Overhead Robustness of Scheduling Heuristics in Scientific Workflows

Weiwei Chen*, Rafael Ferreira da Silva‡, Ewa Deelman*, Rizos Sakellariou§
*University of Southern California, Information Sciences Institute, Marina Del Rey, CA, USA
{wchen,deelman}@isi.edu
‡University of Lyon, CNRS, INSERM, CREATIS, Villeurbanne, France
rafael.silva@creatis.insa-lyon.fr
§University of Manchester, School of Computer Science, Manchester, U.K.
rizos@cs.man.ac.uk

Abstract—**Workflow overheads play an important role in workflow performance particularly the scheduling performance. However, existing task scheduling strategies only provide a coarse-grained approach that relies on an over-simplified workflow model that ignores or underestimate the impact of overheads. In this work, we first identify the major overhead patterns in a general workflow management system. Next, we evaluate the performance of existing scheduling heuristics while varying overhead parameters (duration, etc.). Finally, we propose overhead robust heuristics that leverage overhead patterns even without a overhead duration prediction to improve the reliability of scheduling algorithms. A trace-based simulation shows scheduling heuristics that can release more jobs as soon as possible are more overhead robust.**

Keywords*—*scientific workflow, scheduling, system overhead, log analysis

I. 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 [1], [2] 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 environment, this traditional approach may prove to be grossly inefficient [3], 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 [4], [5] 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 [1] 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 [4]–[7] in the literature has focused on proposing algorithms that are aware of the dynamic changes of runtime environments. Task rescheduling [8]–[10] 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 [4] can be used to estimate the variance. However, rescheduling a task is costly as it implies some extra communication and synchronization costs. Relevant studies [8] 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 [11] 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,the overheads do not simply work as an attachment to the job runtime and it involves a lot more complicated patterns such as periodicity [1].

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.

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

II. Related Work

Some work in the literature has attempted to define and model robustness. In [12], 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 [13] 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. [14] 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 [4], [5], [7], [15] 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 [4], [5] 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 [16]–[18] are used to capture and abstract the common structure within a workflow and they give insights on designing new workflows and optimization methods. Yu [16] proposed a taxonomy that characterizes and classifies various approaches for building and executing workflows on Grids. They also provided a survey of several representative Grid workflow systems developed by various projects world-wide to demonstrate the comprehensiveness of the taxonomy. Juve [17] 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 [19]. Liu [18] 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 [2] [1] is a topic of great interest in the grid community. Stratan [20] evaluates workflow engines including DAGMan/Condor and Karajan/Globus in a real-world grid environment. Sonmez [21] 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 [2] offers a grid workflow overhead classification and a systematic measurement of overheads. Our prior work [1] 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.

III. Model and Design

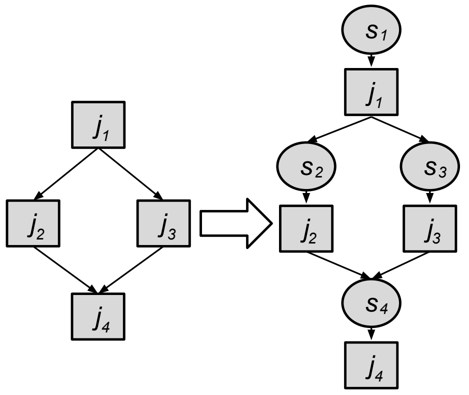

Fig. 1: Extending DAG to o-DAG.

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 [19], Askalon [22], and Taverna [23].

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:

a) Workflow Mapper: generates an executable workflow based on an abstract workflow provided by the user or workflow composition system. It also restructures the workflow to optimize performance and adds jobs for data management and provenance information generation.

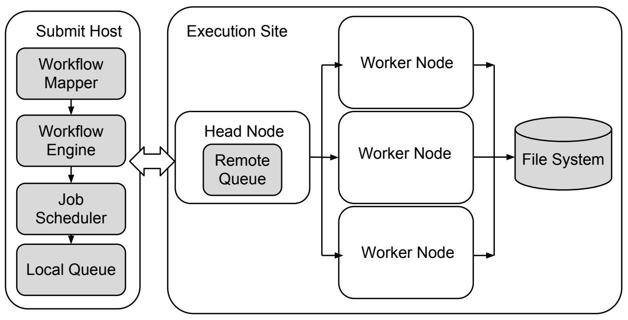

Fig. 2: A workflow system model.

b) 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.

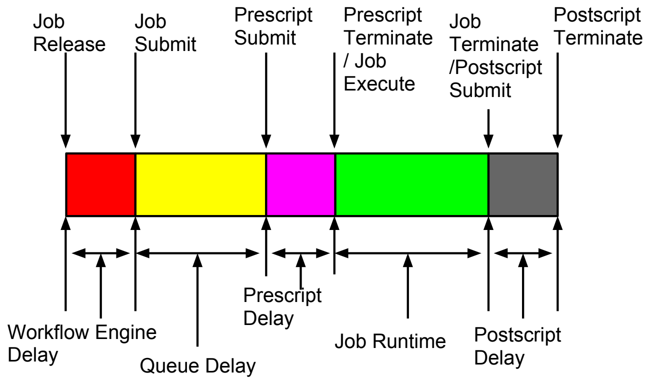

Fig. 3: Overhead Classification

c) 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.

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 3 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 [1], 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 [24]).
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 [25]) 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.

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 tasks are poorly clustered. 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.

A. *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 and temporal verification normally require 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. A scientific workflow overhead duration time series, or overhead pattern, is composed of ordered duration samples obtained from scientific workflow system logs or other forms of historical data. Pattern discovery [18] 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. Also, due to the dynamic nature of underlying resources, it it difficult to provide an accurate forecasting of overheads in practice but we can utilize overhead pattern to improve the overhead robustness without sacrificing much performance gain.

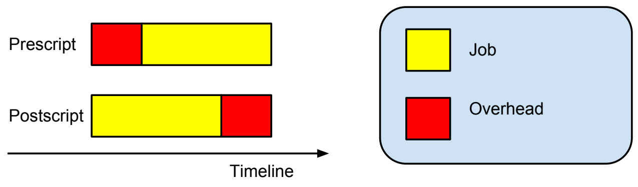

Fig. 4: Adherence Pattern

Most of the work [] 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 4. 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 [1] 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. 5 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 and statistically convincing overhead pattern the Incremental and Periodical Pattern (IPP). We observe the Queue Delay still 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. 6 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. 5 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.

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

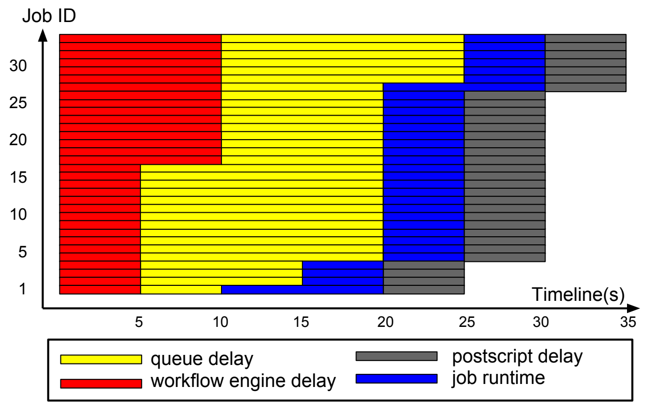

Fig. 5: Workflow Execution Gantt Chart of a Real Trace

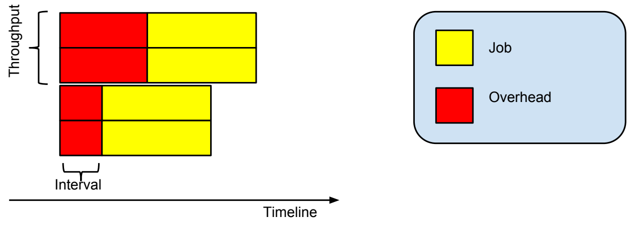

Fig. 6: Incremental and Periodical Pattern

IV. 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 of workflows is not affordable. Mostly speaking, heuristics utilize unique features of workflows or resources to guide the mapping of jobs to resources. For example, both of the MINMIN algorithm and the MAXMIN 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 that performs better than the other one 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. 7, 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.

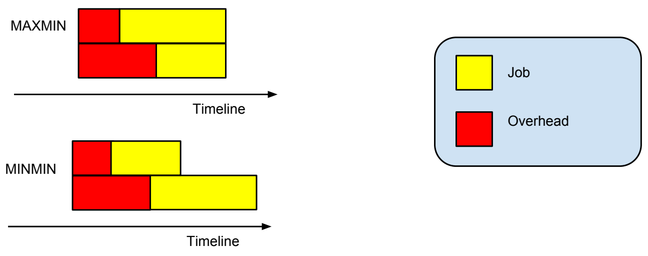

Fig. 7: MAXMIN vs. MINMIN

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 **??**. 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.

Important First and *Unimportant First*: 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:

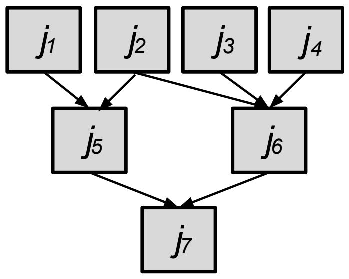

Fig. 8: Impact Factor

$$IF(j_u) = \sum_{j_v \in Child(j_u)} \frac{IF(j_v)}{L(j_v)} \tag{1}$$

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. 8) as 1.0. For instance, consider the workflow in Fig. 8. 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.

V. 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.

A. *Experiment Conditions*

We extended the WorkflowSim [3] 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 bandwidth. As 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

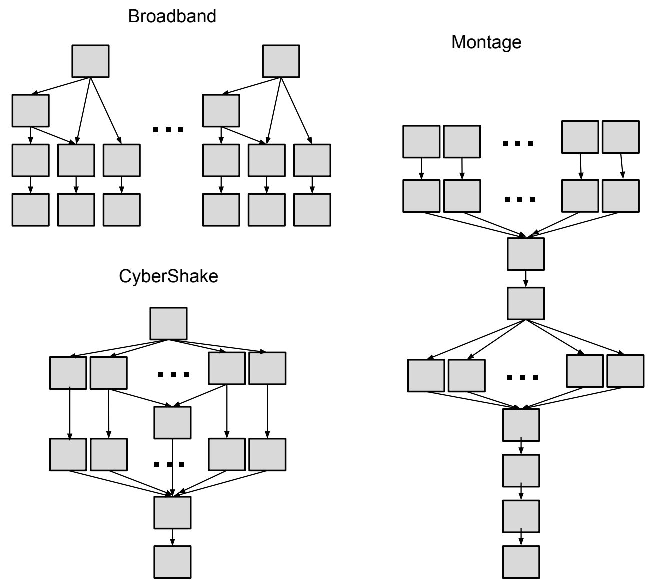

Fig. 9: A simplified visualization of the Broadband workflow, the Montage workflow and the CyberShake workflow.

quota per user of some typical distributed environments such as Amazon EC2 [26] and FutureGrid [27]. Each machine has 512MB of memory and the capacity to process 1,000 million instructions per second.

Heuristics	Overhead Friendly	Overhead Unfriendly
Experiment 1	MAXMIN	MINMIN
Experiment 2	BFS	DFS
Experiment 3	FFS	UFFS
Experiment 4	IFS	UIFS

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 [28] is an astronomy application used to construct large image mosaics of the sky. CyberShake [29] 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. 9. Runtime (average and task runtime distribution) and overhead (workflow engine delay, queue delay, and network bandwidth) information were collected from real traces production environments [1], [17], 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 friendly heuristic (MAXMIN in this experiment) over the overhead unfriendly heuristic (MINMIN), while varying the average workflow engine delay (d) and bandwidth of workflow engine (b). Table V-A shows the heuristics compared in these experiments. Simulation results present a confidence level of 95%. Thus, for values of *Performance Gain* > 0, the overhead friendly heuristics perform better than the respective overhead unfriendly heuristics. Otherwise, the overhead friendly heuristics perform poorer.

In these experiment sets, we vary the average bandwidth of workflow engine from 1 to 20 and we show the results of $b = 1, 5, 15, 20$. The original bandwidth of workflow engine in these traces is 5 and a bandwidth 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 workflow engine delay from 0 to 100 seconds, which represents a typical range of workflow engine delay as shown in [1] and this range is able to show the difference of overhead robustness in these heuristics.

B. Results and Discussion

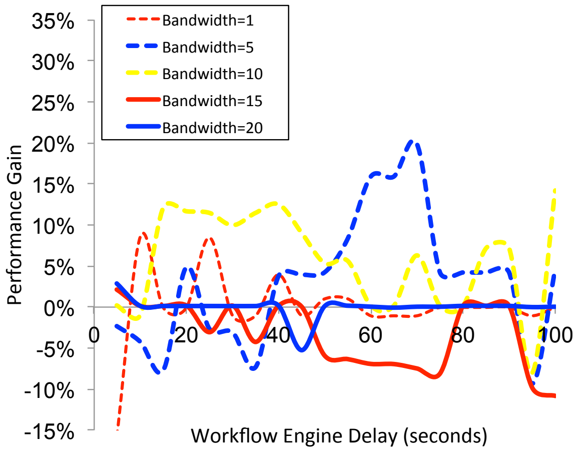

Fig. 10: Broadband

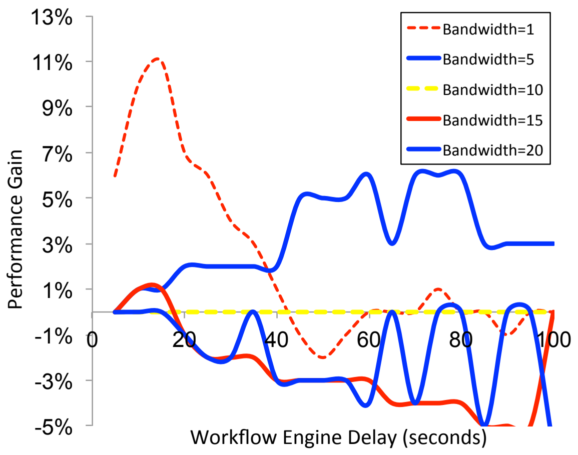

Fig. 11: CyberShake

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

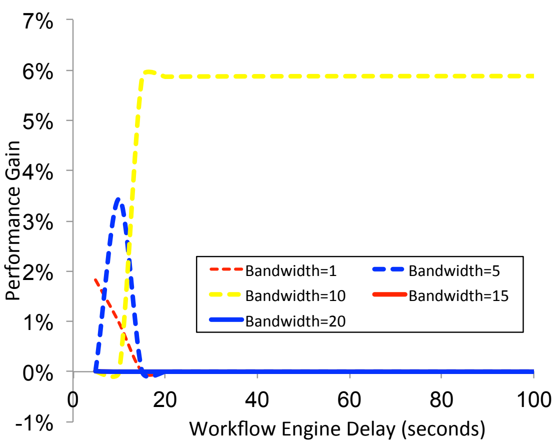

Fig. 12: Montage

Experiment 1: Fig. 10, 11, 12 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 friendly 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 task runtime, workflow engine delay, number of resources and bandwidth.

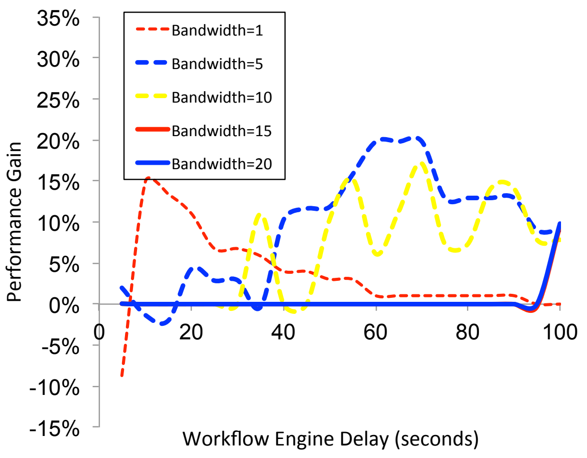

Fig. 13: Broadband

Experiment 2: Fig. 13, 14, 15 shows 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 14 shows with the increase of average bandwidth, the *Performance Gain* is more significant. We can also see that when the average bandwidth is high, the *Performance Gain* increases with the average workflow engine delay. This suggests us in a real environment with a large overhead, we should use BFS instead of DFS.

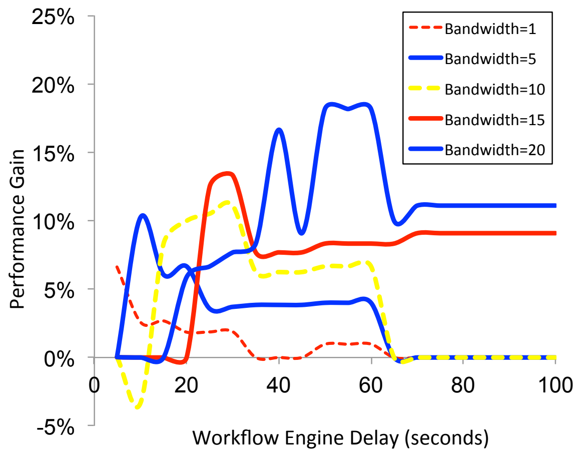

Fig. 14: CyberShake

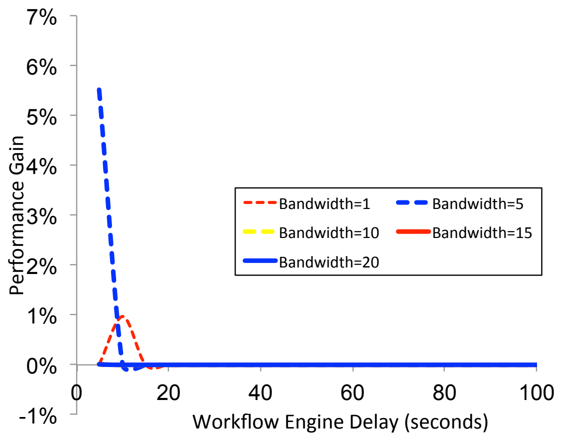

Fig. 15: Montage

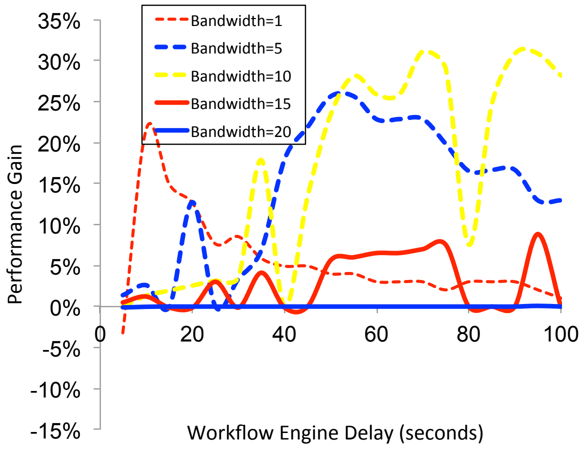

Fig. 16: Broadband

Experiment 3: Fig. 16, 17, 18 shows the *Performance Gain* of FFS over UFFS for the three workflows. We observe that most *Performance Gain* > 0 and thus FFS performs better

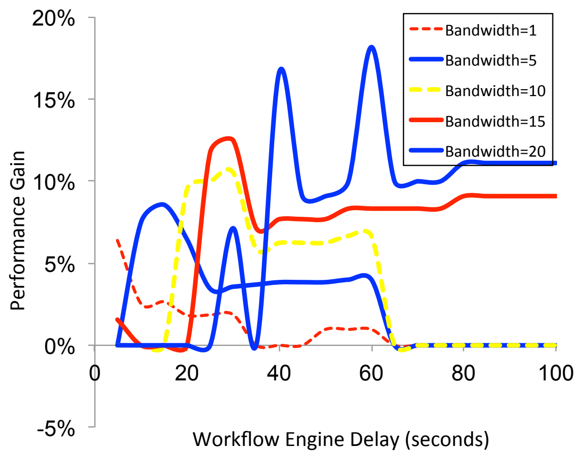

Fig. 17: CyberShake

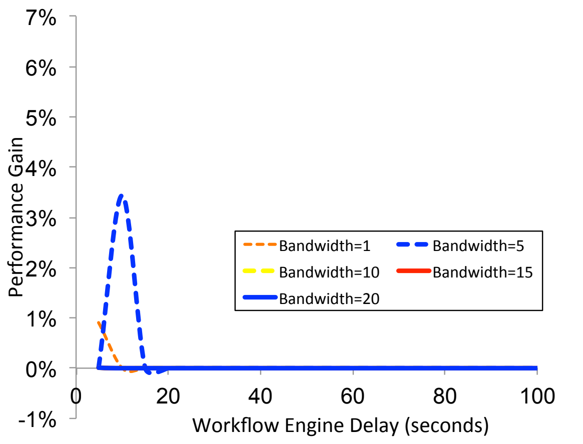

Fig. 18: Montage

than UFFS in terms of overhead robustness, which is similar to Experiment 2. Comparing Fig. 13 and Fig. 16 we can see the *Performance Gain* of FFS over UFFS is more significant (30%) than that of BFS over DFS (20%).

Experiment 4: Fig. 19, 20, 21 shows 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.

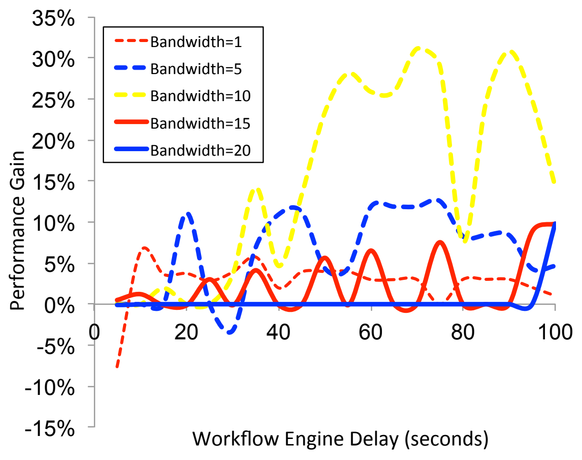

Fig. 19: Broadband

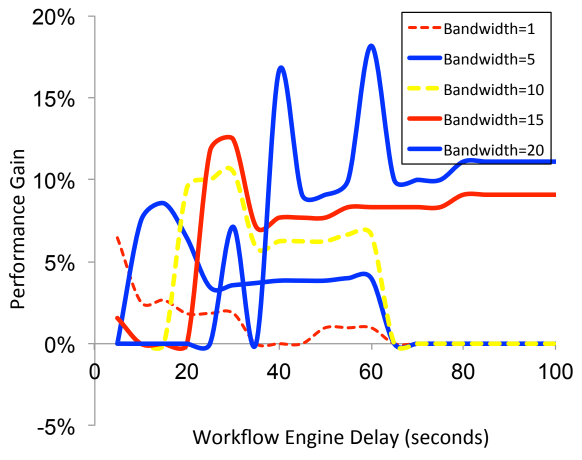

Fig. 20: CyberShake

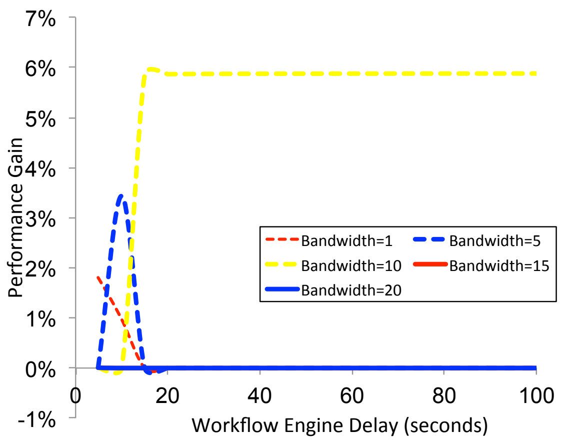

Fig. 21: Montage

VI. Conclusion

References

[1] W. Chen and E. Deelman, "Workflow overhead analysis and optimizations," in *Proceedings of the 6th workshop on Workflows in support of large-scale science*, ser. WORKS '11. New York, NY, USA: ACM, 2011, pp. 11–20.

[2] R. Prodan and T. Fahringer, "Overhead analysis of scientific workflows in Grid environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 3, pp. 378–393, March 2008.

[3] W. Chen and E. Deelman, "Workflowsim: A toolkit for simulating

scientific workflows in distributed environments," in *The 8th IEEE International Conference on eScience*, Oct. 2012.

[4] F. Dong, J. Luo, A. Song, and 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.

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

[6] I. Ahmad, M. K. Dhodhi, and R. Ul-Mustafa, "Dps: Dynamic priority scheduling heuristic for heterogeneous computing systems," *Computers and Digital Techniques, IEE Proceedings-*, vol. 145, no. 6.

[7] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real-Time Systems 2.3*, 1990.

[8] R. Sakellariou and H. Zhao, "A low-cost rescheduling policy for efficient mapping of workflows on grid systems," *Sci. Program.*, vol. 12, no. 4, pp. 253–262, Dec. 2004.

[9] Y. Zhang, C. Koelbel, and K. Cooper, "Hybrid re-scheduling mechanisms for workflow applications on multi-cluster grid," in *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, 2009, pp. 116–123.

[10] W. Chen, K. Fekete, and Y. C. Lee, "Exploiting deadline flexibility in grid workflow rescheduling," in *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, 2010, pp. 105–112.

[11] M. Zaharia, K. Elmeleegy, D. Borthakur, S. Shenker, J. S. Sarma, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *In Proc. EuroSys*, 2010.

[12] S. Ali, A. Maciejewski, H. Siegel, and J.-K. Kim, "Measuring the robustness of a resource allocation," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 15, no. 7, pp. 630–641, 2004.

[13] L.-C. Canon and E. Jeannot, "Comparative evaluation of the robustness of dag scheduling heuristics," in *Grid Computing*. Springer, 2008.

[14] T. D. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. A. Hensgen, and R. F. Freund, "A comparison of eleven static heuristic for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, 2001.

[15] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, and et al., "Task scheduling strategies for workflow-based applications in grids," in *IN CCGRID*. IEEE Press, 2005, pp. 759–767.

[16] J. Yu and R. Buyya, "A taxonomy of workflow management systems for grid computing," *Journal of Grid Computing*, vol. 3.

[17] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682 – 692, 2013.

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

[19] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Sci. Program.*, vol. 13, no. 3, pp. 219–237, Jul. 2005.

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

[21] O. Sonmez, N. Yigitbasi, A. Iosup, and D. Epema, "Trace-based evaluation of job runtime and queue wait time predictions in grids," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*, ser. HPDC '09. New York, NY, USA: ACM, 2009, pp. 111–120.

[22] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. Seragiotto, Jr., and H.-L. Truong, "Askalon: a tool set for cluster and grid computing: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 2-4, pp. 143–169, Feb. 2005.

[23] 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, and C. Wroe, "Taverna: lessons in creating a workflow environment for the life sciences: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 18, no. 10, pp. 1067–1100, Aug. 2006.

[24] "DAGMan: Directed Acyclic Graph Manager," http://cs.wisc.edu/condor/dagman. [Online]. Available: http://cs.wisc.edu/condor/dagman

[25] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-G: a computation management agent for Multi-Institutional grids," *Cluster Computing*, vol. 5, no. 3, pp. 237–246, 2002.

[26] Amazon.com, Inc., "Amazon Web Services," http://aws.amazon.com. [Online]. Available: http://aws.amazon.com

[27] "FutureGrid," http://futuregrid.org/. [Online]. Available: http://futuregrid.org/

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

[29] S. Callaghan, P. Maechling, E. Deelman, K. Vahi, G. Mehta, G. Juve, K. Milner, R. Graves, E. Field, D. Okaya, and 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.