Dynamic Resource Provisioning in Grid Environments

Ioan Raicu[*], Yong Zhao[*], Catalin Dumitrescu[*], Ian Foster[#*+], Mike Wilde[#+]

{iraicu,yongzh,catalind}@cs.uchicago.edu, {foster,wilde}@mcs.anl.gov
[*]Department of Computer Science, University of Chicago, IL, USA
[+]Computation Institute, University of Chicago & Argonne National Laboratory, USA
[#]Math & Computer Science Division, Argonne National Laboratory, Argonne IL, USA

Abstract

Batch schedulers commonly used to manage access to parallel computing clusters are not typically configured to enable easy configuration of application-specific scheduling policies. In addition, their sophisticated scheduling algorithms can be relatively expensive to execute. Thus, for example, applications that require the rapid execution of many small tasks often do not perform well. It has been proposed that these problems be overcome by separating the two tasks of provisioning and scheduling. This paper focuses on resource provisioning, the various allocation and de-allocation policies, and how dynamic and adaptive provisioning can be in light of varying workloads. We couple the proposed dynamic resource provisioning (DRP) with an existing system, Falkon, which is used for the scheduling of tasks to the provisioned resources. We describe the DRP architecture and implementation, and present performance results for both microbenchmarks and applications. Microbenchmarks show that DRP can allocate resources on the order of 10s of seconds across multiple Grid sites and can reduce average queue wait times by up to 95% (effectively yielding queue wait times within 3% of ideal); furthermore, applications (executed by the Swift parallel programming system) reduce end-to-end run time of up to 90% for large-scale astronomy and medical applications, relative to versions that execute tasks via separate scheduler submissions.

Keywords: dynamic resource provisioning, batch scheduler, interactive resource usage, Grid computing

1 Introduction

Many interesting computations can be expressed conveniently as data-driven task graphs, in which individual tasks wait for input to be available, perform computation, and produce output. Systems such as DAGMan [1], Karajan [2], Swift [3], and VDS [4] support this model. These systems have all been used to encode and execute thousands of individual tasks.

In such task graphs, as well as in the popular master-worker model [5], many tasks may be logically executable at once. Such tasks may be dispatched to a parallel compute cluster or (via the use of grid protocols [6]) to many such clusters. The batch schedulers used to manage such clusters receive individual tasks, dispatch them to idle processors, and notify clients when execution is complete.

This strategy of dispatching tasks directly to batch schedulers has two disadvantages. First, because a typical batch scheduler provides rich functionality (e.g., multiple queues, flexible task dispatch policies, accounting, per-task resource limits), the time required to dispatch a task can be large—30 secs or more—and the aggregate throughput relatively low (perhaps two tasks/sec). Second, while batch schedulers may support different queues and policies, the policies implemented in a particular instantiation may not be optimized for many tasks. For example, a scheduler may allow only a modest number of concurrent submissions for a single user. These factors can cause problems when dealing with many tasks.

One solution to this problem is to transform applications (manually or automatically) to reduce the number of tasks. However, such transformations can be complex and/or may place a burden on the user. Another approach is to employ multi-level scheduling [7, 8]. A first-level request to a batch scheduler allocates resources to which a second-level scheduler dispatches tasks. The second-level scheduler can implement specialized support for task graph applications. Frey et al. [9] and Singh et al. [10] create an embedded Condor pool by "gliding in" Condor workers to a compute cluster, while MyCluster [11] can embed both Condor pools and SGE clusters. Singh et al. [12, 13] report 50% reductions in execution time relative to a single-level approach.

We seek to achieve further improvements by:

- Using an adaptive provisioner to acquire and/or release resources as application demand varies

- Reducing average queue wait times by amortizing high overhead of resource allocation over the execution of many tasks

To explore these ideas, we defined and implemented an architecture that permits the embedding of different provisioning and scheduling strategies. We have implemented a range of provisioning strategies, and evaluate their performance. We also integrated provisioning into Falkon, a Fast and Light-weight tasK executiON framework, which handles the scheduling and dispatching of independent tasks to provisioned resources. We use synthetic applications to demonstrate the benefits of adaptive provisioning, and quantify the effects of various allocation and de-allocation policies. Finally, results for two applications involving many small tasks demonstrate that substantial speedups can be achieved for real scientific applications.

2 Related Work

Frey and his colleagues pioneered the application of resource provisioning to clusters via their work on Condor "glide-ins" [9]. Requests to a batch scheduler (submitted, for example, via Globus GRAM) create Condor "startd" processes, which then register with a Condor resource manager that runs independently of the batch scheduler. Others have also used this technique. For example, Mehta et al. [13] embed a Condor pool in a batch-scheduled cluster, while MyCluster [11] creates "personal clusters" running Condor or SGE. Such "virtual clusters" can dedicated to a single workload; thus, Singh et al. find, in a simulation study [12], a reduction of about 50% in completion time. However, because they rely on heavyweight schedulers to dispatch work to the virtual cluster, the per-task dispatch time remains high, and hence the wait queue times are likely to remain significantly higher than in the ideal case due to the schedulers' inability to push work out faster.

In a different space, Bresnahan et al. [25] describe a multi-level scheduling architecture specialized for the dynamic allocation of compute cluster bandwidth. A modified Globus GridFTP server varies the number of GridFTP data movers as server load changes.

Appleby et al. [23] were one of several groups to explore *dynamic resource provisioning within a data center*. Ramakrishnan et al. [24] also address *adaptive resource provisioning* with a focus primarily on resource sharing and container level resource management.

In summary, this work's innovation is the combination of dynamic resource provisioning and Falkon to provide a fast and lightweight scheduling overlay on top of virtual clusters with the use of standard grid protocols for adaptive resource allocation. This combination of techniques allows us to achieve lower average queue wait times, lower end-to-end application run times, while also offering applications the ability to trade-off system responsiveness, resource utilization, and execution efficiency.

3 Architecture and Implementation

3.1 Execution Model

The *resource acquisition policy* determines when and for how long to acquire new resources, and how many resources to acquire. The *resource release policy* determines when to release resources.

Resource Acquisition Policy. We have implemented various *resource acquisition policies,* which decide when and how to acquire new resources. This policy determines the state information that will be used to trigger new resource acquisitions. It also determines the number of resources to acquire based on the appropriate state information, as well as the length of time for which the resources should be required. Having decided that *n* resources should be acquired, we then need to determine what request(s) to generate to the LRM to acquire those resources. We have implemented four different strategies, with a fifth that we could implement if the LRMs support it.

The first strategy, *Optimal***,** assumes that we can query the resource manager to determine the maximum number of resources available to us. We then simply request that number if it is less than *n*, and request *n* otherwise. This policy has not been implemented due to the fact that this feature is not the common case among LRMs and what they expose to applications. The TeraGrid [26] for example is a case where such information is available, but it is done via batch queue prediction mechanisms which can be used to statistically predict the number of resources that could be allocated in a relatively short period of time.

The other strategies assume that we cannot obtain this maximum number via a query. In the *One-at-a-time* strategy, we submit *n* requests for a single resource. In the *All-at-once* strategy, we issue a single request for *n* resources. In the *Additive*, strategy, for *i*=1, 2, …, the *i*th request requests *i* resources; thus, $\left\lfloor (\sqrt{8n+1}-1)/2 \right\rfloor$ requests are required to allocate *n* resources. Finally, in the *Exponential* strategy, for i=1, 2, …, the *i*th request requests 2^{i-1} resources. Thus, $\lceil \log_2(n+1) \rceil$ requests are required to allocate *n* resources. For the purpose of this paper and the experiments conducted in this paper that pertained to the resource provisioning, we used the all-at-once strategy and did not explore the other strategies due to space restrictions.

Resource Release Policy. We distinguish between centralized and distributed resource release policies. In a *centralized* policy, decisions are made based on state information available at a central location. For example: "if there are no tasks to process, release all resources," and "if the number of queued tasks is less than *q*, release a resource." In a *distributed* policy, decisions are made at individual resources based on state information available at the resource. For example: "if the resource has been idle for time t, the acquired resource should release itself." Note that resource acquisition and release policies are typically not independent: in most batch schedulers, one must release all resources obtained in a single request at once. In the experiments reported in this paper, we used a distributed policy, releasing resources after a specified idle time.

3.2 Architecture

As illustrated in Figure 1, our dynamic resource provisioning (DRP) system comprises: (1) user(s); (2) a DRP Utilizing Application (i.e. a Web Service such as Falkon); (3) the Provisioner; (4) a Resource Manager (i.e. GRAM, Condor, PBS, etc); and (5) a resource pool. The interaction between these various components is as follows.

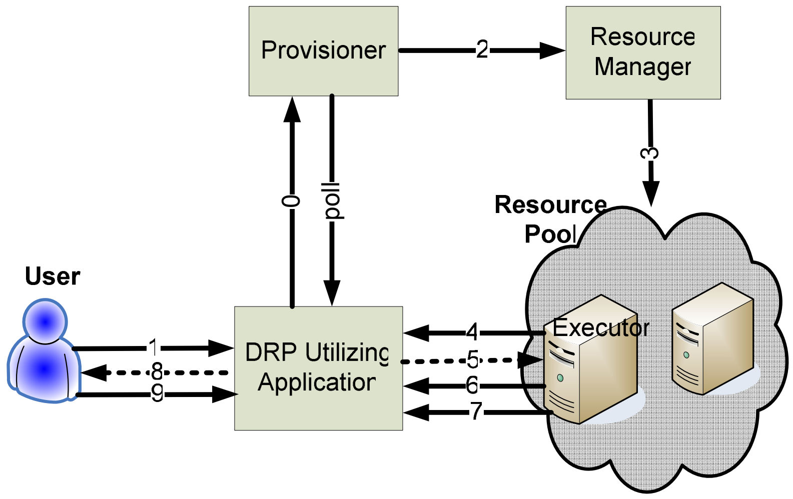

Figure 1: Dynamic Resource Provisioning Architecture

The DRP utilizing application initializes the provisioner with a set of configuration parameters via message (0). These parameters include: the state that needs to be monitored and how to access it, the rule(s) and conditions under which the provisioner should allocate/de-allocate resources, the location of the worker code that is specific to the DRP utilizing application, the minimum/maximum number of resources it should allocate, the minimum/maximum length of time resources should be allocated for, and the allowed idle time per resource before resources are de-allocated. Once the provisioner was initialized, the application would be ready to interact with its users and process work.

The users submit work to their application via message (1), which internally queues up the work making it ready for processing by an executor. The provisioner monitors the internal queue state of the application via message (poll), and based on the rules and conditions from the initialization phase, the provisioner makes the decision how many resources and for how long to allocate. When the provisioner detects the need to allocate more resources, it contacts the Resource Manager with the appropriate resource allocation via message (2). In our implementation, these resources are allocated using GRAM in order to abstract away all the local resource managers that could be used in Grids (PBS, LSF, Condor, etc). The resource manager is used to bootstrap the executor that is specific to the DRP utilizing application with message (3), which then registers with message (4) with the application and becomes ready to process work. Once the application has executors available for work, it sends notifications in message (5) directly to executors that work is available for pickup, after which the executors that received the notifications contact the application directly to pick up the relevant work in message (6). When the work results are complete, the executors delivers the results back to the application in message (7), which then triggers a notification in message (8), and finally leading to the user collecting the final result from the application in message (9).

At first sight, this seems to be relatively complex and likely to add overhead to the execution of application work. It should be noted that while these executors are available (which is dictated by the resource de-allocation policies), any subsequent work requests from the user can simply use the same resources that have already been allocated (according to the resource allocation policy), without the need to go through the entire allocation process. After the initial phases in which resources are allocated and executors are started, a high volume of work broken down into many smaller tasks can essentially be performed using just messages 1, 5, 6, 7, 8, and 9. With some optimizations (task bundling and piggy-backing [cite Falkon]), these messages can be reduced on average to only message 7 and 8 per task.

3.3 Multi-Site Provisioning

Comment [IR1]: Should these be part of the execution model?

Although single site resource provisioning can have many benefits, it has the limitation of limited resources that are typically found at any single site. The main advantage of single site provisioning is the fact that it is simple and trivial to decide where to allocate resources when they are needed. We believe it is very useful to expand the provisioner to enable it to allocate resources across multiple Grid sites. This adds the complexity of deciding what sites to use in the allocation of all of a partial set of the needed resources. We explore several policies for multi-site provisioning: random, round-robin, over-allocation, and probabilistic provisioning.

Random provisioning: Randomly selects a site with enough available resources to allocate the needed resources. If no site has enough available resources, the allocation is en-queued for later (when enough resources are available) evaluation, processing, and allocation.

Round-robin provisioning: Iterates over all Grid sites in a round-robin fashion in order to offer load balancing of allocated resources.

Over-allocation provisioning: Allocates the needed resources at n different Grid sites, where n is configurable and allows the control of how much over-allocation to perform.

Probabilistic Provisioning: Assuming that the Grid sites have a batch queue wait prediction system [cite] in place, such as the one the TeraGrid [cite] has, the provisioner can query the various Grid sites to determine which sites will have the least queue wait times for the needed resources.

3.4 Provisioning in Falkon

Falkon, a Fast and Light-weight tasK executiON framework, provides a system for scheduling and dispatching of independent tasks to a set of executors. Integrating the provisioning mechanisms proposed in this paper into Falkon gives Falkon expanded capabilities to dynamically deploy and run executors across multiple Grid sites based on Falkon's load (i.e. wait queue length). Falkon consists of a dispatcher and zero or more executors (Figure 2); the provisioner is added as a third component that acts as a mediator between he dispatcher and the Grid resources on which the executors are to run on. The dispatcher accepts tasks from clients and implements the dispatch policy. The provisioner implements the resource acquisition policy. Executors run tasks received from the dispatcher. Components communicate via Web Services (WS) messages, except for notifications are performed via a custom TCP-based protocol.

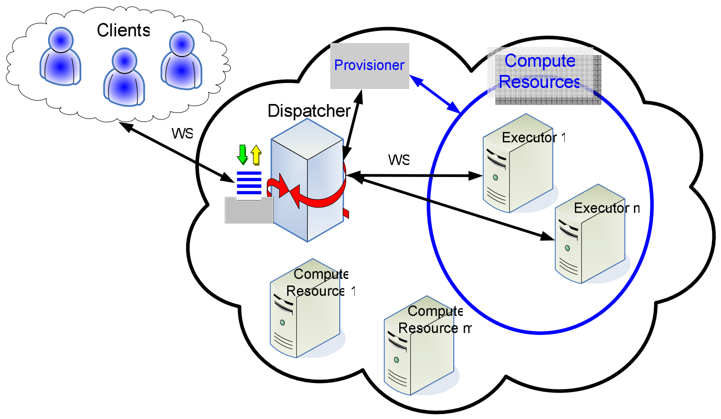

Figure 2: Falkon architecture overview

The **dispatcher** implements the factory/instance pattern, providing a *create instance* operation to allow a clean separation among different clients. To access the dispatcher, a client first requests creation of a new instance, for which is returned a unique endpoint reference (EPR). The client then uses that EPR to submit tasks, monitor progress, retrieve results, and (finally) destroy the instance. Each instance can be thought of as a separate instantiation of the dispatcher, maintaining its own task queue and related state. The dispatcher runs within a Globus Toolkit 4 (GT4) [28] WS container, which provides authentication, message integrity, and message encryption mechanisms, via transport-level, conversation-level, or message-level security [29].

The **provisioner** is responsible for creating and destroying executors. It is initialized by the dispatcher with information about the state to be monitored and how to access it; the rule(s) under which the provisioner should create/destroy executors; the location of the executor code; bounds on the number of executors to be created; bounds on the time for which executors should be created; and the allowed idle time before executors are destroyed. The provisioner periodically monitors dispatcher state and, based on the supplied rules, determines whether to create additional executors, and if so, how many, and for how long. Creation requests are issued via GRAM4 [27], to abstract away LRM details.

A new **executor** registers with the dispatcher. Work is then supplied as follows: (1) the dispatcher notifies the executor when work is available; (2) the executor requests work; (3) the dispatcher returns the task(s); (4) the executor executes the supplied task(s) and returns results, including return code and optional standard output/error strings; and (5) the dispatcher acknowledges delivery.

4 Performance Evaluation

4.1 Allocation Policies

We use two metrics to evaluate our DRP system: *Provisioning Latency* (i.e., the time required to obtain all required resources) and *Accumulated CPU Time* (i.e., the total CPU time obtained since the first request to the DRP system). We expect these metrics to help us identify the best dynamic resource provisioning strategies in real world systems (i.e. TeraGrid).

We perform experiments in two scenarios on the ANL/UC TeraGrid site, which has 96 IA32 processors and is managed by the PBS local resource manager. In the first case, the site is relatively idle with only 2 of the 96 resources utilized; these results are shown in solid lines in Figure 3. Thus, our requests (for up to 48 resources) can be served ”immediately.” Due to PBS overheads, it takes about 40 seconds for the first resource to be allocated in all cases, despite the queue being idle; we observed this overhead vary between 30 seconds to as high as 100 seconds in other experiments we performed. Figure 3 shows the number of worker resources that have registered back at the application and are ready to receive work as the experiment time progressed; this time includes several steps: time to allocate the resources with GRAM, time needed to coordinate between GRAM and PBS the resource allocation, time PBS needed to prepare the physical resource for use, time needed to start up the worker code, and the time needed for the worker code to register back at the main application.

We see that the one-at-a-time strategy is the slowest, due to the high number of batch scheduler submissions: it takes 105 seconds to allocate all 48 resources vs. 22 to 36 seconds for the other strategies. Note that the accumulated CPU time after 3 minutes of the experiment for one-at-a-time is almost 30 CPU minutes behind the other strategies.

In a more realistic setting, sites are rarely idle, and hence some resource requests will end up waiting in the local resource manager's queue. To explore this case, we consider a scenario in which the site has only 47 resources available until the 160 second mark, at which point availability increases to 48; these results are shown in dotted lines in Figure 3. Thus, each strategy has their last resource request held in the wait queue until the 160 second mark. Those last requests are for 1, 3, 17, and 48 resources for One-at-a-time, Additive, Exponential, and All-at-once, respectively. On one extreme, the 1-at-a-time strategy manages to allocate 47 resources and has only 1 resource in the waiting queue; the other extreme, the all-at-once strategy has all 48 resources in the waiting queue waiting for a single resource to free up before it can process the entire request. This is evidence of the back-filling strategies of the local resource manager. Therefore, the all-at-once is now the worst overall, being over 60 CPU minutes behind One-at-a-time and Exponential, and almost 90 CPU minutes behind Additive. Note that these lags in accumulated CPU time will remain until the resources begin to de-allocate, at which time the strategies that received their resources later will also hang on to the resources later; in the end, all strategies should get the same accumulated CPU time eventually.

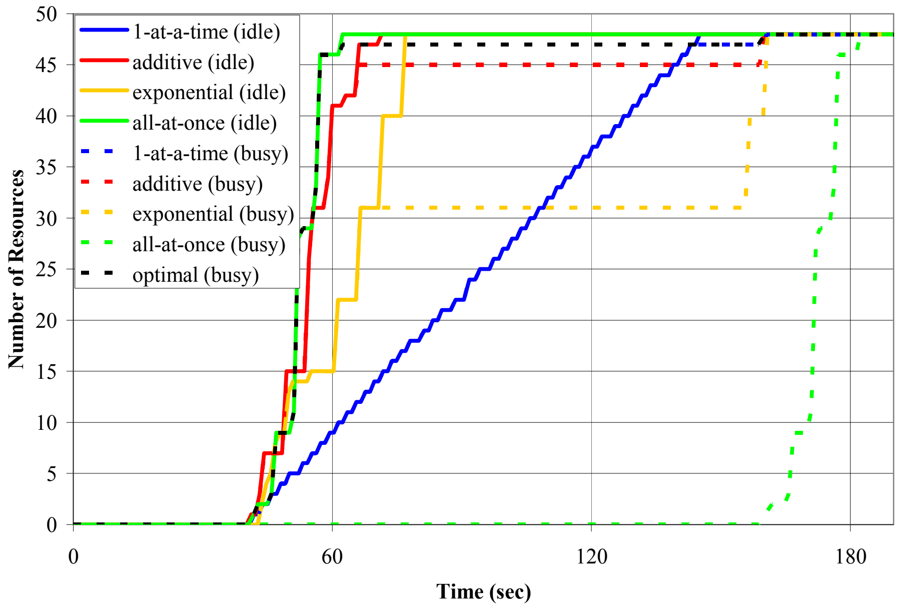

Figure 3: Provisioning latency in acquiring 48 resources for various strategies; the solid lines represent the time to acquire the resources in an idle system, while the dotted lines is the time to acquire the resources in a busy system

Table 1: Accumulated CPU time in seconds after 180 seconds in both an idle and busy system

Strategy	Accumulated CPU Time IDLE	Accumulated CPU Time BUSY
1-at-a-time	4220 sec	4205 sec
additive	6048 sec	5773 sec
exponential	5702 sec	4267 sec
all-at-once	6156 sec	409 sec
optimal	6059 sec	6059 sec

We conclude that different provisioning strategies must be used depending on how utilized a given set of resources are, with the all-at-once strategy being preferred if the resources are mostly idle, the additive and exponential strategies being appropriate for medium loaded resources, and the one-at-a-time being preferred when the resources are heavily loaded. Note that the finer grained the request sizes, the more likely it will be that DRP will be able to benefit from the back-filling of the local resource managers, but the higher the cost will be in terms of how fast the resources can be allocated. Notice

Another important issue, not addressed in this work, concerns the length of time for which resources should be requested. Many batch schedulers give preference to short requests and/or can schedule short requests into empty slots in their schedule (what is termed "backfilling"). Short requests may also minimize idle time. On the other hand, short requests increase more scheduling overhead and may cause problems for long-running user tasks. We envision the length of time to allocate resources to be application dependent, depending on the tasks complexity and granularity. Ideally, the length of time resources are allocated for should be large enough to ensure that several tasks can be performed on each resource, effectively amortizing the cost of the queue wait times for the coarse granular resource allocation.

4.2 De-Allocation Policies

To study provisioner performance, we constructed a synthetic 18-stage workload, in which the numbers of tasks and task lengths vary between stages. Figure 4 shows the number of tasks per stage and the number of machines needed per stage if each task is mapped to a separate machine (up to a maximum of 32 machines). Note the exponential ramp up in the number of tasks for the first few stages, a sudden drop at stage 8, and a sudden surge of many tasks in stages 9 and 10, another drop in stage 11, a modest increase in stage 12, followed by a linear decrease for several stages, and finally an exponential decrease until the last stage has only a single task. All tasks run for 60 secs except those in stages 8, 9, and 10, which run for 120, 6, and 12 secs, respectively. In total, the 18 stages have 1,000 tasks, summing to 17,820 CPU secs, and can complete in an ideal time of 1,260 secs on 32 machines.

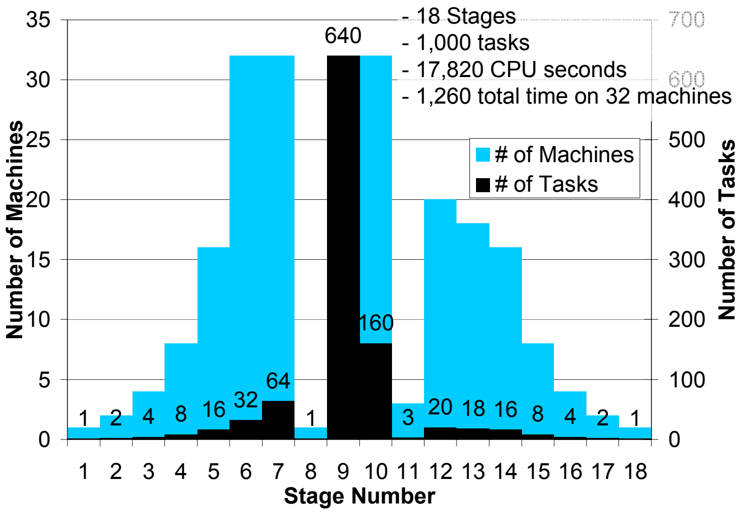

Figure 4: The 18-stage synthetic workload.

We configured the provisioner to acquire at most 32 machines from TG_ANL_IA32 and TG_ANL_IA64, both of which were relatively lightly loaded. (100 machines were available of the total 162 machines.) We measured the execution time in six configurations:

- *GRAM+PBS:* Each task was submitted as a separate GRAM task over PBS, without imposing any hard limits on the number of machines to use; there were about 100 machines available for this experiment.
- *Falkon-15, Falkon-60, Falkon-120, Falkon-180:* Falkon configured to use a minimum of zero and a maximum of 32 machines; the allocation policy we used was *all-at-once*, and the resource release policy idle time was set to 15, 60, 120, and 180 secs (to give four separate experiments).
- *Falkon-∞:* Falkon, with the provisioner configured to retain a full 32 machines for one hour.

Table 2 gives, for each experiment, the average per-task queue time and execution time, and also the ratio exec_time/(exec_time+queue_time). The queue_time includes time waiting for the provisioner to acquire nodes, time spent starting executors, and time tasks spend in the dispatcher queue. We see that the ratio improves from 17% to 28.7% as the idle time setting increases from 15 secs to 180 secs; for Falkon-∞, it reaches 29.2%, a value close to the ideal of 29.7%. (The ideal is less than 100% because several stages have more than 32 tasks, which means tasks must be queued when running, as we do here, on 32 machines.) GRAM+PBS yields the worst performance, with only 8.5% on average, less than a third of ideal.

Table 2: Average per-task queue and execution times for synthetic workload

	GRAM +PBS	Falkon-15	Falkon-60	Falkon-120	Falkon-180	Falkon-∞	Ideal (32 nodes)
Queue Time (sec)	611.1	87.3	83.9	74.7	44.4	43.5	42.2
Execution Time (sec)	56.5	17.9	17.9	17.9	17.9	17.9	17.8
Execution Time %	8.5%	17.0%	17.6%	19.3%	28.7%	29.2%	29.7%

The average per-task queue times range from a near optimal 43.5 secs (42.2 secs is ideal) to as high as 87.3 secs, more than double the ideal queue time. In contrast, GRAM+PBS experience a queue time that is 15 times larger than the ideal at 611.1 secs. Also, note the execution time for Falkon with the resource provisioning (both static and dynamic) is the same across all the experiments, and is within 100 ms of ideal (which essentially accounts for the dispatch cost and delivering the result); in contrast, GRAM+PBS have an average execution time of 56.5 secs, significantly larger than the ideal time. This large difference in execution time is attributed to the large per task overhead GRAM and PBS have, which further strengthens our argument that they are not suitable for short tasks.

Table 3 shows the total time to complete the 18 stages, the resource utilization, the execution efficiency, and the number of resource allocations. We define resource utilization as the ratio of resources used to resources used + resources wasted (i.e., resources consumed but not used for task execution), and execution efficiency as the ratio of ideal time to actual time.

The resources used are the same (17,820 CPU secs) for all cases, as we have fixed run times for all 1000 tasks.

As for resources wasted, we expected GRAM+PBS to not have any as each machine is released after one task is run; in reality, the measured execution times were longer than the actual task execution times, and hence the resources wasted was high in this case: 41,040 secs over the entire experiment. (We define task execution time in the GRAM+PBS case to be from the time GRAM sends a notification of the task changing its state to being "Active"—meaning that PBS has taken the task off the wait queue and placed into the active queue assigned to some physical machine—to the time the state changes to "Done," at which point the task has finished its execution.) The average execution time of 56.5 secs shows that GRAM+PBS is slower than Falkon in dispatching the task to the remote machine, preparing the remote machine to execute the task, and cleaning up and releasing the machine. Note that the reception of the "Done" state change in GRAM4 does not imply that the utilized machine is ready to receive another task—PBS takes even longer to make the machine available again for more work, which makes GRAM+PBS resource wastage yet worse.

Falkon with dynamic resource provisioning fairs better from the perspective of resource wastage. Falkon-15 has the least amount of wasted resources with 2032 CPU secs, and Falkon-∞ (which never de-allocates nodes during the experiment) has the worst with 22,940 CPU secs for the duration of the experiment.

The resource utilization shows the fraction of time the machines were executing tasks vs. idle. Due to its high resource wastage, GRAM+PBS achieves a utilization of only 30%, while Falkon-15 reaches 89%. Falkon-∞ is 44%. Notice that as the resource utilization increases, so does the time to complete—as we assume that the provisioner has no foresight regarding future needs, delays are incurred allocating machines previously de-allocated due to a shorter idle time setting. Note the number of resource allocations (GRAM4 calls requesting resources) for each experiment, ranging from 1000 allocations for GRAM+PBS to less than 11 for Falkon with provisioning. For Falkon-∞, the number of resource allocations is zero since machines were provisioned prior to the experiment starting, and that time is not included in the time to complete the workload.

If we had used a different allocation policy (e.g., one-at-a-time), the Falkon results would have been less close to ideal, as the number of resource allocations would have grown significantly. The relatively slow handling of such requests by GRAM+PBS (~1/sec on TG_ANL_IA32 and TG_ANL_IA64) would have delayed executor startup and thus increased the time tasks spend in the queue waiting to be dispatched.

The higher the desired resource utilization (due to more aggressive dynamic resource provisioning to avoid resource wastage), the longer the elapsed execution time (due to queuing delays and overheads of the resource provisioning in the underlying LRM). This ability to trade off resource utilization and execution efficiency is an advantage of Falkon.

Table 3: Summary of overall resource utilization and execution efficiency for the synthetic workload

	GRAM +PBS	Falkon-15	Falkon-60	Falkon-120	Falkon-180	Falkon-∞	Ideal (32 nodes)
Time to complete (sec)	**4904**	**1754**	**1680**	**1507**	**1484**	**1276**	**1260**
Resouce Utilization	30%	89%	75%	65%	59%	44%	100%
Execution Efficiency	26%	72%	75%	84%	85%	99%	100%
Resource Allocations	1000	11	9	7	6	0	0

To communicate how provisioning works in practice, we show in Figures 10 and 11 details of experiment execution for Falkon-15 and Falkon-180, respectively.

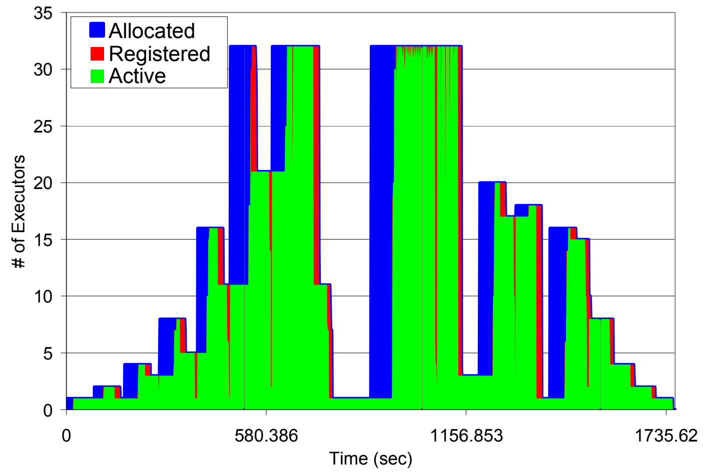

Figure 5: Synthetic workload for Falkon-15

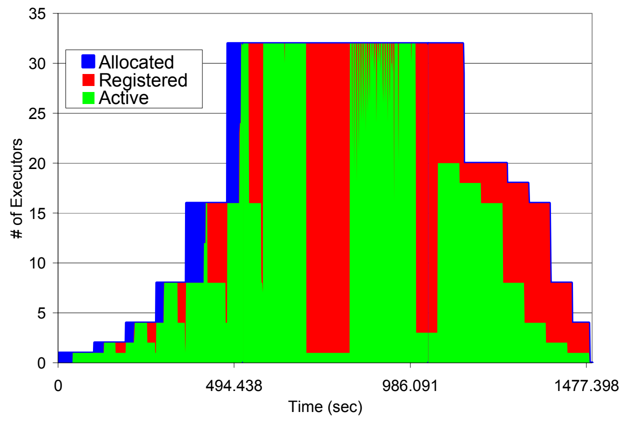

Figure 6: Synthetic workload for Falkon-180

These figures show the instantaneous number of allocated, registered, and active executors over time. Allocated executors (blue) are those for which creation and registration are in progress. Creation and registration time can vary between 5 and 65 secs, depending on when a creation request is submitted relative to the PBS scheduler polling loop (which we believe occurs at 60 sec intervals). (JVM startup time and registration generally consume less than five secs.) Registered executors (red) are ready to process tasks, but are not active. Finally, active executors (green) are actively processing tasks. In summary, blue is startup cost, red is wasted resources, and green is useful work.

4.3 Multi-Site Provisioning

Investigate the various policies: 1) random provisioning, 2) round-robin provisioning, 3) over-allocation provisioning, and 4) probabilistic provisioning.

Comment [IR2]: To do!

4.4 Application Experiments

Comment [IR3]: These results only show static resource provisioning… they are good, but we also have to show their performance when we use dynamic resource provisioning!

As we have stated earlier, we have integrated the dynamic resource provisioning into Falkon. Falkon has already been integrated into the Karajan [2, 3] workflow engine, which in term is used by the Swift parallel programming system. Thus, Karajan and Swift applications can use Falkon along with the dynamic resource provisioning without modification. Using the dynamic resource porivsioning and the light-weight task dispatch mechanisms from Falkon, we demonstrated reductions in end-to-end run time by as much as 90% when compared to traditional approaches in which the applications used the batch schedulers directly.

Swift has been applied to a variety of science applications in disciplines such as physical sciences, biological sciences, social sciences, humanities, computer science, and science education. Table 4 characterizes some applications in terms of the typical number of tasks and the number of stages.

Table 4: A list of potential applications that could benefit from the use of Falkon

Application	#Jobs/workflow	#Levels
ATLAS: High Energy Physics Event Simulation	500K	1
fMRI DBIC: AIRSN Image Processing	100s	12
FOAM: Ocean/Atmosphere Model	2000	3
GADU: Genomics	40K	4
HNL: fMRI Aphasia Study	500	4
NVO/NASA: Photorealistic Montage/Morphology	1000s	16
QuarkNet/I2U2: Physics Science Education	10s	3 ~ 6
RadCAD: Radiology Classifier Training	1000s	5
SIDGrid: EEG Wavelet Processing, Gaze Analysis	100s	20
SDSS: Coadd, Cluster Search	40K, 500K	2, 8

We illustrate the distinctive dynamic features in Swift using an fMRI [21] analysis workflow from cognitive neuroscience, and a photorealistic montage application from the national virtual observatory project [32, 22].

4.4.1 Functional Magnetic Resonance Imaging

Comment [IR4]: Also discuss wait queue time, and possibly resource utilization…. Similar to table 2 and 3, and figure 10 and 11 as we did for the synthetic workload.

This medical application is a four-step pipeline [21]. An fMRI *Run* is a series of brain scans called volumes, with a *Volume* containing a 3D image of a volumetric slice of a brain image, which is represented by an *Image* and a *Header*. We ran this application for four different problem sizes, ranging from 120 volumes (480 tasks for the four stages) to 480 volumes (1960 tasks). Each task can run in a few secs on a TG_ANL_IA64 processor.

We compared three implementation approaches: task submission via GRAM+PBS, a variant of that approach in which tasks are clustered into eight groups, and Falkon with a fixed set of eight executors. In each case, we ran the client on UC_IA32 and application tasks on TG_ANL_IA64.

In Figure 7 we show execution times for the different approaches and for different problem sizes. Although GRAM+PBS could potentially have used up to 62 nodes, it performs badly due to the small tasks. Clustering reduced execution time by more than four times on eight processors. Falkon further reduced the execution time, particularly for smaller problems.

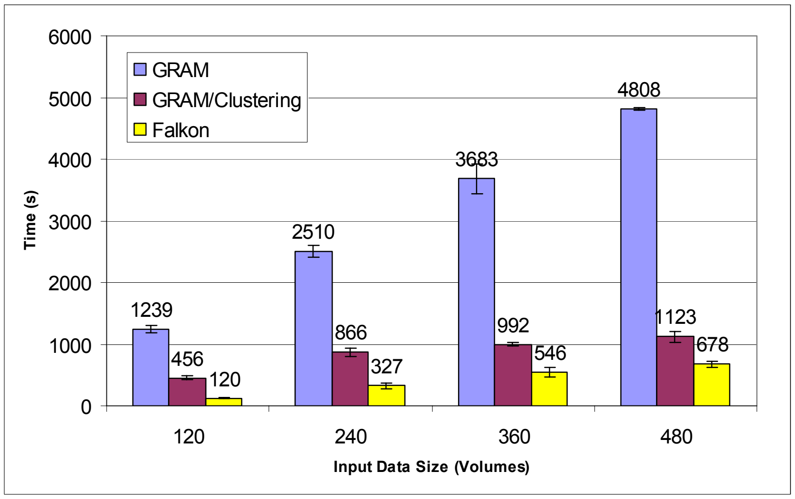

Figure 7: Execution Time for the fMRI Workflow

4.4.2 Montage Image Mosaicing

Comment [IR5]: Also discuss wait queue time, and possibly resource utilization…. Similar to table 2 and 3, and figure 10 and 11.

Montage generates large astronomical image mosaics by composing multiple small images [32, 22]. A four-stage pipeline reprojects each image into a common coordinate space; performs background rectification (calculates a list of overlapping images; computes image difference between each pair of overlapping images; and fits difference images into a plane); performs background correction; and co-adds the processed images into a final mosaic. (To enhance concurrency, we decompose the co-add into two steps.)

We considered a modest-scale computation that produces a 3°x3° mosaic around galaxy M16. There are about 440 input images and 2,200 overlapping image sections between them. The resulting task graph has many small tasks.

Figure 8 shows execution times for three versions of Montage:

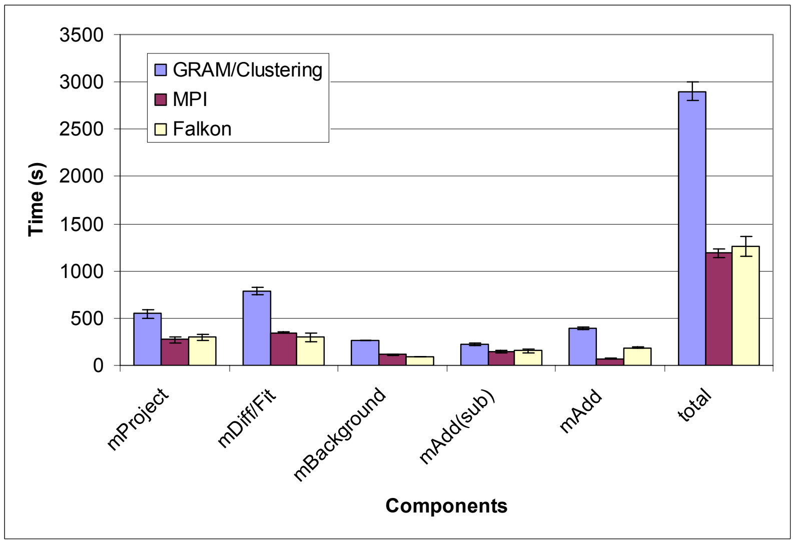

Figure 8: Execution time for Montage application

Swift with clustering, submitting via GRAM+PBS; Swift submitting via Falkon; and an MPI version constructed by the Montage team. The second co-add stage was only parallelized in the MPI version; thus, Falkon performs poorly in this step. Both the GRAM and Falkon versions staged in data, while the MPI run assumed data was pre-staged. Despite these differences, Falkon achieved performance similar to that of the MPI version.

Deelman et al. have also created a task-graph implementation of the Montage code, using Pegasus [33]. They do not implement quite the same application as us: for example, they run two tasks (mOverlap and mImgtlb) on the portal rather than on compute nodes, they combine what for us are two distinct tasks (mDiff and mFit) into a single task, mDiffFit, and they omit the final mAdd phase. Thus, direct comparison is difficult. However, if the final mAdd phase is omitted from the comparison, Swift+Falkon is faster by about 5% (1067 secs vs. 1120 secs) when compared to MPI, while Pegasus is reported as being somewhat slower than MPI. We attribute these differences to two factors: first, the MPI version performs initialization and aggregation actions before each step; second, Pegasus uses Condor glide-ins, which are heavy-weight relative to Falkon.

5 Conclusions

The schedulers used to manage parallel computing clusters are not typically configured to enable easy configuration of application-specific scheduling policies. In addition, their sophisticated scheduling algorithms and feature-rich code base can result in significant overhead when executing many short tasks.

Falkon, a Fast and Light-weight tasK executiON framework, is designed to enable the efficient dispatch and execution of many small tasks. To this end, it uses a multi-level scheduling strategy to enable separate treatment of resource allocation (via conventional schedulers) and task dispatch (via a streamlined, minimal-functionality dispatcher). Clients submit task requests to a dispatcher, which in turn passes tasks to executors. A separate provisioner is responsible for creating and destroying provisioners in response to changing client demand; thus, users can trade off application execution time and resource utilization.

Dynamic resource provisioning can lead to significant savings in end-to-end application execution time, enable the use of batch-scheduled Grids for interactive use, and alleviate the high queue wait times typically found in production Grid environments. We have described a dynamic resource provisioning architecture and presented performance results we obtained on the TeraGrid. We have also integrated the dynamic resource provisioning into Falkon, a Fast and Light-weight tasK executiON framework, which allowed us to measure various performance aspects of resource provisioning with both real applications and synthetic workloads.

Microbenchmarks show that DRP can allocate resources on the order of 10s of seconds across multiple Grid sites and can reduce average queue wait times by up to 95% (effectively yielding queue wait times within 3% of ideal). Furthermore, applications (executed by the Swift parallel programming system) reduce end-to-end run time of up to 90% for large-scale astronomy and medical applications, relative to versions that execute tasks via separate scheduler submissions.

6 References

[1] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience" Concurrency and Computation: Practice and Experience, Vol. 17, No. 2-4, pages 323-356, February-April, 2005.

[2] Swift Workflow System: www.ci.uchicago.edu/swift

[3] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, M. Wilde. “Swift: Fast, Reliable, Loosely Coupled Parallel Computation”, under review at IEEE Workshop on Scientific Workflows 2007.

[4] I. Foster, J. Voeckler, M. Wilde, Y. Zhao. “Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation”, SSDBM 2002.

[5] J.-P Goux, S. Kulkarni, J.T. Linderoth, and M.E. Yoder, "An Enabling Framework for Master-Worker Applications on the Computational Grid," Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing, 2000.

[6] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid", Int. Supercomputing Applications, 2001.

[7] G. Banga, P. Druschel, J.C. Mogul. “Resource Containers: A New Facility for Resource Management in Server Systems.” Symposium on Operating Systems Design and Implementation, 1999.

[8] J.A. Stankovic, K. Ramamritham,, D. Niehaus, M. Humphrey, G. Wallace, “The Spring System: Integrated Support for Complex Real-Time Systems”, Real-Time Systems, May 1999, Vol 16, No. 2/3, pp. 97-125.

[9] J. Frey, T. Tannenbaum, I. Foster, M. Frey, S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids," Cluster Computing, vol. 5, pp. 237-246, 2002.

[10] G. Singh, C. Kesselman, E. Deelman, “Optimizing Grid-Based Workflow Execution.” Journal of Grid Computing, Volume 3(3-4), December 2005, Pages 201-219.

[11] E. Walker, J.P. Gardner, V. Litvin, E.L. Turner, “Creating Personal Adaptive Clusters for Managing Scientific Tasks in a Distributed Computing Environment”, Workshop on Challenges of Large Applications in Distributed Environments, 2006.

[12] G. Singh, C. Kesselman E. Deelman. “Performance Impact of Resource Provisioning on Workflows”, ISI Tech Report 2006.

[13] G. Mehta, C. Kesselman, E. Deelman. “Dynamic Deployment of VO-specific Schedulers on Managed Resources,” USC ISI, 2006.

[14] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the Grid", in Fran Berman, Anthony J.G. Hey, Geoffrey Fox, editors, Grid Computing: Making The Global Infrastructure a Reality, John Wiley, 2003. ISBN: 0-470-85319-0

[15] E. Robinson, D.J. DeWitt. “Turning Cluster Management into Data Management: A System Overview”, Conference on Innovative Data Systems Research, 2007.

[16] B. Bode, D.M. Halstead, R. Kendall, Z. Lei, W. Hall, D. Jackson. “The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters”, Usenix, Proceedings of the 4th Annual Linux Showcase & Conference, 2000.

[17] S. Zhou. “LSF: Load sharing in large-scale heterogeneous distributed systems,” Workshop on Cluster Computing, 1992.

[18] W. Gentzsch, "Sun Grid Engine: Towards Creating a Compute Power Grid," 1st International Symposium on Cluster Computing and the Grid, 2001.

[19] D.P. Anderson. “BOINC: A System for Public-Resource Computing and Storage.” 5th IEEE/ACM International Workshop on Grid Computing. November 8, 2004.

[20] D.P. Anderson, E. Korpela, R. Walton. “High-Performance Task Distribution for Volunteer Computing.” IEEE Int. Conference on e-Science and Grid Technologies, 2005.

[21] The Functional Magnetic Resonance Imaging Data Center, http://www.fmridc.org/, 2007.

[22] G.B. Berriman, et al. ”Montage: a Grid Enabled Engine for Delivering Custom Science-Grade Image Mosaics on Demand.” Proceedings of the SPIE Conference on Astronomical Telescopes and Instrumentation. 2004.

[23] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger, "Oceano - SLA Based Management of a Computing Utility," in 7th IFIP/IEEE International Symposium on Integrated Network Management, 2001.

[24] L. Ramakrishnan, L. Grit, A. Iamnitchi, D. Irwin, A. Yumerefendi, J. Chase. “Toward a Doctrine of Containment: Grid Hosting with Adaptive Resource Control,” IEEE/ACM SuperComputing 2006.

[25] J. Bresnahan, I. Foster. “An Architecture for Dynamic Allocation of Compute Cluster Bandwidth”, MS Thesis, Department of Computer Science, University of Chicago, December 2006.

[26] TeraGrid, http://www.teragrid.org/

[27] M. Feller, I. Foster, and S. Martin. “GT4 GRAM: A Functionality and Performance Study”, TeraGrid 07.

[28] I. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems," Conference on Network and Parallel Computing, 2005.

[29] The Globus Security Team. “Globus Toolkit Version 4 Grid Security Infrastructure: A Standards Perspective,” Technical Report, Argonne National Laboratory, MCS, September 2005.

[30] I. Raicu, I. Foster, A. Szalay. “Harnessing Grid Resources to Enable the Dynamic Analysis of Large Astronomy Datasets”, IEEE/ACM SC 06.

[31] I. Raicu, I. Foster, A. Szalay, G. Turcu. “AstroPortal: A Science Gateway for Large-scale Astronomy Data Analysis”, TeraGrid Conference 2006.

[32] J.C. Jacob, et al. ”The Montage Architecture for Grid-Enabled Science Processing of Large, Distributed Datasets.” Proceedings of the Earth Science Technology Conference 2004

[33] E. Deelman, et al. "Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems", Scientific Programming Journal, Vol 13(3), 2005, Pages 219-237.