2014 15th International Conference on Parallel and Distributed Computing, Applications and Technologies

Running Data-Intensive Scientific Workflows in the Cloud

Chiaki Sato
University of Sydney, Australia
Email: csat9577@uni.sydney.edu.au

Luke M. Leslie
University of Illinois at Urbana-Champaign, USA
Email: lmlesli2@illinois.edu

Young Choon Lee
(이영춘)
University of Sydney, Australia
Email: young.lee@sydney.edu.au

Albert Y. Zomaya
University of Sydney, Australia
Email: albert.zomaya@sydney.edu.au

Rajiv Ranjan
Commonwealth Scientific and Industrial Research Organisation, Australia
Email: rajiv.ranjan@csiro.au

***Abstract*—The scale of scientific applications becomes increasingly large not only in computation, but also in data. Many of these applications also concern inter-related tasks with data dependencies; hence, they are scientific workflows. The efficient coordination of executing/running scientific workflows is of great practical importance. The core of such coordination is scheduling and resource allocation. In this paper, we present three scheduling heuristics for running large-scale, data-intensive scientific workflows in clouds. In particular, the three heuristic algorithms are designed to leverage slot queue threshold, data locality and data prefetching, respectively. We also demonstrate how these heuristics can be collectively used to tackle different issues in running "data-intensive" workflows in clouds although each of these heuristics can be used independently. The practicality of our algorithms has been realized by actually implementing and incorporating them into our workflow execution system (DEWE). Using Montage, an astronomical image mosaic engine, as an example workflow, and Amazon EC2 as the cloud environment, we evaluate the performance of our heuristics in terms primarily of completion time (*makespan*). We also scrutinize workflow execution showing different execution phases to identify their impact on performance. Our algorithms scale well and reduce makespan by up to 27%.**

I. Introduction

Applications in science and engineering are increasingly large-scale and complex. These applications are often composed of multiple inter-dependent tasks (i.e., precedence constraints) represented by directed acyclic graph (DAG or simply workflow). Examples of scientific workflows are Montage [1], CyberShake [2], [3], LIGO [4], [5], Epigenomics [6] and SIPHT [7]. As precedence constraints are dictated by data dependencies and data size continues to increase, the overhead of data transfers accounts for a significant amount of completion time. For example, a Montage workflow with the 6.0 degree data set, we used in our experiments, contains 8,586 jobs, 1,444 input data files, 22,850 intermediate files, and has a total data footprint of 38GB. Thus, running data-intensive scientific workflows with the explicit consideration of data locality and transmission overhead is of great practical importance.

There have been a myriad of studies on workflow scheduling with the primary focus of high performance (*makespan*). Although some recent works have considered data involvement in scientific workflows, e.g., [8], [9], such as data sharing options on Amazon EC2 and clustering workflows using graph partitioning techniques, data-aware scheduling in clouds is still in its infant state. Besides, existing techniques and algorithms for running scientific workflows are still limited in taking full advantage of the underlying execution environment.

In this paper, we design three scheduling algorithms that can be collectively used to run data-intensive scientific workflows in clouds. Each of these algorithms is designed to optimize a particular objective, such as load balancing, minimization of data transfers or asynchronous data transfers. Specifically, three algorithms are based on slot queue threshold, data locality and data prefetching, respectively.

To evaluate the efficacy of our heuristic algorithms, we have conducted extensive experiments in Amazon EC2 using a real scientific workflow application, Montage, with several different data sets. We also dissect the execution of workflows showing usage patterns of different resources, e.g., CPU and I/O. The running of workflows in Amazon EC2, including resource acquisition, data staging and actual execution, has been managed by our workflow execution framework, DEWE (Distributed, Elastic Workflow Execution)[1]. Based on our experimental results, our algorithms are capable of reducing makespan on average by 11% and up to 27% compared with a baseline round-robin algorithm.

The rest of this paper is organized as follows. In Section II, we provide the background and related work. Section III briefly describes our workflow management system (DEWE) used in this study focusing on its job scheduling component. Section IV presents and details our three scheduling heuristics. Section V show experimental results. We draw our conclusion in Section VI.

[1]DEWE including its source code and visualization toolkit used in this study is available from https://bitbucket.org/lleslie/dwf/wiki/Home.

978-1-4799-8334-6/14 $31.00 © 2014 IEEE
DOI 10.1109/PDCAT.2014.30

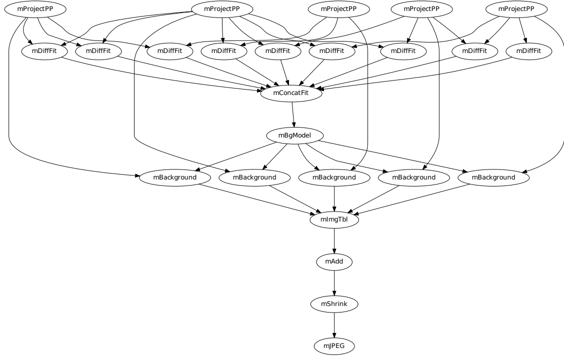

Figure 1. An example Montage workflow.

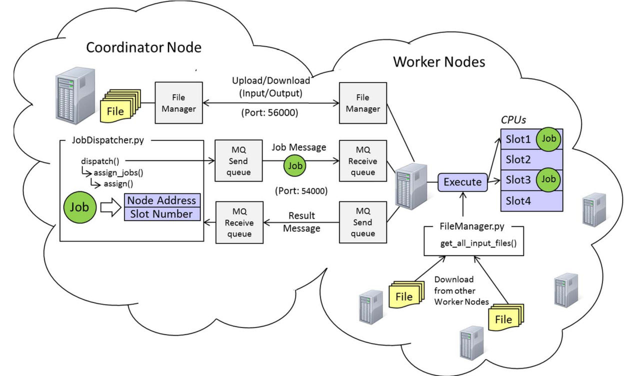

Figure 2. DEWE's Job Scheduling.

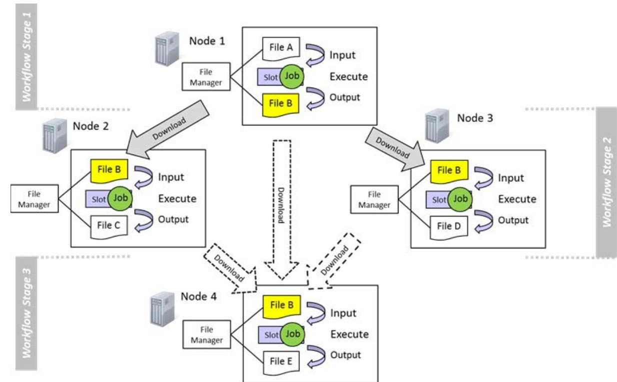

Figure 3. DEWE File Manager.

II. Background and Related Work

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

A. Scientific Workflows

Applications in science and engineering are becoming increasingly large-scale and complex. These applications are often amalgamated in the form of workflows 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), i.e., 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 leaf node in the DAG).

B. Workflow Scheduling

The execution of scientific workflows is typically planned and coordinated by schedulers/resource managers (e.g., [10], [11]) 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., [12], [9]. Critical-path base scheduling is one particularly popular approach to makespan minimization [13], [14]. Clustering-based scheduling is another approach getting much attention in the recent past with the emergence of many data-intensive workflows, such as Montage [9].

More recently with the adoption and prevalence of cloud computing, the trade-off between costs and performance has been extensively studied [15], [16], [17]. Most works on workflow scheduling in clouds study the elasticity of cloud resources, i.e., dynamic resource provisioning for cost minimization in particular. Although there are some efforts made on workflow execution with data awareness, e.g., [8], [9], the efficiency of scheduling algorithms for data-intensive scientific workflows is still limited particularly in clouds.

III. Workflow Scheduling and Execution

In this section, we describe our workflow management system (DEWE), used in our experiments, focusing on the job scheduling components.

DEWE adopts the master-slave model using **Worker** and **Coordinator** nodes (Figure 2). The Coordinator node contains components for DAG creation (DAG Manager), job scheduling and assignment (Job Dispatcher), slot management and job execution (Slot Manager), file management (File Manager), and Worker node management (Worker Manager). Since our focuses are job scheduling and data management, we give more details of *Job Dispatcher* and *File Manager*.

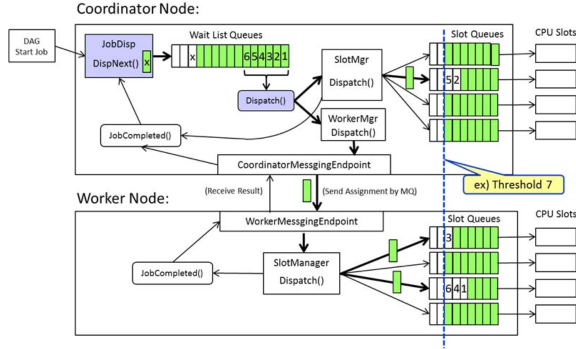

Figure 4. Slot Queue Threshold Algorithm (DAS^{th}).

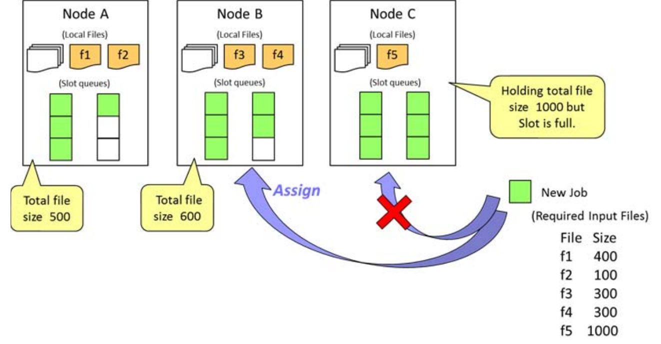

Figure 5. Data Locality Algorithm (DAS^{lc}).

The Job Dispatcher component in the Coordinator node schedules and distributes jobs to DEWE nodes including the coordinator node itself. The Slot Manager component puts the received job from the dispatcher into a queue corresponding to a slot (or a CPU core), and the slot handler component takes the job out from the queue and sends it to the slot (Figure 2). A job is assigned by the dispatcher to a node slot from the list of registered DEWE node slots managed in the coordinator node. This registration list of active nodes can be dynamically changed, i.e., new DEWE nodes can be added during the execution of a workflow.

Every DEWE node including the coordinator node runs an instance of the File Manager to deal with data staging, i.e., acquiring input files and storing output files for the job execution and after the execution, respectively. When the input files required for the job execution do not exist on the node the job is assigned (local node), the File Manager on the local node looks up the location list of files it maintains and downloads the required input files from remote DEWE nodes. The location list is distributed and synchronized among the File Managers on all DEWE nodes. All input files initially exist in the coordinator node and therefore, at the start of a workflow execution every worker node needs to download the input files from the coordinator node in order to execute a job assigned to them. Since the input and output files are left on the worker nodes after each workflow execution, as the workflow progresses, each worker node is likely to store many different files locally which are in turn downloaded and used by other nodes for a job execution. Figure 3 shows that, in workflow stage 1 (higher level of workflow in DAG), Node 1 outputs File B after the execution of a job. In stage 2, Nodes 2 and 3 download File B from Node 1 for the execution of assigned jobs. In stage 4, Node 4 can acquire input file B from either Node 1, 2 or 3.

IV. Data-Aware Scheduling

In this section, we present three data-aware scheduling (DAS) algorithms that concern slot queue threshold, data locality and data prefetching, respectively; hence, the names, DAS^{th}, DAS^{lc} and DAS^{pf}.

A. Slot Queue Threshold

DAS^{th} is designed to evenly distribute the workloads of jobs across all node slots. It assigns a job to a node slot that has the minimum (least) number of jobs in the slot queue among all node slots. This is expected to prevent from a skewed workload distribution and avoid a slot from completing the assigned jobs far later than the other slots. However, this may not evenly balance the actual workloads among the node slots since a particular job could be much heavier than the other jobs in terms of workload, e.g., computation and I/O. To this end, we adopt 'Slot threshold' to mitigate such an imbalance that in turn negative effect performance and resource usage. The threshold limits the number of jobs that can be assigned to the slot queues. When all slot queues are filled up to the threshold, the unassigned jobs simply wait in the wait list queue until a room on a slot queue becomes available. Thereby, in this algorithm, even the jobs with much heavier loads than the other jobs can be assigned just up to the threshold in the worst case, and then the balance in workloads on slots is likely to be kept to some extent. Figure 4 shows that the jobs represented by numbers 1 to 6 are retrieved from the wait list queue in order, then assigned to slot queues with a threshold value of 7 (indicated by the dotted line).

B. Data Locality

DAS^{lc} is designed to explicitly consider the communication cost, or the downloading time of the input files for a job execution. As the volume of data for scientific workflows becomes increasingly large, moving job/computation to data is rather obvious and intuitive. That is, a job should be assigned to a node slot where the most of its input files already exist locally on the node. In this algorithm, the "best" node to assign a job is decided by the total size of input files existing locally for the job execution. As consequence, the slot queues on the coordinator node are always filled up first due to its advantage in the file locality. For each job, the assignment steps are as follows.

1) Choose the node that holds the most amount of input files for the job; hence, the node is the "best" for the job (or best node).

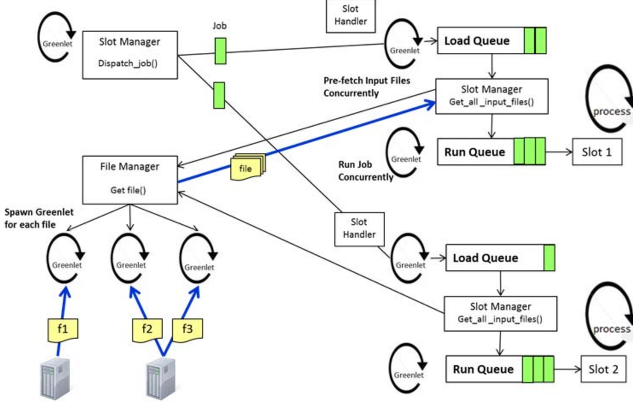

Figure 6. Data Prefetching Algorithm (DAS^{pf}).

2) If all slot queues on the node found in Step 1 are full with other jobs, choose the next node that contains the largest amount of input files; this step may be performed iteratively until the node with its slot queues being not completely full is found.
3) If all slot queues on all nodes are full, the job waits in the wait list queue until a slot queue becomes available and go back to Step 1.
4) Otherwise, assign the job to a slot on the chosen node.

Figure 5 illustrates the assignment of a new job which requires five files (`f1`–`f5`) as input. Node C is the best node in that it already has `f5` (file size of 1000); however, as all slot queues on Node C are already occupied by other jobs, the second best node, B that holds `f3` and `f4` (a total of 600 in file size) is chosen for the new job.

C. Data Prefetching

DAS^{pf} is designed to make use of resources efficiently by consolidating I/O-intensive jobs and CPU-intensive jobs with the expectation of their resource usage being complementary. In particular. while the former jobs are accessing I/O resources (e.g., disks and network devices) for downloading input files, the latter jobs are accessing CPU resources for computation. As the name indicates. DAS^{pf} prefetches (downloads) the input files for a (waiting) job while the other jobs are running on the slot, rather than downloading the files and executes the job sequentially. In Figure 6, the main application process on the assigned node running a greenlet (gevent utility for pseudo-threading, https://pypi.python.org/pypi/greenlet) takes out a job from the load queue and downloads the input files for the job before sending the job into the run queue. The main process waits for the physical slot being available and when it becomes available, it takes one job out from the run queue, and executes it on the physical slot as a separate process from the main process.

V. Evaluation

In this section, we evaluate our scheduling algorithms using Montage and DEWE on Amazon EC2. We compared (combinations of) our algorithms with a naive round-robin algorithm (RR) that simply dispatches jobs to nodes in turn. Experiments were conducted using `m1.xlarge` Amazon EC2 instances with Montage workflows of three different data sets, 2.0, 4.0 and 6.0 degree data; the number of instances in our experiments varies from 1 to 8.

A. Results

The performance metric is makespan (completion time). As Amazon EC2 instances may exhibit some performance variations, each experiment was run 3 times and results are averaged. Note that we used a slot queue threshold of 5 as different thresholds (between 5 and 10) showed similar results in our preliminary experiments.

Results are shown in Figures 7, 8 and 9, and summarized in Table I with respect to different numbers of instances, data sets and algorithms. Clearly, the collective solution with all three of our algorithms (DAS^{combo}) always outperforms the other two algorithms by 15% and 7%, respectively, and up to 27%. Besides, the number of instances used affects performance. However, the slopes of curves for results of RR and DAS^{pf} are not as smooth as that of DAS^{combo} as shown in Figures 7 and 8. This effect is due primarily to not explicitly considering data locality that may incur more communications/data transfers between instances; as the number of instances increases, this may be worsened unexpectedly.

Another thing to note is the speed up particularly with 6.0 degree data (Figure 9). In other words, Montage workflows with large data set (e.g., 6.0 degree data) spawns significantly more jobs that can leverage the use of more instances.

B. Performance Breakdown

To identify the impact of different execution factors on performance (i.e., makespan), we have analyzed workflow execution in detail. In particular, for a given workflow, we plotted the execution of each job in five stages shown in different colors (Figures 10 and 11):

1) *Job dependency wait (light blue).* For a given job, its job dependency wait is the waiting time for all the parent jobs being completed.
2) *Slot queue busy wait (green).* Slot queue busy wait only incurs with the Slot Queue Threshold algorithm DAS^{th}, where all the slot queues to assign a job are filled up to the threshold.
3) *Assigned slot wait (gray).* Assigned slot wait is the waiting time after DEWE scheduler assigns a job to a queue on a particular CPU before the job actually starts to be executed on the assigned CPU. Assigned slot wait could be caused either by waiting for the

Table I
OVERALL RESULTS. MAKESPAN RESULTS ARE IN SECONDS.

Montage workflow data	algorithm	the number of instances								average
		1	2	3	4	5	6	7	8	
2.0 degree	RR	571	522	530	495	482	520	474	460	507
	DAS^{pf}	574	534	490	444	458	457	490	402	481
	DAS^{combo}	566	463	477	441	423	417	397	381	446
4.0 degree	RR	1498	1150	1076	1046	1011	909	948	1031	1084
	DAS^{pf}	1491	1219	996	930	927	1065	898	927	1057
	DAS^{combo}	1491	1107	911	972	919	911	861	909	1010
6.0 degree	RR	4000	3345	3187	2381	2266	2109	1943	2019	2656
	DAS^{pf}	4028	3326	3018	2287	2377	2093	1994	1856	2623
	DAS^{combo}	4056	3130	2848	2134	2135	2007	1889	1861	2508

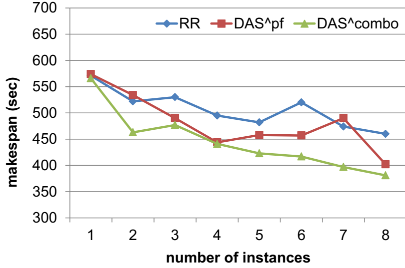

Figure 7. Montage workflow with 2.0 data.

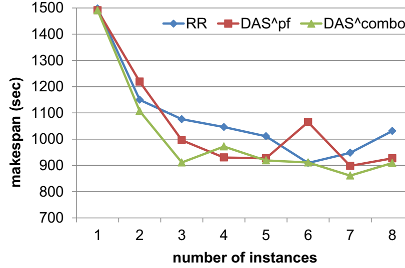

Figure 8. Montage workflow with 4.0 data.

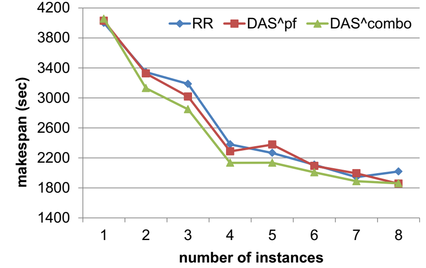

Figure 9. Montage workflow with 6.0 data.

completion of data staging (input/output files) or by waiting for an available CPU.

4) *Data staging (yellow).* Before the execution of a job, the files required for the execution, that do not exist on the local node, are downloaded from remote nodes. The input and output files for a job remain on the local node once the job is executed, for the other nodes to download if they need.
5) *Job execution (blue).* Job execution time is the actual Montage API execution time. In addition to CPU time for the execution, the job execution time includes the time for file reading/writing.

We show the performance breakdown of DAS^{pf} and DAS^{combo} in Figures 10 and 11. For each figure, the left chart is the result on a single DEWE node and the right chart is the result on 6 nodes.[2] As visually identified for both algorithms, in a single node they have a similar makespan with around 435 seconds, and data staging times are almost zero because all input files exist locally. The reduction of makespan between the single-node case and the 6-node case is realized in the time span before the execution of `mConcatFit` job (in Montage workflow) even though different types of waiting are reduced for each algorithm. As 301 `mProjectPP` jobs and 838 `mDiffFit` jobs exist in the 2.0-Degree Montage workflow, this reduction indicates that these jobs are effectively distributed across 6 nodes. Besides, a negative influence of the job distribution is observed in the 6-node case for both algorithms. The data staging time has grown since the input files to be acquired are dispersed across 6 nodes, and the naive RR algorithm has suffered from much larger waiting time than DAS^{combo} in particular.

The performance enhancement of workflows with an increasing number of nodes for both algorithms (DAS^{pf} and DAS^{combo}) is the result of the positive effect of the job distribution which is much larger than the negative influence. As a result, makespans of around 435 seconds on a single node for both algorithms are improved by nearly 15% and 27% on 6 nodes with 368.20 and 318.50 seconds. The primary source of such performance gains is from significant reductions of data staging times.

[2]Note that the actual runtimes in Figures 10 and 11 do not match those in Table I because as stated earlier, results in Table I are averages of three runs.

VI. CONCLUSION

As cloud computing is increasingly adopted not only for enterprise applications, but also for scientific applications, we have addressed the effective execution of data-intensive scientific workflows in the cloud. In particular, we have developed three data-aware scheduling algorithms that can be collectively used. As the scale and complexity of scientific workflows continues to increase particularly in data, running these workflows with the explicit consideration of data locality and the use of data prefetching are rather essential. Our experimental results obtained using a real scientific workflow (Montage) on Amazon EC2 clearly demonstrate the efficacy of our algorithms and prove our claims.

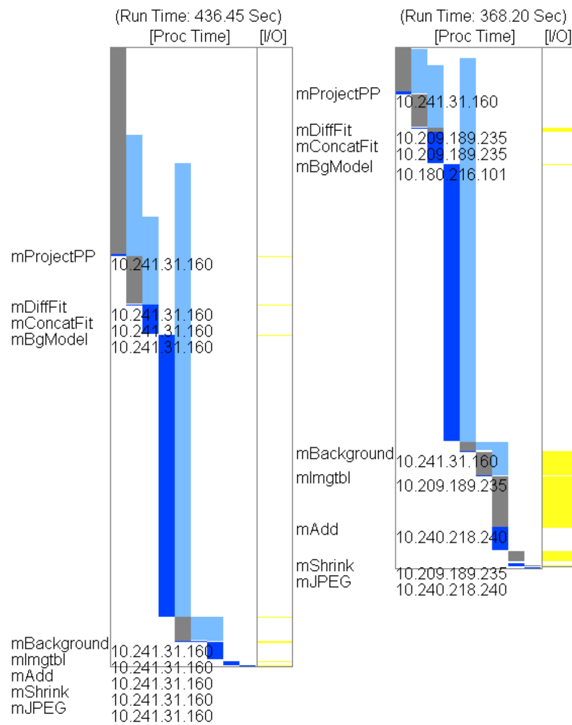

Figure 10. Performance breakdown of DAS^{pf}.

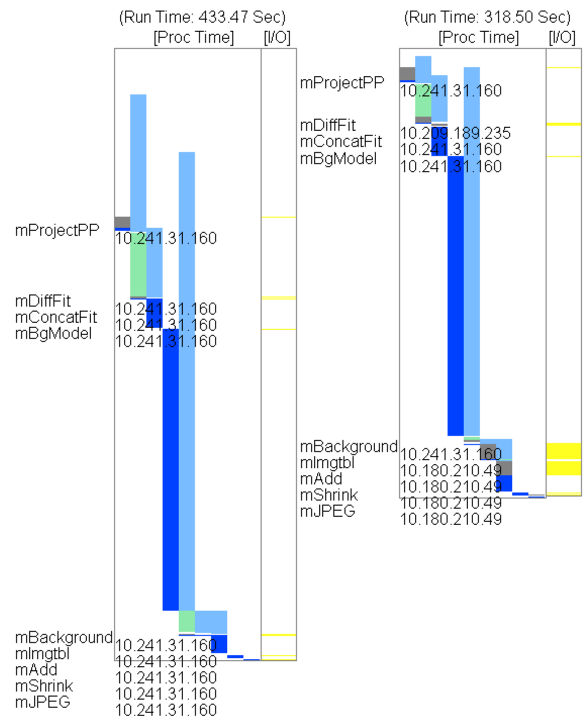

Figure 11. Performance breakdown of DAS^{combo}.

REFERENCES

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

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

[3] 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.

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

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

[6] 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.

[7] 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.

[8] D. E. Juve, Gideon and et al., "Data sharing options for scientific workflows on amazon ec2," in *Proc. the 2010 ACM/IEEE Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010, pp. 1–9.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

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