# DEWE: A Framework for Distributed Elastic Scientific Workflow Execution

**Luke M. Leslie**[1] **Chiaki Sato**[2] **Young Choon Lee**[2] **Qingye Jiang**[2]
**Albert Y. Zomaya**[2]

[1] Department of Computer Science,
University of Illinois at Urbana-Champaign,
Champaign, IL 61801, USA.
Email: lmlesli2@illinois.edu

[2] Centre for Distributed and High Performance Computing,
School of Information Technologies,
University of Sydney,
NSW 2006, Australia.
Emails: csat9577@uni.sydney.edu.au, qjiang@ieee.org {young.lee, albert.zomaya}@sydney.edu.au

**Abstract**

As cloud computing is increasingly adopted, the open, on-demand nature of public clouds makes explicit consideration of the underlying environment highly advantageous in terms of decreasing cost and increasing elasticity. In this paper, we address the exploitation of such cloud capabilities and present DEWE, a framework for the distributed, elastic execution of scientific workflows. DEWE is designed to be easily extensible and customizable, and to provide a simple interface to automate deployment and elasticity in public clouds. Using Montage, an astronomical image mosaic engine, as a case study, and Amazon Web Services (AWS) as the cloud environment, we demonstrate the benefits DEWE can provide to scientists seeking to design job scheduling, data management, and resource allocation strategies with potentially unlimited on-demand resources at hand. Further, DEWE's visualization tool much leverages the analysis and evaluation of those strategies.

*Keywords:* resource management; cloud computing; scientific workflows; resource efficiency; scheduling

## 1 Introduction

Cloud computing offers users access to near-infinite computational resources in an on-demand, open manner. Scientific workflows (such as Montage (1; 2), CyberShake (3; 4), LIGO (5; 6), Epigenomics (7) and SIPHT (8)) in particular can take great advantage of abundant resources as they are mostly resource-intensive and they scale well with the number of resources (resource capacity). However, the efficient use of such resource abundance for scientific workflows often requires specialist skills and tools (e.g., (9; 10; 11; 12)) due to the structural complexity and data dependency of these workflows (Figure 1).

Frameworks and management systems for distributed scientific workflow execution, such as Pegasus (10), have historically focused on providing independence from the underlying execution environment, with many systems instead relying on middleware such as Apache Mesos to shape the underlying environment and share resources among applications. However, IaaS (Infrastructure as a Service) cloud providers such as AWS provide users with the tools to not only assemble large compute clusters on the fly, and with pay-as-you-go rates, but also to shape and resize these clusters during execution through calls to various provider-specific APIs and SDKs. While providing independence is useful in handling the diversity of private systems, the new, open computing paradigm offered by the cloud, coupled with the increasingly large data sizes and corresponding required computing resources of modern scientific applications, provides an appealing alternative. Allowing the user to design algorithms that modify the underlying execution environment to fit usage requirements is a powerful strategy in terms of both performance and cost-efficiency.

Another problem inherent in current research on scientific workflow execution revolves around the absence of a single, easy-to-deploy framework on which to design, deploy, debug, and test experimental algorithms (e.g., for job scheduling). Testing new strategies on heavyweight systems and with different workflow applications can be a time-consuming process. As data sets grow in size and complexity, scientists without the resources to maintain these systems must instead seek out alternatives.

In this paper, we present DEWE[1], a framework designed for the distributed, elastic execution of scientific workflows. DEWE abstracts away underlying networking functionality, asynchronously disperses jobs across a dynamic pool of workers, and provides a simple interface to automate elasticity when deployed in Amazon EC2. DEWE is designed with extensibility, customizability, and ease-of-deployment in mind, and to enable research in workflow scheduling algorithms that are able to shape and expand their environment during runtime. In comparison with existing workflow execution systems, such as Pegasus, Kepler (11), and Apache Airavata (https://airavata.apache.org/), DEWE:

- Is lightweight and highly extensible, allowing scientists to quickly modify and test components, such as the implementation of experimental job-scheduling, data management, and fault-tolerance strategies.



[1] DEWE including its source code and visualization web service is available from https://bitbucket.org/lleslie/dwf/wiki/Home.


Figure 1: An example Montage (an astronomical image mosaic engine) workflow. Job names and their execution times, and precedence constraints (dictated by data dependencies) are shown in vertices and along edges, respectively.

- Aims to expose and shape the underlying execution environment. With a simple function call, DEWE can lease an EC2 instance that will automatically add itself to the worker pool.
- Has an asynchronous design that provides high scalability, greatly increasing the feasibility of resource addition and removal during execution, and the speed and efficiency of file-dependency transfers.
- Provides a visualization tool as a post-processing aid. It facilitates analysis of workflow execution, including resource utilization and performance bottleneck, and evaluation of resource/data management and job scheduling strategies.

Our evaluation of DEWE uses the data-intensive scientific workflow application Montage (1), an astronomical image mosaic engine, as a case study. We override the default (random) job-scheduling algorithm with algorithms designed to incorporate available data and abstractions, such as file locality and the various workloads on each node. Using a 6.0 degree Montage workflow[2] on clusters in Amazon EC2 comprised of up to 15 *m1.xlarge* instances, we demonstrate the ability to quickly and efficiently develop, test, and compare experimental algorithms.

The remainder of this paper is organized as follows. Section 2 describes scientific workflows and discusses related work. Section 3 provides a system overview of DEWE, including descriptions of nodes, components, abstractions, and methods to shape the underlying environment. We present our results using DEWE in Section 4, and our conclusions are drawn in Section 5.

[2] A 6.0-degree Montage workflow creates a 6-by-6 degree square mosaic centered at a particular region of the sky (e.g., M16). The number of jobs in each workflow increases with the number of degrees.

## 2 Background and Related Work

In this section, we describe scientific workflows and provide a brief review of related work on workflow scheduling and execution frameworks.

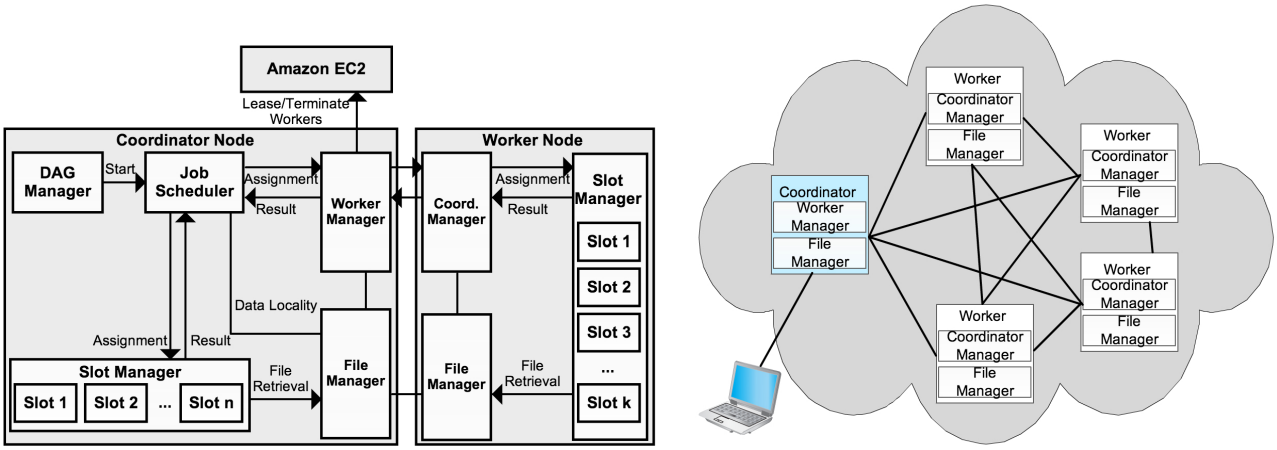
### 2.1 Scientific Workflows

Applications in science and engineering are becoming increasingly large-scale and complex. These applications are often amalgamated in the form of workflows (such as Montage (1), CyberShake (3; 4), LIGO (5; 6), Epigenomics (7) and SIPHT (8)) with a large number of composite software modules and services, often numbering in the hundreds or thousands.

More formally, a scientific workflow consists of a set of precedence-constrained jobs represented by a directed acyclic graph (DAG), $G = \langle V, E \rangle$ comprising a set $V$ of vertices, $V = \{v_0, v_1, ..., v_n\}$, and a set $E$ of edges, each of which connects two jobs. The graph in Figure 1 depicts a Montage workflow with vertices for jobs and edges for data dependences or precedence constraints. Sibling vertices/jobs are most likely to run in parallel and get assigned onto different resources, i.e., they are executed in a distributed manner. A job is regarded as ready to run (or simply as a 'ready job'). The readiness of job $v_i$ is determined by its predecessors (parent jobs), more specifically the one that completes the communication at the latest time.

The completion time of a workflow application is denoted as *makespan*, which is defined as the finish time of the exit job (or the left node in the DAG).

The longest path of a graph is the critical path (*CP*, shown with jobs connected through red edges in Figure 1). For a given DAG, the critical path determines the theoretical min makespan and the critical path length (CPL) is defined as the summation of computation costs of jobs in CP.




(a) Component interaction during execution in EC2.

(b) Network communication on a private cloud.

Figure 2: Inter/intra-node interaction during execution.

### 2.2 Workflow Scheduling

The execution of scientific workflows is typically planned and coordinated by schedulers/resource managers (e.g., (9; 13)) particularly with distributed resources. At the core of these schedulers are scheduling algorithms/policies.

Traditionally, workflow scheduling focuses on the minimization of makespan (i.e., high performance) within tightly coupled computer systems like compute clusters with an exception of grids. Various scheduling approaches including list scheduling and clustering are exploited, e.g., (14; 15). Critical-path base scheduling is one particularly popular approach to makespan minimization (16; 17). Clustering-based scheduling is another approach getting much attention in the recent past with the emergence of many data-intensive workflows, such as Montage (15).

More recently with the adoption and prevalence of cloud computing, the trade-off between costs and performance has been extensively studied (18; 19; 20). Most works on workflow scheduling in clouds study the elasticity of cloud resources, i.e., dynamic resource provisioning for cost minimization in particular. However, such dynamic provisioning still remains mostly in the initial workflow deployment; that is, the full elasticity capability of public clouds including that during the execution of workflows is not well explotied with scientific workflows.

### 2.3 Workflow Execution Frameworks

As scientific workflows are becoming increasingly large-scale and complex, their distributed execution across multiple resources is far beyond an average task involving workflow composition, resource provisioning/reservation, job scheduling, fault tolerance and data staging. Coinciding with this increase in scale and complexity have been efforts on developing workflow execution frameworks including Condor (DAGMan in particular) (9), Kepler (11), Pegasus (10), Taverna (21), Trident (22), and more recently Apache Tez (http://tez.incubator.apache.org/) and Apache Airavata. These frameworks tend to be heavyweight and inaccessible to scientists who lack dedicated hardware and support staff, e.g., Condor and Pegasus. Moreover, many of these frameworks, such as Kepler, Taverna and Trident have focused on providing independence from the underlying execution environment. DEWE on the other hand is designed specifically to take full advantage of elasticity of cloud with the capability of dynamically expanding and shrinking the resource pool (or the pool of workers).

An interesting approach pursued recently is a hybrid workflow system (23). B. Plale et al. in (23) weave multiple workflow execution frameworks—including Trident, Kepler and Apache ODE (http://ode.apache.org/)—to support a wider range of workflows.

## 3 System Overview

In this section, we present a system overview of DEWE, listing and describing each component and node, and their roles. We also describe the methods used to expose the environment to the user and allow for the design of algorithms that shape this environment during runtime.

### 3.1 Components

Nodes in DEWE are divided into two categories: **Workers**, responsible for job execution and data sharing/replication, and **Coordinators**, responsible for workflow creation, job scheduling, and data assignment, as well as job execution. Each DEWE application contains only a single Coordinator and potentially many Workers.

Individual job execution takes place through a **slot** model. Each slot represents an isolated portion of the available system resources; by default, this resource is identified as the number of (virtual) CPUs. The Coordinator Node contains components for DAG creation (**DAG Manager**), job scheduling and assignment (**Job Scheduler**), slot management and job execution (**Slot Manager**), file management (**File Manager**), and Worker node management (**Worker Manager**) (see Figure 2a). The main functionality of each component can be overridden or extended by the user as desired. Worker nodes contain the Slot Manager and File Manager modules, as well as a component for interactions with the Coordinator node (**Coordinator Manager**). An overview of network communications on a private cloud is given in Figure 2b.

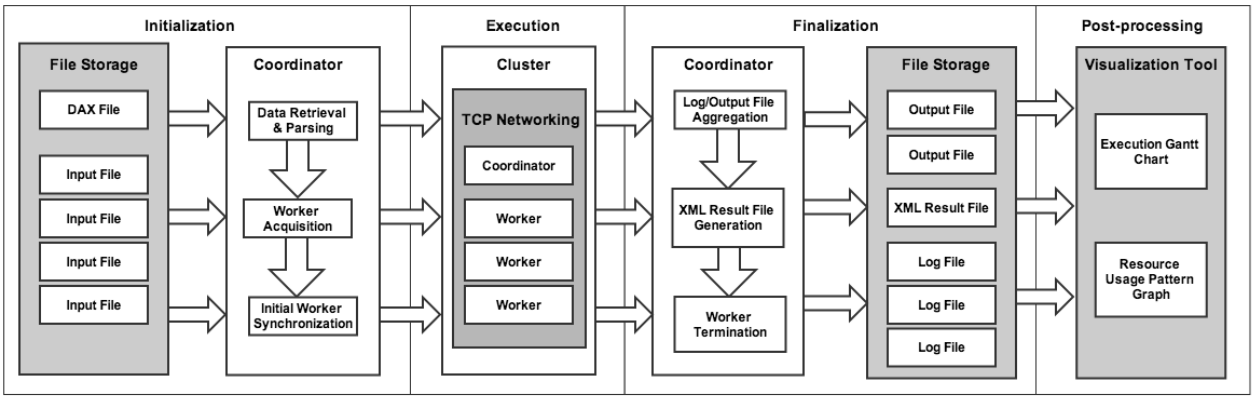



Figure 3: High-level system overview of DEWE.

The Coordinator node's Worker Manager exposes the queued and executing jobs of Worker nodes for use in job scheduling strategies, etc. Similarly, the Slot Manager exposes the queued and executing jobs of the local node. The Worker Manager also keeps track of heartbeats sent periodically by Worker nodes (every 5 seconds by default), and performs an overridable action when a Worker node misses some number of consecutive heartbeats. The Coordinator node's File Manager provides up-to-date file information, such as size and the nodes on which the file is available for retrieval. The main functionality of the Job Scheduler involves, upon receipt of a newly completed job assignment, the determination of which (if any) of the job's children now have all dependencies satisfied. Any jobs newly satisfying this criteria may be assigned to a specific node-slot for execution, and the assignment of file dependency locations are determined through communication with the File Manager, which also assumes the role of a file server.

### 3.2 Abstractions

DEWE provides abstractions during the workflow process for job assignment and data dependencies. As mentioned in the preceding section, a scientific workflow can be represented by a DAG, $G = \langle V, E \rangle$, where $V$ represents the set of jobs. When jobs are assigned to a particular slot on a chosen node, a **Job Assignment**, $a = \langle v, n, s, r \rangle$ is constructed, where $v \in V$ is the job, $n \in N$ is the chosen node, $s \in S_n$ is the slot chosen from those in the specified node, and $r$ is the resulting runtime of the job in the chosen assignment (initially 0).

The Coordinator node also maintains abstractions of each Worker node, called a **Worker Rep**, in the Worker Manager module. These abstractions include up-to-date representations of the slots on the worker, called a **Slot Rep**. Each worker is represented as $w = \langle I, S, F \rangle$, where $I$ is a set of identifying information (e.g., IP address, EC2 instance ID, etc.), $S = \{s_1, \ldots, s_k\}$ is the set of all slot representations for the worker, and $F = \{f_1, \ldots, f_m\}$ is the set of files locally available on the worker. Each slot is represented by $s = \langle FQ, EQ \rangle$, where $FQ$ is a FIFO queue of job assignments waiting to have input files fetched from other nodes, and $EQ$ is a FIFO queue of assignments ready for execution respectively.

### 3.3 Workflow Execution

The overall process of initialization, workflow execution, finalization is illustrated in Figure 3 and proceeds as follows. At the start of each workflow, the Coordinator retrieves and extracts the initial input files from the location specified by the user (e.g., on the local disk, or from Amazon S3 if DEWE is deployed on EC2). The Coordinator then creates the DAG representing the workflow from these initial input files. At startup, Worker nodes connect to the Coordinator node. For both Coordinator and Worker nodes, $n$ slots are considered for job execution, where $n$ equals the number of CPUs in the node. The Coordinator synchronizes with any initial workers if requested by the user, and then begins asynchronously assigning jobs with all dependencies satisfied to slots on available nodes. After creation, the assignment is sent to the specified node and placed into a slot-specific queue ($FQ$) for input file retrieval. Input files are retrieved from assigned nodes (if not local); this assignment is designed to be extended, and currently randomly selects a node from those containing the file. After all input files are local, the assignment is popped from $FQ$ and placed into another slot-specific queue ($EQ$) for execution on the assigned slot. After successful completion of the assignment, the Coordinator is notified and loops through the children of the completed assignment, assigning those which now have all dependencies satisfied. The process is repeated until the workflow is finished. Upon completion, an XML file (`results.xml`) describing job assignments (e.g., the assigned node, etc.), execution times, and file sizes is generated, and log file from each Worker are retrieved by the Coordinator.

### 3.4 Post-processing: Visualizing Workflow Execution

DEWE's visualization tool as a post-processing analysis facility takes the XML execution log file and provides two visual aids, i.e., detailed workflow execution visualization (Figure 4) and resource usage pattern visualization. The online web service for this visualization is available from the project web site.

For a large-scale workflow with thousands of jobs, scientists can only calculate the overall resource utilization such as the total or percentage amount of time being used for computation and data staging, but the details about the scheduling and execution are often overwhelmed by the size of the output. DEWE's visualization facility much leverages the analysis and evaluation on the resource utilization status of all worker nodes during the whole workspan. Such visualization enabled by DEWE provides insights into the idling time slots in the computing environment, which will help researchers design better workflow scheduling algorithms or resource allocation strategies.

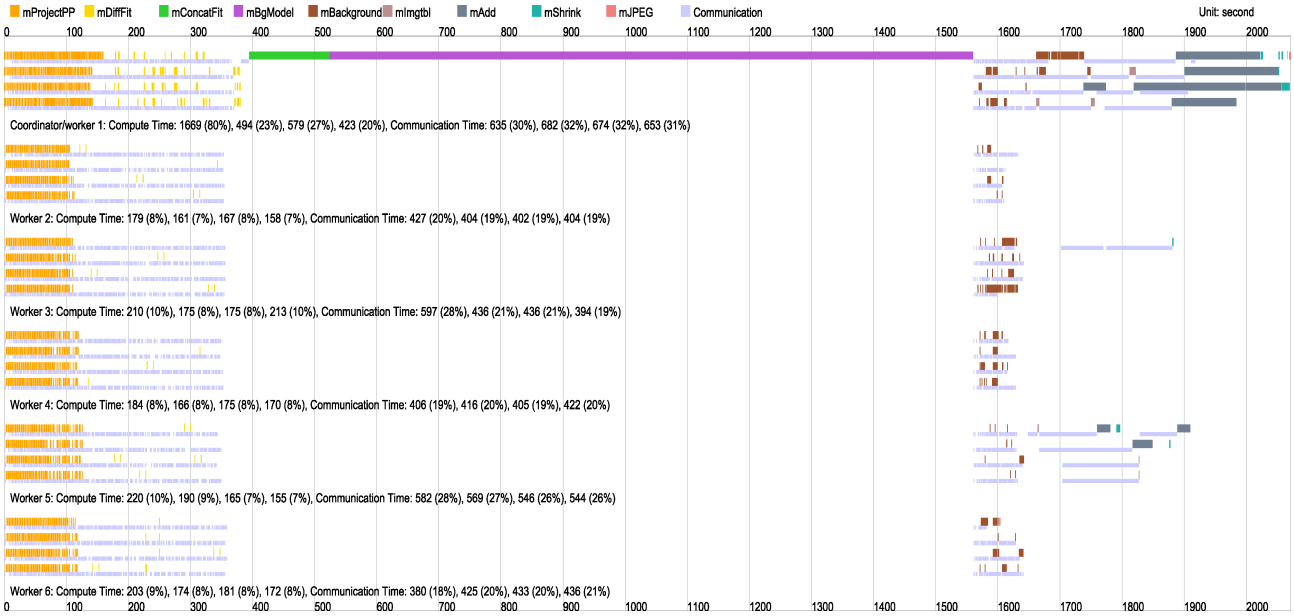



Figure 4: A visualization of Montage workflow execution on six *m1.xlarge* Amazon EC2 instances (each with 4 vCPUs; hence 4 slots) with 6.0 degree data set. Data are initially in the coordinator node, the third node in the figure; and thus, no data staging is required for early jobs. Due to the time scale of the figure, some jobs with their execution times less than 1 second are not clearly visible.

### 3.5 Exposing the Environment

One of DEWE's main contributions lies in methods designed to expose the underlying execution environment to users, allowing the design of algorithms that modify this environment during runtime. To this end, when deployed in EC2, users may deploy a variant of the Coordinator node that extends the Worker Manager and File Manager modules, providing an interface for Worker addition and removal (by leasing and terminating instances), and for downloads of input files and uploads of output files to S3.

**Elasticity with EC2:** Today, scientific workflows can contain thousands of jobs and data files, often following fluctuating concurrency patterns defined by dependencies. These patterns can potentially lead to low resource efficiency in fixed clusters, as many resources may be idle while waiting for a small number of job dependencies to complete (Figure 4). The elasticity supplied by IaaS cloud providers such as Amazon EC2 provides a means to obtain all the benefits of using a large cluster of resources, without incurring the associated costs and inefficiencies of idle resources. Moreover, the on-demand, pay-as-you-go nature of the cloud provides an efficient means to facilitate this elasticity.

For example, Montage workflows typically follow a regular structure (see Figure 1), with opportunities for concurrency fluctuating around certain bottlenecks (such as *mConcatFit* and *mImgTbl*). For large workflows, where the number of jobs in the first two levels can number in the thousands, a strategy involving dynamically shrinking and expanding the Worker pool around these bottlenecks can significantly decrease cost with minimal effect on makespan.[3]

To allow this resizing of the Worker node pool, DEWE provides an easy interface for elasticity. The process for Worker addition by DEWE when deployed on Amazon EC2 is as follows. During execution, a user-defined component makes a call to the Worker Manager module with a specific instance type and availability zone for provisioning. The Worker Manager module makes a call to Amazon's SDK, dynamically generating a shell script to be passed as `user-data` for execution by the `cloud-init` package at instance startup. The shell script starts a Worker node on the provisioned instance with parameters including the Coordinator's networking info. (e.g., IP address, port), workflow ID, S3 bucket info., etc.

Once the Worker node has initialized, it sends a registration message to the Coordinator node, including node-specific information such as the number of available slots. The Worker Manager adds the new Worker node to the list of available instances so that jobs may be dispatched to the Worker during the next iteration of the Job Scheduler's assignment phase. Once the Worker node is no longer required, e.g., during a bottleneck and after all file dependencies for future jobs have been replicated elsewhere, the Worker node may be terminated via another call to the Worker Manager module. The Worker Manager instructs the Worker node to upload log files to Amazon S3. After the upload completes, the Worker Manager terminates the instance through another call to Amazon's SDK.

**File management and provenance with S3:** When deployed in Amazon EC2, DEWE can utilize S3 for input and output file storage. At the start of each workflow, users can specify (e.g., via the command line or web application) an S3 bucket and associated file containing the input files. DEWE will automatically retrieve and extract this file at startup. The workflow output files (e.g., the image mosaic in the case of Montage), log files, and generated XML file are uploaded to the same S3 bucket after workflow completion or node termination. Files uploaded to S3 are accessible via public URLs, providing a further means to ease testing and debugging for the user; these URLs are listed in the web application, or can be navigated to directly within the user's S3 bucket.

[3] Acquisition times for EC2's On-Demand Linux instances are around 100 seconds on average (24). Thus, preemptive requisitioning of Worker nodes (e.g., via runtime knowledge) may be necessary for smaller workflows.

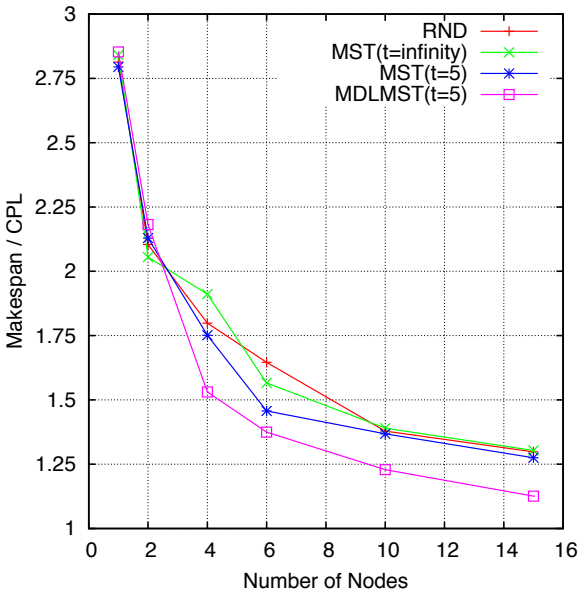



Figure 5: Ratio of makespan to CPL for different cluster sizes and scheduling algorithms.

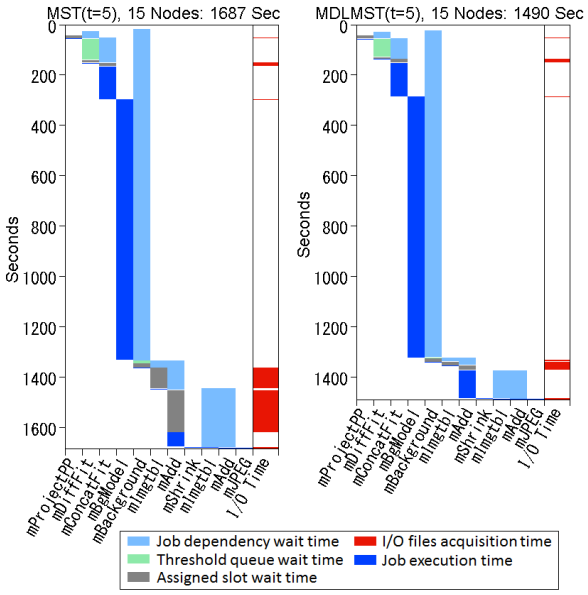



Figure 6: Evaluation of jobs on the critical path for MST(t=5) and MDLMST(t=5), using 15 nodes.

## 4 Evaluation

In this section, we describe our evaluation of DEWE in Amazon EC2, and provide a comparison of the results. To this end, we have developed two job scheduling algorithms incorporating DEWE's elastic resource (worker) provisioning feature.

### 4.1 Job Scheduling Algorithms

In addition to the default (random or RND) job scheduling algorithm, the following two algorithms were developed and tested to measure the evaluation capabilities of DEWE.

#### 4.1.1 Minimum Slot Threshold (MST)

Jobs are assigned to the node-slot with the minimum number of assignments currently executing or awaiting execution, provided the number of assignments is less than some threshold value, $t$. Let $N$ be the set of all nodes (Workers and Coordinator). The algorithm thus determines the slot $s^*$, where:

$$s^* = \underset{s \in \bigcup_{n \in N} S_n}{\arg\min} ||EQ_s|| + ||FQ_s||. \qquad (1)$$

If $||EQ_{s^*}|| + ||FQ_{s^*}|| \geq t$, the job is placed into a FIFO queue to await an opening. This algorithm helps ensure that existing slots are rarely idle, and newly added slots (such as from worker addition) are immediately utilized, rather than potentially waiting for the completion of an entire level before new jobs are available for assignment.

Since the Coordinator holds up-to-date representations of each Worker and its associated slots (via the Worker Rep and Slot Rep abstractions), the user is guaranteed that the $EQ_s$ and $FQ_s$ queues closely represent the current internal state of the node.

#### 4.1.2 Maximum Data Locality + MST (MDLMST)

Jobs are assigned to the node with the maximum input file locality, and to the slot on that node determined by MST. As mentioned in Section 3.2, each Worker Rep maintains a set of files locally available on the node, $F_n$. In addition, the file manager maintains the size (in bytes) of each file, $size(f)$ for $f \in F$. Thus, for a job, $v$, with input files $F_v$, the algorithm locates the node $n^*$ such that:

$$n^* = \underset{n \in N}{\arg\max} \left( \sum_{f \in F_n \cap F_v} size(f) \right), \qquad (2)$$

and we then determine $s^*$ as in MST, but with $N = \{n^*\}$.

In comparison to the default random scheduling algorithm and the MST algorithm by itself, this algorithm seeks to significantly reduce the communication overhead associated with each job, and thus minimize the delay between job assignment and execution. Utilizing a low queue threshold also helps minimize the potential for jobs waiting on other slots to retrieve common input files and become more suitable candidates, without explicitly considering those jobs during assignment.

### 4.2 Experimental Setup

In our experiments, Montage workflows are used as a case study.[4] As mentioned in Section 1, Montage is an astronomical image mosaic engine that assembles individual images of the sky into a mosaic (1). Montage includes modules to auto-generate the DAX file, an XML-based DAG (*mDAG*), and retrieve the input files from corresponding URLs in the `cache.list` file (*mArchiveExec*). Montage workflows typically follow a regular structure (Figure 1), with each stage of the workflow often taking place in discrete levels separated by bottlenecks, further described in Section 3.5. Montage workflows are I/O bound, and total data footprints can range up to hundreds or thousands of gigabytes.

Specifically, experiments were performed with a 6.0 degree Montage workflow, using clusters composed of up to 15 *m1.xlarge* instances/nodes, each with 4 CPUs and 15GB of memory. The 6.0 degree

[4] Note that the DEWE's DAG Manager can be extended to handle any workflow, as long as job, file, and precedence constraint info is available.

Montage workflow contains 8,586 jobs, 1,444 input data files, 22,850 intermediate files, and has a total data footprint of 38GB. Performance was measured in comparison to the CPL. All Worker nodes were automatically provisioned by the Coordinator after initialization.

### 4.3 Results

We first show makespans normalized by the CPL and resource usage reductions.

#### 4.3.1 Workflow Execution Times

The results for our experiments are illustrated in Figures 5 and 6. As demonstrated in Figure 5, the makespan approaches the CPL as the number of nodes increases, and begins to plateau at just above the CPL due to processing and communication costs. For 1 and 2 nodes, MDLMST is slightly less efficient than other algorithms due to the additional processing time required for file locality examination. However, as communication costs increase in larger clusters, MDLMST achieves the best results by a significant margin (generally more than 10% vs. the next best, MST). Furthermore, algorithms without thresholds – RND and MST($t = \infty$) – tend to exhibit lower performance than MST(t=5) due to a tendency to fill slots with long-running jobs, rather than the implicit consideration of *EQ* progress when $t$=5. All algorithms achieved between 53.9% (RND) to 60.5% (MDLMST) decreases in execution time with 14 Workers. In addition, RND and MST converge as the number of instances increases due to the abundance of available resources, while the ratio of makespan to CPL continues to decrease for MDLMST.

Figure 6 provides a comparison of the execution schedule of jobs on the critical path when running the MST (t=5) and MDLMST (t=5) algorithms with 15 nodes (Coordinator and 14 Workers). Levels with high concurrency (e.g., *mDiffFit* jobs) are significantly reduced in overall makespan when distributed across Workers, while bottlenecks such as *mBgModel* comprise the majority of the CPL and, hence, the execution time with 15 nodes. MDLMST is able to significantly reduce input file acquisition time (red bars). However, file acquisition still comprises a large portion of the waiting time for aggregation jobs such as *mAdd*, even with data locality considerations, preventing convergence to the CPL.

#### 4.3.2 Resource Usage

As the design of sophisticated dynamic resource provisioning strategies is out of scope of this paper, we adopt a simple elastic resource provisioning strategy. In particular, with the workflow execution status information a job scheduling algorithm provides to **Worker Manager**, DEWE terminates idle worker nodes towards the end of workflow execution as they are no longer needed (see Figure 4). Although this strategy is very simple (even primitive), it still demonstrates the elasticity capability of DEWE in that resource usage in our experiments was reduced by 12% on average and up to 27%.

As *mConcatFit* and *mBgModel* constitute the majority of execution and all worker nodes except the one running those two jobs are idling, DEWE can terminate those idle worker nodes and lease again once those two jobs complete their execution. The demand for such dynamic scaling support might not seem to be very attractive with the hourly billing plan in Amazon EC2, but makes more sense with more fine-grained billing mechanisms such as the per-minute billing plans in Google Compute Engine and Windows Azure.

## 5 Conclusion

Systems for the distributed execution of scientific workflows tend to focus on providing independence from the underlying execution environment, and often require long-running, static resources. However, the on-demand, open nature of public clouds has made explicit consideration and modification of the underlying environment during runtime a powerful strategy in terms of decreasing cost and increasing performance. In this paper, we have presented DEWE, a framework for the distributed, elastic execution of scientific workflows. DEWE is designed to provide an easy-to-deploy and highly customizable framework on which scientists may design algorithms with the ability to shape the underlying execution environment during runtime through automated Worker addition and removal. Evaluations of DEWE using a Montage workflow with AWS as the cloud environment have demonstrated that researchers are able to design, deploy, and compare various resource allocation and job scheduling algorithms on real clusters.

## References

[1] "Montage: An astronomical image mosaic engine," http://montage.ipac.caltech.edu/, 2013.

[2] J. C. Jacob and D. S. e. a. Katz, "Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking," *International Journal of Computational Science and Engineering*, vol. 4, no. 2, pp. 73–87, Jul. 2009.

[3] "Cybershake," http://scec.usc.edu/scecpedia/CyberShake, 2013.

[4] R. Graves, T. H. Jordan, and et. al., "Cybershake: A physics-based seismic hazard model for Southern California," *Pure and Applied Geophysics*, vol. 168, no. 3-4, pp. 367–381, 2010.

[5] A. Abramovici, W. E. Althouse, and et. al., "Ligo: The laser interferometer gravitational-wave observatory," *Science*, vol. 256, no. 5055, pp. 325–333, 1992.

[6] "Ligo: Laser interferometer gravitational-wave observatory," http://www.ligo.caltech.edu/, 2012.

[7] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of scientific workflows," in *Proceedings of the 3rd Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2008, pp. 1–10.

[8] J. Livny, H. Teonadi, M. Livny, and M. K. Waldor, "High-Throughput, Kingdom-Wide Prediction and Annotation of Bacterial Non-Coding RNAs," *PLoS ONE*, vol. 3, pp. e3197+, 2008.

[9] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," in *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, 1988, pp. 104–111.

[10] E. Deelman, G. Singh, and et. al., "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Journal of Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[11] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: An extensible system for design and execution of scientific workflows," in *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2004, pp. 423–424.

[12] N. Killeen, J. Lohrey, M. Farrell, W. Liu, S. Garic, D. Abramson, and G. Egan, "Integration of modern data management practice with scientific workflows," in *Proceedings of 8th IEEE Conference on eScience*, 2012.

[13] D. Abramson, R. Sosic, J. Giddy, and B. Hall, "Nimrod: A tool for performing parameterised simulations using distributed workstations," in *Proceedings of the 4th International Symposium on High Performance Distributed Computing (HPDC)*, 1995, pp. 112–121.

[14] H. Topcuouglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 13, no. 3, pp. 260–274, 2002.

[15] M. Tanaka and O. Tatebe, "Workflow scheduling to minimize data movement using multi-constraint graph partitioning," in *Proceedings of the International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, 2012, pp. 65–72.

[16] Y. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 7, no. 5, pp. 506–521, 1996.

[17] Y. C. Lee and A. Y. Zomaya, "Stretch Out and Compact: Workflow Scheduling with Resource Abundance," in *Proceedings of the International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, 2013.

[18] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 49:1–49:12.

[19] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski, "Cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 22:1–22:11.

[20] L. M. Leslie, Y. C. Lee, P. Lu, and A. Y. Zomaya, "Exploiting performance and cost diversity in the cloud," in *Proceedings of the 6th IEEE International Conference on Cloud Computing (CLOUD)*, 2013, pp. 107–114.

[21] T. Oinn, M. J. Addis, and et. al., "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[22] R. S. Barga, J. Jackson, N. Araujo, D. Guo, N. Gautam, and Y. Simmhan, "The trident scientific workflow workbench," in *Proceedings of 4th IEEE Conference on eScience*, 2008, pp. 317–318.

[23] B. Plale, E. C. Withana, C. Herath, K. Chandrasekar, and Y. Luo, "Effectiveness of hybrid workflow systems for computational science," in *Proceedings of the International Conference on Computational Science (ICCS)*, vol. 9, 2012, pp. 508–517.

[24] M. Mao and M. Humphrey, "A performance study on the VM startup time in the cloud," in *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD)*, 2012, pp. 423–430.