# Model-Driven Multisite Workflow Scheduling

Ketan Maheshwari*, Eun-Sung Jung*, Jiayuan Meng*, Venkatram Vishwanath*, Rajkumar Kettimuthu*
*Argonne National Laboratory
Argonne, IL 60439

***Abstract*—Workflows continue to play an important role in expressing and deploying scientific applications. In recent years, a wide variety of computational sites have emerged with shared access to users. A user may not be able to complete a complex workflow at a single site. It is thus beneficial to run different tasks of a workflow on different sites. For such cases, judicious scheduling strategy is required in order to map tasks in the workflow to resources at multiple sites so that the workload is balanced among sites and the overhead is minimized in data transfer. The key challenge is that the data transfer rate among sites varies based on the network capacity and load. We propose a workflow scheduling technique that tackles the multi-site task distribution challenge by using data movement performance modeling. We applied this technique to schedule an earth observation science workflow over three sites. Executed via the Swift parallel scripting paradigm, we augmented its default schedule and improved the time-to-completion by up to 52%.**

## I. Introduction and Background

Large-scale applications often involve repetitive data- and compute-intensive experiments running over multiple remote sites. These sites vary widely in system characteristics including computation power, memory bandwidth, file system throughput, and performance of the networks. With such heterogeneity among sites, different tasks within the same workflow may perform better at different sites. Additionally, users are confronted with logistical constraints including allocation time and software compatibility.

The dynamics resulted by task-resource adaptation makes it difficult to identify an optimal schedule. To address this issue, one needs to study two factors: (a) how computation and data movement may change given a schedule, and (b) how this change would affect a workflow's overall time-to-solution. This knowledge, however, often remains unknown until the workflow is executed and profiled for a given schedule, making it almost impossible to explore and optimize for a large number of scheduling possibilities. Our specific contributions in this paper are three-fold: 1) Development of the notion of workflow skeletons to capture, explore, and analyze workflow behavior with regard to dynamics of computation and data movement; 2) Formulation of an algorithm to explore and propose an optimized schedule, according to the modeled workflow behavior; and 3) Integration of the workflow skeleton and the scheduling algorithm into a workflow deployment system.

An application is coded as Swift [1] scripts and run over multiple sites. We show that the proposed workflow schedule using our technique augments Swift's default schedule and saves up to 52% in time-to-solution. *Swift* [1] is an application-level scripting framework designed for composing ordinary programs into parallel applications. Applications encoded in Swift have been shown to execute on multiple computational sites via Swift coasters [2], [3] mechanism that implements the pilot jobs paradigm. Swift provides a simple reactive resource scheduling wherein, based on an initial "wave" of jobs, it records the per site job completion rate and adjusts the proportionate number of jobs to be sent to these sites. Even though Swift can use GridFTP [4], [5] for high-speed data movement, it does not take data transfer time into account while picking the sites for executing the tasks. Our approach takes data transfer time into account while picking the execution sites.

*SKOPE* (SKeleton framework for Performance Exploration) is a framework that helps users describe, model, and explore a workload's current and potential behavior [6]. It asks the user to provide a description, called *code skeleton*, that identifies a workload's performance behaviors including data flow, control flow, and computation intensity. The SKOPE back-end explores various transformations, synthesizes performance characteristics of each transformation, and evaluates the transformation with various types of hardware models.

## II. Skeleton-based Multisite Workflow Scheduling

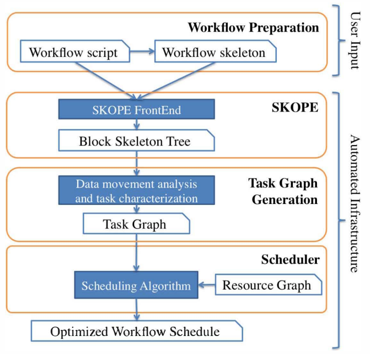


Fig. 1. Skeleton-based workflow scheduling framework

Figure 1 illustrates the overall steps involved in our technique. The user provides a script in Swift, in which a workflow is represented as a sequence of applications, each of which consumes or produces a number of files. In addition to the workflow description, the user needs to provide a workflow skeleton using our extended SKOPE to describe the overall performance properties of the workflow.

The skeleton models the workflow's behavior, including the application's computation resource requirements, as well as characteristics of the input and output files. The skeleton automatically generates a *task graph*, where a *task* refers to one or more applications grouped as a scheduling unit. The

TABLE I. EXECUTION SITES AND THEIR CHARACTERISTICS

| Site | CPU Cores | CPU Speed | Usable Memory per Node | Allocation | Remarks |
|---|---|---|---|---|---|
| LCRC Blues | 310X16=4960 | 2.60GHz | 62.90 GB | unlimited | Early access, 35 jobs cap |
| XSEDE Stampede | 6400X16=102400 | 2.70GHz | 31.31 GB | limited | 50 jobs cap |
| RCC Midway | 160X16=2560 | 2.60GHz | 32.00 GB | limited | Institute-wide access |

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

TABLE II. SYNTAX FOR WORKFLOW SKELETONS

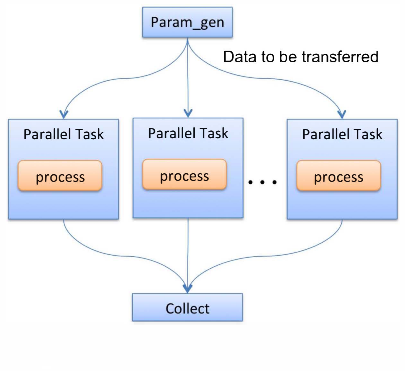
| | |
|---|---|
| **Macros and Data Declarations** | |
| File type and size (in KB) | :MyFile N |
| Constant definition | :symbol = expr |
| Array of files | :type array[N][M] |
| Variable def./assign | var = expr |
| Variable range | var_name=begin:end(exclusive):stride |
| **Control Flow Statements** | |
| 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) |

### A. *Workflow Skeleton and Task Graph*

Given a workflow, its skeleton summarizes the high-level semantics that relate to its performance behavior. The syntax of a workflow skeleton is summarized in Table II. In Figure 2(a), we show 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, application definitions, and the control and data flow among the applications. The size of each type of file are summarized in lines 3-4 of the skeleton. An example skeleton description of an application is demonstrated by lines 17-35 in Figure 2(b).

The skeleton is parsed by SKOPE 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 application 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. Given the high-level nature of workflows and the structural similarity between workflow scripts and skeletons, generating the workflow skeleton can be straightforward and may be automated in the future by a source-to-source translator.

Figure 3 illustrates the task graph generated from the workflow skeleton in Figure 2(b). Nodes refer to tasks and edges refer to data movements. 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.



(c) Task graph generated from the skeleton

Fig. 3. Task graph for the workflow shown in Figure 2(a).

### B. *Procedural Task Graph Generation*

Generating a task graph from a workflow skeleton involves three major steps. First, we obtain the data footprint for each task. Second, we construct the data flow among dependent tasks. Third, we derive the symbolic expression to express the execution time of a task over different systems.

A critical component of our technique is the data movement analysis, for which we apply array section analysis using bounded regular section (BRS) [7]. 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. 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 an application 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 $A[\langle r^l : r^u : r^s\rangle][\langle c^l : c^u : c^s\rangle]$, which is summarized by $\mathbb{P}(A[r][c], \mathbb{T}(r,c))$. To obtain the data footprint of a task, 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 task.

The resulting task graph is output in the form of an adjacency list which is next passed to the scheduler algorithm to generate an optimized mapping among the tasks and resources.

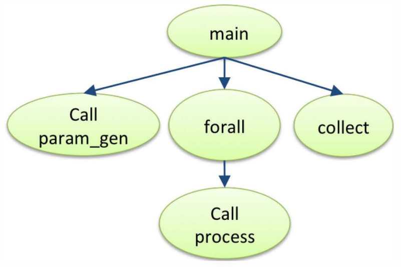
```
type file;
 # Files
int N = 165;
file inputs[N]
file outputs[N]
param_gen(inputs)
foreach n in ●:N
{
    outputs[n] = process(inputs[n]);
}
collect(outputs)

// function definitions are omitted
```

(a) Original workflow script

```
:site = 'stampede'
:N = 165
:InFile = 431 // file size in KB
:OutFile = 1600 // file size in KB
:InFile inputs[N]
:OutFile outputs[N]
def main()
{
  // produces input files
  call param_gen(inputs)
  forall n = ●:N
  { // execute parallel tasks
     call process(outputs[n], inputs[n])
  }
  call collect(outputs) // collect results
}
def process(output, input)
{
 // read from input file
 ld input
 // run time (sec)
 switch(site)
 {
   case 'stampede'
   {
     comp 9972
   }
   break
   case 'blues'
   {
     comp 12650
   }
 // write to output file
 st output
}
```

(b) Workflow skeleton



(c) Block Skeleton Tree for the main skeleton function

Fig. 2. Workflow script (a), skeleton (b), and the corresponding block skeleton tree (c) for a pedagogical workflow.

## C. Multisite Scheduling

We use a joint scheduling algorithm [8] that takes into account both compute resources and network paths. Our algorithm considers both resources holistically by converting a scheduling problem into a network flow problem. In order to schedule parallel jobs in a workflow while considering concurrently runnable jobs at a compute resource, a new notion of task-resource affinity has been devised by taking into account the number of concurrently runnable jobs at computation sites. Task clustering based on our algorithm is discussed later in this section.

Our scheduling algorithm also needs a resource graph which describes the underlying hardware architecture to generate an optimized schedule. 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.

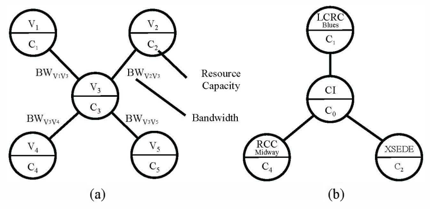


Fig. 4. (a) Resource graph model (b) Resource graph in our experiments.

Table III shows network bandwidth among our execution sites. The network bandwidths among them is measured by *iperf* benchmark tool. Figure 4 (b) is the resource graph corresponding to Table I. The bandwidth values in Table III are associated with edges in Figure 4.

TABLE III. DISK-TO-DISK BANDWIDTHS BETWEEN SITES

| **Site** | **Blues** | **Stampede** | **Midway** |
|---|---|---|---|
| **Submit Host** | 896 Mbit/s | 592.88 Mbit/s | 430 Mbit/s |

We next set the resource capacities, $C_n$, which represents computation power at site $n$, associated with nodes in Figure 4. $d_i$ denotes the amount of compute resource that task $i$ demands and $d_i$ is associated with task $i$. So $\frac{d_i}{C_s}$ represents how fast a task demand, $d_i$, can be processed by compute resources at site $s$, $C_s$. $C_s$ and $d_i$ are relative values. To describe that task $i$ takes 1 sec at compute resource site $s$, we can assign either 100 or 10 to both of $C_s$ and $d_i$. We have execution time of a task $i$ on site $s$, $t_s^i$ through performance modeling. Equation 1 is task-resource affinity equation where $CE_s$ is a random variable of the number of concurrently runnable tasks at site $s$. We define task-resource affinity as $\frac{t_s^i}{E(CE_s)}$. $\frac{t_s^i}{E(CE_s)}$ is the expected run time per task if multiple tasks are run at computation site $s$. For example, if 10 same parallel tasks are run at a site that can run 10 tasks at the same time, the expected run time per task is one tenth of the tasks's run time.

$$\frac{t_s^i}{E(CE_s)} = \frac{d_i}{C_s} \tag{1}$$

Equation 1 means task-resource affinity equals $\frac{d_i}{C_s}$, which is the runtime of task $i$ at site $s$. We can thus set $C_s$ for a computation resource site with fewest computation resource to 100. Then for each task, we can get $d_i$ and assign this to the corresponding task in the workflow. To compute $C_n$, when $n \neq s$, we can use Equation 2, where $T$ is a set of tasks. Since $C_n$ can be arbitrary values relative to $d_i$ according to Equation 1, we should normalize $C_n$ regarding the base case by Equation 2.

$$C_n = 100 \times \frac{1}{|T|} \sum_{i \in T} \frac{t_s^i}{t_n^i} \cdot \frac{E(CE_n)}{E(CE_s)} \tag{2}$$

Equation 2 averages affinities of tasks to resources. We can easily extend our model such that resource affinity per task is considered. For instance, while $t^1$ can be executed two times faster on site 1 than on site 2, $t^2$ may have similar execution times regardless of sites. We can define $d_s^i$ representing the demand of task $i$ at site $s$ so that we can assign different demands of tasks per each resource to the edges of the auxiliary graph. [8]

The task graph in Figure 2 (c) has 634 parallel tasks, which could result in much higher execution time of the scheduling algorithms. In this paper, we partition the parallel tasks into

10 groups with same number of tasks, and use this reduced task graph for scheduling.

## III. Experimental Setup

We use a mock application–MODIS (*modis.gsfc.nasa.gov*), derived from NASA's MODIS (Moderate Resolution Imaging Spectroradiometer). The workflow model for this application is depicted Figure 5 shows the workflow model with data and job numbers for each computational stage.

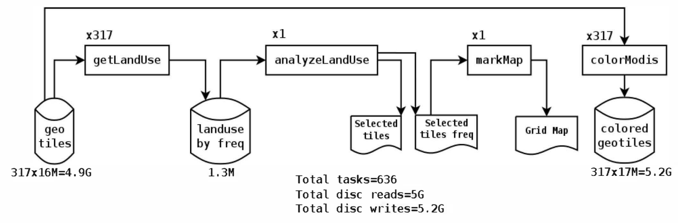


Fig. 5. MODIS application workflow depiction.

We selected three execution sites–XSEDE Stampede (*www.xsede.org*), LCRC (Laboratory Computing Resource Center) Blues (*www.lcrc.anl.gov/about/blues*) and RCC (Research Computing Center) Midway (*rcc.uchicago.edu*) to demonstrate our approach.

## IV. Evaluation

In this section we present an evaluation of our approach. We use the MODIS application workflow encoded in Swift for these evaluations. We chose 16 cores on each of the three clusters (Blues, Midway and Stampede) resulting in a capability of running 48 application jobs in parallel across these clusters. We use a remote machine for submissions. The input data is stored on the disk on this remote machine. We ran the application on individual sites to measure the makespan time on each site. This involves the total amount of time spent in data movement, execution, site-scheduler overhead and Swift's startup and shutdown overhead. Figure 6(a) shows the application makespan for each site. The least execution time on Blues can be attributed to a higher bandwidth from submit host and the lightly loaded queues.

We then evaluate the workflow performance over multiple 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. Note that in figure 6(b) higher "throttle" results in larger makespan. This is because higher "throttle" value results in more parallel jobs and thus more input data files in each batch sent to the execution sites.

In the second set of experiments, we alter the Swift script and distribute the jobs according to a schedule proposed by our scheme. The first such schedule, shown by the bar labeled 'sched1' takes the data movement into account assigns 256, 124 and 256 jobs to Stampede, Midway and Blues respectively. The second proposed schedule takes the difference in job execution time of 'landuse' and 'colormodis' into account and assigns 124, 256 and 256 jobs to Stampede, Midway and Blues respectively. Based on the resource description, our scheme automatically picks the optimal "throttle" value. Note that from the results in figure 6, we achieve a minimum makespan with an informed schedule and saving the effort of fine tuning with throttle changes. Our scheme achieves a 52% improvement in makespan over the default scheme ('th:123' in the Figure) and a 10% improvement over the best performance obtained with manual tuning ('th:48' in the Figure).

## V. Related work

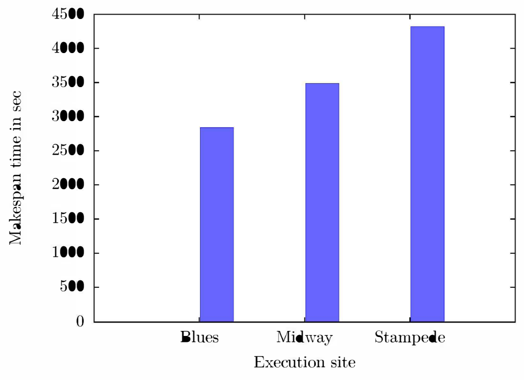
Large scale applications have been shown to benefit significantly on heterogeneous systems [9] for data-intensive science [10] and under multiple sites infrastructure [11]. There has also been much prior work on workflow management and performance modeling, which we discuss below.

Some of the well-known workflow management systems include Condor DAGMan [12], Pegasus [13] and makeflow [14]. Condor DAGMan provides a minimal set of keywords for directed acyclic graph (DAG)-based workflows. The workflow model of Condor DAGMan does not require additional information other than task precedence requirements given by a DAG. Pegasus requires task execution time information related to each task in a workflow. Swift provides a rich set of keywords for parallel task execution. Differences among workflow management systems result in different scheduling capabilities. HEFT or other heuristics use averaged execution times of a task over every possible resources or try earliest/latest completion task first approach. These heuristics do not consider the resource affinity per task effectively.

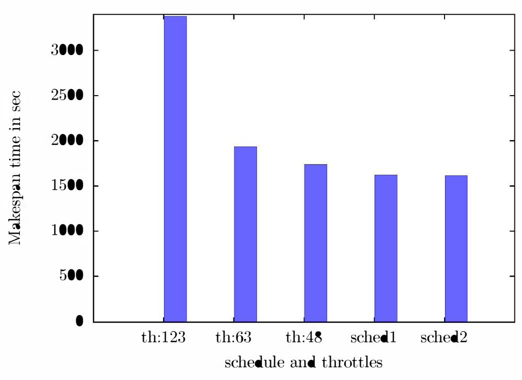
Simulation studies on multi-site resources have been done in the past such as Workflowsim [15] on generic wide-scale environments. While they provide detailed analysis of workflow deployment, simulations take a significant amount of time. Our work models the high level behavior of workflows so that the scheduler can suggest an optimized schedule online when deploying a workflow.

Overall, 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 e-science community.

Performance modeling has been widely used to analyze and optimize workload performance. Application or hardware specific models have been used in many scenarios to study workload performance and to guide application optimizations [16], 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 [17] 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 [18], [19]. SKOPE [6] provides a generic framework to model workload behavior. It has been used to explore code transformations when porting computational kernels to emerging parallel hardware [20]. We apply the same principles in modeling kernels and parallel applications and



(a) Makespan time of application execution on individual resources



(b) Performance on combined resources

Fig. 6. Makespan times of MODIS application execution

extend SKOPE to model workflows. In particular, we propose workflow skeletons and use that to generate task graphs, which are in turn used to manage workflow.

## VI. Conclusion

In this paper, we proposed a multi-site scheduling approach for scientific workflows using performance modeling. We introduced the notion of workflow skeletons and extended the SKOPE framework to capture, analyze and model the computational and data movement characteristics of workflows. We developed a resource and task aware scheduling algorithm that utilizes the task graph generated using the workflow skeleton and the resource graph generated using the resource description. We incorporated our approach into Swift, a script-based workflow framework and showed that our approach can improve the total execution time of the workflows by as much as 52%.

## Acknowledgments

We thank our colleagues David Kelly, John Valdes and Gail Pieper for technical and proofreading help. This work was supported in part by the U.S. Department of Energy, Office of Science, ASCR, under Contract DE-AC02-06CH11357.

## References

[1] M. Wilde et. al., "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, pp. 633–652, 2011.

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

[3] K. Maheshwari et. al., "Evaluating cloud computing techniques for smart power grid design using parallel scripting," in *CCGrid, 2013 13th IEEE/ACM International Symposium on*, 2013.

[4] B. Allcock et. al., "The Globus striped GridFTP framework and server," in *SC'2005*, 2005.

[5] R. Kettimuthu et. al., "Instant gridftp." in *IPDPS Workshops*. IEEE Computer Society, 2012, pp. 1104–1112.

[6] J. Meng et. al., "SKOPE: A Framework for Modeling and Exploring Workload Behavior," *ANL Tech. report*, 2012.

[7] P. Havlak and K. Kennedy, "An implementation of interprocedural bounded regular section analysis," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, 1991.

[8] E.-S. Jung, S. Ranka, and S. Sahni, "Workflow scheduling in e-science networks," in *Computers and Communications (ISCC), 2011 IEEE Symposium on*, 2011, pp. 432–437.

[9] R. Ramos-Pollan, F. Gonzalez, J. Caicedo, A. Cruz-Roa, J. Camargo, J. Vanegas, S. Perez, J. Bermeo, J. Otalora, P. Rozo, and 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.

[10] F. De Carlo, X. Xiao, K. Fezzaa, S. Wang, N. Schwarz, C. Jacobsen, N. Chawla, and 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.

[11] 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.

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

[13] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy, "Task scheduling strategies for workflow-based applications in grids," in *IEEE International Symposium on Cluster Computing and the Grid, 2005. CCGrid 2005*, vol. 2, 2005, pp. 759–767 Vol. 2.

[14] M. Albrecht, P. Donnelly, P. Bui, and 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*, ser. SWEET '12. New York, NY, USA: ACM, 2012, pp. 1:1–1:13. [Online]. Available: http://doi.acm.org/10.1145/2443416.2443417

[15] W. Chen and 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.

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

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

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

[19] V. Taylor, X. Wu, and R. Stevens, "Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications," *SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 4, pp. 13–18, Mar. 2003.

[20] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram, "GROPHECY: GPU performance projection from CPU code skeletons," in *SC*, 2011.