Workflow Performance Improvement using Model-based Scheduling over Multiple Clusters and Clouds[☆]

Ketan Maheshwari[a,*], Eun-Sung Jung[a], Jiayuan Meng[b], Vitali Morozov[b], Venkatram Vishwanath[b], Rajkumar Kettimuthu[a]

[a] *Mathematics and Computer Science Division*
Argonne National Laboratory
Argonne, IL, USA 60439
[b] *Leadership Computing Facility Division*
Argonne National Laboratory
Argonne, IL, USA 60439
[c] *Google Inc., Mountain View, CA, USA 94043*

Abstract

In recent years, a variety of computational sites and resources have emerged, and users often have access to multiple resources that are distributed. These sites are heterogeneous in nature and performance of different tasks in a workflow varies from one site to another. Additionally, users typically have a limited resource allocation at each site capped by administrative policies. In such cases, judicious scheduling strategy is required in order to map tasks in the workflow to resources so that the workload is balanced among sites and the overhead is minimized in data transfer. Most existing systems either run the entire workflow in a single site or use naïve approaches to dis-

[*]Principal corresponding author

Email addresses: ketan@anl.gov (Ketan Maheshwari), esjung@mcs.anl.gov (Eun-Sung Jung), meng.jiayuan@gmail.com (Jiayuan Meng), morozov@anl.gov (Vitali Morozov), venkatv@mcs.anl.gov (Venkatram Vishwanath), kettimut@mcs.anl.gov (Rajkumar Kettimuthu)

tribute the tasks across sites or leave it to the user to optimize the allocation of tasks to distributed resources. This results in a significant loss in productivity. We propose a multi-site workflow scheduling technique that uses performance models to predict the execution time on resources and dynamic probes to identify the achievable network throughput between sites. We evaluate our approach using real world applications using the Swift parallel and distributed execution framework. We use two distinct computational environments–geographically distributed multiple clusters and multiple clouds. We show that our approach improves the resource utilization and reduces execution time when compared to the default schedule.

Keywords: System Modeling, Workflow, Optimization, Swift, Clouds

---

1. Introduction

Large-scale scientific applications involve repetitive communication-, data-, memory- or compute-intensive tasks. These applications are often encoded as workflows in order to improve productivity, and are deployed over remote computational sites. The workflow framework must schedule tasks over available sites and manage data movement among the tasks. In recent years, given the increasing prevalence of computation, these sites have been significantly grown in number and size and have diversified in terms of their underlying architecture. They vary widely in system characteristics including raw compute power, per-node memory, file system throughput, and performance of the network. With such heterogeneity, different tasks within the same workflow may perform better at different sites. Even in a single large supercomputer, we are witnessing this heterogeneity both at a node-level, as well as at a system level. In the latter case, we now have systems where larger memory footprint nodes are interconnected with compute intensive nodes via a high-performance interconnect. Furthermore, emerging computational infrastructures such as clouds have considerably altered the course of computational research in the recent years. The existing computational models are still not well-aligned with the cloud model of computation.

In addition to the issue of resource heterogeneity, users confront logistical constraints in using these systems including allocation time and software compatibility. Users often subscribe to a multitude of sites, spanning geographical regions, connected through various types of networks. It is often desired, therefore, to deploy an application over multiple sites in order to

best utilize the resources collectively.
Unfortunately, the resource allocation for each site may be limited and the system configuration at each site may be suited for some tasks but not others. Such resource-task affinity constraints must be taken into account while scheduling these workflows. Given these constraints and the dynamic nature of the network connecting these sites, it is non-trivial to compute a schedule that will optimally utilize the resources across sites to achieve the best time-to-solution. An ideal scenario from a user's perspective is:

1. Construct a workflow [1].
2. Provide the list of resources.
3. Execute the workflow by spreading the tasks to distributed resources (provided by the user in the previous step) without any intervention from the user.

This work tackles the aforementioned step 3. In particular, our work addresses the following key challenge:

> *Efficiently schedule and run data and compute-intensive workflows over multiple, heterogeneous, and geographically dispersed computational sites.*

We use the Swift framework [2] for workflow execution, SKOPE framework [3] for workflow performance modeling, and a network scheduling algorithm for optimizing mapping between tasks and resources. A scheme of our framework with steps and their interconnections is shown in Figure 1. The framework takes the workflow description encoded as a Swift script and profiles different tasks in the workflow on available resources to generate a workflow skeleton in the format required by SKOPE (see Section 3 for details). Using the workflow skeleton, SKOPE builds analytical models about the data transfers between tasks, and empirical models about performance scalability of tasks. SKOPE then constructs a job graph describing the estimated computation and data transfer according to the models. The job graph is used as inputs to the scheduling algorithm, which generates an optimized schedule by taking into account the performance scalability of tasks and network condition between the relevant sites. Eventually, the Swift framework executes the workflow using the recommended schedule.
Although considerable work has been done in the past on scientific workflow management systems [4, 5, 6, 7, 8] and metascheduling systems [9, 10, 11],

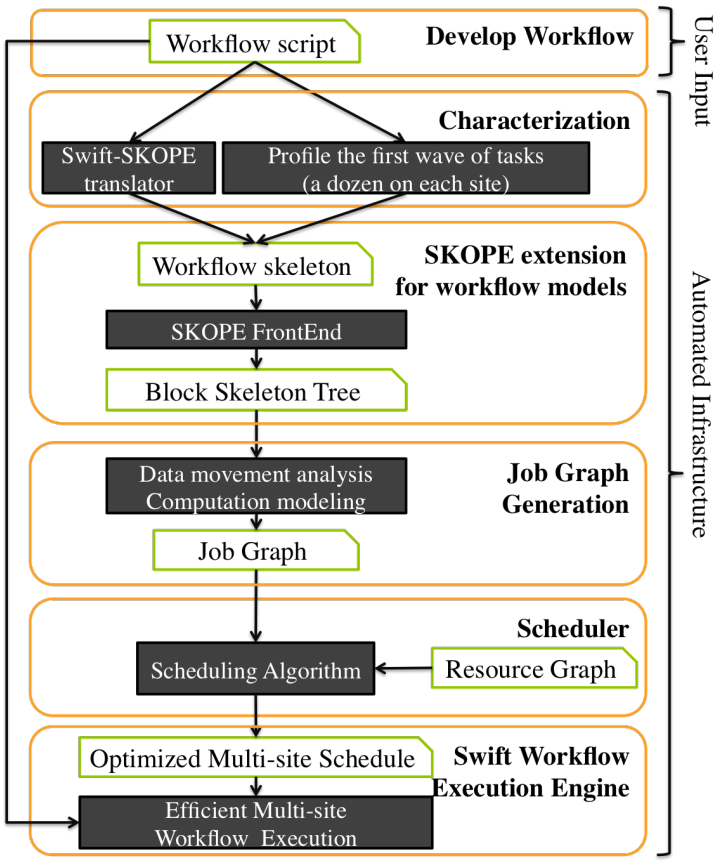


Figure 1: A Conceptual framework for multi-site workflow scheduling.

optimizing the execution of workflows across heterogeneous resources at multiple geographically distributed sites has not received much attention. This is due in part to the lack of access to multiple independent resources and in part to the lack of workflow enactment capabilities. Most metascheduling systems run the entire application or workflow at a single site. Systems such as Swift enables the execution of various tasks of a workflow at different sites but they do not have sophisticated scheduling algorithms to optimize the execution of workflow across different sites. A good scheduling algorithm must take into account not only the heterogeneous nature of the compute infrastructure at various sites but also the network connectivity and load between the computational sites and the data source(s) and sink(s). Our goal

is to develop better schedules for workflows across geographically distributed resources.
Our specific contributions in this work is as follows:

- Development of the notion of *workflow skeletons framework to capture, explore, analyze and model empirical workflow behavior* with regard to dynamics of computation and data movement.
- An algorithm to *construct an optimized schedule*, according to the modeled workflow behavior.
- *Integration of the workflow skeleton and the scheduling algorithm into a deployment system.*
- Demonstration of the effectiveness of our approach over two distinct distributed environments: a collection of traditional clusters and multiple clouds. We use the *Amazon AWS, the Google Compute Engine and the Microsoft Azure cloud platforms* in this work.

The remainder of the paper is organized as follows. Section 2 presents an overview of Swift, a workflow expression and execution framework; typical scheduling mechanisms; and SKOPE, a workload modeling framework. Section 3 presents our optimized scheduling technique. Section 4 describes our experimental setup. Section 5 presents an evaluation of the proposed approach using real scientific workflows over multiple sites with distinct characteristics. Section 6 discusses related works. Conclusions are given in Section 7.

2. Background

In this section we introduce parallel workflow scripting, resource scheduling, and workload behavior modeling techniques which forms the basis of our work. The following terminology is used. A *workflow* is a process that involves the execution of many programs. The invocation of an individual program is referred to as a *task*. These tasks may be dependent on each other or can run in parallel. Tasks are dispatched to various sites in groups of scheduling units, or *jobs*. Jobs define the granularity in which the schedule maps tasks to resources. A job consists of one or multiple tasks that correspond to the same program but different input data.

2.1. Swift: Parallel Workflow Scripting

Swift is a workflow framework for parallel execution of ordinary programs [12]. The Swift runtime contains a powerful platform for running user programs on a broad range of computational infrastructures, such as clouds, grids, clusters, and supercomputers out of the box. A user can define an array of files, and use `foreach` loops to create a large number of implicitly parallel tasks. Swift will then analyze the script and execute tasks based on their dataflow; a task is executed only after any of its dependent tasks finish.

Applications encoded as Swift scripts have been shown to execute on multiple computational sites (clusters, clouds, supercomputers) via Swift coasters [13, 12, 14] mechanism which implements the pilot jobs paradigm. The pilot job paradigm dispatches a pilot task to each of the sites and measures the task completion rate. The task completion rate for the corresponding task-site combination then serves as an indicator to increase or decrease the number of tasks assigned to each site. However, it does not distinguish the latency in task execution from the overhead in data transfer. Moreover, the number of tasks to be executed on each node remains a global constant; it may starve some CPUs if the number is too low, or thrash the shared cache or memory if the number is too high. Therefore, the resulting schedule is sub-optimal.

2.2. Workflow Scheduling

The resource and job scheduling problem is a classic NP-hard problem [15]. It can be formally stated by resource and job definitions, and the algorithms vary depending on characteristics of resources and jobs. For instance, job-shop scheduling [16] is for multiple independent jobs with varying sizes and multiple homogeneous resources.

In the context of distributed computing, jobs may have dependencies among them and take input data from remote sites and send output to a different set of remote sites. The sites themselves may also have a broad spectrum of system architectures and capacities. In order for scheduling to take into account all these factors, sophisticated algorithms are needed. In this paper, we tackle a workflow scheduling problem of minimizing the time-to-completion of a workflow where performance models of tasks in the workflow are given as mathematical functions with regard to all remote sites. We extend our previous linear programming based scheduling algorithms [17] to incorporate performance models as well as network and compute resources. In general, scheduling algorithms considering several factors (e.g., network and compute resources) at the same time are termed as *joint scheduling* algorithms [18, 19].

Joint scheduling algorithms are advantageous for better performance while they may need more sophisticated mechanisms and may lead to high time complexities. Even though many previous studies including our work [17] have addressed these issues, many more factors as described above are left unconsidered for reasons such as time complexities. Our scheduling algorithms extended for task performance models for multiple computation sites are unique from the perspective of incorporating all factors such as network and compute resources and site-specific task performance models. In addition, model-driven job graphs will be able to provide rich semantics for flexible and efficient scheduling.

2.3. SKOPE: A Workload Behavior Modeling Framework

SKOPE (SKeleton framewOrk for Performance Exploration) is a framework that helps users describe, model, and explore a workload's current and potential behavior [3]. It asks the user to provide a description, called *code skeleton*, that identifies a workload's performance behavior including data flow, control flow, and computational intensity. These behavioral properties are intrinsic to the application and is agnostic of any system hardware. They are interdependent; the control flow may determine data access patterns, and data values may affect control flow. Given different input data, they may result in diverse performance outcomes. They also reveal transformation opportunities and help users understand how workloads may interact with and adapt to emerging hardware. According to the semantics and the structure in the code skeleton, the SKOPE back-end explores various transformations, synthesizes performance characteristics of each transformation, and evaluates the transformation with various types of hardware models.

In this work, we adopt and extend SKOPE to model workflows with data transfer requirements over wide area networks. The SKOPE front-end is extended with syntax and semantics to describe files and computational tasks. The resulting code skeleton is called the *workflow skeleton*. We further add a back-end procedure that constructs job graphs from workload skeletons.

3. Model-Driven Multisite Workflow Scheduling

In this section, we describe how we use modeling to schedule a workflow on multiple sites. User provides the workflow as a Swift script or using some other interface. In case of latter, an approach such as [1] can be used to obtain a workflow. From the Swift script, we generate a performance property

description of the workflow, which includes tasks' site-specific scaling and the data dependency among them. Such a description is generated in the form of our extended SKOPE language, and is referred to as a *workflow skeleton*, or *skeleton* in short. This process is done manually now but will be automated in the future. User also provides a list of resources available to execute the workflow. SKOPE then automatically generates a *job graph*, where a *job* refers to one or more tasks of the workflow grouped as a scheduling unit. The technique of grouping multiple tasks into a job is generally called *task clustering*. This technique helps reduce the complexity of a job graph and queue waiting time when tasks are scheduled on multiple HPC resources. In particular, the reduced complexity of a job graph leads to less running time of our LP-based scheduling algorithms, which requires higher computing time than simple heuristics.

The job graph depicts the jobs' site-specific resource requirements as well as the data flow among them. Such a job graph is then used as input to a scheduling algorithm, which also takes into account the resource graph that describes the underlying properties at multiple sites and the network connecting them. The output of the algorithm is an optimized mapping between the job graph and the resource graph, which the default Swift scheduler then uses to dispatch the jobs.

Table 1: Syntax for workflow skeletons

Macros and Data Declarations	
File type and size (in KB)	:MyFile N
Constant definition	:symbol = expr
Array of files	:type array[N][M]
Variable def./assign	var = expr
Variable range	var_name=begin:end(exclusive):stride
Control Flow Statements	
Sequential `for` loop	for var_range {list_of_statements}
Parallel `for` loop	forall list_of_var_ranges {list_of_statements}
Branches	if(conditional probability){list_of_statements}
	else{list_of_statements}
Data Flow Statements	
file input/load	ld array[$expr_i$][$expr_j$]
file output/store	st array[$expr_i$][$expr_j$]
Characteristic Statements	
Run time (in sec.)	comp T
Task description	
Application definition	def app(arg_list){list_of_statements}
Application invocation	call app(arg_list)

3.1. Workflow Skeleton

We use workflow skeletons for two specific purposes: (1) define *tasks*; (2) identify data movements among tasks.

The syntax of a workflow skeleton is summarized in Table 1. Figure 2 presents an illustrative workflow from the geoscience area involving *ab initio* simulations. In Figure 2(a), shows the script for the workflow. Its skeleton is listed in Figure 2(b). The skeleton is structured identically to its original workflow script in terms of file types, task definitions, and the control and data flow among the tasks.

A skeleton is parsed by the SKOPE framework into a data structure called the *block skeleton tree* (BST). Figure 2(c) shows the BST corresponding to the skeleton in Figure 2(b). Each node of the BST corresponds to a statement in the skeleton. Statements such as task definitions, loops, or branches may encapsulate other statements, which in turn become the children nodes. The loop boundaries and data access patterns can be determined later by propagating input values.

To describe a task's distinct behavior over various sites, the user can represent its skeleton with a `switch` statement. Each of its `case` statements describes the task's performance model for the corresponding site. An example skeleton description of a task is demonstrated by lines 33-36 in Figure 2(b). This performance model is then used by the scheduler to determine how many tasks to assign to a site.

Given the high-level nature of workflows and the structural similarity between a workflow script and the skeleton, generating the workflow skeletons is straightforward and will be automated in the future with a source-to-source translator. The major effort in writing a workflow skeleton falls on profiling tasks over available systems in order to obtain site-specific performance models. For each task, we measure its single node execution time when it is co-executed with multiple tasks, up to a point where all computing resources (i.e., cores) on the same node are exploited.

We then apply quadratic curve fitting to obtain an empirical performance model. Since a typical workflow is repeatedly executed, such performance information can be obtained from historical data, either by explicit user measurement or by implicit profiling.

3.2. Procedural Job Graph Generation

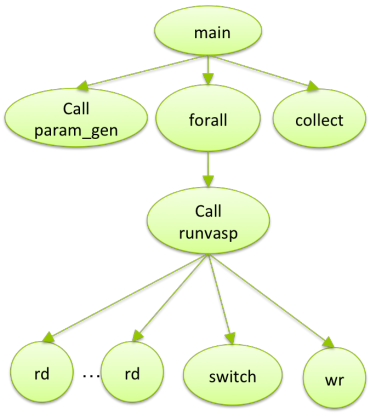
The workflow skeleton produces a job graph as input to the scheduling algorithm. A job graph is a DAG describing the performance characteristics of

```
1.  type file;
2.  # Files
3.  file incar <"INCAR">;
4.  file poscar <"POSCAR">;
5.  file potcar <"POTCAR">;
6.  file kpoints <"KPOINTS">;
7.  int init_vol=400;
8.  float init_factor=1.03574416884776;
9.  int[] kpts=[336, 448, 6612];
10. int[] encuts=[600, 700, 800, 900, 1000];
11. int[] vols=[200, 220, 240, 260, 280, 300, 320,
    340, 360, 380, 400];
12. foreach kpt in kpts{
13.     foreach encut in encuts{
14.         foreach vol in vols{
15.             file outbundle <single_file_mapper;
    file=@strcat("output/vasp_outbundle_", kpt,
    "_", encut, "_", vol, ".tgz")>;
16.             (output, error, outbundle) =
    runvasp(incar, poscar, potcar, kpoints,
    init_vol, init_factor, kpt, encut, vol);
17.         }
18.     }
19. }
```

(a) Original workflow script in Swift

```
1.  // File size description
2.  :InFile = 1 // in kilo bytes
3.  :PosFile = 4
4.  :PotFile = 431
5.  :KptFile = 1
6.  :OBFile = 1000000
7.  :site = 'stampede'
8.  :N = 165
9.  def main()
10. { // file declaration
11.   :InFile incar
12.   :PosFile poscar
13.   :PotFile potcar
14.   :KptFile kpoints
15.   :OBFile outbundle[N]
16.   // produces incar, poscar, potcar, kpoints
17.   call param_gen(incar, poscar, potcar, kpoints)
18.   forall g = 0:N
19.   { // execute parallel tasks
20.     call runvasp(incar, poscar, potcar, kpoints,
    outbundle[n])
21.   }
22.   call collect(outbundle) // collect results
23. }
24. def runvasp(incar, poscar, potcar, kpoint,
    outbundle)
25. { // read from four files
26.   rd incar
27.   rd poscar
28.   rd potcar
29.   rd kpoint
30.   // performance scalability model
31.   switch(site)
32.   {
33.     case 'stampede'
34.     { // site-specific weak scaling model
35.       // "tpn" refers to # of tasks per node
36.       comp   (0.2*tpn*tpn-0.1*tpn+0.3)/tpn
37.     }
38.     break
39.     case 'blues'
40.     {
41.       comp   (0.1*tpn*tpn-0.2*tpn+0.5)/
    tpn
42.     }
43.   }
44.   // write to output file
45.   wr outbundle
46. }
```

(b) Workflow skeleton



(c) Block Skeleton Tree for the main skeleton function

Figure 2: Illustrative workflow script (a), skeleton (b), and the corresponding block skeleton tree (BST) (c) for a pedagogical workflow related to a geoscience application.

each job and the data movement among them. Figure 3 illustrates the job graph generated from the workflow skeleton in Figure 2(b). In a job graph, nodes refer to jobs and edges refer to data movements. Note that the structure of the job graph is independent of the hardware resources. Moreover,

a node is annotated with the amount of computation resources, or the execution time needed by the corresponding task for each available system. An edge is annotated by the amount of data that is transferred from the source node to the sink node.
Note that a job is a scheduling unit that may refer to a group of tasks. Grouping multiple tasks can be achieved by simply transforming nested parallel `for` loops in the workflow skeleton into a two level nested loop, where the inner loop defines a job with a number of tasks, and the outer loop is sized according to the desired number of jobs, which is a predefined constant according to the available number of sites. In this work, we adopt the heuristic where iterations of a parallel `for` loop are grouped into a number of jobs no more than 10 times the number of sites. Such a granularity enables the scheduler to balance the workloads among sites, and at the same time does not lead to a significant overhead in probing a large number of possibilities.

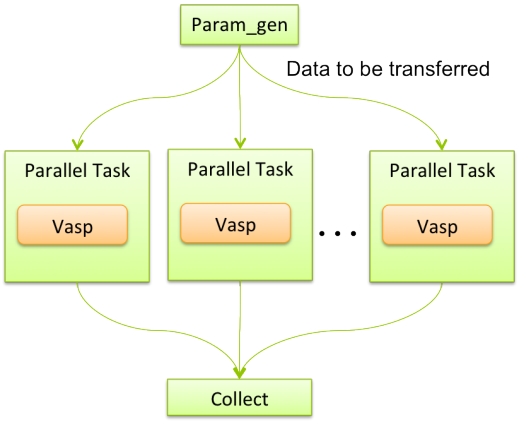


Figure 3: Job graph for the workflow shown in Figure 2(a).

Generating a job graph from a workflow skeleton involves four major steps. First, we decide the group size according to the number of tasks and the number of sites. Second, we obtain the data footprint for each job by aggregating the data footprints for tasks within a group. Third, we construct the data flow among dependent jobs. Finally, we derive a symbolic expression to express the execution time of a job over different systems; this also involves aggregating the execution time of tasks within a group.
The key to our technique is data movement analysis, for which we apply array section analysis using bounded regular section (BRS) [20]. BRS has

been conventionally used to study stencil computation's data access patterns within loops. It is adopted in our study to analyze data access patterns over arrays of files. In BRS, an array access can be regarded as a function that maps a range of loop iterators to a set of elements over the array. In this paper, we refer to the range of loop iterators as a *tile* ($\mathbb{T}$) and the set of accessed array elements as a *pattern* ($\mathbb{P}$). For example, suppose A is a 2-D array of files and a task accesses $A[r][c]$ in a nested `for` loop with two iterators, r and c. The tile corresponding to the loop is denoted as $\mathbb{T}(r,c) = \{r : \langle r^l : r^u : r^s \rangle; c : \langle c^l : c^u : c^s \rangle\}$, where each of the three components represents the lower bound, upper bound, and stride, respectively. The overall pattern accessed within this loop is denoted as $A[\langle r^l : r^u : r^s \rangle][\langle c^l : c^u : c^s \rangle]$, which is summarized by $\mathbb{P}(A[r][c], \mathbb{T}(r,c))$. Patterns can be intersected and/or unioned.

To obtain the data footprint of a job, we identify its corresponding node in the BST and obtain the tile $\mathbb{T}$ corresponding to `one` iteration of all loops in its ancestor nodes (i.e., the outer loops) and `all` iterations of its child nodes (i.e., the inner loops). Given an access to a file array, $\mathbb{A}$, we apply $\mathbb{T}$ to obtain a pattern, $\mathbb{P}(\mathbb{A}, \mathbb{T})$, which symbolically depicts the data footprint of the job. We then build the data flow among jobs. First, we scan all BST nodes that correspond to jobs. Pairs of nodes producing and consuming the same array of files become candidate dependent jobs. Next, we perform intersection operations between produced and consumed patterns to determine the exact pattern that caused the dependency. The size of the dependent patterns is the amount of data movement associated with an edge in the job graph.

Then, we derive the execution time of each job for different systems. We simply traverse the BST of the code skeleton once for each site; in each traversal, the *switch* statement is evaluated, and its execution time for that particular site is obtained. We then aggregate the per task execution time into the per job execution time by multiplying it with the number of tasks within a job.

The resulting job graph is output in the form of an adjacency list. In addition, each node has two attributes, one is the number of independent tasks within this job, and the other is the performance modeling which estimates the execution time of each job given the number of assigned cores. It is then passed to the scheduler algorithm to generate an optimized mapping among the jobs and resources.

3.3. Multisite/Multicloud Scheduling

In this section, we present how a scheduler interact with other components in our framework, and we describe our scheduling algorithm in detail.

3.3.1. General procedure

The scheduler in our framework needs a resource graph as well as the given job graph. While a job graph provides a description about the requirement and behavior of a workflow, a resource graph provides a description about the distributed compute resource where a workflow is deployed. In particular, the knowledge includes available compute nodes at each site, the number of cores per node, and the network bandwidth amongst computation sites. We use a resource graph to describe such information about underlying distributed systems.Figure 4 (a) illustrates an example of the resource graph. Nodes and edges denote compute resources and network paths among those resources. Even though a network path can span multiple physical network links, we use only one logical link between two sites because we cannot setup paths at our discretion in these experiments. However, if we have control over network path setup in connection-oriented networks, a physical network topology can also be used as a resource graph because our algorithm is a network-centric joint scheduling algorithm and can take into account a real network topology for data transfers. Figure 4 (b) is the resource graph for multisite scheduling, corresponding to Table 2. The resource graph is for grid computing environments, and the resource graph for cloud computing environments can be constructed in a similar way. We ran disk-to-disk transfer probes to identify the achievable throughput among our computation sites. Note that the resource graph is a complete graph, where each node is connected to all other nodes.

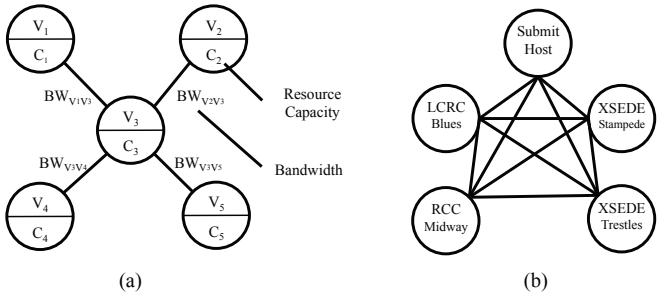


Figure 4: (a) Resource graph model (b) Resource graph in our experiments.

Figure 5 shows the basic ideas of converting a task scheduling problem into a network flow problem. Given a task T_1 and three resources R_1, R_2, and R_3 where D is the T_1's demand for CPU resource and C_1, C_2, and C_3 are capacities of R_1, R_2, and R_3 as shown in Figure 5(a), a task scheduling problem is equal to find the optimal path for a data flow with the amount D among the three paths from the node T_1 to the resources, where the bandwidth of a path is set to the capacity of a destination resource node as in Figure 5(b). For example, if $D = 10$ and C_1, C_2, and C_3 are 1, 2, 5, respectively, this means that the run times of task T_1 on R_1, R_2, and R_3 are $10(= \frac{10}{1})$, $5(= \frac{10}{2})$, and $2(= \frac{10}{5})$, respectively. More details on our algorithm will be described in the following sections.

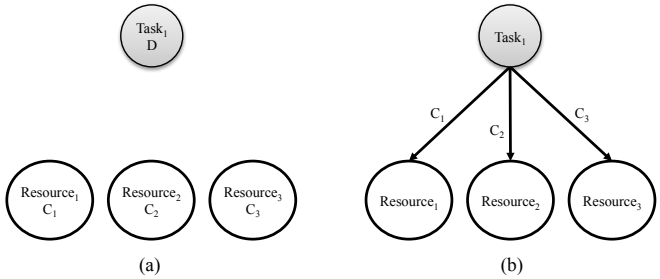


Figure 5: (a) Task and resources (b) Task to resource mapping.

Multicloud scheduling works in the same way as multisite scheduling does. Though cloud computing can provision infinite amount of resources, cloud providers do not allow users to use as many virtual machines as they want on demand. In practice, users can get virtual machines up to a fixed number (e.g., 20), and users should get permission from cloud providers ahead of using a large number of virtual machines. Accordingly, we can assume that a limited number of virtual machine instances are available for on-demand provisioning at each cloud, which is similar to the assumption on HPC resources. Furthermore, it is important to note that the virtual machine instances are drawn from underlying hardware with different characteristics and load. In this scenario, it is crucial to identify true capability of the virtual machine. In order to find out processing speeds and memory bandwidth, we benchmarked each of the clouds used in this work. We take the results shown in Table 3 into account for the final multicloud schedule.
Alternatively, our scheduler combined with intelligent SKOPE job graph generation can achieve better performance. A job in a job graph that SKOPE

provides to the scheduler may correspond to multiple parallel tasks and is a basic scheduling unit. Previously, partitioning techniques for multiple same-level tasks, called *task clustering*, groups all the tasks into fixed number of partitions (e.g., 2 or 4) with the same number of tasks to reduce scheduling overhead or task queue wait time. The SKOPE job graph generator may provide as many parallel jobs in a job graph as the number of available computation sites. In this way, our scheduler can maximize parallel execution of jobs according to the current resource environment while static task clustering approach would not utilize idle resources.

3.3.2. Task-resource affinity aware joint scheduling

In this paper, we extend our joint scheduling algorithm presented in [17]. Joint scheduling means that it takes into account both compute resources and network paths together in order to further optimize the execution of a workflow. In distributed workflow scheduling, data movement among sites is not trivial, especially when the network resources are not abundant. That means independent scheduling of compute resource and network paths may not give a near optimal schedule. Our previous algorithm converts a scheduling problem into a network flow formulation, for which there are well-known linear programming approaches, as shown in Figure 4.

However, these formulations lack task performance models for heterogeneous compute resources. In order to schedule workflows among multisite compute resources, a new notion of task-resource affinity is proposed and incorporated into our previous workflow scheduling algorithms. This is novel in a sense that all factors including network and compute resources and task performance models per site are incorporated into one formulation.

Table 2: Cluster execution sites and their characteristics

Site	**CPU Cores**	**CPU Speed**	**Memory/Node**	**Remarks**
LCRC Blues	310X16=4960	2.60GHz	62.90 GB	35 jobs cap
XSEDE Stampede	6400X16=102400	2.70GHz	31.31 GB	50 jobs cap
XSEDE Trestles	324X32=10368	2.4GHz	63.2 GB	unknown
RCC Midway	160X16=2560	2.60GHz	32.00 GB	Institute-wide access

The one of key issues in extending the previous algorithm is how to incorporate the different task performance models on different resources and how to account for the amount of resources in a task performance model. We define task-resource affinity as $T_s^i(n)$ which represents the run time of task i on the

Table 3: Cloud execution sites and their characteristics

Cloud	**Processing** (GFLOPs)	**Memory Bandwidth** (GB/sec)	**Cost per instance per hour**	**Limitations**
Microsoft Azure	8.6	10.2	$0.14	50 instances/subscription
Google Compute	16.9	11.0	$0.14	24 CPUs/region
Amazon AWS	10.5	12.1	$0.14	20 instances/region

resource site s where n nodes are provisioned. This task-resource affinity is provided by the SKOPE framework for all pairs of tasks and resources. For example, given $T_s^i(n) = \frac{10}{n}$, the run time of task i is 1 when ten nodes at site s are provisioned while the run time of task i is 10 when one node at site s is provisioned. To incorporate these task-resource affinity into the network flow models as shown in Figure 5, we have to set the values of C_s and d_i appropriately where C_s, denotes computation power at site s and d_i denotes the amount of compute resource demand of task i. Note that this capacity is used for a bandwidth of a link representing a compute resource in our network flow formulation [17]. Equation 1 is the task-resource affinity equation showing the relationship among $T_s^i(n)$, d_i, and C_s. So $\frac{d_i}{C_s}$ represents how fast a task demand, d_i, can be processed by compute resources at site s, C_s. This is analogous to a situation where d_i amount of water flows through a water pipe with capacity C_s.

$$T_s^i(n) = \frac{d_i}{C_s} \tag{1}$$

Note that C_s and d_i are relative values. To describe that task i takes 1 second at compute resource site s, we can assign either 100 or 10 to both of C_s and d_i. We can set C_s for a computation resource site with fewest computation resource to 100. Then we can get d_i and assign this to the corresponding task in the formulation. To compute C_m, when $m \neq s$, we should normalize C_m regarding the base case by Equation 2.

$$C_m = 100 \times \frac{T_s^i(n)}{T_m^i(n')} \tag{2}$$

We do the same to each task repetitively such that we assign different C to the edges connecting other tasks and resources in the auxiliary graph [17].

3.3.3. Cost optimization

In the context of grid and cloud computing, many heuristics for optimizing additional objectives such as the execution cost of a workflow as well as the makespan of a workflow have been proposed. Many of them are proposed newly from the scratch and some of them are extended from the existing heuristics such as heterogeneous-earliest-finish-time (HEFT).

Our algorithm is a linear programming based approach, which has a major advantage of extensibility over other heuristics. The cost of a workflow can be incorporated as a constraint or an objective in our linear programming formulations, which means there is no need to develop new heuristics or to put efforts in extending existing heuristics to adapt to new constraints.

4. Experimental Setup

In this section we introduce the application and computation sites used in our experiments.

4.1. Application Characteristics

Synthetic Workflow. We use a synthetic application consisting of various computation and memory intensive tasks. The computation intensive task is matrix multiplication while the memory intensive tasks are array summation, scale, and triad operations from the STREAM benchmarks [21]. The graph in Figure 6 shows the overall data flow of a synthetic workflow. The application consists of a total of 255 tasks.

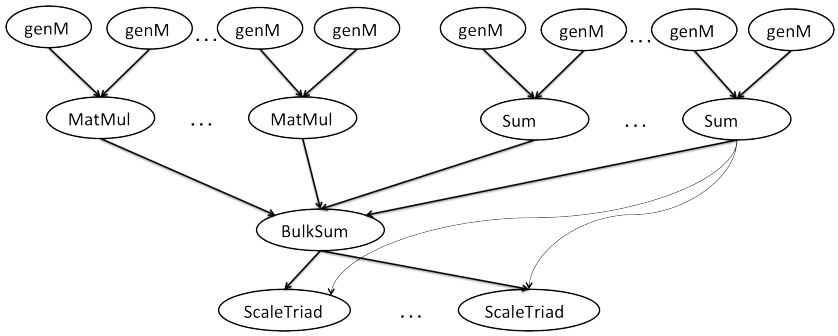


Figure 6: The synthetic application workflow representation

Image Reconstruction. The Advanced Photon Source (APS) at Argonne National Laboratory provides the Western Hemispheres most brilliant X-ray beams for research. Typically, during the experiments, the data generated at the beamlines is typically moved to a local HPC cluster for quasi real-time processing through a LAN. At the end of the experiments, raw and processed data is moved to the user's home institution typically over a WAN. The science workflows at APS is changing in nature rapidly as a result of double exponential growth in both detector resolution and experiment repetition rate. As data volumes increase, we see increased need to use remote computation facilities to process and analyze the data. For example, a near-real-time analysis of a few TB of APS data at a remote compute cluster at Pacific Northwest National Laboratory was done recently. In our experiments, we use a downsized APS application consisting of two tasks, a raw image reconstruction task and an image analysis task, which are both compute-intensive tasks and can be distributed among remote computation sites. through 2D slice images reconstructed from raw tomography data at computation sites in our experiments. We also assume a workflow in which ten datasets are generated by ten experiments performed at X-ray beamlines (note that there are more than 60 beamlines at APS) and will be processed using remote computation sites.

PowerGrid. The electrical power prices in a region are a result of combination of many stochastic and temporal factors, including variation in supply and demand due to market, social, and environmental factors. Evaluating the feasibility of future generation power grid networks and renewable energy sources requires modeling and simulation of this complex system. In particular, the power grid application described here is used to statistically infer the changes in the unit commitment prices with respect to small variations in random factors. The application involves running a stochastic model for a large number of elements generated via a three-level nested `foreach` loop. A numerical algorithm is run to compute lower and upper bounds, which converge for large enough samples. A moderate sample size of five samples can generate hundreds of thousands of tasks. Each task makes call to the Python-implemented sample generation and AMPL [22] models making it an interlanguage implementation spanning Python and AMPL interpreters, as depicted in Figure 7.

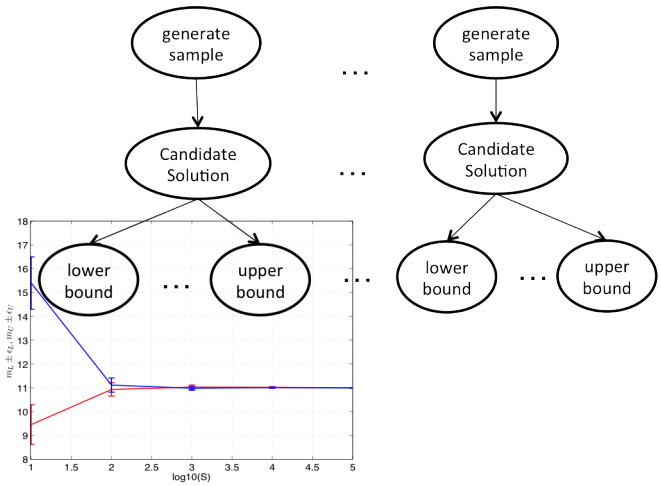


Figure 7: Electrical power price analysis application components: tasks and a plot showing convergence of upper and lower bounds for large sample sizes.

4.2. Cluster Sites

We used four execution sites–XSEDE's Stampede and Trestles clusters, Argonne's Laboratory Computing Resource Center (LCRC) cluster 'Blues' and University of Chicago Research Computing Center (RCC) cluster 'Midway' to evaluate our approach. A summarized characterization of these four sites is given in Table 2.

XSEDE (*www.xsede.org*) is an NSF-funded, national cyberinfrastructure comprising multiple large-scale computation systems on sites across the US. Stampede is one of the supercomputing systems offered by XSEDE. Stampede runs the SLURM scheduler for submitting user jobs. Similarly, Trestles is another supercomputing environment offered by XSEDE. It consists of 324 compute nodes and over 100 TFlop/s of peak performance. It employs a PBS based resource manager.

LCRC Blues (*www.lcrc.anl.gov/about/blues*) is a recently-acquired cluster available to science users at the Argonne National Laboratory. It comprises of 310 16-core nodes. Blues runs the PBS scheduler.

RCC Midway (*rcc.uchicago.edu*) is the University of Chicago Research Computing Center cluster supporting University-wide high-end computational needs. The cluster has multiple resource partitioning dedicated to specialized computing such as HPC, HTC and GPU computing and runs a SLURM batch queue scheduler.

4.3. Cloud sites

We used three of the most popular contemporary public cloud offerings: the Amazon AWS cloud, the Google Compute Engine and the Microsoft Azure cloud.

Amazon AWS is the oldest and most mature cloud provider service and offers many features for compute, network and storage management in the cloud. Google Compute Engine and Microsoft Azure are relatively newer offerings and provide similar but limited functionality compared to Amazon AWS (as of this writing). Both AWS and Google Compute Engine offer command line tools for cloud resource management and administration. This allows for fast and automated shutdown of instances when they are not used during the experiment.

We chose three distinct cloud offerings because of the following reasons:

1. Avoid vendor lock in from a single provider.
2. Demonstrate application adaptability to more than one cloud.
3. Aggregate multi-cloud resources to circumvent limitations on resources set by individual providers.
4. Diversify the types of middleware, hypervisors and resource providers, with the data centers being spread globally.

The aforementioned reasons also form the motivation for cloud based part of current work. As new infrastructure such as clouds mature, more and more applications will be ported to them. With the experiments we discuss in this work, we demonstrate an ability to readily port traditional applications to clouds via a versatile system that is well suited to both traditional clusters and clouds.

We used 10 instances from each of the clouds for our experiments. We selected medium sized Linux instances with 2 cores and 7 GB of memory each costing $0.14 per instance per hour. For the purpose of uniformity of our experiments, we did not use any of the provider specific load-balancing, or advanced network provisioning features. All the resources were chosen from the nearest zones from the US Midwest region.

5. Evaluation

In this section we present an evaluation of our approach. We use the 'synthetic', 'powergrid' and 'image reconstruction' application workflows encoded in Swift. We submit the application workflows to four sites (Blues, Midway,

Stampede and Trestles) from a remote machine (located outside of the network domains of the clusters), where all input data resides. We first gather data needed to build empirical performance models. To do so, we conduct a pilot run, where we allocate one node for each site and execute each task on each site, with different number of tasks per node. This measures the execution time of a task when there are other tasks running on other cores of the same node. Figure 8 shows one such set of measurements on each site for the powergrid workflow. This provides the node-level weak scaling trend for each task. We use quadratic curve fitting and generate a scaling model, which is incorporated into the workflow skeleton.

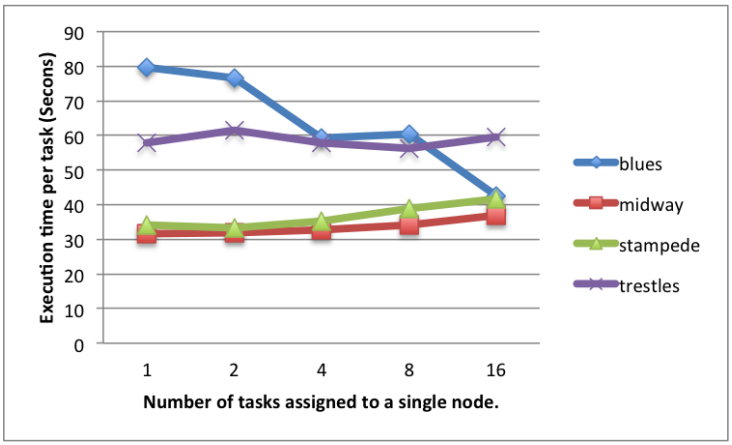


Figure 8: Profiled single-node weak scaling for the PowerGrid workflow on different sites.

In the first set of experiments, we use the default scheduler in Swift and merely tune a configuration parameter, "throttle", which controls the number of parallel jobs to send to sites and hence the number of parallel data transfers to sites. The default scheduler distributes an equal number of jobs to each of the execution sites.
In the second set of experiments, we alter the Swift configuration and distribute the jobs according to a schedule proposed by our scheme. We plot the execution trace log in order to generate a cumulative task completion plot as shown in Figures 9 to 12. The plot labeled 'baseline' and 'enhanced' show cumulative task completion with default and proposed schemes respectively. We notice an initial poor performance (especially Figure 10) as the schedule starts acquiring resources via local resource managers which rapidly

improves as the resources become available on remote sites. We note a sharp and steady increase in task completion for short tasks as seen in most of the synthetic workflow. On the other hand, a steps-like plot is seen in the case of compute intensive APS workflow of Figure 12. Similarly the poor performance of default scheme can be attributed to ramping up of jobs at sites with poor affinity for those jobs and/or a lower bandwidth to carry data to the site once the bandwidths are saturated by initial ramp-up.
It can be noted from the results in Figures 9 to 12, we achieve a shorter makespan with an informed schedule and save the effort for the users to fine tune the performance in the default Swift schedule. In complex and large real workflows interfaced with multiple remote sites, such fine tuning will consume a lot of time or even impossible. Our scheme achieves up to 60% improvement in makespan over the default scheme. This is possible because of the following key decisions taken by our schedule:

1. Accounts for both computational and data movement characteristics of tasks and capacities of resources.
2. Steers computation according to a proactive plan based on task-resource affinity.
3. Groups tasks into chunks to send out to sites ensuring load-balancing from the beginning of execution.

Figures 13 and 14 shows the activity share plot of each of the clouds during the execution of the synthetic workflow application over 10 instances of each of the three clouds (totalling 30 instances). The horizontal axis shows the time progression and the vertical axis shows the number of active tasks over each of the cloud. Each task in the application was modified such that it prints a timestamp as it begins and ends. An aggregate of the timestamps of all tasks from each of the 30 cloud instances were recorded and plotted. As shown in the Figure 13 the average time it takes for a default schedule is 53 minutes to complete over the three clouds resulting in a total cost of $3.85 (30 * 0.14*55/60). Note that since the schedule is default, all the instances form a single pool and it is not determined instances from which cloud needs to be shut down as there is no deterministic assurance as to which instances will not be used.
On the other hand, Figure 14 shows the same workflow over the same cloud resources with our schedule allocating a pre-determined number of tasks to each cloud. This divides the clouds into three distinct resource pools thus

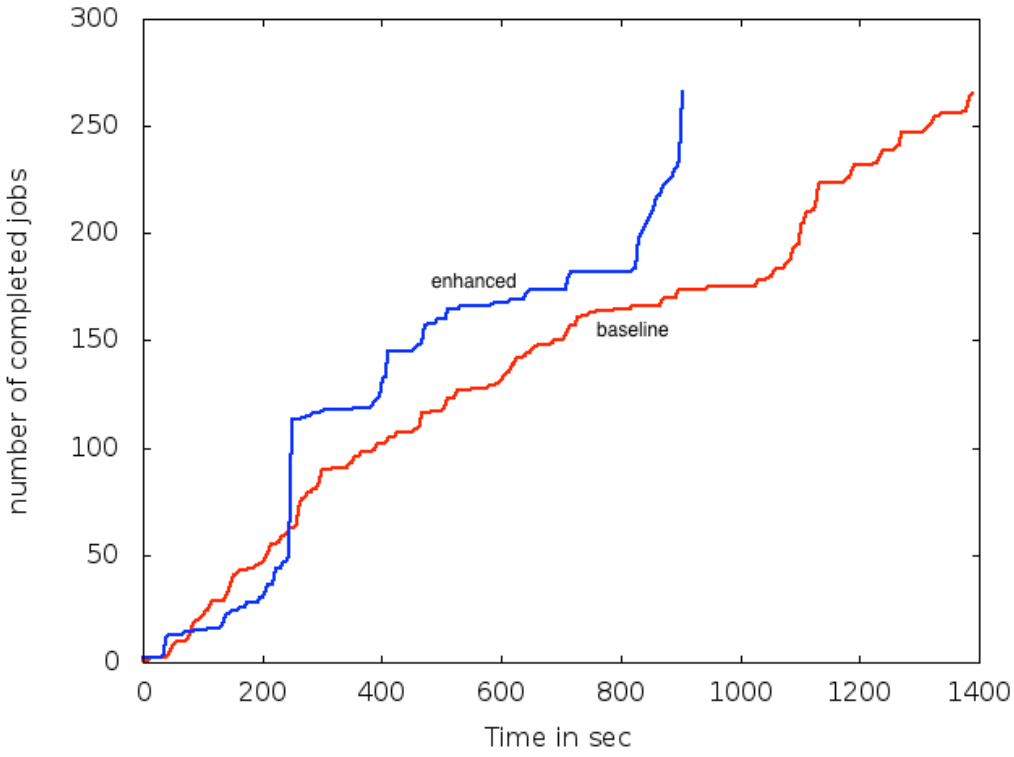


Figure 9: Synthetic workflow performance with a load of 255 tasks.

making it possible to shutdown each individual cloud instances as they are finished doing computation. This results in cost saving over and above the cost that are saved by a better schedule resulting in a faster time-to-completion. The workflow finishes in 46 minutes with the improved schedule resulting in a total cost of $3.22 (30 * 0.14*46/60) if computed conservatively. With an ability to shut down the Azure cloud at 24 minutes and AWS at 38 minutes saves additional amount resulting in the total cost to be $2.52 for this workflow. In practice, such workflows can be executed many times over for different datasets resulting in a significant savings over the time duration of an application lifecycle. A one time analysis of application and resources results in a schedule which can be used repeatedly.

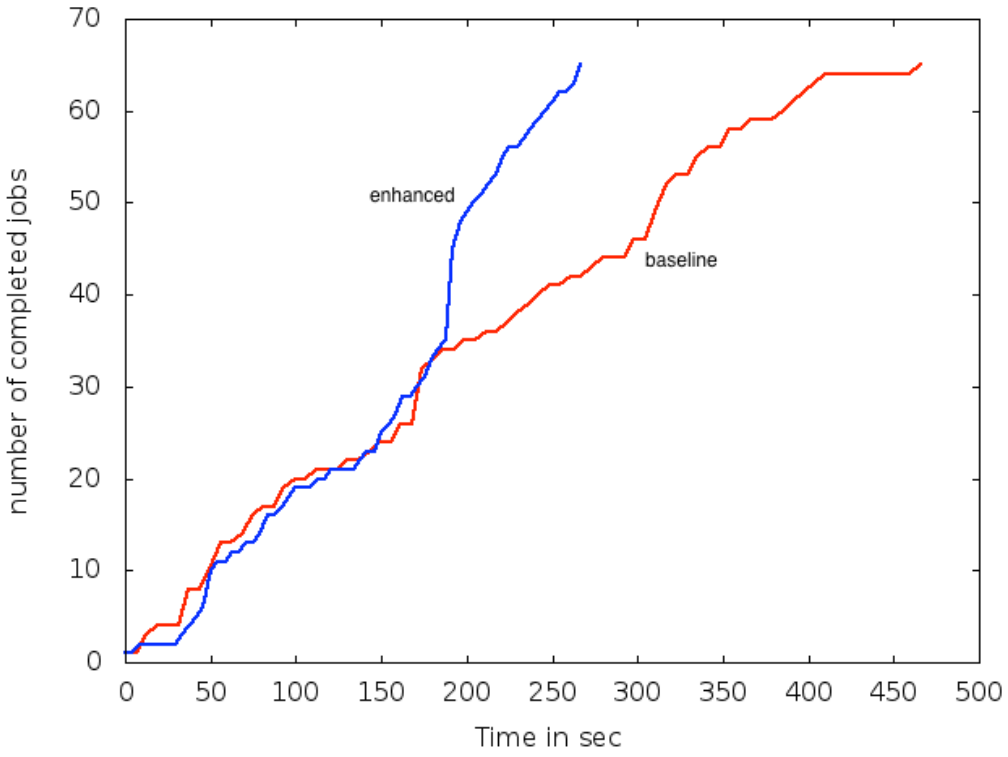


Figure 10: PowerGrid workflow performance for a small sample size resulting in 65 tasks.

6. Related work

Large scale applications have been shown to benefit significantly on heterogeneous systems [23] for data-intensive science [24] and under multiple sites infrastructure [25]. We demonstrate the value of these arguments in a realistic scenario. There has been much prior work on workflow management and performance modeling, which we discuss below.

6.1. Workflow Management

Some of the well-known workflow management systems include Pegasus [5, 26], HTCondor DAGMan [27, 28], Taverna [4], Triana [29] and makeflow [30]. Pegasus, in particular, keeps the separation of workflow description and system environment description. Pegasus Mapper takes an *abstract workflow* describing tasks' inputs, outputs, execution times, and data transfer times in XML format, which is given by users. The execution times are absolute

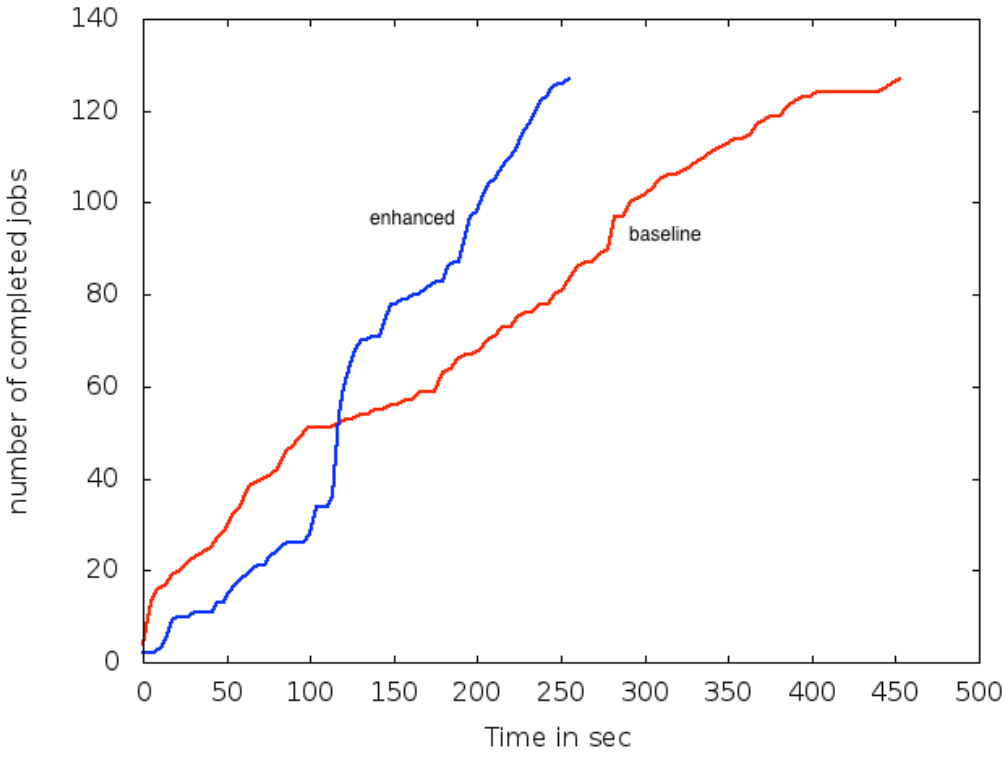


Figure 11: PowerGrid workflow performance for a large sample size resulting in 127 tasks.

values and are not given with regard to all available resources. Pegasus Mapper outputs an *executable workflow* describing the precedence among tasks and mapping of tasks and resources. At this phase, Pegasus Mapper deploys heuristics such as heterogeneous-earliest-finish-time (HEFT) [31], considering task execution time and/or data transfer times. HTCondor DAGMan, which is default Pegasus workflow execution engine, reads the *executable workflow*, and carries out best-efforts batch-mode scheduling together with remote execution engines such as PegasusLite and Pegasus MPI Cluster.
Our work differs from those previous work in two aspects. First, instead of having users manually measure execution time and data transfer overhead for individual tasks, we model tasks' execution time on heterogeneous resources and data transfer overhead through a workflow skeleton. The workflow skeleton, which is the equivalent of *abstract workflow* in Pegasus, has richer information (e.g., the control flow and data access patterns of a task) than *abstract*

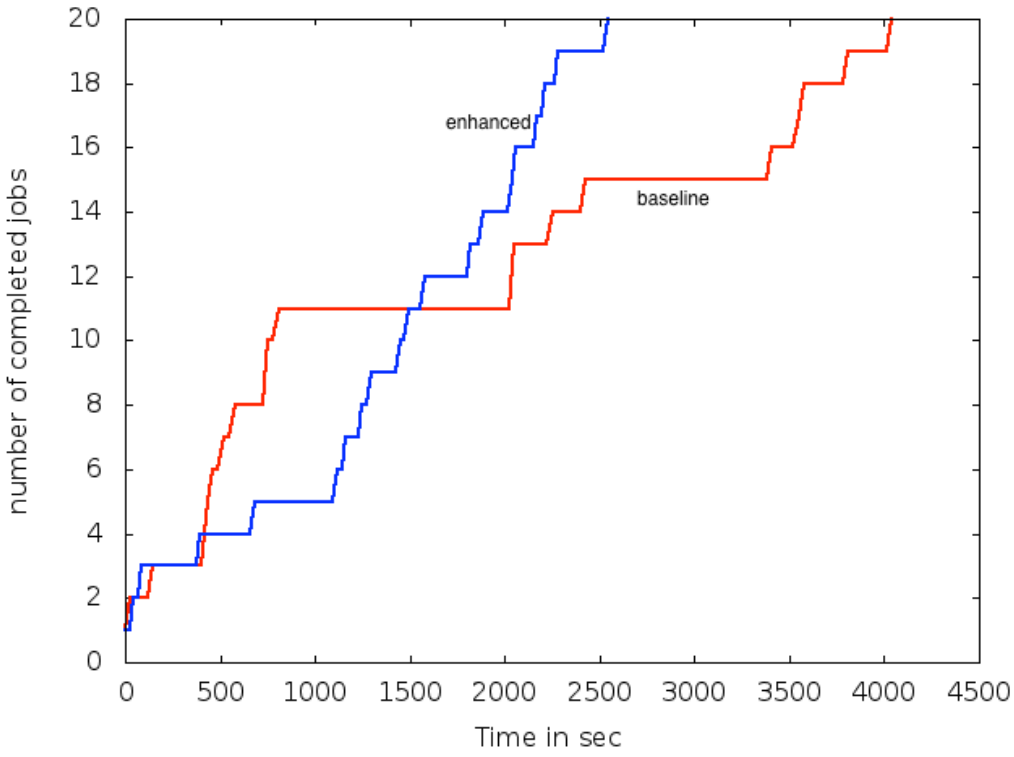


Figure 12: APS application performance with a load of 20 tasks.

workflow. In addition, SKOPE analyzes this workflow skeleton and output a job graph, which will be an input to our scheduler. The job graph is still the equivalent of *abstract workflow* in Pegasus. We can say that we introduce one more step, workflow skeleton, to incorporate task models into workflow management systems. As a result, we can project the overall time-to-solution without executing each possible schedule, which may not be feasible. Second, our scheduler takes into account the variance of task execution time over different sites and resource affinity among the tasks more accurately. Many variations of HEFT heuristic go through two phases. In the first phase, they determine ranks of tasks, the order of scheduling, based on averaged execution and communication time over all available resources. In the second phase, they evaluate each task on every resource in the determined order. This is not globally optimized method, and if a task show great variable in execution time among resources, the schedule by those heuristics can be far

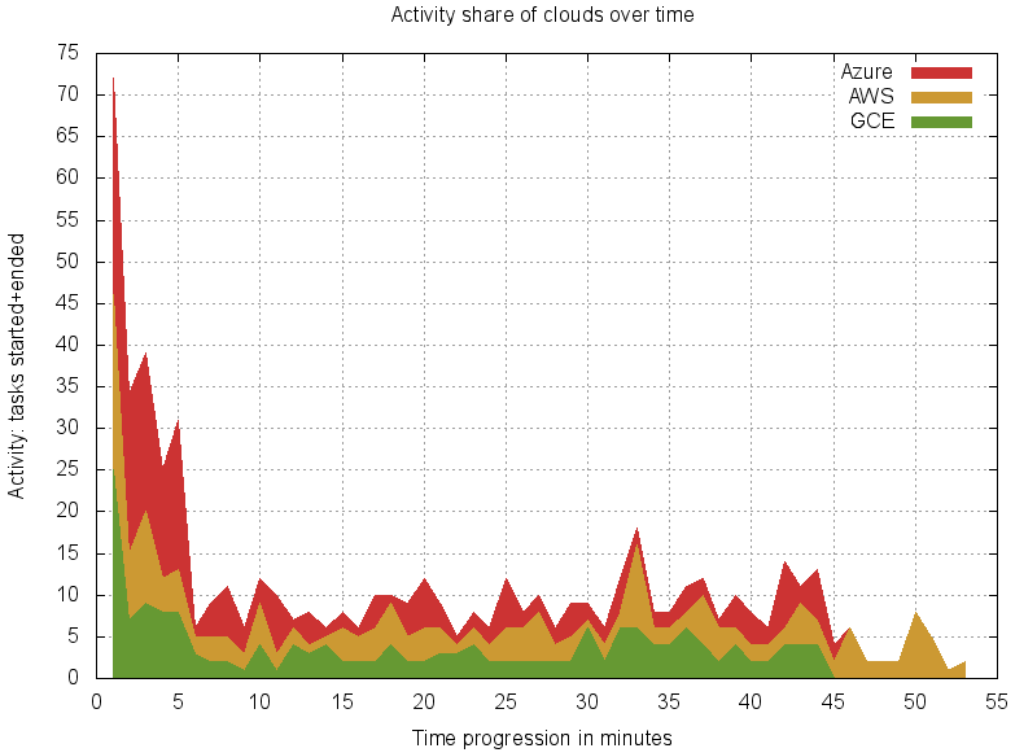


Figure 13: Synthetic application performance over clouds with default schedule.

from the optimal schedule. Our scheduling algorithm incorporates different execution times on heterogeneous resources into LP-based formulation, which is more globally optimized. Even though our LP-based algorithm may have some disadvantages when the size of workflow grows, it can be taken care of by task clustering and repetitive scheduling of partitioned small groups of a whole workflow. With the advent of cloud computing, existing workflow management systems including Pegasus [32] extended their support to cloud, and many workflow scheduling algorithms focusing on costs of cloud have been proposed [33]. We show that our approach based on task models can be extended to multi-cloud environments and our LP-based scheduling algorithm can easily incorporate cost- or deadline- constrained scheduling.
Simulation studies on multi-site resources have been done in the past such as Workflowsim [34] on generic wide-scale environments. While they provide detailed analysis of workflow deployment, simulations often take a significant

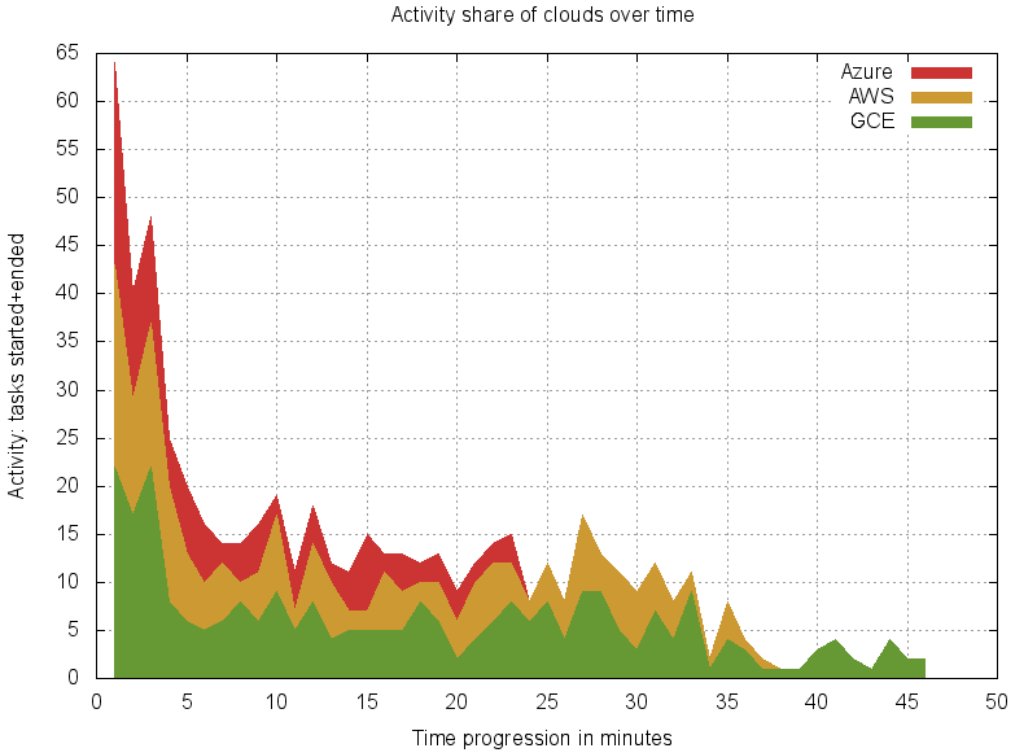


Figure 14: Synthetic application performance over clouds with improved schedule.

amount of time to emerge with and accurate picture of real-world scenarios and tend to lag behind in mapping the new architectures.

A related but dissimilar work on application skeleton have been recently undertaken by Zhang et. al. [35]. The work differs from ours in the sense that the goals of the work is mainly to address the requirement of generating varying workloads and dataflow patterns in the context of workflows.

Our approach based on workflow skeleton captures the application characteristics while offloading the execution responsibility to Swift which leads to a better division of responsibility. This approach makes our work distinct and a valuable contribution to parallel and distributed processing community.

6.2. Performance Modeling

Performance modeling has been widely used to analyze and optimize workload performance. SKOPE [3] provides a generic framework to model workload behavior. It has been used to explore code transformations when porting

computational kernels to emerging parallel hardware [36]. We apply the same principles in modeling kernels and parallel applications and extend SKOPE to model workflows. In particular, we propose workflow skeletons and use them to generate job graphs, which are in turn used to manage workflow. Application or hardware specific models have been used in many scenarios to study workload performance and to guide application optimizations [37, 38], where applications are usually run at a small scale to obtain knowledge about the execution overhead and their performance scaling. Snavely et al. developed a general modeling frameworks [39] that combine hardware signatures and application characteristics to determine the latency and overlapping of computation and data movement. An alternative approach uses black-box regression, where the workload is executed or simulated over systems with different settings, to establish connections between system parameters and run time performance [40, 41, 42]. All the above techniques target both computational kernels and full parallel applications.

7. Conclusion

In this paper, we propose a multi-site scheduling approach for scientific workflows using performance modeling. We introduce the notion of workflow skeletons and extended the SKOPE framework to capture, analyze and model the computational and data movement characteristics of workflows.
We develop a resource and job aware scheduling algorithm that utilizes the job graph generated using the workflow skeleton and the resource graph generated using the resource description. We incorporate our approach into Swift, a script-based parallel workflow execution framework. We evaluate using real-world applications in image reconstruction for an experimental light-source and for modeling of power-grid in a multi-site environment. We show that our approach reduces the total execution time of the workflows by as much as 60%. We demonstrate our approach using three application workflows over two distinct distributed, multisite computational environments: traditional clusters and clouds.

Acknowledgment

We thank Gail Pieper of Argonne for proofreading help. This work was supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, and the RAMSES project.

References

[1] K. Maheshwari, A. Rodriguez, D. Kelly, R. Madduri, J. Wozniak, M. Wilde, I. Foster, Extending the galaxy portal with parallel and distributed execution capability, in: SC'13 workshops, 2013.

[2] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, I. Foster, Swift: A language for distributed parallel scripting, Parallel Computing 37 (2011) 633–652.

[3] J. Meng, X. Wu, V. A. Morozov, V. Vishwanath, K. Kumaran, V. Taylor, C.-W. Lee, SKOPE: A Framework for Modeling and Exploring Workload Behavior, ANL Tech. report (2012).

[4] 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 (2004) 3045–3054.

[5] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, K. Kennedy, Task scheduling strategies for workflow-based applications in grids, in: IEEE International Symposium on Cluster Computing and the Grid, 2005. CCGrid 2005, volume 2, 2005, pp. 759–767 Vol. 2. doi:10.1109/CCGRID.2005.1558639.

[6] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, Y. Zhao, Scientific workflow management and the kepler system: Research articles, Concurr. Comput. : Pract. Exper. 18 (2006) 1039–1065.

[7] E. Bartocci, F. Corradini, E. Merelli, L. Scortichini, Biowms: a web-based workflow management system for bioinformatics., BMC Bioinformatics 8 (2007).

[8] P. Mhashilkar, Z. Miller, R. Kettimuthu, G. Garzoglio, B. Holzman, C. Weiss, X. Duan, L. Lacinski, End-to-end solution for integrated workload and data management using glideinwms and globus online, Journal of Physics: Conference Series 396 (2012) 032076.

[9] G. Sabin, R. Kettimuthu, A. Rajan, Scheduling of parallel jobs in a heterogeneous multi-site environment, in: in the Proc. of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes In Computer Science, 2003, pp. 87–104.

[10] V. Subramani, R. Kettimuthu, S. Srinivasan, P. Sadayappan, Distributed job scheduling on computational grids using multiple simultaneous requests, in: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, HPDC '02, IEEE Computer Society, Washington, DC, USA, 2002, pp. 359–. URL: http://dl.acm.org/citation.cfm?id=822086.823345.

[11] E. Huedo, R. S. Montero, I. M. Llorente, A framework for adaptive execution in grids, Softw. Pract. Exper. 34 (2004) 631–651.

[12] K. Maheshwari, K. Birman, J. Wozniak, D. V. Zandt, Evaluating cloud computing techniques for smart power grid design using parallel scripting, in: Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on, 2013.

[13] M. Hategan, J. Wozniak, K. Maheshwari, Coasters: uniform resource provisioning and access for scientific computing on clouds and grids, in: Proc. Utility and Cloud Computing, 2011.

[14] 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. URL: http://doi.acm.org/10.1145/2286996.2287000. doi:10.1145/2286996.2287000.

[15] O. Sinnen, Task scheduling for parallel systems, Wiley-Interscience, Hoboken N.J., 2007.

[16] R. Graham, Bounds for certain multiprocessing anomalies, Bell System Tech. Journal 45 (1966) 1563–1581.

[17] E.-S. Jung, S. Ranka, S. Sahni, Workflow scheduling in e-science networks, in: Computers and Communications (ISCC), 2011 IEEE Symposium on, 2011, pp. 432–437. doi:10.1109/ISCC.2011.5983875.

[18] W. Guo, W. Sun, Y. Jin, W. Hu, C. Qiao, Demonstration of joint resource scheduling in an optical network integrated computing environment [topics in optical communications], Communications Magazine, IEEE 48 (2010) 76–83.

[19] Y. Wang, Y. Jin, W. Guo, W. Sun, W. Hu, M.-Y. Wu, Joint scheduling for optical grid applications, Journal of Optical Networking 6 (2007) 304–318.

[20] P. Havlak, K. Kennedy, An implementation of interprocedural bounded regular section analysis, IEEE Trans. Parallel Distrib. Syst. 2 (1991).

[21] J. D. McCalpin, STREAM: Sustainable Memory Bandwidth in High Performance Computers, Technical Report, University of Virginia, 1991-2007.

[22] R. Fourer, D. M. Gay, B. Kernighan, Algorithms and model formulations in mathematical programming, Springer-Verlag New York, Inc., New York, NY, USA, 1989, pp. 150–151. URL: http://dl.acm.org/citation.cfm?id=107479.107491.

[23] R. Ramos-Pollan, F. Gonzalez, J. Caicedo, A. Cruz-Roa, J. Camargo, J. Vanegas, S. Perez, J. Bermeo, J. Otalora, P. Rozo, J. Arevalo, Bigs: A framework for large-scale image processing and analysis over distributed and heterogeneous computing resources, in: E-Science (e-Science), 2012 IEEE 8th International Conference on, 2012, pp. 1–8. doi:10.1109/eScience.2012.6404424.

[24] F. De Carlo, X. Xiao, K. Fezzaa, S. Wang, N. Schwarz, C. Jacobsen, N. Chawla, F. Fusseis, Data intensive science at synchrotron based 3d x-ray imaging facilities, in: E-Science (e-Science), 2012 IEEE 8th International Conference on, 2012, pp. 1–3. doi:10.1109/eScience.2012.6404468.

[25] M. Silberstein, Building an online domain-specific computing service over non-dedicated grid and cloud resources: The superlink-online experience, in: Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on, 2011, pp. 174–183. doi:10.1109/CCGrid.2011.46.

[26] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, K. Wenger, Pegasus, a workflow management system for science automation, Future Generation Computer Systems 46 (2015) 17–35.

[27] P. Couvares, T. Kosar, A. Roy, J. Weber, K. Wenger, Workflow management in condor, in: I. J. Taylor, E. Deelman, D. B. Gannon, M. Shields (Eds.), Workflows for e-Science, Springer London, London, 2007, pp. 357–375. URL: http://www.springerlink.com/content/r6un6312103m47t5/.

[28] HTCondor - Directed Acyclic Graph Manager, 2015. URL: http://research.cs.wisc.edu/htcondor/dagman/dagman.html.

[29] I. Taylor, M. Shields, I. Wang, A. Harrison, The triana workflow environment: Architecture and applications, in: I. Taylor, E. Deelman, D. Gannon, M. Shields (Eds.), Workflows for e-Science, Springer London, 2007, pp. 320–339. URL: http://dx.doi.org/10.1007/978-1-84628-757-2_20. doi:10.1007/978-1-84628-757-2_20.

[30] M. Albrecht, P. Donnelly, P. Bui, D. Thain, Makeflow: a portable abstraction for data intensive computing on clusters, clouds, and grids, in: Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies, SWEET '12, ACM, New York, NY, USA, 2012, pp. 1:1–1:13. URL: http://doi.acm.org/10.1145/2443416.2443417. doi:10.1145/2443416.2443417.

[31] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, Parallel and Distributed Systems, IEEE Transactions on 13 (2002) 260–274.

[32] E. Deelman, G. Juve, M. Malawski, J. Nabrzyski, Hosted science: managing computational workflows in the cloud, Parallel Processing Letters 23 (2013) 1340004.

[33] E. N. Alkhanak, S. P. Lee, S. U. R. Khan, Cost-aware challenges for workflow scheduling approaches in cloud computing environments: Taxonomy and opportunities, Future Generation Computer Systems (2015).

[34] W. Chen, E. Deelman, Workflowsim: A toolkit for simulating scientific workflows in distributed environments, in: E-Science (e-Science),

2012 IEEE 8th International Conference on, 2012, pp. 1–8. doi:10.1109/eScience.2012.6404430.

[35] Z. Zhang, D. Katz, Using application skeletons to improve escience infrastructure, in: e-Science (e-Science), 2014 IEEE 10th International Conference on, volume 1, 2014, pp. 111–118. doi:10.1109/eScience.2014.9.

[36] J. Meng, V. A. Morozov, V. Vishwanath, K. Kumaran, Dataflow-driven GPU performance projection for multi-kernel transformations, in: SC, 2012.

[37] J. W. Choi, A. Singh, R. W. Vuduc, Model-driven autotuning of sparse matrix-vector multiply on GPUs, in: PPoPP, 2010.

[38] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, W. Gropp, Modeling the performance of an algebraic multigrid cycle on HPC platforms, in: ICS, 2011.

[39] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, A. Purkayastha, A framework for performance modeling and prediction, in: SC, 2002.

[40] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, M. Schulz, A regression-based approach to scalability prediction, in: ICS, 2008.

[41] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, S. A. McKee, Methods of inference and learning for performance modeling of parallel applications, in: PPoPP, 2007.

[42] V. Taylor, X. Wu, R. Stevens, Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications, SIGMETRICS Perform. Eval. Rev. 30 (2003) 13–18.