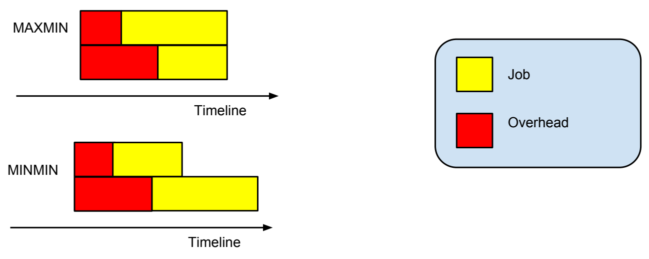

Figure 21: MAXMIN vs. MINMIN

Breadth First and *Depth First*: Namely, the Breadth First (BFS) algorithm iterates the jobs at the same workflow level (or depth within a workflow directed acyclic graph) first while the Depth First (DFS) algorithm iterates the jobs at the deepest workflow level first. Intuitively speaking, we believe BFS performs better than DFS in terms of overhead robustness since BFS releases more jobs at the same workflow level for most scientific workflows as shown in Fig. 23. Therefore, with enough jobs in queue, BFS can fully utilize the resources without wasting this overhead interval.

Fertile First and *Unfertile First*: The Fertile First (FFS) algorithm sorts all the unmapped jobs based on the number of children jobs that they have and assigns the jobs with most children jobs first. The Unfertile First (UFFS) algorithm works in the opposite way and assigns the jobs with lest children jobs first. Similar to the case of BFS and DFS, we believe FFS should perform better than UFFS since FFS releases more jobs at the next level compared to UFFS and thus FFS can fully utilize the available resources.

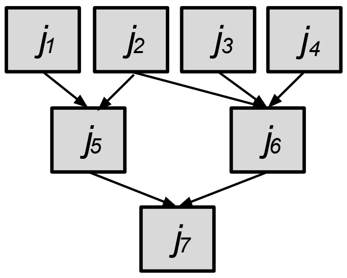

Figure 22: Impact Factor

Important First and *Unimportant First*: Here we propose a pair of Impact Factor based scheduling heuristics. The Important First (IFS) algorithm sorts all the unmapped jobs based on their *Impact Factor* (IF) and assigns the jobs with the largest IF first while the Unimportant First (UIFS) algorithm assigns the jobs with the smallest IF first. The **Impact Factor** (IF) of a job j_u is defined as follows:

$$IF(j_u) = \sum_{j_v \in Child(j_u)} \frac{IF(j_v)}{L(j_v)} \tag{6}$$

where $Child(j_u)$ denotes the set of child jobs of j_u, and $L(j_v)$ the number of parent jobs of j_v. For simplicity, we assume the IF of a workflow exit job (e.g. j_7 in Fig. 22) as 1.0. For instance, consider the workflow in Fig. 22. IF for j_1, j_2, j_3, and j_4 are computed as follows:

$$IF(j_7) = 1.0, IF(j_6) = IF(j_5) = IF(j_7)/2 = 0.5$$
$$IF(j_1) = IF(j_5)/2 = 0.25$$
$$IF(j_2) = IF(j_5)/2 + IF(j_6)/3 = 0.42$$
$$IF(j_3) = IF(j_4) = IF(j_6)/3 = 0.17$$

Thus, IFS algorithm should schedule j_2 first while UIFS algorithm should schedule j_3 or j_4 first. The intuition of Impact Factor is that we aim to measure the relative importance of a job to the entire graph. Intuitively speaking, tasks with larger impact factors should have more impacts on the remaining jobs compared to tasks with smaller impact factors. For example, a bottleneck usually has a larger IF since it controls the release of more jobs. Similar to the case of FFS and UFFS, we believe IFS has a better overhead robustness than UIFS since IFS releases more jobs at the next few levels compared to UFFS.

Table 5 summaries the heuristics and their potential overhead robustness.

Table 5: Scheduling Heuristics

Heuristics	Overhead Robust	Overhead Unrobust
Experiment 1	MAXMIN	MINMIN
Experiment 2	BFS	DFS
Experiment 3	FFS	UFFS
Experiment 4	IFS	UIFS

10. Experiment and Evaluation

The experiments presented hereafter evaluate the performance of the four pairs of heuristics mentioned ahead, which are widely used by workflow management systems.

10.1. Experiment Conditions

We extended the WorkflowSim [**?**] simulator with the overhead model to simulate a distributed environment where we could evaluate the overhead robustness of scheduling algorithms when varying the average overheads and throughput. As

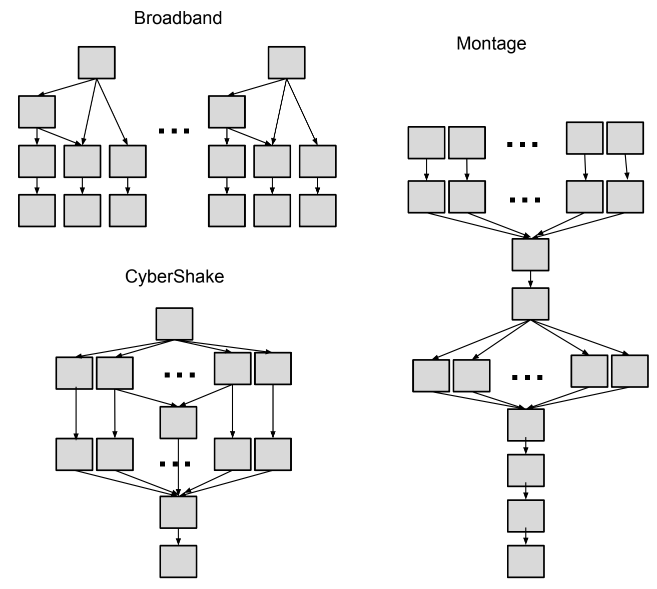

Figure 23: A simplified visualization of the Broadband workflow, the Montage workflow and the CyberShake workflow.

an initial attempt, we focus on the workflow engine delay (d) in this paper. The simulated computing platform is composed by 20 single core virtual machines (worker nodes), which is the quota per user of some typical distributed environments such as Amazon EC2 [64] and FutureGrid [55]. Each machine has 512MB of memory and the capacity to process 1,000 million instructions per second.

Three workflows are used in the experiments: Broadband [?] is an application that enables researchers to combine long-period deterministic seismograms with high-frequency stochastic seismograms. Montage [3] is an astronomy application used to construct large image mosaics of the sky. CyberShake [67] is a seismology application that calculates Probabilistic Seismic Hazard curves for geographic sites in the Southern California region. All workflows are generated and varied using the WorkflowGenerator[1]. Each workflow instance is composed by around 100 tasks and its workflow structure is presented in Fig. 23. Runtime (average and task runtime distribution) and overhead (workflow engine delay and queue delay) information were collected from real traces production environments [18, 8], then used as input parameters for the simulations.

Four sets of experiments are conducted. Experiment 1 evaluates the relative robustness of MAXMIN and MINMIN by comparing the performance gain of the overhead robust heuristic (MAXMIN in this experiment) over the overhead unrobust heuristic (MINMIN), while varying the average interval of workflow engine (d) and throughput of workflow engine (t). Table 5 shows the heuristics compared in these experiments. Simulation results present a confidence level of 95%. Thus, for values of *Performance Gain* > 0, the overhead robust heuristics perform better than the respective overhead unrobust heuristics. Otherwise, the overhead robust heuristics perform poorer.

In these experiment sets, we vary the average throughput of workflow engine from 1 to 20 and we show the results of $t = 1, 5, 15, 20$. The original throughput of workflow engine in these traces is 5 and a throughput that is larger than 20 does not influence the performance much since we have only 20 worker nodes in our experiments. We also vary the average interval of workflow engine from 0 to 100 seconds, which represents a typical range of workflow engine delay as shown in [18] and this range is able to show the difference of overhead robustness in these heuristics.

[1]https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator

10.2. Results and Discussion

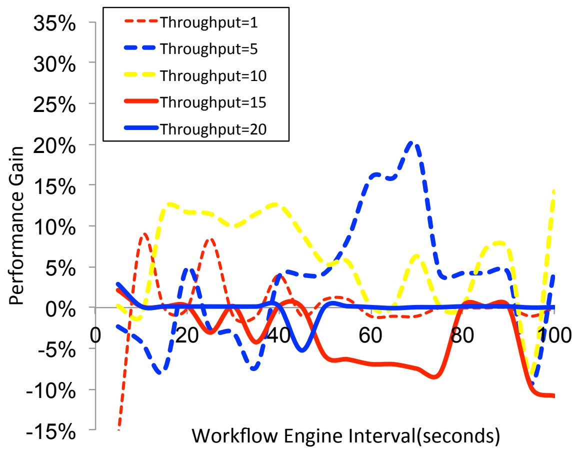

Figure 24: Broadband

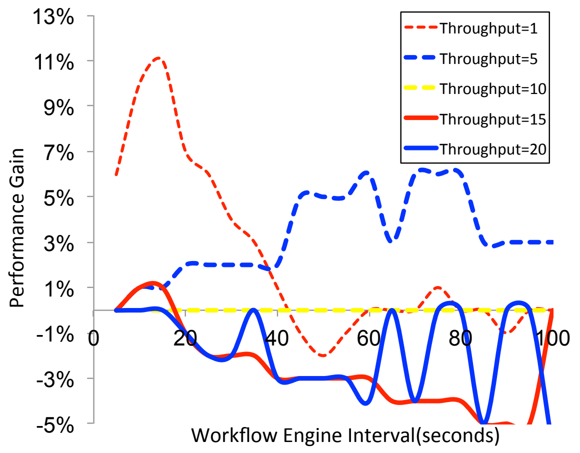

Figure 25: CyberShake

Experiment 1: Fig. 24, 25, 26 show the *Performance Gain* of MAXMIN over MINMIN for the three workflows. We expected to see most *Performance Gain* $>$ 0 if MAXMIN is a overhead robust heuristic compared to MINMIN. However, except for the Montage workflow, the *Performance Gain* is not significant for all of parameter settings, which concludes that MAXMIN is not globally overwhelming MINMIN in terms of overhead robustness. The reason we believe is that the *Performance Gain* in this comparison highly depends on the ratio of

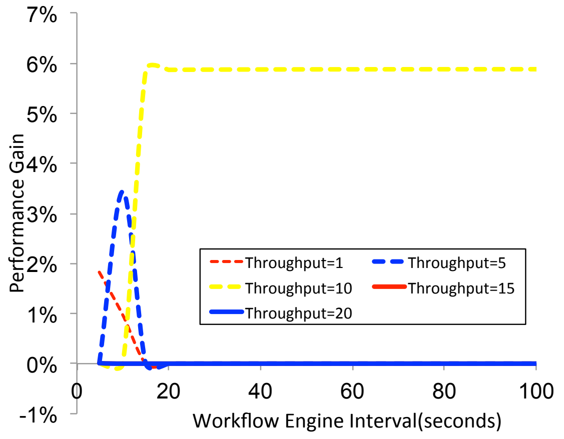

Figure 26: Montage

job runtime, workflow engine delay, number of resources and throughput of workflow engine.

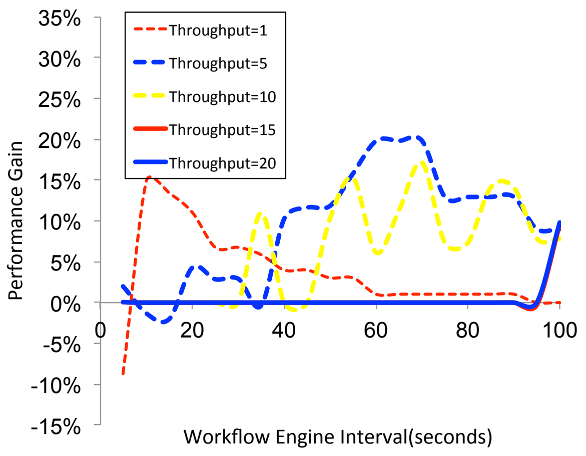

Figure 27: Broadband

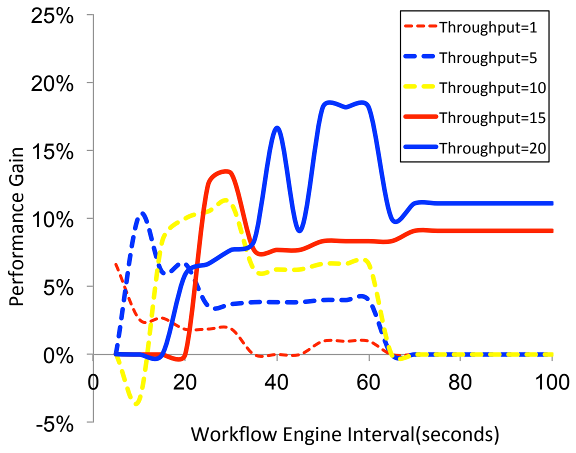

Figure 28: CyberShake

Experiment 2: Fig. 27, 28, 29 show the *Performance Gain* of BFS over DFS for the three workflows. We observe that most *Performance Gain* > 0 and thus BFS performs better than DFS in terms of overhead robustness. What is more, Fig 28

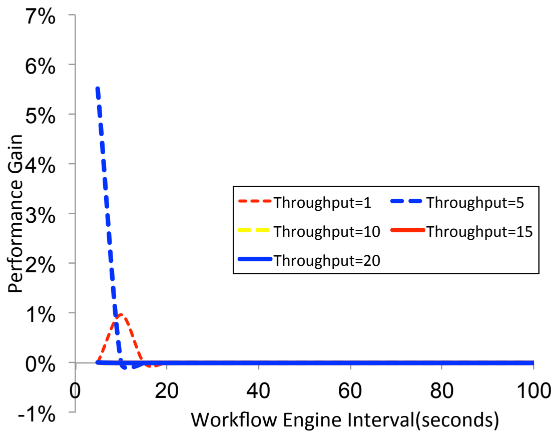

Figure 29: Montage

shows with the increase of average throughput, the *Performance Gain* is more significant. We can also see that when the average throughput is high, the *Performance Gain* increases with the average interval of workflow engine. This suggests us in a real environment with a large overhead, we should use BFS instead of DFS.

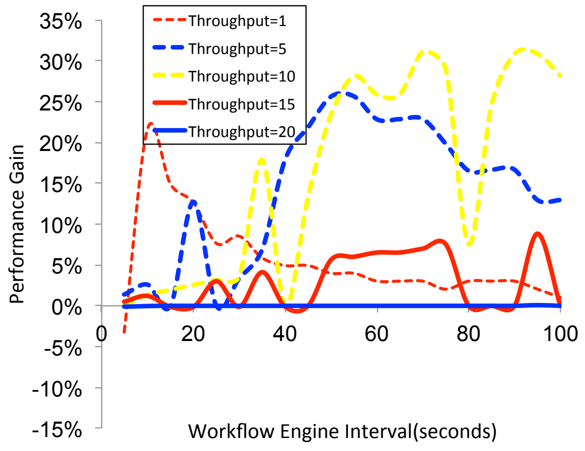

Figure 30: Broadband

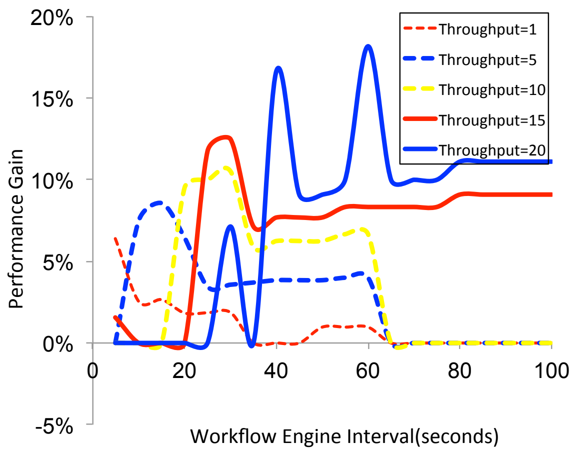

Figure 31: CyberShake

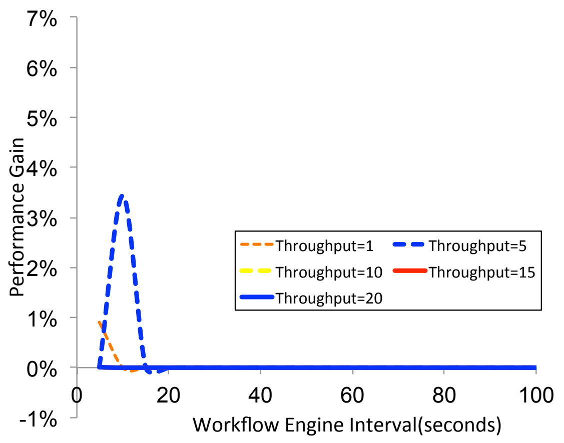

Figure 32: Montage

Experiment 3: Fig. 30, 31, 32 show the *Performance Gain* of FFS over UFFS for the three workflows. We observe that most *Performance Gain* > 0 and thus FFS performs better than UFFS in terms of overhead robustness, which is similar to Experiment 2. Comparing Fig. 27 and Fig. 30 we can see the *Performance Gain* of FFS over UFFS is more significant (30%) than that of BFS over DFS (20%).

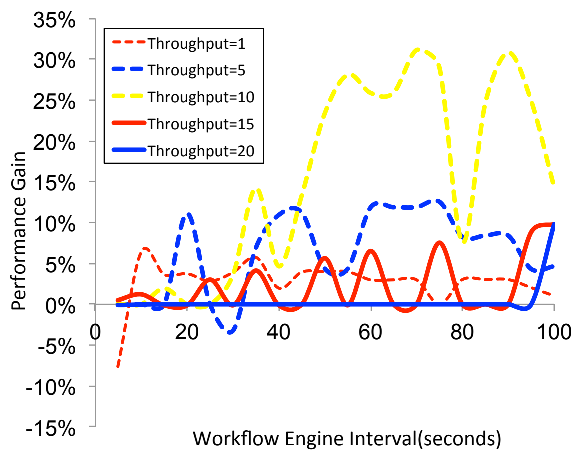

Figure 33: Broadband

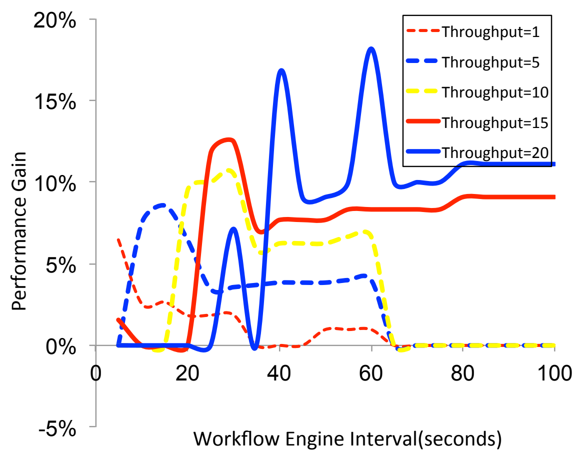

Figure 34: CyberShake

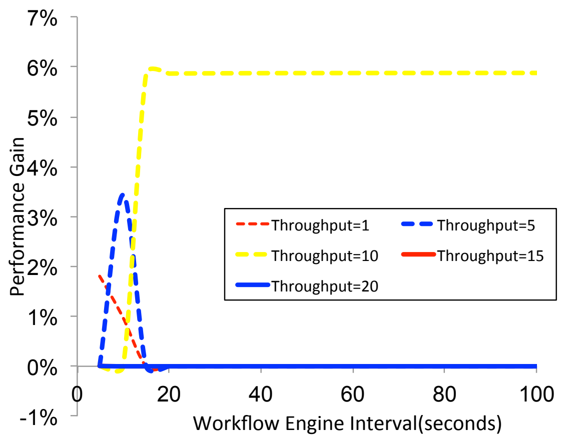

Figure 35: Montage

Experiment 4: Fig. 33, 34, 35 show the *Performance Gain* of IFS over UIFS for the three workflows. We observe that most *Performance Gain* > 0 and thus IFS performs better than UIFS in terms of overhead robustness, which is similar to Experiment 3. The reason is that for these workflows, IF based heuristics can produce similar schedule as the heuristics based on the number of children. Most of the workflows used in this paper is not irregular enough and thus we are not able to show the difference of IF based heuristics and the heuristics based on the number of children.

11. Conclusion and Future Work

A more sophisticated alternative, however, is to conduct a statistical analysis based on Principal Component Analysis (PCA). This method determines in a more precise way the existing correlation between the collected metrics and their impact on resilience. However, undertaking an automated analysis using PCA requires prior experimental data to be available. For this reason, in or- der to better enable a comparison, the aggregation of all the metrics into a single value is proposed in this section. Aggregations of multiple variables are used in many well- known disciplines such as statistical or economic measure- ment. However, combining the metrics we have proposed above provides three main challenges. First, the metrics are quantitative and can be measured, however on their own they are hard to compare against each other and to determine whether one metric is more significant than another. Second, having multiple metrics makes any comparison diffi- cult. Third, resilience increases in direct proportion to some metrics, whereas it reduces with respect to others. Identify- ing to what degree each measured metric impacts resilience (directly or inversely) must also be qualified through previ- ously recorded experimental data or by an expert.

References

[1] T. Hey, S. Tansley, K. Tolle, The Fourth Paradigm: Data-Intensive Scientific Discovery, Microsoft Research, 2009.

[2] E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams, 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.

[3] R. Sakellariou, H. Zhao, E. Deelman, Mapping Workflows on Grid Resources: Experiments with the Montage Workflow, in: F. Desprez, V. Getov, T. Priol, R. Yahyapour (Eds.), Grids P2P and Services Computing, 2010, pp. 119–132.

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

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

[6] M. Wieczorek, R. Prodan, T. Fahringer, Scheduling of scientific workflows in the askalon grid environment, in: ACM SIGMOD Record, Vol. 34, 2005, pp. 56–62.

[7] 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, E. Field, SCEC CyberShake WorkflowsAutomating probabilistic seismic hazard analysis calculations, in: I. Taylor, E. Deelman, D. Gannon, M. Shields (Eds.), Workflows for e-Science, Springer, 2007, pp. 143–163.

[8] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, K. Vahi, Characterizing and profiling scientific workflows, in: Future Generation Computer Systems, 2013, p. 682692.

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

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

[11] S. Callaghan, P. Maechling, P. Small, K. Milner, G. Juve, T. Jordan, E. Deelman, G. Mehta, K. Vahi, D. Gunter, K. Beattie, 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) (2011) 274–285.

[12] D. Gunter, E. Deelman, T. Samak, C. H. Brooks, M. Goode, G. Juve, G. Mehta, P. Moraes, F. Silva, M. Swany, et al., Online workflow management and performance analysis with stampede, in: Proceedings of the 7th International Conference on Network and Services Management, International Federation for Information Processing, 2011, pp. 152–161.

[13] U. Yildiz, A. Guabtni, A. H. Ngu, Towards scientific workflow patterns, in: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science, ACM, 2009, p. 13.

[14] L. Ramakrishnan, D. Gannon, A survey of distributed workflow characteristics and resource requirements, Tech. Rep. TR671, Indiana University (2008).

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

[16] Y. Gu, Q. Wu, Performance analysis and optimization of distributed workflows in heterogeneous network environments, in: IEEE Transactions on Computers, IEEE, 2013.

[17] D. Garijo, O. Corcho, Y. Gil, Detecting common scientific workflow fragments using templates and execution provenance, in: Proceedings of the seventh international conference on Knowledge capture, ACM, 2013, pp. 33–40.

[18] W. Chen, E. Deelman, Workflow overhead analysis and optimizations, in: Proceedings of the 6th workshop on Workflows in support of large-scale science, WORKS '11, 2011, pp. 11–20.

[19] R. Prodan, T. Fabringer, Overhead analysis of scientific workflows in grid environments, in: IEEE Transactions in Parallel and Distributed System, Vol. 19, 2008.

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

[21] Y. Caniou, G. Charrier, 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, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2011, pp. 15–24.

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

[23] S.-M. Park, M. Humphrey, Data throttling for data-intensive workflows, in: IEEE Intl. Symposium on Parallel and Distributed Processing, 2008.

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

[25] K. Maheshwari, A. Espinosa, D. S. Katz, M. Wilde, Z. Zhang, I. Foster, S. Callaghan, 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, ACM, New York, NY, USA, 2012, pp. 3–12.

[26] R. Ferreira da Silva, T. Glatard, F. Desprez, On-line, non-clairvoyant optimization of workflow activity granularity on grids, in: F. Wolf, B. Mohr, D. Mey (Eds.), Euro-Par 2013 Parallel Processing, Vol. 8097 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 255–266.

[27] W. Chen, R. F. da Silva, E. Deelman, R. Sakellariou, Balanced task clustering in scientific workflows, in: The 9th IEEE International Conference on e-Science, IEEE, 2013.

[28] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, 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.

[29] H. Casanova, A. Legrand, D. Zagorodnov, F. Berman, Heuristics for scheduling parameter sweep applications in grid environments, in: 9th Heterogeneous Computing Workshop, 2000.

[30] F. Dong, J. Luo, A. Song, J. Jin, Resource load based stochastic dags scheduling mechanism for grid environment, in: High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on, 2010, pp. 197–204.

[31] L. Yang, J. M. Schopf, I. Foster, Conservative scheduling: using predicted variance to improve scheduling decisions in dynamic environments, 2003, pp. 1–16.

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

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

[34] L.-C. Canon, E. Jeannot, Comparative evaluation of the robustness of dag scheduling heuristics, in: Grid Computing, Springer, 2008.

[35] 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, 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) (2001) 810–837.

[36] H. Chetto, M. Silly, T. Bouchentouf, Dynamic scheduling of real-time tasks under precedence constraints, Real-Time Systems 2.3.

[37] J. Yu, R. Buyya, A taxonomy of workflow management systems for grid computing, Journal of Grid Computing 3.

[38] X. Liu, J. Chen, K. Liu, Y. Yang, Forecasting duration intervals of scientific workflow activities based on time-series patterns, in: eScience, 2008. eScience '08. IEEE Fourth International Conference on, 2008, pp. 23–30.

[39] 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, D. S. Katz, Pegasus: A framework for mapping complex scientific workflows onto distributed systems, Scientific Programming 13 (3) (2005) 219–237.

[40] C. Stratan, A. Iosup, D. H. Epema, A performance study of grid workflow engines, in: Grid Computing, 2008 9th IEEE/ACM International Conference on, IEEE, 2008, pp. 25–32.

[41] O. Sonmez, H. Mohamed, 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, IEEE, 2006, pp. 79–79.

[42] N. Muthuvelu, J. Liu, N. L. Soe, S. Venugopal, A. Sulistio, 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 - Volume 44, 2005, pp. 41–48.

[43] N. Muthuvelu, I. Chai, C. Eswaran, An adaptive and parameterized job grouping algorithm for scheduling grid jobs, in: Advanced Communication Technology, 2008. ICACT 2008. 10th International Conference on, Vol. 2, 2008, pp. 975 –980.

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

[45] N. Muthuvelu, C. Vecchiolab, I. Chaia, E. Chikkannana, R. Buyyab, Task granularity policies for deploying bag-of-task applications on global grids, FGCS 29 (1) (2012) 170 – 181.

[46] W. K. Ng, T. F. Ang, T. C. Ling, C. S. Liew, Scheduling framework for bandwidth-aware job grouping-based scheduling in grid computing, Malaysian Journal of Computer Science 19.

[47] T. Ang, W. Ng, T. Ling, L. Por, C. Lieu, A bandwidth-aware job grouping-based scheduling on grid environment, Information Technology Journal 8 (2009) 372–377.

[48] Q. Liu, Y. Liao, Grouping-based fine-grained job scheduling in grid computing, in: ETCS '09, Vol. 1, 2009, pp. 556 –559.

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

[50] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, C. Wroe, Taverna: lessons in creating a workflow environment for the life sciences: Research articles, Concurr. Comput. : Pract. Exper. 18 (10) (2006) 1067–1100.

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

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

[53] L. Page, S. Brin, R. M. andTerry Winograd, The pagerank citation ranking: bringing order to the web, 1999.

[54] Amazon.com, Inc., Elastic Compute Cloud (EC2), `http://aws.amazon.com/ec2`.

[55] FutureGrid, `http://futuregrid.org/`.

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

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

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

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

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

[61] J. Lifflander, S. Krishnamoorthy, 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, HPDC '12, New York, NY, USA, 2012, pp. 137–148.

[62] W. Chen, E. Deelman, Integration of workflow partitioning and resource provisioning, in: Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on, 2012, pp. 764–768.

[63] G. Singh, M.-H. Su, K. Vahi, E. Deelman, B. Berriman, J. Good, D. S. Katz, G. Mehta, Workflow task clustering for best effort systems with pegasus, in: Mardi Gras'08, 2008, pp. 9:1–9:8.

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

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

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

[67] S. Callaghan, P. Maechling, E. Deelman, K. Vahi, G. Mehta, G. Juve, K. Milner, R. Graves, E. Field, D. Okaya, T. Jordan, Reducing Time-to-Solution Using Distributed High-Throughput Mega-Workflows: Experiences from SCEC CyberShake, in: Proceedings of the 4th IEEE International Conference on e-Science (e-Science '08), 2008.