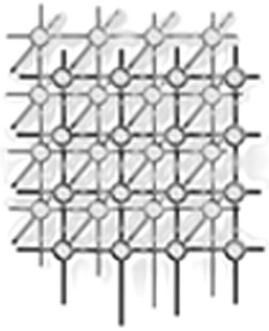
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2008; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Adaptive Workflow Processing and Execution in Pegasus



Kevin Lee[1], Norman W. Paton[1], Rizos Sakellariou[1], Ewa Deelman[2], Alvaro A. A. Fernandes[1], Gaurang Mehta[2]

[1]*School of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, U.K,* {*klee, norm, rizos, alvaro*}*@cs.man.ac.uk*
[2]*University of Southern California, Information Sciences Institute, 4676 Admiralty Way, Marina Del Ray, CA 90292, USA,* {*deelman, gmehta*}*@isi.edu*

SUMMARY

Workflows are widely used in applications that require coordinated use of computational resources. Workflow definition languages typically abstract over some aspects of the way in which a workflow is to be executed, such as the level of parallelism to be used or the physical resources to be deployed. As a result, a workflow management system has responsibility for establishing how best to execute a workflow given the available resources. The Pegasus workflow management system compiles abstract workflows into concrete execution plans, and has been widely used in large-scale e-Science applications. This paper describes an extension to Pegasus whereby resource allocation decisions are revised during workflow evaluation, in the light of feedback on the performance of jobs at runtime. The contributions of this paper include: (i) a description of how adaptive processing has been retrofitted to an existing workflow management system; (ii) a scheduling algorithm that allocates resources based on runtime performance; and (iii) an experimental evaluation of the resulting infrastructure using grid middleware over clusters.

1. Introduction

A number of workflow environments have been developed in recent years to provide support for the specification and execution of scientific workflows. We distinguish scientific workflows (as supported, for example, by Pegasus [1, 2], Askalon [3], Taverna [4], Kepler [5] and Triana [6]), being typically compute and/or data intensive, as opposed to business workflows, which are typically transactional and are beyond the scope of this paper. Workflow languages are used to provide a high-level characterization of the pattern of activities that need to be carried out to support a user task. Workflows written in such languages typically leave open a number of decisions as to how a workflow is enacted, such as where the workflow is to be run, what level

of parallelism is to be used and what resources are to be made available to the workflow. As a result, a collection of decisions must be made before a workflow can be enacted, for example by a compilation process that translates a workflow from an abstract form into a more concrete representation that resolves various of the details as to how the workflow is to make use of available resources.

Most existing workflow systems provide static approaches for mapping (e.g., [7, 8]) on the basis of information that provides a snapshot of the state of the computational environment. Such static decision making involves the risk that decisions may be made on the basis of information about resource performance and availability that quickly becomes outdated. As a result, benefits may result either from incremental compilation, whereby resource allocation decisions are made for part of a workflow at a time (e.g., [1]), or by dynamically revising compilation decisions that gave rise to a concrete workflow while it is executing (e.g., [9, 10, 11, 12, 13]). In principle, any decision that was made statically during workflow compilation can be revisited at runtime (e.g., [14]). Workflow decisions can be broadly classified as either mapping or scheduling decisions. Mapping decisions can lead to adaptations involving changing the mapping of abstract tasks to concrete tasks, increasing or reducing the abstract task to concrete task numbers and changing the type of service for a mapping. Scheduling decisions can lead to adaptations that change the levels of parallelism, replace services, or move tasks between execution nodes (see [14] for more opportunities). The focus of this paper is on scheduling adaptations involving moving concrete tasks between execution nodes.

Proposals that describe adaptive approaches to mapping (e.g., [9]) are often quite intrusive, in that the adaptive behaviour of their engine exercises fine-grained control over the workflow engine, implying that significant effort may be required to incorporate such capabilities into existing mainstream workflow systems. In contrast, the work described in this paper implements adaptivity as a separate module which is loosely-coupled with an existing workflow system. This loosely-coupled approach enables experimentation with adaptive strategies that transcend the specifics of a given execution engine. In this way, one can aim to identify a space of widely-useful policies without precluding their later implementation as specific mechanisms inside specific software artifacts.

This paper describes an approach to adaptive resource allocation and scheduling in the Pegasus workflow management system [1]. Pegasus already accommodates uncertainty about the runtime environment by incremental compilation, which both defers certain decisions as to how workflow activities are mapped to resources and forms the basis for fault tolerance, whereby a workflow partition, which is the unit of incrementality, can be retried if it fails. In line with [9], our adaptive system is purely reactive in that it monitors information and reacts to it. Thus, the emphasis is on adaptations on the basis of specific observable behaviour rather than mechanisms to predict what future behaviour is going to be. Our objectives in this work have been: (i) to dynamically adjust resource allocation decisions in the light of runtime feedback on the performance of the clusters onto which workflows are being compiled; and (ii) to obtain that dynamic behavior through minimal intervention into the existing Pegasus infrastructure. As a case study for the evaluation of our adaptive system we consider resource allocation on clusters that might be used by several users at the same time. This allows us to introduce adaptivity into an environment whose performance is not well known in advance, and in which there is limited control over the execution of individual jobs; studies that focus on

the evaluation of scheduling heuristics usually require more information about the environment than is assumed here [15, 11, 12, 16]. Yet, without using sophisticated heuristics, our adaptive engine yields demonstrable benefits.

The remainder of this paper is structured as follows. Section 2 provides the technical context for this work by describing the Pegasus workflow management system. Section 3 details both what adaptations are carried out and how these have been integrated with the Pegasus infrastructure. Section 4 describes the results of experiments conducted using both synthetic and real-world scientific workflows. Section 5 draws some overall conclusions.

2. Technical Context

2.1. Overview

The Pegasus Workflow Management System (Figure 1) consists of the Pegasus workflow mapper [1] and the Directed Acyclic Graph Manager (DAGMan) workflow executor for Condor-G [17]. Pegasus takes high-level descriptions of complex applications structured as workflows (abstract workflows), automatically maps them to available cyberinfrastructure resources (concrete workflows), and submits them to DAGMan for execution.

Pegasus has been used in a wide range of applications including earthquake science and astronomy. Using Pegasus, earthquake scientists are able to generate more accurate hazard maps that can be used by civil-engineers to design new construction in earthquake-prone areas [18]. Astronomers use Pegasus to generate large-scale (6 and 10 square degree), mosaics of the sky that allow them to see structures not observed before [19]. Gravitational-wave physicists are using Pegasus to run sophisticated analysis in the hopes of finding gravitational waves [20].

2.2. Compilation

The *workflow mapping engine* is a compiler that translates (maps) between the high-level specifications of an abstract workflow and the underlying execution system and optimizes the executables based on the target architecture. The translation includes finding the appropriate software and computational resources where the execution can take place, as well as finding copies of the data indicated in the workflow instance. The mapping process can also involve workflow restructuring geared towards optimizing the overall workflow performance as well as workflow transformation geared towards data management. The result of the mapping process is an executable or concrete workflow, which can be executed by a workflow engine that follows the dependencies defined in the workflow and executes the activities defined in the workflow tasks. DAGMan, the workflow engine used relies on the resources (compute, storage, and network) defined in the workflow to perform the necessary actions.

Mapping the workflow instance to an executable form involves finding the resources that are available and can perform the computations, the data used in the workflow, and the necessary software. We assume that data may be replicated in the environment and that users publish their data products into some data registry. This registry can be a private or community

resource and can be either static or dynamic (see Figure 1). Some communities, such as Laser Interferometer Gravitational-Wave Observatory [21] maintain project-wide registries of the data coming off the detectors.

Pegasus uses the logical filenames referenced in the workflow to query the Globus Replica Location Service (RLS) [22], which is an example of a dynamic data location service, to locate the replicas of the required data. Given the set of logical filenames RLS returns a corresponding set of physical file locations. A local Replica Catalog provides a static source of physical data locations. In order to be able to find the location of the logical application component names (transformations) defined in the workflow, Pegasus queries the static Transformation Catalog (TC) [23] and obtains the physical locations of the transformations (on possibly several systems) and the environment variables and libraries necessary for the proper execution of the software. The executables are transferred to the remote grid sites along with any input data.

Like with locating replicas and transformations, Pegasus can find resources for execution from static and dynamic sources of information. Pegasus can query cyberinfrastructure monitoring services (e.g., the Globus Monitoring and Discovery Service (MDS) [24]) or local Site Catalogs to find the available resources and their characteristics (machine load, scheduler queue length, available disk space, and others). This information is combined with information from the Transformation Catalog to make scheduling decisions. Schedulers are one of the pluggable components of Pegasus. Up to now Pegasus included four different scheduling algorithms: random, round-robin, min-min [7] and HEFT [15]. In this work, we designed and incorporated a new scheduler into Pegasus. As opposed to the static nature of the existing four (and the large body of relevant work in the literature, e.g., [16]), the key feature of our new algorithm is that it takes into account runtime information (see Section 3).

Pegasus also uses information services to find the location of the data movement services (e.g., GridFTP [25] or SRB [26]) that can perform wide-area data transfers, job managers [27] that can schedule jobs on the remote sites, storage locations, shared execution directories, site-wide environment variables, etc. This information is necessary to produce the executable workflow that describes the necessary data movement, computation and catalog updates. Registries of code and data as well as information services allow Pegasus to provide a level of abstraction to the user and give the freedom to automatically optimize workflow execution.

2.3. Submission

After the abstract workflow has been compiled it is in a concrete or executable form. Pegasus submits this concrete workflow to the DAGMan workflow executor. DAGMan then uses this workflow description to determine when to submit jobs to the resources described in the workflow (determined from the monitoring and Discovery Service [24] or the local Site Catalog). It then manages the submission of subsequent jobs using the dependencies in the workflow.

2.4. Execution

Individual jobs are submitted by DAGMan to remote Globus interfaces representing clusters where they are executed. These include jobs that setup the remote site, transfer data via GridFTP and execute tasks. The jobs pass through the Globus interfaces to the Globus

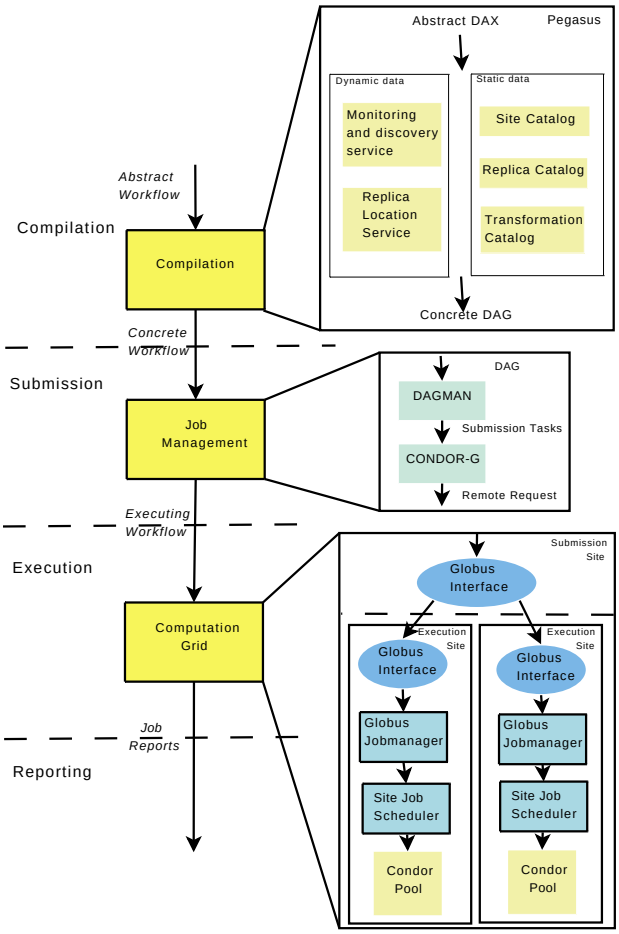


Figure 1. Workflow Execution using Pegasus.

jobmanagers at each execution site before finally being executed by the Site Scheduler on the resources. The experiments in this paper use the Condor-G submission client and Condor clusters though a Globus interface, as illustrated in Figure 1. During execution, workflow tasks can register the final and intermediate workflow data products into the various registries. Future Pegasus compilations can easily discover and use these new data products.

2.5. Reporting

During execution, events are passed back to DAGMan via the Site Scheduler, Globus Jobmanager, Globus interfaces and DAGMan. These events indicate the current status of an individual job's execution, e.g., if it is held, queued, executing, completed or failed. This log of the execution provides a snapshot at any point in the workflow execution. DAGMan uses this information to determine when jobs can be submitted, which jobs or workflows have failed and when a workflow is complete.

3. Adaptive Pegasus

As stated in Section 1, the focus of this paper is on dynamically adjusting resource allocation decisions in response to feedback on the performance of workflow execution. A wealth of potentially useful performance information is available about the execution of a workflow, however the characteristics of the machines used for workflow execution yield a specific opportunity. The environment to which Pegasus is targeted utilises batch queues to assign jobs to cluster resources. Experience with workflow executions shows that a major factor in overall execution time is the amount of time a job is queued. The batch queue time for a job is dependent on the external load on the cluster and any load we assign to it. In a grid environment, however, this cannot be obtainted directly because of different ownership of resources so this information must be obtained by observing the changing queue times on the clusters. The strategy proposed in this section adapts workflow execution to the varying batch queue times at clusters.

The adaptive strategy used to achieve this is structured around the *MAPE* functional decomposition [28] which partitions adaptive functionality into four areas, *Monitoring*, *Analysis*, *Planning* and *Execution*. The MAPE functional decomposition is a useful framework for systematic development of adaptive systems, and can be applied in a wide range of applications, including different forms to workflow adaptation [14]. The use of MAPE to structure the adaptive strategies in this paper is illustrated in Figure 2, which shows how it is retrofitted with minimal intervention to a Pegasus-planned executing workflow.

In the adaptation strategy described in this paper, an executing workflow instance is monitored for the relevant events at the assigned resources. These events are constantly analysed for patterns, which may lead to planning. Planning updates the information available to Pegasus, and reruns Pegasus on the current workflow. The revised plan for the work that remains to be done is compared with the current plan, and the new plan is adopted if it is predicted to give an improved overall response time. Changes to the workflow execution proposed by Planning are implemented in an execution step that removes and replaces the executing workflow. The following paragraphs discuss the components in Figure 2 in more detail. The diagram shows how the sensor and effector of the adaptive software interact with parts of the Pegasus environment to adapt executing workflows.

Monitoring: To monitor the progress of an executing workflow, *job queue*, *execute* and *termination* events are tracked. These, respectively, indicate when Condor submits a task to the remote scheduler, when the remote scheduler indicates that the task has started to execute,

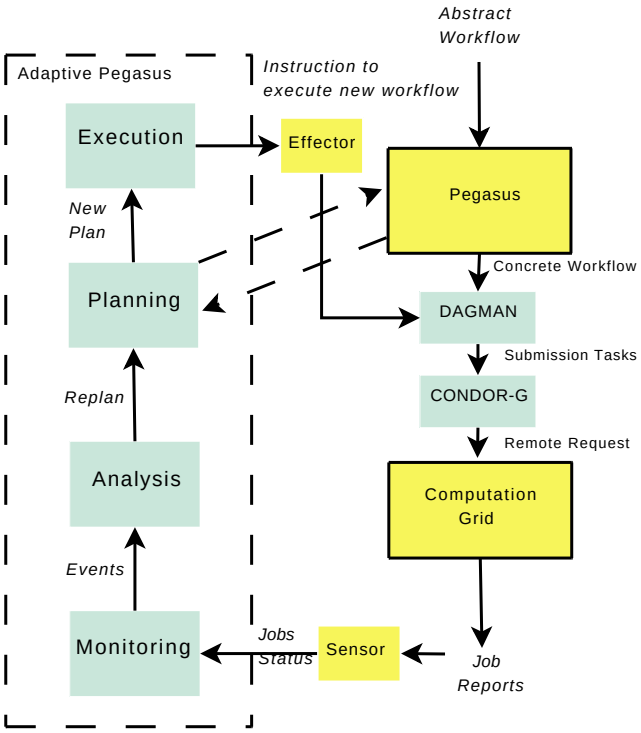


Figure 2. Pegasus Adaptive Support

and when the remote scheduler indicates that the task has completed. These are sensed using a *LogSensor* that polls for new entries in the DAGMan log file every 100 milliseconds. The DAGMan log file records all events about the progress of a workflow's execution. Each entry of the log file is parsed to determine if it contains an event of interest and passed to Analysis.

Analysis: The role of the analysis step is to establish whether the workflow is performing according to expectations when it was compiled. If expectations are not being met, then it may be possible to improve on the plan that is being pursued. To support the concise and declarative description of patterns in the monitoring data, the CQL continuous query language [29] is used to group and analyse the events produced by monitoring. The CQL queries that implement the analysis are given in Figure 3. Although the style of the queries is like SQL, it differs in that they operate in a continuous way. Every time a new tuple arrives in a stream, the queries are evaluated and any output produced. The queries can operate over varying windows of time or number of tuples. In this case, four intermediate queries operate on the raw input stream or the output of the other queries to enable a final Analysis query to be evaluated. The queries look for a sustained substantial increase or decrease in batch queue times per site compared

Input Streams:
events :
  `int` *timestamp*, `char` *event*, `char` job
assignments :
  `char` *job*, `char` *site*, `int` *estimate*

Queries:
jobqueued :
  `select` *timestamp*, *job* `from` *events* `where` *event* $=' ULOG\_SUBMIT'$;
  `register stream` *jobqueued* (`int` *timestamp*, `char` *job*);

jobexecuted :
  `select` *timestamp*, *job* `from` *events* `where` *event* $=' ULOG\_EXECUTE'$;
  `register stream` *jobexecuted* (`int` *timestamp*, `char` *job*);

queuedtime :
  `select` *execute.timestamp* − *queued.timestamp*, *execute.job*
  `from` *jobqueued* `as` *queued*, *jobexecute* `as` *execute* `where` *queued.job* = *execute.job*;
  `register stream` *queuedtime* (`int` *queuetime*, `char` *job*);

queuetimeandestimate :
  `select` *queue.timestamp*, *queue.job*, *assignment.site*, *assignments.estimate*
  `from` *queuetime* `as` *queue*, *jobassignments* `as` *assignments* `where` *queue.job* = *assignments.job*;
  `register stream` *queuetimeandestimate* (`int` *timestamp*, `char` *job*, `char` *site*);

Analysis:
  `select` "*LongQueue*", *site* `from` *queuetimeandestimate*[`Rows` 3] `where` `AVG`(*time* − *estimate*) > *threshold*;
  `select` "*ShortQueue*", *site* `from` *queuetimeandestimate*[`Rows` 3] `where` `AVG`(*estimate* − *time*) > *threshold*;

Figure 3. Filtering monitoring events in CQL

to the job batch queue predictions created by the scheduler. If there is an output from this analysis, the planner is notified. In addition to determining if adaptations may be necessary, *Analysis* also generates average queue times for each available site for use by the scheduling algorithm. Queue times are derived using relevant event information from Monitoring.

Planning: When analysis detects a sustained change in batch queue times for a site, rescheduling may need to be performed. To examine this, the Pegasus planner is called to propose an alternative schedule taking into account recent queue times.

To ensure that jobs are not unnecessarily repeated, the replica catalogues used by Pegasus to share results within and between workflows are updated with results already produced by the workflow. This is because each job in a concrete workflow outputs its results as intermediate data in the form of a file. The relevant folders on the execution sites are scanned for intermediate results, which are added to the replica catalogue.

As discussed in Section 2.2, Pegasus currently has four different schedulers which it uses to assign jobs to resources. However, these were designed to schedule statically using limited (or statically estimated) information about the performance characteristics of execution resources. To enable adaptive behavior, a scheduling algorithm is needed that takes account of

```
Input:
   Workflow W
   List of Sites S
   List of Average Queue Times SQ

1. Calculate PS_s the proportion of the workflow each site s should process.
   for Site s ∈ S
      PS_s = (1/SQ_s)/ Σ_{i∈S}(1/SQ_i)

2. Calculate Num_s the number of jobs each site s should process.
   for Site s ∈ S
      Num_s = PS_s * size(W)

3. Create AS a queue of assignable sites.
   for Site s ∈ S
      for Int i = 1 to Num_s
         AS.push_back(s)

4. Randomise the list of assignable sites.
   AS.randomise()

5. Create A_j, the job to site assignment list.
   for Job j ∈ W
      A_j = AS.pop_front()
```

Figure 4. Adaptive Scheduling Algorithm

information gleaned by Monitoring. To this end, we implemented a new scheduler which uses data collected about the average queue times of each available site to decide where to schedule each job in the workflow. Figure 4 shows this scheduling algorithm that enables adaptivity.

The scheduler depends on the presence of historic data containing the average queue times for each available site. This is generated by *Analysis*; when no prior data on average site queue times is available a default value of 0 is used. The scheduler allocates work to each site in inverse relation to the average queue time since the start of the execution of the workflow. It is as follows: Step *1* calculates the proportion of a workflow (in number of tasks) that should be assigned to each site, based on average batch queue times. Step *2* calculates the number of jobs each site should process, by multiplying the number of jobs of the workflow by the proportion each site should be assigned. Steps *3* and *4* create a randomised list of sites based on the number of jobs each site should be assigned from Step *2*. Step *5* creates the final job-to-site assignment list.

Not every new schedule proposed by the scheduler is deployed; new schedules are compared with the existing executing schedule to see if they are predicted to improve on the current plan. The cost of adaptation (recorded from previous adaptations) is also taken into account when deciding whether or not to deploy a new schedule. If it is decided to deploy it, the next component, execution, is called. In addition to returning a list of job assignments to sites, the scheduler also generates a list of predicted batch queue times for each job on each site. These

predictions are later used by *Analysis* to detect substantial deviations from actual running times. The predicted batch queue time is the average queue time for the site to which the job has been assigned. These are made available to *Analysis* in the form of the *assignments* input stream in Figure 3.

A possible further improvement to the scheduling strategy proposed here is to use a previous schedule from our scheduler as the basis of the schedule for the next workflow to be deployed. This would result in a possibly better initial schedule, however there is also a potential for an incorrect schedule if the cluster queue times changed drastically between workflow executions.

Execution: At the stage that execution is called, there is a currently executing workflow. Execution stops the executing workflow and deploys the new one using Pegasus commands.

4. Experimental Evaluation

4.1. Experiment Setup

The aim of the experimental evaluation is to explore the effect of the adaptive approach on response time in a range of scenarios. The experiments use two abstract workflow styles. The first type is a *linear workflow*, which is simply a DAG were each subsequent task is dependent on the file created by the previous task, and may contain any number of tasks. With these dependencies present, the tasks in the workflow will execute in series. In our experiments we considered an instance with 50 tasks.

The second workflow type is that of a *Montage workflow*, which creates a large mosaic image from many smaller astronomical images [1]. These can be of varying sizes depending on the size of the area of sky of the mosaic. A simple Montage workflow is illustrated in Figure 5. The numbers represent the level of each task in the overall workflow. This corresponds to the size used in our experiments (25 tasks, equivalent to a 0.2 degree area).

For these experiments two clusters were used, which we designate *Cluster 1* and *Cluster 2*. Cluster 1 has as submission site a 2.4Ghz Xeon with 2GB of RAM, and 8 worker nodes each with a 2.4Ghz Xeon with 2GB of RAM connected together by Gigabit Ethernet. Cluster 2 has as submission site a 2Ghz dual core Opteron with 4GB RAM, and 112 worker nodes each with a dual core 1Ghz P3 with 4GB RAM connected together by 100 Megabit Ethernet. All jobs are setup and submitted from the Cluster 1 submission site.

For each of the experiments, we submitted two workflows in parallel, a non-adaptive one and an adaptive one. The non-adaptive workflow uses simple round-robin scheduling, whereas the adaptive workflow uses the adaptive scheduling mechanism described in Section 3. If no historic data is available, the scheduling of the adaptive workflow should be equivalent to that of the non-adaptive workflow.

It should be noted that the resources, as detailed above, are not dedicated to the experiments in this paper, so they may be influenced by submissions from other users. However, in order to test the effect of the adaptivity strategy better, in some experiments we also introduced additional (controlled) loads to the clusters. Thus, we group the results of the experiments according to the model for additional external load considered:

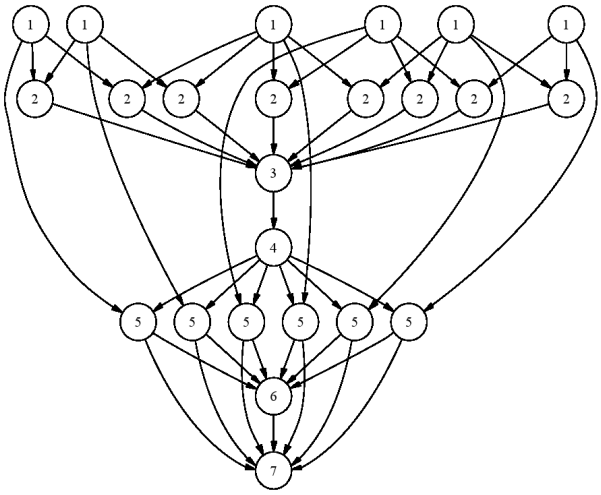


Figure 5. A Simple Montage Workflow [1]

- **No Additional External Load:** For the purposes of the experiment, no external load is applied; the clusters are still, however, subject to third-party external load.
- **Constant Additional External Load:** For the duration of the experiment, additional linear workflows are submitted to a cluster. This has the effect of providing a constant additional external load above any third-party load on the clusters.
- **Temporary Additional External Load:** For a period of time specified in each experiment, linear workflows of a specified size and number are submitted to a cluster, creating a temporary increase in load.

At the end of each experiment, the log files were parsed to produce the results. In order to illustrate long waiting times in the queue for individual jobs of a workflow, the graphs presented plot both the queue time and execution time for each job separately; even though this distinction may not be immediately obvious in the case of experiments using workflows with a relatively large number of tasks, the graphs still indicate trends. The vertical axis of the graphs shows wall-clock time in the form *hours:minutes:seconds*. For each experiment, graphs for non-adaptive and adaptive workflow execution are plotted side-by-side to allow comparison.

4.2. No Additional External Load

Experiment 1: The objective of this experiment is to compare the adaptive and non-adaptive approaches where no additional external load has been submitted to the clusters and there is no historical information on cluster performance. Adaptive and non-adaptive linear workflows (50 tasks each) are submitted in parallel, with access to Clusters 1 and 2, with no

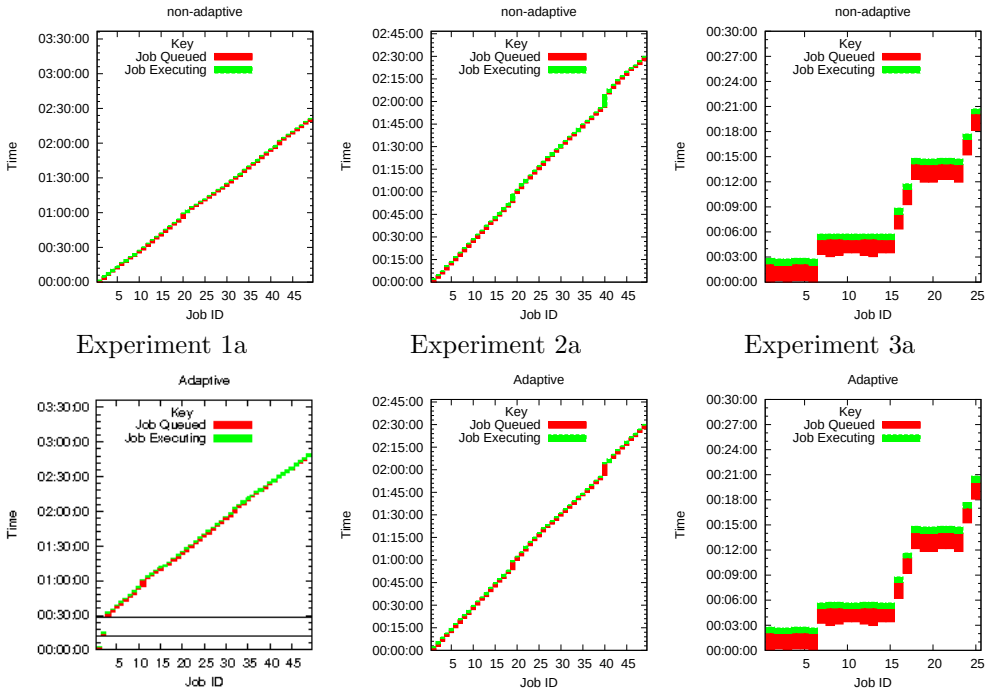


Figure 6. Results of Experiments with No Additional External Load

additional external load. The results of experiment 1 are presented in Figure 6, which shows that the adaptive workflow (1b) performs less well than non-adaptive one (1a). This is because it has to build up knowledge about execution that can form the basis for informed adaptations. When enough knowledge has been gained, an adaptation is performed, which is visible on the graph as a gap in the linear workflow execution. The point in time when, as a result of an adaptation, a new schedule is applied is denoted with an black horizontal line in the graph. The adaptive workflow adapts twice. The gains that result from adaptation are too modest to make up for the cost of adapting. This is because the clusters are performing similarly and consistently across the execution, and thus the original non-adaptive schedule is efficient.

Experiment 2: The objective of this experiment is to compare the adaptive and non-adaptive approaches where no additional external load has been submitted to the clusters and historical information on cluster performance is available. The same workflows are submitted as in Experiment 1. With prior knowledge about the environment (from Experiment 1), the results of experiment 2 are as shown in Figure 6. No adaptations are carried out for this run, and the non-adaptive (2a) and adaptive (2b) workflows perform similarly.

Experiment 3: The objective of this experiment is to compare the adaptive and non-adaptive approaches with no additional external load in the presence of historical information with a more complex workflow. Adaptive and non-adaptive Montage workflows are submitted in parallel, with access to Clusters 1 and 2, with no additional external load. Prior knowledge is available about the environment (from Experiment 1). The results of experiment 3 are shown in Figure 6, which indicates that the clusters act as expected and no adaptations are carried out for this run for either the non-adaptive (3a) or adaptive (3b) execution. Where tasks are run in parallel, this reflects the inherent parallelism of Montage (see Figure 5).

Summary: Once the adaptive infrastructure has been primed with current information about the environment, it correctly refrains from performing adaptations where none are required. The remainder of the experiments assume the availability of historical information about the clusters.

4.3. Constant Additional External Load

Experiment 4: The objective of this experiment is to compare the adaptive and non-adaptive approaches with constant additional external load on the smaller cluster. The same linear workflows are submitted as in Experiment 1, with additional constant external load supplied by the submission of 50 linear workflows (100 tasks each) to Cluster 1 at the start. The results are presented in Figure 7, which shows that the adaptive workflow (4b) changes its schedule early in the workflow execution, leading to a significant improvement in response time of the adaptive workflow compared to the non-adaptive (4a) workflow. The adaptive response time is 17% less than that in the non-adaptive case.

Experiment 5: The objective of this experiment is to compare the adaptive and non-adaptive approaches with constant additional external load on the larger cluster. The same linear workflows are submitted as in Experiment 4, with additional constant external load supplied by the submission of 50 linear workflows (100 tasks each) to Cluster 2 at the start. The results are presented in Figure 7, which shows that the adaptive workflow (5b) changes its schedule once, early in the workflow execution. In this case, the adaptive workflow completes within 1 minute of the non-adaptive workflow (5a) despite having performed an adaptation. This can be explained because the larger cluster can accommodate more load before its queue times increase, leading to adaptations that have less effect.

Experiment 6: The objective of this experiment is to compare adaptive and non-adaptive approaches with constant additional external load on a small cluster with a complex workflow. Adaptive and non-adaptive Montage workflows are submitted in parallel to Clusters 1 and 2, with additional constant external load supplied by submitting 50 linear (100 task each) workflows to Cluster 1 at the start. The results are presented in Figure 7, which shows that the adaptive workflow (6.b) changed the schedule early on in the workflow execution, leading to a significant improvement in performance when compared to the non-adaptive workflow (6.a) . By moving work away from the heavily loaded Cluster 1, long queue times have been avoided, especially for the jobs with *Job Id 10*, *11*, *14* and *15*. The adaptive response time is 38% less than that in the non-adaptive case.

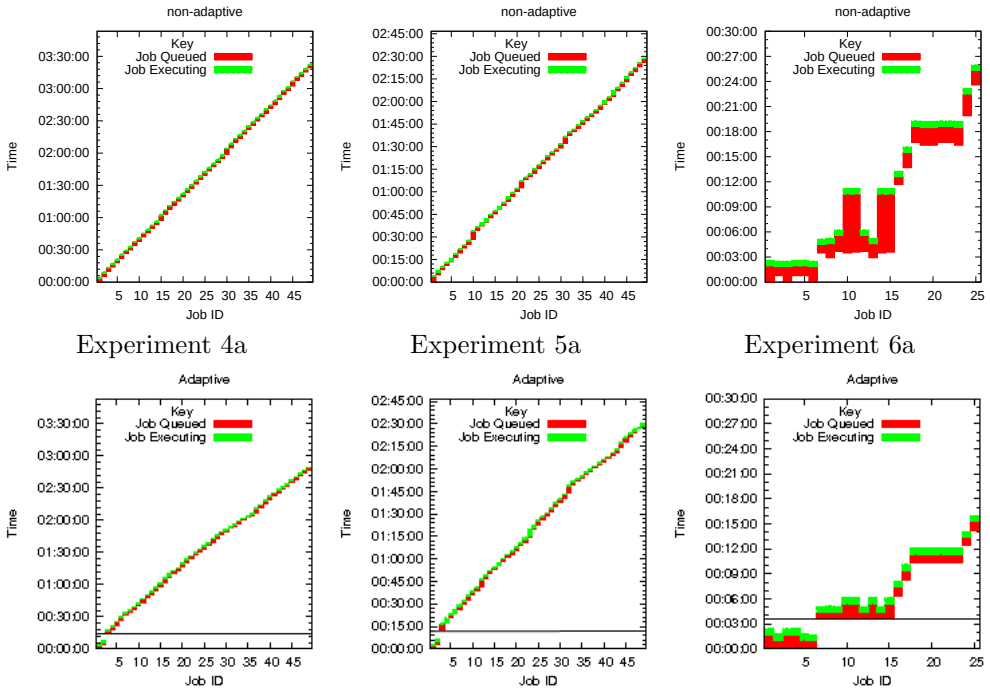
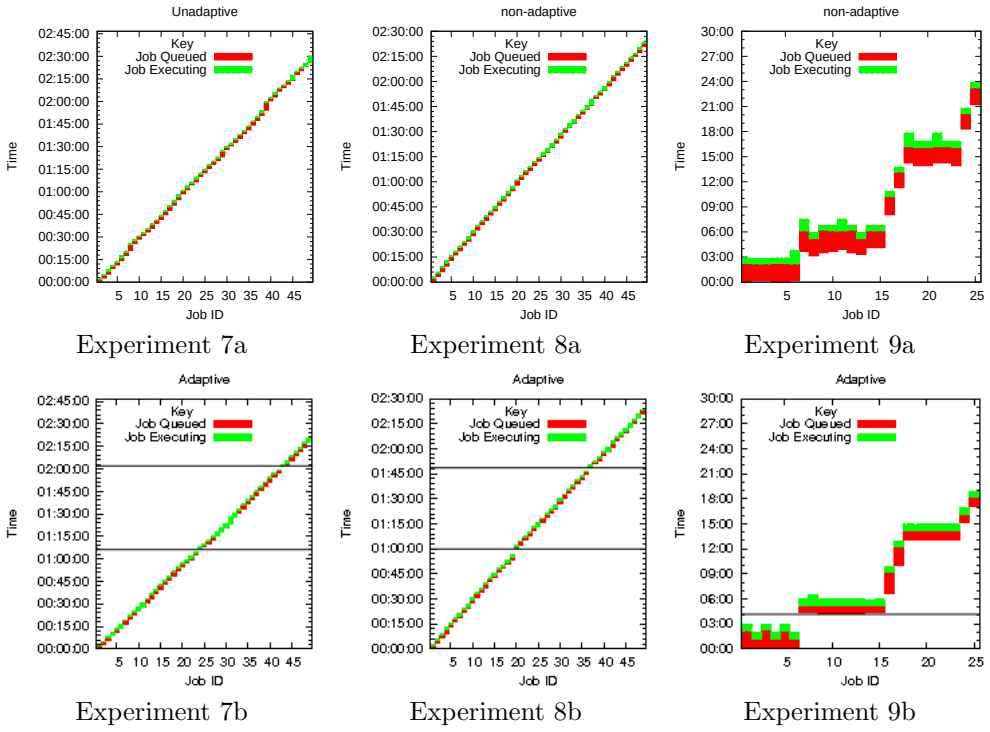


Figure 7. Results of Experiments with Constant Additional External Load

Summary: The constant external load is handled well by the adaptive scheduling scheme; few adaptations are required, but these provide lasting benefits, and significant response time improvements are observed.

4.4. Temporary Additional External Load

Experiment 7: The objective of this experiment is to compare adaptive and non-adaptive approaches with temporary external load on the small cluster with a linear workflow. The same workflows are used as in Experiment 1, with a temporary external load supplied by submitting 50 linear (10 tasks each) workflows to Cluster 1 at 60 minutes into the experiment. The results of the experiment are shown in Figure 8, in which one adaptation is performed just after 60 minutes and another when the temporary workflows complete after 120 minutes. The adaptation has reduced average queue times during the time of additional load, by moving jobs away from the heavily loaded cluster. The adaptive workflow (7b) response time is 7% less than that in the non-adaptive case (7a).



Experiment 7a Experiment 8a Experiment 9a

Experiment 7b Experiment 8b Experiment 9b

Figure 8. Results of Experiments with Temporary Additional External Load

Experiment 8: The objective of the experiment is to compare adaptive and non-adaptive approaches with temporary external load on the large cluster with a linear workflow. The same workflows are used as in Experiment 1, with a temporary external load supplied by submitting 50 linear (10 tasks each) workflows to Cluster 2 at 60 minutes into the experiment. The results for the adaptive (8b) and non-adaptive (8a) workflows for this experiment are shown in Figure 8. In this case, the adaptive workflow performs an adaptation at 60 minutes into the experiment, at the time that additional load is detected on Cluster 2. A further adaptation is performed at around 105 minutes into the experiment after the , after the temporary workflows have completed. Both workflows finish at a remarkably similar time (within 10 seconds), this can be attributed to the load having a minimal overall effect on Cluster 2.

Experiment 9: The objective of the experiment is to compare adaptive and non-adaptive approaches with temporary external load on a small cluster with a complex workflow. Adaptive (9b) and non-adaptive (9a) Montage workflows are submitted in parallel to Clusters 1 and 2, with temporary external load supplied by submitting 50 linear (10 task) workflows to Cluster 1 at 10 minutes into the experiment. The results of the experiment are shown in Figure 8. The

results show that an adaptation is performed once, after 3 minutes. The adaptive workflows jobs are then subject to shorter queue times than the non-adaptive one. Even after the temporary workflows are complete no more adaptations are performed due to the jobs performing well on Cluster 2. The adaptive response time is 21% less than that in the non-adaptive case.

Summary: A temporary external load impedes the progress of a static workflow less than one that is present all the time, so the potential improvements available from the adaptive techniques are reduced compared with the constant external load case. However, adaptation takes place when the temporary external load is introduced, and in one case when it is removed, providing significantly reduced response times.

5. Conclusions

We have presented an approach to adaptive workflow processing that: (i) adds adaptive scheduling to an existing workflow infrastructure with minimal intrusion; (ii) illustrates the use of the MAPE functional decomposition from the autonomic computing community in a new setting, including the use of stream queries for identifying patterns of interest in monitoring events; and (iii) demonstrates significant performance improvements in experiments involving different forms of imbalance and workflows, even though the environment provides limited fine-grained control over the execution timing of individual jobs. Adaptive workflow processing promises to provide more robust performance in uncertain environments. Our experiments also indicate that workflows with a higher degree of inherent parallelism, such as Montage, may benefit more from adaptation. Finally, our work has demonstrated that effective adaptation can be added to an established grid workflow infrastructure at modest development cost, making use of existing facilities for monitoring and control.

Acknowledgments: We are pleased to acknowledge the support of the UK Engineering and Physical Science Research Council. The ISI work was supported by the National Science Foundation under grants: CNS-0615412 and OCI-0722019.

REFERENCES

1. Deelman E, *et al.*. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming* 2005; **13**(3):219–237.
2. Deelman E, Gannon D, Shields M, Taylor I. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems, In Press, Corrected Proof* 2008; .
3. Fahringer T, *et al.*. Askalon: A development and grid computing environment for scientific workflows. *in Workflows for eScience, Scientific Workflows for Grids*, Springer Verlag, ISBN: 978-1-84628-519-6, 2005.
4. Oinn T, *et al.*. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics* 2004; **20**(17):3045–54.
5. Altintas I, Berkley C, Jaeger E, Jones M, Ludascher B, Mock S. Kepler: An extensible system for design and execution of scientific workflow. *in 16th Intl. Conf. on Scientific and Statistical Database Management (SSDBM'04), 21-23 June*, 2004.
6. Taylor I, Shields M, Wang I, Harrison A. The triana workflow environment: Architecture and applications. *Workflows for e-Science*, 2007; 320–339.
7. Blythe J, Jain S, Deelman E, Gil Y, Vahi K, Mandal A, Kennedy K. Task scheduling strategies for workflow-based applications in grids. *IEEE Symposium on Cluster Computing and the Grid*, 2005.

8. Wieczorek M, Prodan R, Fahringer T. Scheduling of scientific workflows in the askalon grid environment. *in SIGMOD Record Volume 34(3)*, 2005.
9. Heinis T, Pautasso C, Alonso G. Design and evaluation of an autonomic workflow engine. *2nd International Conference on Autonomic Computing*, IEEE Computer Society, 2005; 27–38.
10. Duan R, Prodan R, Fahringer T. Run-time optimisation of grid workflow applications. *Proc. Intl. Conference on Grid Computing*, IEEE Press, 2006; 33–40.
11. Lee JH, Chin SH, Lee HM, Yoon T, Chung KS, Yu HC. Adaptive workflow scheduling strategy in service-based grids. *GPC*, Springer, 2007; 298–309.
12. Yu Z, Shi W. An adaptive rescheduling strategy for grid workflow applications. *IPDPS*, IEEE Press, 2007; 1–8.
13. Sakellariou R, Zhao H. A low-cost rescheduling policy for efficient mapping of workflows on grid systems. *Scientific Programming* December 2004; **12**(4):253–262.
14. Lee K, Sakellariou R, Paton NW, Fernandes AAA. Workflow adaptation as an autonomic computing problem. *Proceedings of 2nd Workshop on Workflows in Support of Large-Scale Science*, 2007; 29–34.
15. Topcuoglu H, Hariri S, Wu MY. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.* 2002; **13**(3):260–274.
16. Zhao H, Sakellariou R. Advance reservation policies for workflows. *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, LNCS 4376, Springer-Verlag, 2006; 47–67.
17. Frey J, Tannenbaum T, Livny M, Foster IT, Tuecke S. Condor-G: A computation management agent for multi-institutional grids. *HPDC*, 2001; 55–63.
18. Deelman E, *et al.*. Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, IEEE Computer Society: Washington, DC, USA, 2006.
19. Berriman GB, Others. Montage: A grid enabled engine for delivering custom science-grade mosaics on demand. *Proceedings of the SPIE Conference on Astronomical Telescopes and Instrumentation, 2004* ; .
20. Brown DA, *et al.*. A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis. *Workflows for e-Science, Springer* 2006; .
21. Barish BC, Weiss R. LIGO and the detection of gravitational waves. *Physics Today* 1999; **52**:44–50.
22. Chervenak A, *et al.*. Giggle: A framework for constructing sclable replica location services. *Proceedings of Supercomputing 2002 (SC2002)* November 2002; .
23. Deelman E, *et al.*. Transformation catalog design for griphyn. *Technical Report, GriPhyN-2001-17, www.griphyn.org* 2001; .
24. Fitzgerald S. Grid information services for distributed resource sharing. *Proc. 10th IEEE Intl Symposium on High Performance Distributed Computing*, 2001.
25. Allcock B, Bester J, Bresnahan J, Chervenak AL, Foster I, Kesselman C, Meder S, Nefedova V, Quesnel D, Tuecke S. Data management and transfer in high-performance computational grid environments. *Parallel Computing Journal* 2002; **28**(5):749–771.
26. Baru C, Moore R, Rajasekar A, Wan M. The SDSC Storage Resource Broker. In *CASCON'98*, 1998.
27. Czajkowski K, Foster I, Karonis N, Kesselman C, Martin S, Smith W, Tuecke S. A resource management architecture for metacomputing systems. *Lecture Notes in Computer Science* 1998; **1459**:62–82.
28. Kephart J, Chess D. The Vision of Autonomic Computing. *IEEE Computer* 2003; **36**(1):41–50.
29. Arasu A, Babu S, Widom J. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 2006; **15**(2):121–142.