Journal of Grid Computing (2006) 3: 201–219
DOI: 10.1007/s10723-005-9011-7

Optimizing Grid-Based Workflow Execution

Gurmeet Singh★, Carl Kesselman and Ewa Deelman
Information Sciences Institute, University of Southern California, Marina Del Rey, CA 90292, USA
E-mail: {gurmeet, carl, deelman}@isi.edu

Received 28 May 2005; accepted in revised form 6 December 2005

Key words: Grid computing, optimizing workflow performance, resource allocation, scheduling, workflow execution engine

Abstract

Large-scale applications can be expressed as a set of tasks with data dependencies between them, also known as application workflows. Due to the scale and data processing requirements of these applications, they require Grid computing and storage resources. So far, the focus has been on developing easy to use interfaces for composing these workflows and finding an optimal mapping of tasks in the workflow to the Grid resources in order to minimize the completion time of the application. After this mapping is done, a workflow execution engine is required to run the workflow over the mapped resources. In this paper, we show that the performance of the workflow execution engine in executing the workflow can also be a critical factor in determining the workflow completion time. Using Condor as the workflow execution engine, we examine the various factors that affect the completion time of a fine granularity astronomy workflow. We show that changing the system parameters that influence these factors and restructuring the workflow can drastically reduce the completion time of this class of workflows. We also examine the effect on the optimizations developed for the astronomy application on a coarser granularity biology application. We were able to reduce the completion time of the Montage and the Tomography application workflows by 90% and 50%, respectively.

1. Introduction

Large-scale applications are being built by scientific collaborations in physics [1], astronomy [2, 3], biology [4], earthquake science [5] etc. These applications are often structured as workflows that express an application by specifying a set of interdependent tasks. Due to factors such as the location of or the need for large amounts of computing or storage resources, it is often not possible to execute all of the steps in a workflow on a single computer. The resources available to the scientists are often geographically distributed and belong to different administrative domains. The resources are typically shared among users as part of a Grid infrastructure. Hence, it is often desirable to view the Grid as the target execution environment for application workflows. An application workflow is a set of tasks with data dependencies between them. It can be represented as a directed acyclic graph (DAG) where the vertices are the compute tasks and the edges are the data dependencies between the tasks. The input data required by a task should be available before the task begins execution and the output data produced can be transferred to its child tasks only when the task has completed execution. This is in contrast to the data pipeline model where data is streamed between the tasks [6]. In this paper, we focus on large-scale workflows containing thousands of tasks.

★ Corresponding author.

Much work has focused on developing heuristics for mapping the application tasks to appropriate resources based on performance models in order to minimize the application makespan [7–9]. The makespan of the workflow is defined as the time interval between the start time of the first task and the completion time of the last task in the workflow. The makespan is theoretical in nature and does not include the overhead of executing the application across the mapped Grid resources. The overhead is incurred since the workflow execution engine has to parse the workflow description, identify resources for the tasks, submit tasks to these resources, monitor their execution and analyze the dependencies in the workflow. The execution overhead can be non-negligible due to the distributed nature of the resources, the large number of tasks in the workflow (in thousands) and the complex dependencies between the tasks. The overhead which can be an order of magnitude more than the theoretical makespan, (for large scale, fine computational granularity workflows) can have a significant impact on the overall workflow execution. In this paper, we show that the behavior of the workflow execution engine in executing the workflow can be crucial in determining the actual application makespan. An important distinction is that the experiments in this paper were done using dedicated resources and hence we do not include the wait time associated with provisioning the resources in the execution overhead. This wait time can be arbitrary depending on the workload of the resources and its optimization is a separate problem outside the scope of this paper.

We examined the behavior of a commonly used grid-based workflow execution engine, Condor [10, 11]. We conducted our studies with an astronomy application called Montage [2] and a biology application called Tomography [4]. Montage is a small computational granularity application and for the particular workflow used in this paper, the average runtime of the tasks is 3.3 s on a 1.3 GHz Intel Itanium 2 processor machine with 4,469 tasks in the workflow. Tomography is a coarser granularity application than Montage and the average runtime is 2 min on similar processors with 2,946 tasks in the workflow. Table 1 shows the difference between the theoretical (assuming no overhead) and the actual makespan of these workflows (found by actually executing the workflow) when using 100 dedicated processors with the default behavior of the workflow execution engine.

Table 1. Difference between theoretical and actual runtimes of Montage and Tomography workflows.

	Theoretical	Actual
Montage	<5 min	303 min
Tomography	60 min	135 min

The actual makespan is 60 times more than the theoretical one for Montage and is more than twice the theoretical one for the Tomography workflow. Thus depending on the computational granularity and the scale of the workflow, the execution overhead can be a significant part of the actual makespan even with dedicated resources. Apart from the average runtimes and number of tasks, the workflows used in this work differ in the fact that the Montage workflow has dependencies between the tasks in the workflow whereas the Tomography workflow is a set of independent tasks. We use the Montage workflow for developing a set of optimizations and study how these optimizations perform for a coarse granularity application such as Tomography. Using these optimizations, we were able to reduce the actual makespan by 90% and 50% in case of the Montage and Tomography workflows, respectively.

The paper is structured as follows: Section 2 describes the workflow execution model that identifies the major costs involved in executing the workflow. Section 3 describes the approach, the execution engine and the Montage workflow. Experimental results with the Montage workflow and basic optimizations based on reconfiguring the execution engine are presented in Section 4. Section 5 describes some further optimization methods based on workflow restructuring and distributed workflow execution. Experimental results with the Tomography application are presented in Section 6. Section 7 discusses the results obtained in the paper and their generalization. Related work is presented in Section 8 followed by conclusions.

2. Workflow Execution Model

The workflow is expressed as a set of tasks with dependencies between them. A task becomes executable when its dependencies are met and may be

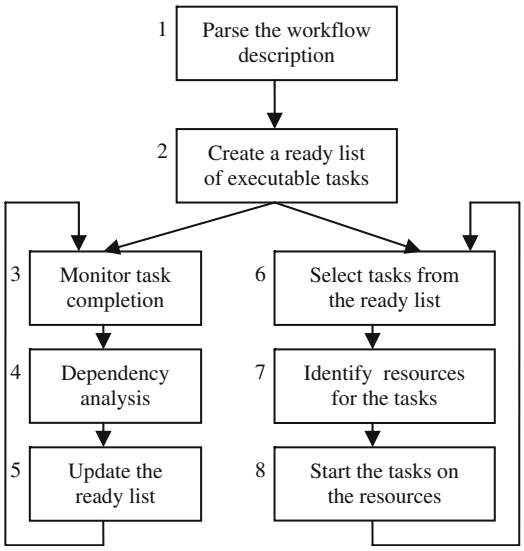

Figure 1. The workflow execution model.

scheduled by submitting it to a resource for execution. Figure 1 shows the major functions that a workflow execution engine must perform to execute a workflow. Prior to workflow execution, a high level mapper is used to map the workflow tasks to Grid resources based on some performance model and create a workflow description where each task is annotated with its target resource. Before starting the workflow execution, the engine parses the workflow description and creates a ready list for holding the executable tasks.

Once the execution has started, there are three main tasks associated with the actual execution of a workflow:

- *Updating the ready list (steps 3,4,5 in Figure 1.)* The ready list is the list of executable tasks. Updating the ready list involves maintaining a status of all the tasks in the workflow, monitoring task completions, determining when tasks have become eligible for execution based on the task completion events and the dependencies in the workflow and adding them to the ready list. Initially, when the workflow starts execution, this list will contain tasks that have no predecessors.
- *Resource matching (steps 6,7 in Figure 1.)* This involves identifying the resources for a subset of tasks in the ready list. The subset may include any combination of tasks in the list. The resource matching may be done every time a task is added to the ready list or periodically. Note that a high level mapper has already mapped the workflow tasks to Grid resources. In case the mapped resource is a single host, the resource identification part is trivial. However, when the mapped resource is a cluster, the resource identification step is responsible for low-level mapping and tries to find an available host from that cluster for running the task.
- *Task submission (step 8 in Figure 1.)*. This involves contacting the identified host for the task, sending the task description to it, transferring the input files, and starting a process which monitors the progress of the submitted task on the resource. This might also involve authorization and authentication with the host.

Due to the large number of tasks (in thousands) and dependencies in the workflow and the nature of the execution resources (shared or dedicated, remote or local), there is a cost associated with each of the above operations. The experiments in this paper explore how the efficiency of the workflow execution engine in performing these operations can affect the makespan of large scale, fine granularity workflows. In the rest of this paper, the term job and task would be used interchangeably to refer to a compute node in the workflow graph.

3. Approach and Experimental Setup

In order to understand the costs associated with the significant operations in workflow execution, we performed a series of experiments in which a large-scale scientific workflow with thousands of tasks was executed using a commonly used workflow engine on an operational Grid infrastructure. In this section, we describe the workflow execution engine and the application workflow used for these experiments.

3.1. *Condor as the Workflow Execution Engine*

We use Condor [12] as the workflow execution engine (ver 6.7.1). Condor was originally designed as a high-throughput resource management system and was initially used for opportunistically scheduling jobs on a set of distributively owned machines

known as the Condor pool. Note that the application workflow could be mapped to any Grid resource that need not be managed by the Condor resource management system. The only requirement is that the Grid resources must use the Globus GRAM service [13]. For our experiments, initially the Condor pool consists of couple of machines (local to the user), one of them acts as the central manager of the pool and the other would be used for workflow submission purposes (submit host). Then, a Condor function called a *glide-in* [14] is used to extend the pool with the remote Grid resources using GRAM for certain duration. During this duration, these resources appear to be dedicated nodes in the Condor pool and can be used for task execution using the Condor services.

Condor can be thought of as consisting of two major components: a task sequencing engine called DAGMan [11] and a task scheduler (*schedd*). DAGMan submits ready jobs to the scheduler by monitoring the status of currently executing jobs and analyzing the dependencies in the workflow. The scheduler maintains a job queue and schedules jobs onto resources in the pool by consulting a *negotiator* running on a central manager (the resource manager of the pool). It then passes the job to an execution service called *startd*. The job status is recorded in a log file that is parsed by DAGMan to update dependency information. The components of Condor and their relationship are summarized in Figure 2. DAGMan and the task scheduler (*schedd*) need to be collocated on the same machine which is known as the submit host. As mentioned earlier, our Condor pool consists of two local machines, one of them acts as the submit host and the other as the central manager of the pool. All the machines used for task execution, also known as worker nodes are remote Grid resources, drafted in temporarily into the Condor pool using *glide-in* [14]. A worker node can possibly run multiple tasks concurrently by hosting multiple *startd* execution services. However, for the experiments described in this paper, a worker node

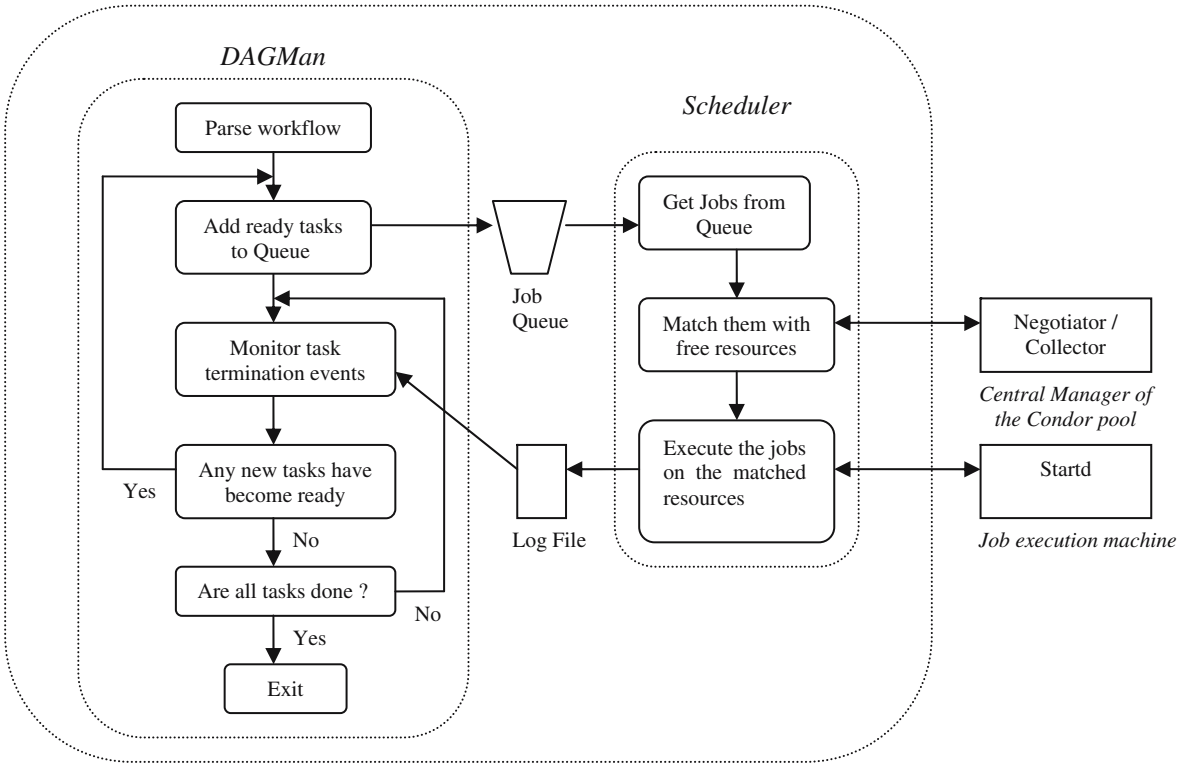

Figure 2. Workflow execution in Condor.

only hosts one *startd* service and executes one task at a time.

Earlier we discussed the costs associated with workflow execution in abstract terms. In the context of the Condor resource management system these costs (along with the responsible entities) would translate into the following.

1. Updating the ready list is equivalent to submitting jobs into the job queue on the submit host (DAGMan, scheduler).
2. Resource matching is equivalent to scheduling jobs to available machines in the pool. (scheduler, negotiator)
3. Submitting the tasks for execution is equivalent to dispatching them to the matched machines for execution (scheduler, startd).

In Section 4 we explore how the job submission rate, the scheduling interval and the dispatch rate affect the makespan of the workflow.

3.2. *Montage Workflow*

Montage is a data-intensive astronomy application to create custom image mosaics of the sky on demand [2]. There are four major tasks involved in building a mosaic using Montage [15].

At first the input images are reprojected to a common spatial scale and coordinate system. Then the background radiation in images is modeled to achieve common flux scale and background level by minimizing the inter-image differences. The images are then rectified to a common flux scale and background level. Lastly the reprojected and background corrected images are coadded to yield the final mosaic.

Figure 3 shows the structure of a small Montage workflow. The vertices in the workflow are the various tasks such as image reprojection, background modeling, coaddition etc. The edges represent the data dependency between the tasks. We assign a level to each task in the workflow indicated by the number inside the vertices in Figure 3. This level assignment will help us in workflow restructuring based optimizations as will be described in Section 5. The top-level tasks in the workflow (those that do not have any predecessors) are assigned level 1. All the tasks that become ready for execution when the tasks

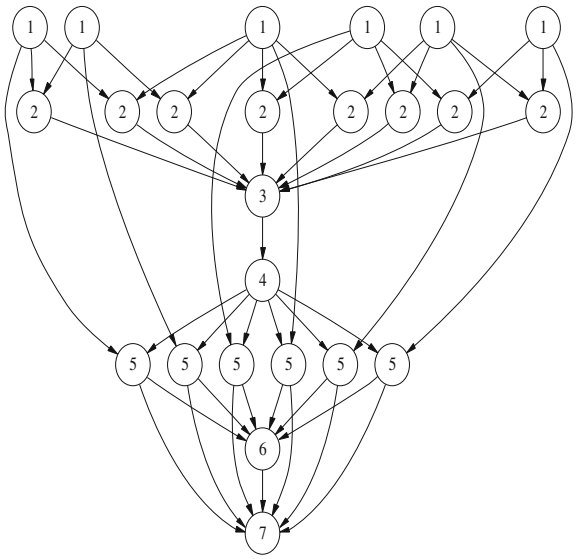

Figure 3. A small Montage workflow.

at level 1 complete are assigned level 2 and so on. All the tasks at level k have a predecessor at level k-1. An important property to note is that the tasks at the same level are independent of each other. Also, all the tasks at the same level in the Montage workflow are similar, i.e., the same program invoked with different input data. Due to this the runtimes of the tasks at the same level do not differ much. This allows us to characterize the workflow by describing the number of tasks at each level of the workflow and their average runtimes.

4. Experiments

The experiments performed can be classified into two categories. The first set of experiments (described in this section) is done by modifying the behavior of the execution engine. This modification is done by changing the configuration parameters of the engine that affect the execution cost such as the scheduling interval (*resource matching*), dispatch rate (*task submission*) and submission rate (*updating the ready queue*). No changes are done to the workflow structure. The second set of experiments (described in next section) restructure the workflow in addition to distributing the workflow submission functionality i.e. using multiple submit hosts instead of only one. All the experiments examine the effect

Table 2. Runtime and number of tasks at various levels of the Montage workflow.

Level	Number of tasks	Runtime (in seconds)
1	892	8.2
2	2,633	2
3	1	68
4	1	56
5	892	1
6	25	6
7	25	40

of the changes on the makespan of the workflow. The Montage workflow used in the experiments creates a five square degree mosaic of the sky centered at M16. The workflow contains 4,469 jobs. Table 2 gives the average runtime and the number of tasks at each level of the workflow.

The Condor pool used for the experiments contains one submit host (used for submitting the workflow), one central manager (resource manager for the pool) and 100 worker nodes as shown in Figure 4. In order to eliminate variability that may result from shared use, this pool was only populated with worker nodes drafted in temporarily using *glide-in* [14]. For the duration that these worker nodes were part of the pool, they were available exclusively for executing the tasks in the workflow.

To ensure homogeneity, these nodes were all allocated from the NCSA TeraGrid cluster [16]. The cluster nodes were a combination of 1.3 and 1.5 GHz Intel Itanium 2 processors with 2 GB memory per node.

As mentioned before, we do not include the wait time associated with getting the resource allocated in the execution overhead. In order to achieve this, in all the experiments, the workflow is submitted to DAGMan for execution when the worker nodes are present in the Condor pool. The makespan is the time difference between when the workflow is submitted for execution to DAGMan and when the last task in the workflow completes execution. In the rest of this paper, we will use makespan and completion time interchangeably to refer to this time difference. Since Montage is a data intensive application, an important issue is the data transfer between the tasks in the workflow. When the tasks in the workflow are mapped to different resources, then the data will have to be transferred between resources if the parent and child tasks are mapped to different resources. This transfer time will get included in the makespan and will not allow us to measure the execution overhead precisely. In order to avoid such transfers, the entire workflow is mapped to the same resource (NCSA TeraGrid cluster). The data transfer between the tasks is done using a shared file system accessible from all the nodes in the cluster and the runtimes of the tasks include the time to read and write the data to the file system. The input data for the workflow is prestaged to the cluster before the workflow is submitted for execution.

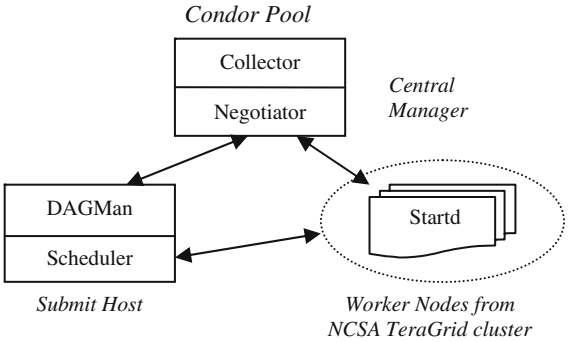

Figure 4. Execution environment.

4.1. *Resource Provisioning*

In our experiments we used Condor *glide-in* [14] to provision the execution resources ahead of time. Resource provisioning implies that even though the resources are usually shared and belong to different administrative domains, they are dedicated for our use for certain timeframe.

If the resources are not provisioned ahead of time, the workflow execution engine may have to interact with multiple resource management systems to identify suitable resources and the submitted tasks may have to wait in the job queue of the identified resources before they can begin execution. The use of *glide-in* eliminates this possibility and allows for experiments to isolate and examine the overheads introduced by the workflow execution engine.

4.2. *Baseline Condor Performance*

First we evaluate the performance obtained using Condor's default configuration. Figure 5 shows the execution timeline of the workflow.

The *X*-axis denotes the time (in minutes) since the workflow was submitted to DAGMan for execution. The *Y*-axis denotes the tasks in the workflow. For

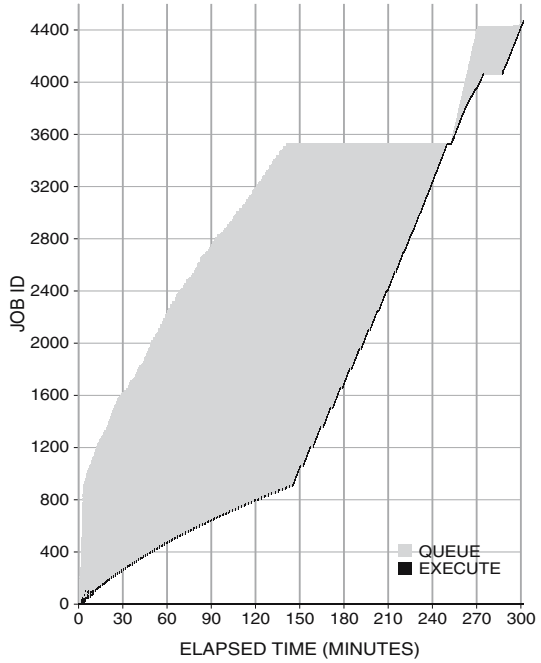

Figure 5. Workflow execution graph with default Condor parameters.

each task, a horizontal line represents its overall runtime. The gray portion in this line denotes the time the task spent waiting in the job queue on the submit host and the black portion denotes the time when it was executing on the worker node. With the default behavior of the execution engine, the tasks spent most of their time waiting in the queue even with 100 nodes in the pool and the workflow takes 303 min to complete.

In the following sections, we study how we can improve the overall runtime by modifying the behavior of the workflow execution system.

4.3. *Scheduling Interval*

The process of identifying resources for the submitted jobs is called a negotiation cycle. During a negotiation cycle, the scheduler on the submit host tries to find available machines for all the jobs in its queue by consulting the negotiator on the central manager. The interval between two successive negotiation cycles is the scheduling interval. The scheduling interval in Condor can be controlled in a variety of ways. The first option is to have a fixed scheduling interval. The second option is to start a negotiation cycle upon submission of each job at a rate no greater than once every 20 s. A negotiation cycle can be a costly operation because the scheduler attempts to find machines for each and every job in its queue even when all the worker nodes are currently busy. With thousands of jobs in the queue, a negotiation cycle can last for 2 to 3 min with most of the time spent in a fruitless search for worker nodes.

4.3.1. *Scheduling at Fixed Intervals*

Figure 6 shows the workflow execution graphs when the scheduling interval is fixed to 30 s, 5 and 10 min

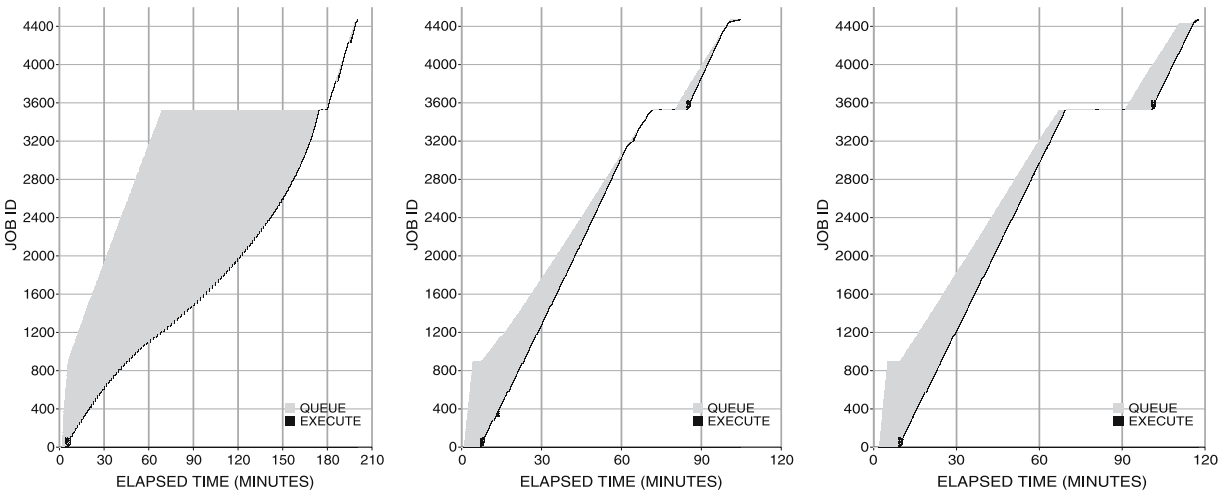

Figure 6. Workflow execution with fixed scheduling intervals; 30 s (*left*), 5 min (*center*), 10 min (*right*).

A short scheduling interval can help in matching jobs with available machines expeditiously. However, since the scheduler must both match resources and dispatch jobs, time spent in resource matching can decrease the job dispatch rate. A longer scheduling interval allows the scheduler to dispatch jobs to matched resources uninterrupted for longer intervals. However, the submitted jobs may have to spend more time in the queue waiting to be matched with the resources.

Due to these reasons, as Figure 6 shows, a scheduling interval of 5 min (completion time 105 min) works better than a scheduling interval of 30 s (completion time 201 min) or 10 min (completion time 118 min).

4.3.2. *Scheduling at Each Job Submission*

Here, each job submission starts a new negotiation cycle (but not within 20 s of the last one). In addition, in the absence of any job submission events, the scheduling occurs at the predefined fixed interval. Figure 7 shows the workflow execution graph when the predefined interval is fixed at 30 s, 5 and 10 min resulting in a workflow completion time of 218, 146 and 145 min, respectively. In this scheduling strategy, scheduling happens every 20 s for most of the workflow lifetime. Such short scheduling interval leads to an increase in the workflow makespan.

Figure 8 gives a comparison of the workflow completion times with various scheduling intervals and scheduling policies. As the figure shows, scheduling at fixed intervals work better than scheduling at each job submission for all values of the scheduling interval. In the latter case, since scheduling happens very frequently (every 20 s) when jobs are being submitted to the queue, the dispatch rate of the scheduler suffers. Hence, the workflow completion time increases. In addition, in this case longer fixed intervals are better since there is no need for a negotiation cycle when jobs are not being submitted. In the limiting case, when the scheduling happens based only on the job submission events (no predefined fixed interval), the workflow takes 140 min to complete. While the above analysis would seem to imply that short scheduling intervals are necessarily bad, this is not always the case. For example when there are only few jobs in the queue or if the scheduler can terminate the negotiation cycle on detecting the first resource allocation failure, then short scheduling intervals are preferable. Both of these conditions lead to short negotiation cycles. Thus the optimum value of the scheduling interval for minimizing the workflow makespan is related to the duration of the negotiation cycle.

The observations above have to be qualified by the fact that once the scheduler has matched a number of jobs with machines during a negotiation cycle, it uses those *claimed* machines for executing

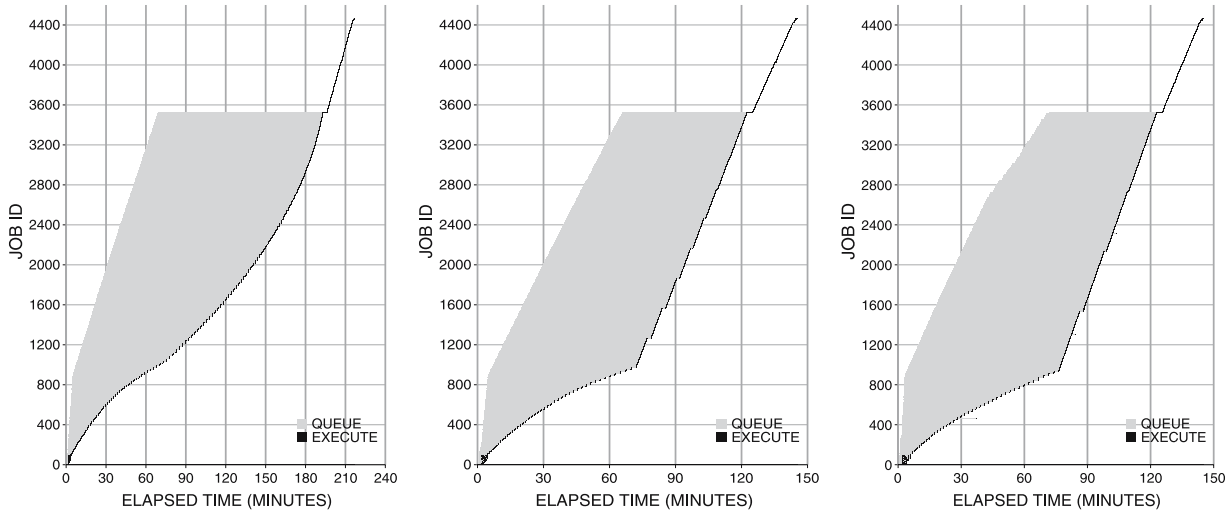

Figure 7. Scheduling at each job submission and at 30 s (*left*), 5 min (*center*), 10 min (*right*) intervals.

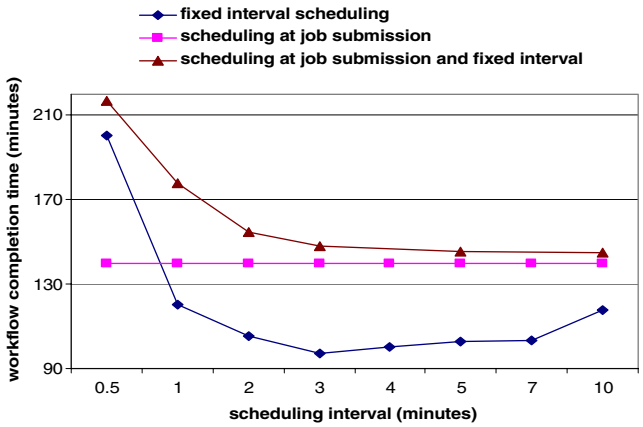

Figure 8. Workflow completion times with various values of the scheduling interval and scheduling policies.

other eligible jobs in its queue until the negotiator forces it to vacate those machines (due to user priorities) or it runs out of jobs. Thus ideally no negotiation cycle is required once there are enough number of tasks in the queue and all the machines in the pool are claimed by the scheduler on the submit host. However, it is not possible to implement this ideal policy with the basic configuration options.

4.4. *Dispatch Rate*

The dispatch rate is the rate at which the scheduler can start the jobs on the remote resources (worker nodes). In Condor, the task of starting a job on a remote resource involves creating a shadow process that is responsible for creating the environment and managing the job on the remote resource. The dispatch rate can be throttled by specifying the JOB_START_DELAY. The scheduler must wait for this duration of time before creating the next shadow process. The recommended delay is 2 s (used in Section 4.2). This artificial throttling reduces the load on the scheduler and on the submit host as it prevents them from having to manage the startup activity all at once. Figure 9 shows the workflow execution graph when this delay is set to 0, 1, and 2 s. The scheduling was done at each job submission and at 5 min intervals (the default behavior). When the dispatch delay is reduced to zero, the number of waiting jobs in the

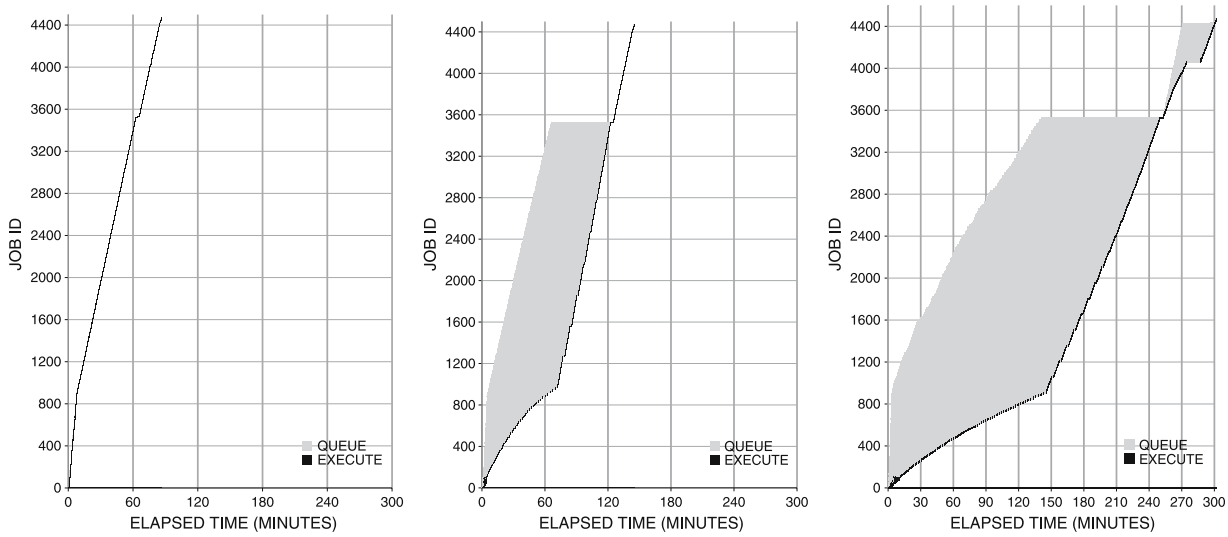

Figure 9. JOB_START_DELAY set to 0 s (*left*), 1 s (*center*) and 2 s (*right*).

queue is drastically reduced, and hence a short scheduling interval works well in this case.

Using the recommended delay of 2 s, the workflow completes in 303 min. The dispatch rate of the scheduler increases when the delay is reduced to 1 and 0 s and the workflow completes in 146 and 87 min, respectively. Because there is a large number of short running jobs in the workflow, even a small dispatch delay incurred for each of them add up to a significant delay for the whole workflow.

The optimal scheduling interval for the workflow also depends on the dispatch rate since the dispatch rate affects the number of jobs in the scheduler queue. The JOB_START_DELAY was 1 s in Section 4.3 resulting in a dispatch rate of one job per second. A fixed scheduling interval of 3 min gives the minimum completion time for this dispatch rate. However, if JOB_START_DELAY is 0 s, then lower value of the scheduling interval is preferable.

Figure 10 shows the workflow completion time with a fixed scheduling interval of 30 s to 10 min and two values of the JOB_START_DELAY. When the JOB_START_DELAY is zero, then the minimum workflow completion time is 80 min with a 30 s scheduling interval. An effective strategy with a fast dispatch rate is to do scheduling at each job submission. In the rest of the Montage experiments described in this paper, we use a JOB_START_DELAY = 0 and scheduling at each job submission (subject to a 20 s minimum gap between two successive scheduling events).

4.5. *Job Submission Rate*

The third factor that we considered is the rate at which DAGMan can submit jobs to the job queue on the submit host. DAGMan has to monitor job completion events and based on the dependencies of the workflow it has to determine when jobs become executable and submit them to the queue. Figure 9 showed that when the dispatch rate is faster than the job submission rate, the latter becomes the limiting factor. In Figure 9 (center and right) due to the slower dispatch rate, the jobs have to wait in the Condor queue (indicated by the gray portion in the graphs). In these cases, a faster submission rate would only lead to longer wait times for jobs. However, in Figure 9 (left), due to a faster dispatch rate the jobs do not have to wait in the queue. Thus there is potential for decreasing the workflow completion time further by increasing the job submission rate. The next section shows how the job submission rate can be increased by exploiting the workflow structure.

5. Restructuring the Workflow

The motivation for restructuring the workflow arises out of Figure 9 (left), which shows the job submission rate of DAGMan. There is a change in the job submission rate after the first 892 jobs have been submitted. The job submission rate slows down from 2.1 job submissions per second to 0.8 job submissions per second after this point. A glance at the application workflow (Table 2) reveals that the first 892 jobs do not have any dependencies. The rest of the jobs in the workflow have dependencies that must be satisfied before these jobs become executable. Thus, it is obvious that the dependency analysis slows down the job submission rate.

We explored the restructuring of the workflow so that the dependencies in the workflow graph can be

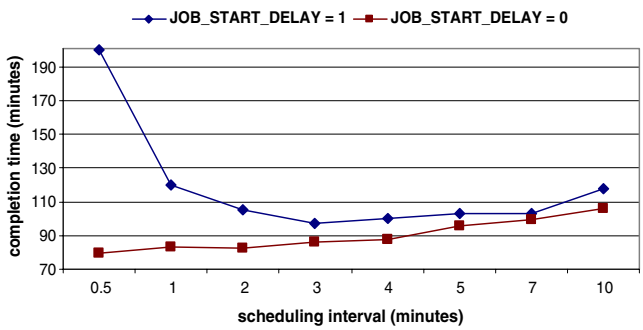

Figure 10. Workflow completion time with various values of the dispatch rate and the scheduling interval.

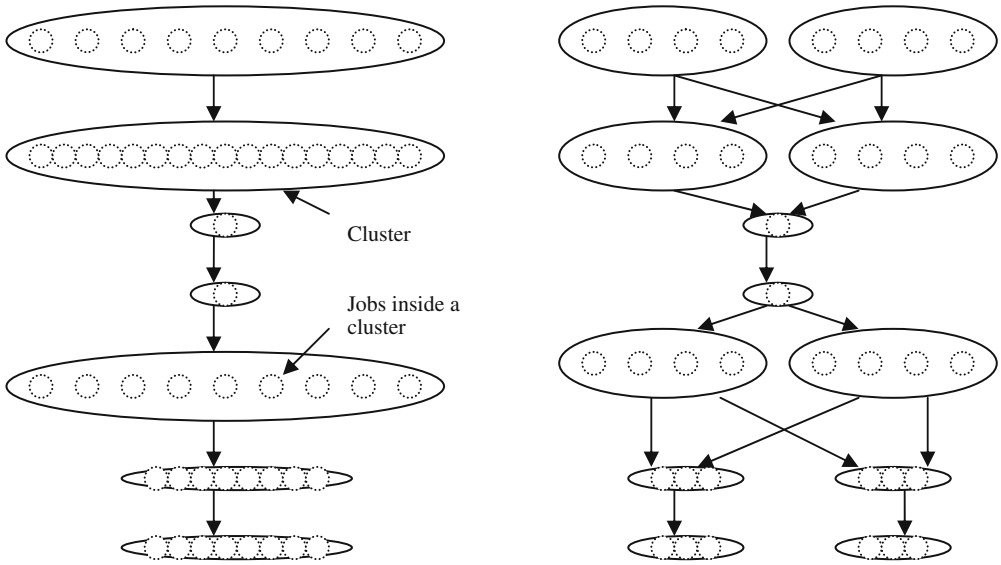

Figure 11. Restructured workflow. One cluster per level (*left*) and two clusters per level (*right*).

reduced. We group independent jobs at the same level (explained in Section 3.2) in the workflow into clusters. Figure 11 shows the restructured workflow graph with one cluster per level (left) and two clusters per level (right).

Now the dependencies are only between the clusters. Since there are only a handful of them, the dependency analysis is simplified. However, it has to be emphasized that this restructuring is only for job submission purpose. Clustering does not imply that all the tasks in a cluster are to executed on the same processor as is done by the traditional clustering algorithms [17]. It also does not imply that the tasks in each cluster might be executed sequentially as might be the case with hierarchical task graphs [18].

Each cluster is implemented using a wrapper program that takes as input a list of jobs in that cluster and submits them into the job queue. So any job in the workflow can still execute on any available worker node. The performance gain achievable can vary based on the implementation of the wrapper program. In the next section we use Condor DAGMan as the wrapper and in Section 5.2 we use a custom written wrapper.

5.1. *Implementing Clusters with DAGMan*

In this section, DAGMan is used as the wrapper program. Since there is no dependency between the jobs in a cluster, DAGMan can submit them at a faster rate. This results in a scenario where DAGMan is used for executing the main workflow as well as the clusters in the workflow. These are separate unrelated instantiations of the DAGMan program. Figure 12 shows the workflow execution graph when using one cluster per level and using two clusters per level.

With one cluster per level, the workflow now completes in 46 min. The job submission rate is constant at about 1.8 job submissions per second. Using two clusters per level, the job submission rate is even faster since two DAGMan processes can run in parallel.

However, in this case the job submission rate becomes more than the job dispatch rate resulting in accumulation of jobs in the job queue. The scheduler on the submit host becomes overwhelmed resulting in an increase in the workflow completion time.

5.2. *Using a Custom Wrapper Program*

In this case, we have written a custom wrapper program that submits all the jobs in the cluster into the job queue. It monitors their progress and terminates when all of them have completed. It utilizes a Condor feature incidentally also called *clustering* which allows multiple jobs to be specified in the same submit file and submitted using a single

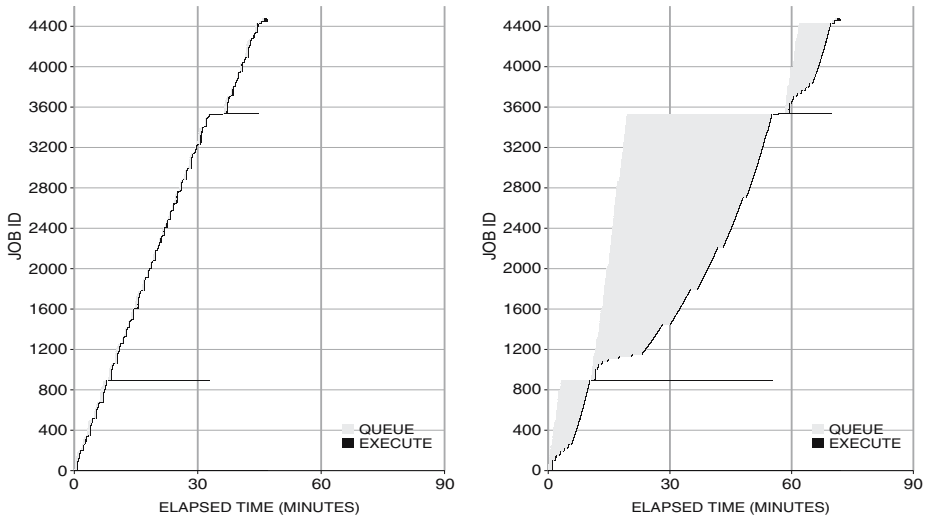

Figure 12. The workflow execution graph with one cluster per level (*left*) and two clusters per level (*right*).

scheduler invocation. The only requirement is that the jobs in the submit file should be independent and have the same program executable. Both of these requirements are satisfied in our application. Thus, all the jobs in the cluster are submitted using a single submit file.

The job submissions are very fast because this involves only a single call to the scheduler for submitting the whole cluster instead of one call per job in the cluster. Another advantage of using clustering is that the Condor scheduler can be configured such that if it fails to find a match for an idle job, it will not try to match any other idle jobs in the same cluster during that negotiation cycle. This leads to shorter and much efficient negotiation cycles. In the following experiments, we show the workflow performance using clustering with centralized (single submit host) and distributed (multiple submit hosts) job submission.

5.2.1. *Centralized Job Submission*

Figure 13 shows the workflow execution graph using this approach with one cluster per level. The jobs in a cluster are submitted in the same time it took to submit a single job earlier and the job submission rate is no longer the limiting factor in the workflow completion time. The vertical edges in the graph show that all the jobs in each cluster were submitted to the queue at the same instant. The whole workflow is submitted and executed using a single submit host and completes in 30 min (a reduction of 90% from the baseline workflow completion time of 303 min, Section 4.2).

The workflow completion time is no longer limited by the job submission rate. Instead, it now depends on the rate at which the scheduler can dispatch the jobs for execution.

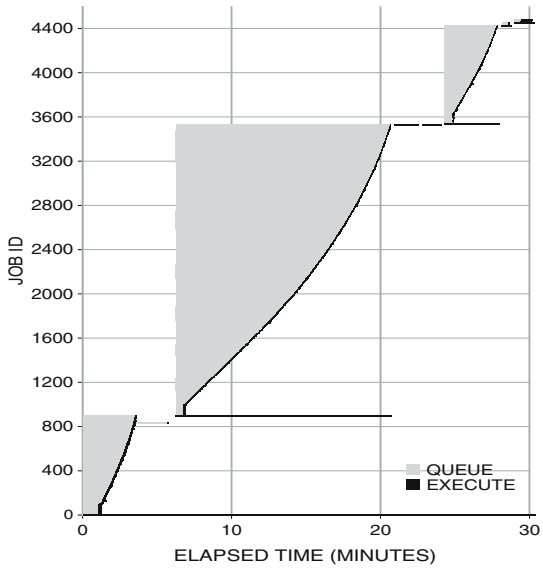

Figure 13. Workflow execution graph with condor clustering with centralized job submission.

5.2.2. *Distributed Job Submission*

In order to increase the dispatch rate, we investigate using multiple schedulers in parallel. Since there is only one scheduler per submit host, we have to increase the number of submit hosts in our Condor pool and submit jobs in a distributed fashion. We modify the execution environment to include multiple submit hosts as shown in Figure 14.

The central manager has job submission capability and the submit hosts have both the job submission and execution capability. The worker machines only have the job execution capability. The workflow is restructured with multiple clusters at each level. The number of clusters at each level is equal to the number of submit hosts in the pool. The main workflow is now submitted from the central manager. The clusters at each level are made to execute on one of the submit hosts by using the *requirements* attribute in Condor.

After the restructured workflow is submitted from the central manager using DAGMan, the sequence of events for the execution of jobs at each level are as follows:

1. Each cluster is matched with one of the submit hosts.
2. When the cluster (our wrapper program) starts executing on the submit host, it submits all the jobs in the cluster to the job queue on that host.
3. The schedulers on the submit hosts try to find suitable nodes for the submitted jobs during the negotiation cycle.
4. The matched jobs are dispatched to the worker nodes for execution.

Figure 15 shows the workflow execution graph when using one, two, and three submit hosts with the same number of clusters per level leading to a workflow completion time of 42, 25, and 34 min, respectively. The decrease in workflow completion time depends on the sharing of worker nodes between the submit hosts. In the case of two submit hosts, both of them get a fair share of the nodes and so both the schedulers can work in parallel. In the case of three submit hosts (Figure 15, right), two of the submit hosts got all the nodes and the third had to wait until either of the first two were done. Thus, only two of the schedulers could work in parallel. The added overhead of distributing the jobs increase the workflow completion time.

Newer versions of Condor (6.7.3) provides features for submitting jobs remotely (Condor-C) that may make it possible to implement distributed job submissions without using any external wrapper. However, the overhead associated with these mechanisms is yet to be evaluated.

6. Tomography Application

Tomography is another application that we use for studying the performance of the execution engine. The Tomography application is a set of independent tasks that are used for deriving 3D structures from a series of 2D electron microscopic images. Tomography allows for the reconstruction and detailed structural analysis of complex structures such as synapses and large structures like dendritic spines [4]. The tasks in the Tomography application are not only independent of each other but also are of coarser granularity than the Montage application. The average runtime of a Tomography task is around 2 min and there are 2,946 tasks in the workflow used for the experiments in this section. This application allows us

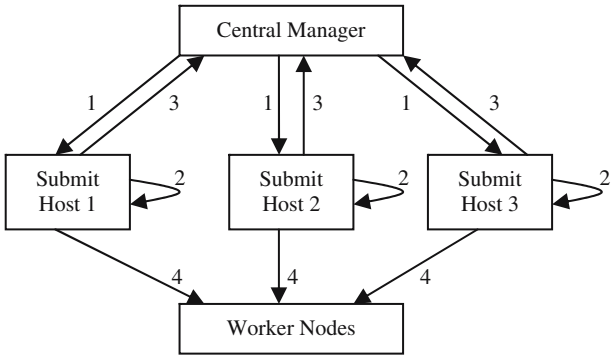

Figure 14. Distributed job submission scenario.

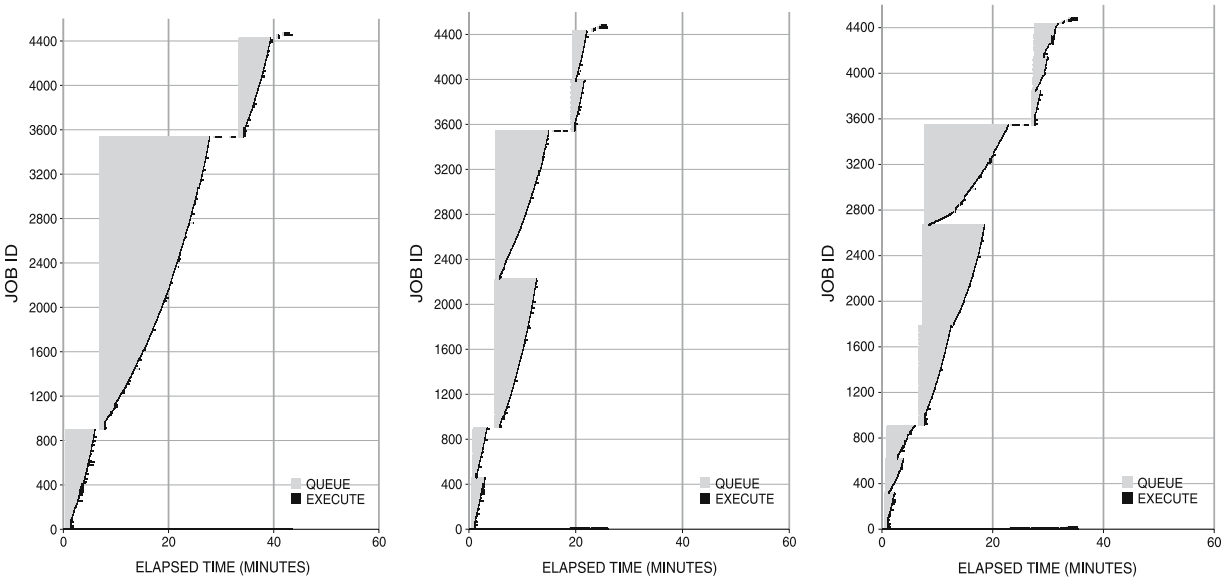

Figure 15. Distributed workflow execution with one (*left*), two (*center*), and three (*right*) clusters per level.

to explore the performance impact of the execution engine on the completion time of a coarse granularity application that does not have dependencies.

6.1. *Baseline Performance*

The workflow takes 135 min to complete with the baseline Condor configuration parameters.

Since the theoretical execution time of the workflow with 100 processors is approximately 1 h, the total execution overhead is less in case of Tomography than it was in case of Montage (Figure 16). For Montage, the completion time was 5 h while the theoretical execution time was less than 5 min.

6.2. *Scheduling Interval*

The Tomography application workflow has no dependencies and all the workflow tasks are submitted to the job queue on the submit host within the

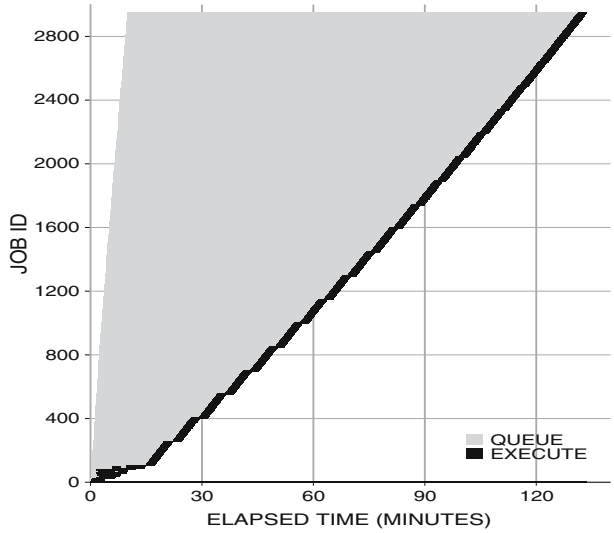

Figure 16. Baseline Tomography workflow execution graph.

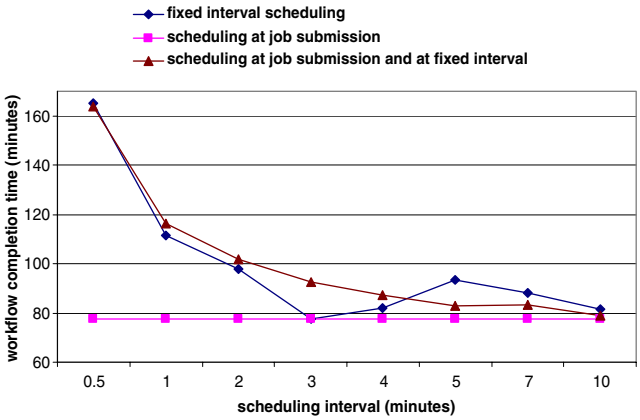

Figure 17. Workflow completion times with various values of the scheduling interval and scheduling policies.

first 10 min. The rest of the time is spent in executing the tasks on the worker nodes in the Condor pool. Figure 17 shows the workflow completion time with various values of the scheduling interval and scheduling policies.

In the case of the Tomography application, scheduling at job submission only works the best because scheduling is required only in the initial 10 min when jobs are being submitted to the queue. Later no scheduling event is required. As can be seen from Figure 17, there is not much difference between scheduling at fixed intervals and scheduling at each job submission in addition to fixed intervals. The reason is that after the first 10 min when all the jobs are submitted, both of them are essentially the same.

6.3. *Dispatch Rate*

In the case of the Montage workflow, the dispatch rate had a large impact on the workflow completion time. In the case of the Tomography application, the dispatch rate does not have a major impact on the completion time. The baseline Condor performance for Tomography is obtained using a scheduling interval of 5 min in addition to scheduling at each job submission with a dispatch rate of one job every 2 s (JOB_START_DELAY = 2). With the same configuration, if we reduce the dispatch delay to 1 s and 0, the completion time is 83 and 89 min, respectively. Thus, there is a large reduction in the completion time by reducing the dispatch delay from

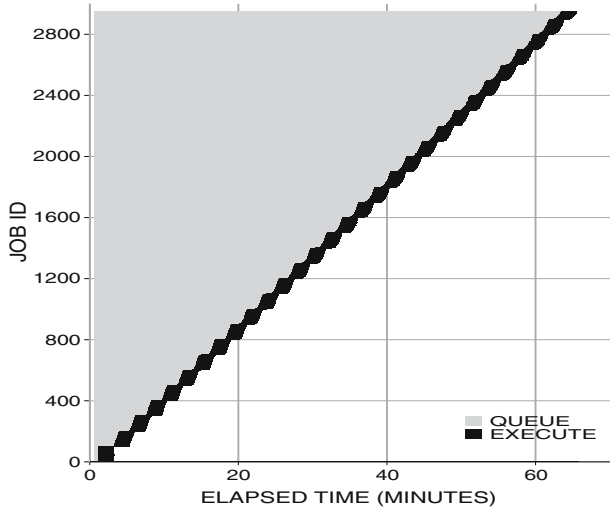

Figure 18. Tomography workflow execution with Condor clustering.

2 to 1 s but further reduction in the dispatch delay leads to a marginal increase in the completion time. This apparent anomaly can be explained by the fact that with a dispatch delay of 1 s or less, the total dispatch delay for 2,946 jobs is 50 min which is less than the theoretical execution time of the workflow. In such a situation, the effect of changing the dispatch rate is marginal and other factors such as the scheduling interval or the load on the submit host are more predominant.

6.4. *Job Submission Rate*

All the jobs in the Tomography application are independent of each other and hence DAGMan can submit them at a faster rate.

The only way to increase the submission rate further is to use Condor clustering as described in Section 5.2. Figure 18 shows the workflow execution graph for the Tomography application when all the jobs in the workflow are put in a single cluster. The workflow completes in 64 min which is within 10% of the theoretical execution time.

We have also performed experiments with another astronomy application called Galaxy Morphology [3]. Galaxy Morphology is also a fine granularity application like Montage. Using a similar set of optimization developed for the Montage workflow, we were able to reduce the workflow completion time of the Galaxy Morphology application by 75% [19].

7. Discussion

The theoretical execution time of the Montage workflow with 100 processors is under 5 min. Thus it is alarming that the workflow took 303 min to complete with the default configuration. This provided the motivation for this study on understanding the costs associated with workflow execution and the factors affecting the workflow completion time. We isolated and studied the effect of job submission rate, the scheduling interval, and dispatch rate on the workflow execution. It is difficult to do so in general since changing one of these may have an indirect influence on the others. For example, in the case of the Montage workflow, the optimal scheduling strategy and the scheduling interval depends on the dispatch rate.

If the dispatch rate is high enough, the job submission rate can become the limiting factor. With the original job submission rate, the workflow could not be completed in less than 87 min. By exploiting the workflow structure to group independent jobs and submitting them in a single call to the scheduler using Condor clustering, the workflow completion time could be reduced to 30 min. Thus it is critical to have a high submission rate and the mechanism that we have used for increasing the submission rate is to restructure the workflow.

Finally, the dispatch rate was the limiting factor. The effect of the dispatch rate on the completion time depends on the number of available resources and the runtime of tasks. With 100 available worker nodes and small jobs, a single scheduler was not able to utilize all of them. In this case, multiple schedulers were used to increase the dispatch rate.

The effect of the scheduling interval (and its duration) depends on the length of the job queue and the behavior of the scheduler (negotiate all the jobs or only a subset). Ideally the duration of the scheduling event should be proportional to the number of available resources and not the number of submitted jobs. This can be achieved in Condor using clustering and configuring the scheduler to stop negotiating as soon as it cannot find a match for an idle job in the cluster. The length of the job queue depends on the job submission rate and dispatch rate. If the job queue is small or the number of jobs considered for scheduling can be restricted then smaller scheduling intervals are preferable otherwise longer intervals tend to work well. For example, in the case of the Montage workflow when the dispatch rate is one job per second and the job queue has a large number of jobs, the best fixed scheduling interval is 3 min but when the dispatch delay is zero and the job queue is small, the best fixed scheduling interval is 30 s.

Workflow restructuring allows us to increase the job submission rate and distribute the workload among multiple submit hosts. The restructuring only implies a change in the job submission mechanism for the jobs in the workflow. The level-based partitioning used for restructuring works well for us since all the jobs at the same level are similar (same runtime). However, it introduces a synchronization point between levels since any job at a lower level cannot be submitted until all the jobs at the higher

level clusters have finished. In spite of this, it helps us to reduce the workflow completion time.

The use of the Tomography application demonstrates the extent to which the performance is affected by the factors considered in this paper when the workflow contains no dependencies and is at a coarser granularity. The techniques developed with the Montage workflow are still useful and lead to a workflow completion time that is within 10% of what is theoretically achievable. Overall, there is a 50% reduction in completion time from 135 to 64 min. There are differences between the Montage and the Tomography application in the effect of the dispatch rate and the scheduling interval on the completion time. The first difference is that since the Tomography application consists of a set of independent jobs, scheduling at each job submission works best because as soon as all the jobs in the workflow are submitted, no more scheduling events are required. Secondly, when the dispatch rate is fast enough to keep the resources in the pool busy, further reduction in the dispatch delay does not bring significant reduction in the workflow completion time.

It is important to note that the experiments in this paper were done in the context of a stable execution environment with resources provisioned ahead of time. If the resources in the execution environment can change dynamically, the best value for the scheduling interval needs to be reexamined. Also for a large scale, fine computational granularity application such as Montage, the load of the submit host can have a significant impact on the workflow performance. Thus it is preferable to use a lightly loaded machine as the submit host.

Though this study was done in the context of the Condor system, we believe the factors affecting the workflow performance in the case of Condor would be present in other workflow engines too. The costs involved in parsing dependencies and submitting tasks, identifying resources for the ready tasks and initiating their execution on the identified resources are not implementation specific but can be explained in terms of the general execution model depicted in Figure 1. Similar costs exist in other workflow execution engines like the Workflow Enactment Engine (WFEE) [20] and GridAnt [21]. In case of the former, a workflow coordinator (WCO) is responsible for monitoring the status of the tasks and activating the child tasks when they become eligible. An event service server (ESS) is used for notification purposes. Active tasks register their status with the ESS, which in turn notifies the WCO. Based on the status received from the ESS, WCO may active the child tasks (similar to DAGMan functionality). The activated tasks query a resource discovery service when they become eligible for execution which is similar to scheduling at each job submission. The WFEE supports multiple middleware components (e.g., Gridbus broker, Globus [22] and web services) and hence the cost associated with initiating the task on the remote resource depends on the middleware used. Authors in [20] note that a gap exists between the parent task end time and the child task start time due to the execution engine running overhead, communication delay, event notification process, resource identification delay and event processing. This is just a different categorization (in the context of the WFEE) of the generic costs that we have investigated in this paper. GridAnt [21] uses a commodity tool called Ant as the workflow execution engine. The cost of dependency management depends on the implementation of this tool. There is no cost associated with identifying resources as the description of each task in the workflow contains identification information for the execution resource. There is a cost associated with dispatching the task on the execution resource as the GridAnt system uses Globus [22] as the dispatch mechanism. This involves contacting the remote GRAM service [13] and submitting the task description to it.

8. Related Work

There are many projects on workflow management and execution on the Grid [20, 21, 23–25]. They differ in the amount of support for workflow composition, dynamic resource scheduling and fault tolerance. The design and implementation of a workflow execution engine (WFEE) is described in [20]. GridAnt [21] is based on an extension of the Java ANT project build tool. The performance of the workflow manager is not discussed in any of the above papers.

The efficiency of resource brokering is considered in [26]. The jobs in this study are independent whereas in our case they belong to a workflow. In addition, the focus is on resource brokering whereas

we also consider the issues related to job submission and dispatching to remote resources. Policies for improving the response time of small jobs in the Condor system is considered in [27]. In our case, improving the response time of a single job in the workflow at the expense of others is not helpful since we are interested in reducing the completion time of the whole workflow.

There is a wealth of literature on clustering algorithms for directed acyclic graphs. A survey can be found in [17]. However, these clusters are actually resource allocation decisions that map tasks in the same cluster to the same processor. In our case each cluster is just a mechanism to reduce the job submission and the scheduling overhead for the tasks in the cluster. The tasks in the cluster are not constrained to execute on the same execution machine. On a similar note, a compiler can partition a program into a hierarchy of parallel tasks by generating hierarchical task graphs (HTG) [18]. The decision whether to expand a task into parallel subtasks or to run it sequentially is made at runtime based on the load conditions. Thus it is used to dynamically adjust the granularity of the program at runtime. The difference between the hierarchical task representation and the workflow restructuring that we have done is that we do not change the execution granularity of the workflow. The workflow executes at its full parallelism. While our clusters are similar to the composite nodes in the HTG, they are never constrained to run sequentially.

The nature of the execution environment also affects the optimal values of the configuration parameters. The effect of the scheduling interval on the makespan of parameter sweep applications with different quality of information has been studied in [28]. A short scheduling interval was found to work well in their case since it allowed the scheduling algorithms to adapt to the dynamic environment. Since we have used a stable execution environment with resource provisioning, we found short scheduling intervals to be undesirable particularly with a large number of idle jobs in the queue. Rescheduling under uncertainty has also been studied in operational research [29]. The focus is on the placement of scheduling events. *Continuous* rescheduling and *periodic* rescheduling as described in the paper are similar to scheduling at each job submission and scheduling at fixed intervals. However both in [28] and [29] the purpose of scheduling is also to handle uncertainty in the execution environment while in our case the purpose of scheduling is only resource allocation and sequencing.

9. Conclusions

Large-scale scientific workflows are complex to compose and execute on the Grid. There are several projects working on workflow composition and execution on the Grid [20, 21, 23, 25, 30]. The focus of the community has been on building easy to use workflow composition interfaces, standardizing workflow description languages, developing mapping heuristics for the tasks in the workflow and developing workflow enactment engines. While these are essential first steps, it is important to start a discussion on the performance aspects of the workflow execution engine. So far, only the performance of the workflow tasks on the target resources is considered for developing an optimal mapping between the tasks and the resources in order to minimize the application makespan. However, the overheads of executing the workflow across Grid resource can significantly impact the actual makespan particularly for large scale, fine granularity workflows.

We conducted our experiments using a widely used workflow execution engine (Condor) and an operational Grid infrastructure (TeraGrid [16]). Our studies were performed in the context of a fine granularity astronomy application and a coarse granularity biology application. As we have shown, the performance of the execution engine is critical for determining the completion time for fine granularity workflows. We were able to reduce the workflow completion time by 90% in case of the Montage and 50% in the case of the Tomography application.

Acknowledgements

We would like to thank the Condor team for answering the queries that we had while performing the experiments and writing the paper. This work is supported by NSF under grants ITR-0086044 (GriPhyN), ITR AST0122449 (NVO), and Cooperative Agreement number CCR-0331645 (VGrADS). This work is supported in part by the National Institutes of Health, National Institute of Neurological Disorders and Stroke

program for Collaborative Research in Computational Neuroscience (CRCNS) (NIH grant # R01 NS046068).

References

1. E. Deelman, et al., "GriPhyN and LIGO, Building a virtual data grid for gravitational wave scientists," in *11th Intl Symposium on High Performance Distributed Computing*, 2002.
2. B. Berriman, et al., "Montage: A Grid-enabled image mosaic service for the NVO," in *Astronomical Data Analysis Software and Systems (ADASS) XIII*, 2003.
3. E. Deelman, et al., "Grid-based galaxy morphology analysis for the national virtual observatory," in *SC*, 2003.
4. E. Deelman, et al., "Pegasus: Mapping scientific workflows onto the grid," in *2nd EUROPEAN ACROSS GRIDS CONFERENCE*, 2004, Nicosia, Cyprus.
5. C. Kesselman, I. Foster and T. Prudhome, "Distributed telepressence: The NEESgrid earthquake engineering collaboratory," in *The Grid: A Blueprint for a New Computing Infrastructure*, I. Foster and C. Kesselman, (eds.), 2004, Morgan Kaufmann.
6. W. Du and G. Agrawal, "Filter decomposition for supporting coarse-grained pipelined parallelism," in *Parallel Processing, 2005. ICPP 2005. International Conference on*, 2005.
7. M. Iverson and F. Ozguner, "Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment," in *Heterogeneous Computing Workshop, 1998. (HCW 98) Proceedings, 1998 Seventh*, 1998.
8. M. Maheswaran and H.J. Siegel, "A dynamic matching and scheduling algorithm for heterogeneous computing systems, in *Heterogeneous Computing Workshop, 1998. (HCW 98) Proceedings, 1998 Seventh*, 1998.
9. P. Markenscoff and Y.Y. Li., "Scheduling a computational dag on a parallel system with communication delays and replication of node execution," in *Parallel Processing Symposium, 1993., Proceedings of Seventh International*, 1993.
10. J. Frey, et al., "Condor-G: A Computation management agent for multi-institutional grids," in *10th International Symposium on High Performance Distributed Computing*, IEEE Press, 2001.
11. DAGMan, http://www.cs.wisc.edu/condor/dagman.
12. M. Litzkow, M. Livny and M. Mutka, "Condor – A hunter of idle workstations," in *Proc. 8th Intl Conf. on Distributed Computing Systems*, 1988, pp. 104–111.
13. K. Czajkowski, et al., "A resource management architecture for metacomputing systems," in *4th Workshop on Job Scheduling Strategies for Parallel Processing*, 1998, Springer, pp. 62–82.
14. Condor_Glidein, http://www.cs.wisc.edu/condor/glidein.
15. D.S. Katz, et al., "A comparison of two methods for building astronomical image mosaics on a grid," in *Parallel Processing, 2005. ICPP 2005 Workshops. International Conference Workshops on*, 2005.
16. C. Catlett, The philosophy of TeraGrid: Building an open, extensible, distributed TeraScale facility. in *Cluster Computing and the Grid 2nd IEEE/ACM International Symposium CCGRID2002*, 2002.
17. Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, 31 (4) (1999) 406–471.
18. C.D. Polychronopoulos, "The hierarchical task graph and its use in auto-scheduling," in *Proceedings of the 5th international conference on Supercomputing 1991*, ACM: Cologne, West Germany pp. 252–263.
19. G. Singh, C. Kesselman and E. Deelman, "Optimizing grid-based workflow execution," in *CS Tech report 05–851. 2005*, University of Southern California, available at http://www.cs.usc.edu/Research/TechReports/05–851.pdf.
20. J. Yu and R. Buyya, "A novel architecture for realizing grid workflow using tuple spaces," in *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, 2004.
21. K. Amin, et al., "GridAnt: A client-controllable grid workflow system," in *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, 2004.
22. I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Journal of Supercomputer Applications*, 11 (2) (1997) 115–128.
23. J. Cao, et al., "GridFlow: Workflow management for grid computing," in *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, 2003.
24. S. Hwang, and C. Kesselman, "Grid workflow: A flexible failure handling framework for the Grid," in *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*, 2003.
25. S. Majithia, et al., "Triana: A graphical web service composition and execution toolkit," in *Web Services, 2004. Proceedings. IEEE International Conference on*, 2004.
26. P. Crosby, D. Colling and D. Waters, Efficiency of resource brokering in grids for high-energy physics computing. Nuclear Science, IEEE Transactions on, 2004, 51(3): pp. 884–891.
27. G.D. Ghare, and S.T. Leutenegger, "Improving small job response time for opportunistic scheduling," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings. 8th International Symposium on*, 2000.
28. H. Casanova, et al., "Heuristics for scheduling parameter sweep applications in grid environments," in *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*, 2000.
29. H. Aytug, et al., "Executing production schedules in the face of uncertainties: A review and some future directions," *European Journal of Operational Research*, 2005, 161(1): pp. 86–110.
30. J. Kim, M. Spraragen and Y. Gil, "An intelligent assistant for interactive workflow composition," in *Intelligent User Interfaces*, J. Vanderdonckt, N.J. Nunes, and C. Rich, Editors. 2004, ACM. pp. 125–131.