Falkon: a Fast and Light-weight tasK executiON framework

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

To enable the rapid execution of many tasks on compute clusters, we have developed Falkon, a Fast and Light-weight tasK executiON framework. Falkon uses (1) multi-level scheduling to separate resource acquisition (via, e.g., requests to batch schedulers) from task dispatch, and (2) a streamlined dispatcher. Multi-level scheduling, introduced in operating systems research in the 1990s, has been applied to clusters by the Condor team and others, while streamlined dispatchers are found in, e.g, BOINC. Falkon's integration of the techniques delivers performance not provided by any other system. We describe Falkon architecture and implementation, and present performance results for both microbenchmarks and applications. Microbenchmarks show that Falkon throughput (440 tasks/sec) and scalability (to 54,000 executors and 2,000,000 tasks processed in just over two hours) are one to two orders of magnitude better than other schedulers. 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: parallel programming, dynamic resource provisioning, scheduling, 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:

- Reducing task dispatch time by using a streamlined dispatcher that eliminates support for features such as multiple queues, priorities, accounting, etc.
- Using an adaptive provisioner to acquire and/or release resources as application demand varies.

To explore these ideas, we have developed Falkon, a Fast and Light-weight tasK executiON framework. Falkon incorporates a lightweight task dispatcher, to receive, enqueue, and dispatch tasks; a simple task executor, to receive and execute tasks; and a provisioner, to allocate and deallocate executors.

Microbenchmarks show that Falkon can process millions of task requests and scale to over 50,000 executor nodes. A synthetic application demonstrates the benefits of adaptive provisioning. Finally, results for two applications involving many small tasks demonstrate that substantial speedups can be achieved for real scientific applications.

2 Related Work

Full-featured local resource managers (LRMs) such as Condor [1], Condor-J2 [15], PBS [16], LSF [17] support client specification of resource requirements, data staging, process migration, dynamic load balancing, check-pointing, accounting, and daemon fault recovery. Falkon, in contrast, is not a full-featured LRM: it focuses solely on efficient task dispatch and thus can omit these features to streamline task submission. This narrow focus is possible because Falkon can rely on LRMs for certain functions (e.g., accounting) and clients for others (e.g., recovery).

The BOINC "volunteer computing" system [19, 20] has a similar architecture to that of Falkon. BOINC's database-driven task dispatcher is estimated to be capable of dispatching 8.8M tasks per day to 400K workers. This estimate is based on extrapolating from smaller synthetic benchmarks of CPU and I/O overhead, on the task distributor only, for the execution of 100K tasks. By comparison, Falkon has been measured to execute 2M artificial tasks in two hours using the actual Falkon code, and has scaled to 54K managed executors with similarly high throughput. This test as well as other throughput tests achieving 440 tasks/sec suggest that Falkon can provide higher throughput than BOINC.

Multi-level scheduling has been applied at the OS level [27, 30] to provide faster scheduling for groups of tasks for a specific user or purpose by employing an overlay that does lightweight scheduling within a heavier-weight container of resources: e.g., threads within a process or pre-allocated thread group.

Frey and his colleagues pioneered the application of this principle 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.

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, Falkon's innovation is its combination of a fast 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 higher task throughput than previous systems, while also offering applications the ability to trade-off system responsiveness, resource utilization, and execution efficiency.

3 Architecture and Implementation

3.1 Execution Model

Each task is dispatched to a computational resource, selected according to the *dispatch policy*. If a response is not received after a time determined by the *replay policy*, or a failed response is received, the task is re-dispatched according to the dispatch policy. 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.

Dispatch policy. We consider here only the *next-available* policy, which dispatches each task to the next available resource. We assumed that all data needed by a task is available in a shared file system. In the future, we will examine other dispatch policies that take into account data availability.

Resource Acquisition Policy. This policy determines the number of resources, *n*, to acquire; the length of time for which resources should be requested; and the request(s) to generate to LRM(s) to acquire those resources. We have implemented five strategies that variously generate a single request for *n* resources; generate *n* requests for a single resource; generate a series of arithmetically or exponentially larger requests; or use system functions to determine available resources. In the experiments reported in this

paper, we considered only first ("all-at-once strategy") 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

Falkon consists of a dispatcher, a provisioner, and zero or more executors (Figure 1). 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.

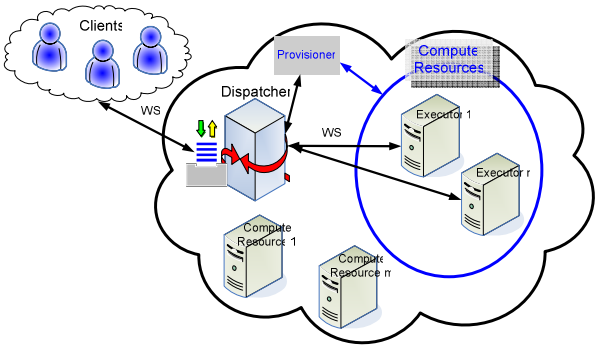

Figure 1: Falkon architecture overview

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.

A client "submit" request takes an array of tasks, each with working directory, command to execute, arguments, and environment variables. It returns an array of outputs, each with the task that was run, its return code , and optional output strings (STDOUT and STDERR contents). Each instance contains its own work queue. A shared notification engine is used to notify executors that work is available for pick up. This engine maintains a queue on which a pool of threads operate to send out notifications. The GT4 container also has a pool of threads that handle WS messages. Profiling shows that most dispatcher time is spent communicating (WS calls, notifications). Increasing the number of threads should allow the service to scale on newer multicore and multiprocessor systems.

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.

3.3 Performance Enhancements

Communication costs can be reduced by *task bundling* between client and dispatcher and/or dispatcher and executors. In the latter case, problems can arise if task sizes vary and one executor gets assigned many large tasks, although that problem can be addressed by having clients assign each task an estimated runtime. We use client-dispatcher bundling in experiments described below, but (lacking runtime estimates) not dispatcher-executor bundling. Another technique that can reduce message exchanges is to *piggy-back* new

task dispatches when acknowledging result delivery (step 5 above).

Using both task bundling and piggy-backing, we can reduce the average number of message exchanges per task to be arbitrarily close to zero, by increasing the bundle size. In practice, we find that performance degrades for bundle sizes of greater than 300 tasks (see Section 4.2)—and, as noted above, bundling cannot always be used between dispatcher and executors.

With client-dispatcher bundling and piggy-backing alone, we can reduce the number of messages to three per task (one message from executor to dispatcher to deliver a result, one associated response from dispatcher to executor to acknowledge receipt and provide a new task, and one notification of task completion from dispatcher to client)—or five, if the clients needs to fetch task output after notification.

3.4 Ease of Use

We modified the Swift parallel programming system to use Falkon for task dispatch. The "Falkon provider" consisted of 840 lines of Java code, and took less than a day to implement. Code size is comparable to that of the GRAM2 provider (850 lines), GRAM4 provider (517 lines), and Condor provider (575 lines).

4 Performance Evaluation

Table 1 lists the platforms used in experiments. The latency between these machines was one to two millisecs. We assume a one-to-one mapping between executors and processors in all experiments.

Table 1: Platform descriptions

Name	# of Nodes	Processors	Memory	Network
TG_ANL_IA32	98	Dual Xeon 2.4GHz	4GB	1Gb/s
TG_ANL_IA64	64	Dual Itanium 1.5GHz	4GB	1Gb/s
TP_UC_x64	122	Dual Opteron 2.2GHz	4GB	1Gb/s
UC_x64	1	Dual Xeon 3GHz w/ HT	2GB	100 Mb/s
UC_IA32	1	Intel P4 2.4GHz	1GB	100 Mb/s

4.1 Throughput

To determine maximum throughput, we measured performance running "sleep 0." We ran executors on TG_ANL_IA32 and TG_ANL_IA32, the dispatcher on UC_x64, and the client generating the workload on TP_UC_x64. As each machine had two processors, we ran two executors on each machine. We measured Falkon's throughput for dispatching and executing short tasks both without any security, and with authentication and encryption. We enable two optimizations discussed below, namely client-dispatcher bundling and piggy-backing. However, every task is transmitted individually from dispatcher to an executor, as is each notification of success or failure from dispatcher to client.

For purposes of comparison, we also tested GT4's performance with all security disabled. We created a simple service that incremented a counter for each WS call made to a counter service, and measured the number of WS calls per second that could be achieved from a varying number of machines.

Figure 2 shows results. The GT4 performance (500 messages/sec) is an upper bound on what Falkon can achieve on the given hardware, given that (in the absence of optimizations discussed below) every Falkon task requires at least one WS message and one notification. Performance reaches between 180 and 440 tasks/sec depending on security. These results compare favorably to other existing systems such as Condor (~2/sec) [15], Condor-J2 (~22/sec) [15], and Boinc (~93/sec) [19, 20]. A single Falkon executor can handle between seven and 28 tasks/sec depending on security level.

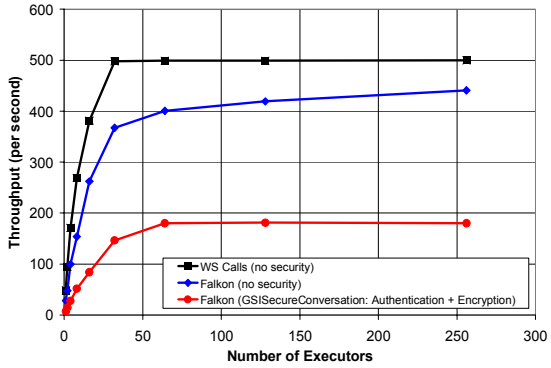

Figure 2: Throughput as function of executor count

4.2 Bundling

We measured performance for a workload of "sleep 0" tasks as a function of task bundle size. We see that performance increases from about 20 tasks/sec without bundling to a peak of almost 1500 tasks/sec. Performance decreases after around 300 tasks per bundle. We attribute this drop to the array data structure implementation in the Axis software that GT4 uses to handle XML serialization and de-serialization. (Axis implements the array data-structure used to store the representation of the bundled tasks as a grow-able array, copying to a new bigger array each time its size increases.) We will investigate this inefficiency further to see if this limitation can be remedied.

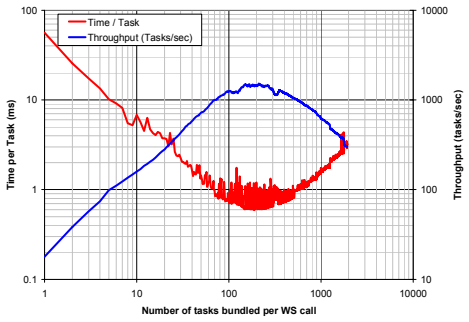

Figure 3: Bundling throughput and cost per task

4.3 *Efficiency & Speedup*

Figures 4 and 5 show speedup ($S_P=T_1/T_P$, where T_n is the execution time on n processors) and efficiency ($E_P=S_P/P$) as a function of number of executors and task length.

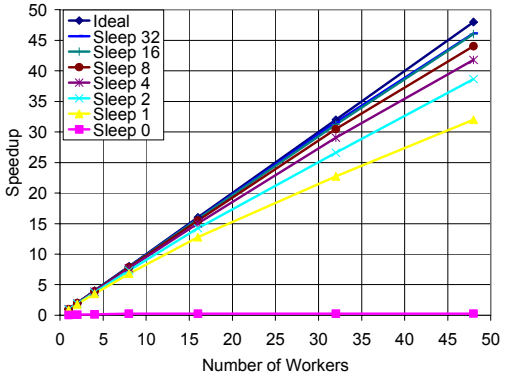

Figure 4: Speedup for varying task length and number of executors

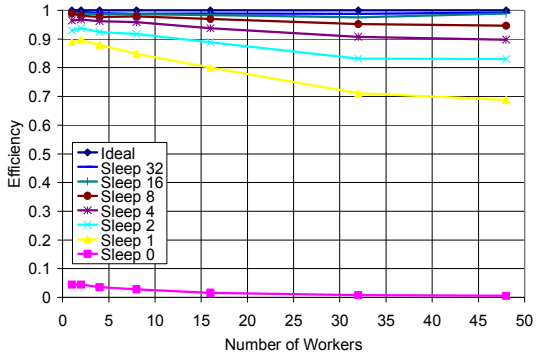

Figure 5: Efficiency as a function of task length and number of executors

We see that even with short tasks, we achieve high efficiencies and speedup. These experiments were conducted on TG_ANL_IA64 with no security and no optimizations, such as bundling or "piggy-backing." We plan to redo the experiments with these optimizations enabled; we expect the lines to be significantly closer to each other, and altogether closer to the ideal speedup and efficiency.

4.4 *Scalability*

To test scalability and robustness, we performed experiments that pushed Falkon to its limits.

Our first experiment studies Falkon's behavior as the task queue increases in length. We constructed a client that submits two million "sleep 0" tasks to a dispatcher configured with a Java heap size set to 1.5GB. We created 32 executors on 16 machines from TG_ANL_IA32 and ran the dispatcher on UC_x64 and the client on TP_UC_x64.

Figure 6 results show the entire run as it progressed in time. The solid black line is the instantaneous queue length, the light blue dots are raw samples (once per sec) of achieved throughput in terms of task completion, and the darker blue line is the moving average (over 120 sample intervals) of raw throughput. Average throughput was 268 tasks/sec, but was clustered at either around 300 tasks/sec or around 225 tasks/sec. Note the slight increase when the queue stopped growing, as the client finished submitting all two million tasks.

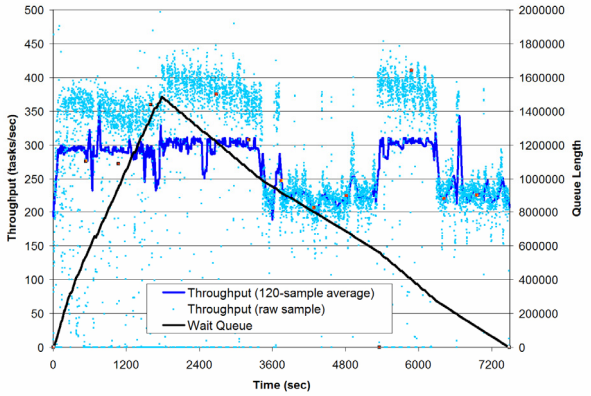

Figure 6: Long running test with 2M tasks

We attribute the considerable second-by-second performance variations to JVM garbage collection. By configuring the JVM to garbage collect more frequently, we can potentially reduce this variation.

In a second experiment, we tested how many executors the dispatcher could handle. As no system available to us could provide us with more than a few hundred physical machines, we ran multiple executors on each physical machine, in essence emulating a larger number of machines. Others [15] have used this experimental method with success.

We performed our experiment on TP_UC_x64, on which we configured one dispatcher machine, one

client machine, and 60 machines to run executors. We ran 900 executors (split over four JVMs) on each machine, for a total of 900x60=54,000 executors. Once we started up the system and all 54K executors registered and were ready to receive work, we started the experiment consisting of 54K tasks of "sleep 480" (8 minutes). For this experiment, we disabled all security, and only enabled bundling between the client and the dispatcher. Note that piggy-backing would have made no difference in this case as each executor only processed one task each.

Figure 7 shows that the dispatch rate (green line) is on par with the submit rate. The black line shows the number of busy executors, going from 0 to 54K in 408 secs. As soon as the first task finishes after 480 secs (that is the task length), results start to be delivered at the client at about the same rate as they were submitted and dispatched. Overall throughput (including ramp up and ramp down time) was about 60 tasks/sec.

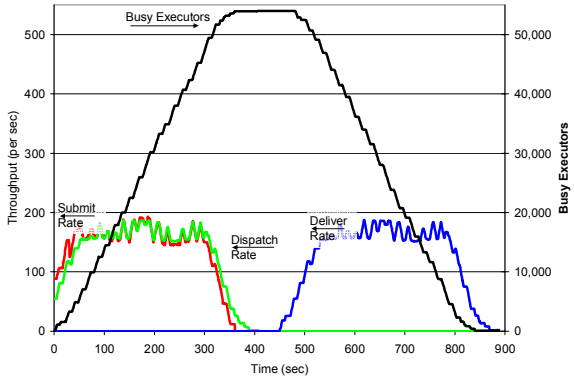

Figure 7: Falkon scalability with 54K executors

Figure 8 shows the per task overhead in millisec for the 54K task experiment from Figure 7, ordered by their start time of each task.

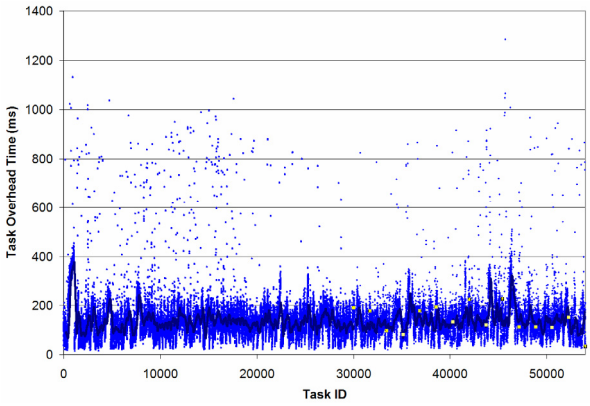

Figure 8: Task overhead with 54K executors

We see that the majority of the overhead was below 200 ms, with just a few higher than that, with a maximum of 1300 ms. The overhead is defined as the time it takes an executor to create a thread to handle the task, pick up a task via one WS call, perform an Java exec on the specified command (sleep 480), and send the result (the exit return code) back via one WS call, minus 480 secs (the task run time). Note that we have 900 executors on each physical machine, so this overhead is higher than normal as each thread was only getting a fraction of the processing power of the machine.

4.5 Dynamic Resource Provisioning

To study provisioner performance, we constructed a synthetic 18-stage workload, in which the numbers of tasks and task lengths vary between stages. Figure 9 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.

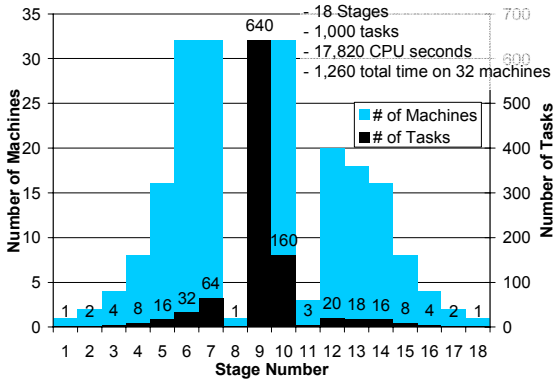

Figure 9: The 18-stage synthetic workload.

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.

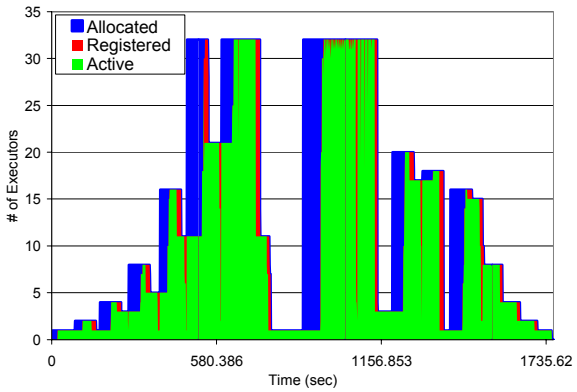

Figure 10: Synthetic workload for Falkon-15

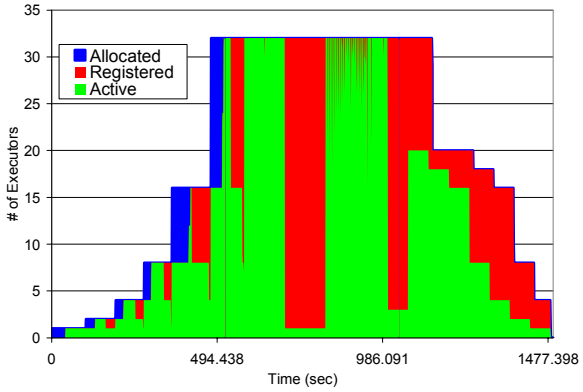

Figure 11: Synthetic workload for Falkon-180

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.

4.6 Security Overhead

Security requirements vary according to application and environment. In Figure 2, we showed that peak throughput declined from 440 tasks/sec without security to 180 tasks/sec when performing both authentication and encryption. We describe here an experiment in which a client submits 30 tasks, each 60 secs in length, which are then dispatched in sequence to a single executor. We measure time from first task submission to last task completion. The experiment is similar to one reported for MyCluster [11]. We ran all processes on TG_ANL_IA32 processors, and disabled all optimizations such as bundling or piggy-backing.

This kind of a workload is generally going to produce the highest overhead as the entire experiment is serialized, and there is no opportunity for the dispatcher to overlap computation and communication.

Table 4 summarizes performance for the various security levels. Falkon took 1803.46 secs to run this 30 task workload without any security. The ideal time with no overhead would have been 30x60 = 1800 secs, so we computed our overhead to be 3.46 secs or 0.19%. The highest level of security we have available in GT4 is authentication + encryption, which surpasses the security level used in the MyCluster experiment using just authentication. The MyCluster overhead ranged from 5% to 25% depending on which underlying scheduling technology they used (Condor or SGE respectively). Our overhead is substantially lower, with overheads ranging from 0.30% to 0.96% for the various security mechanisms.

Table 4: Falkon overhead for various security levels

	Exec Time (sec)	Exec Overhead %
Ideal Tasks Execution	1800.00	0.00%
No Security	1803.46	0.19%
GSI Transport (Authentication + Encryption)	1817.37	0.96%
GSI Secure Conversation (Authentication + Encryption)	1815.58	0.87%

5 Application Experiments

We have integrated Falkon 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 without modification. Using Falkon in this way, 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 5 characterizes some applications in terms of the typical number of tasks and the number of stages.

Table 5: 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].

5.1 Functional Magnetic Resonance Imaging

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 12 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.

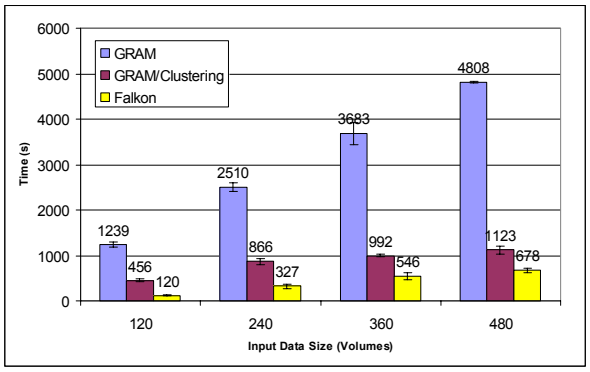

Figure 12: Execution Time for the fMRI Workflow

5.2 Montage Image Mosaicing

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 13 shows execution times for three versions of Montage:

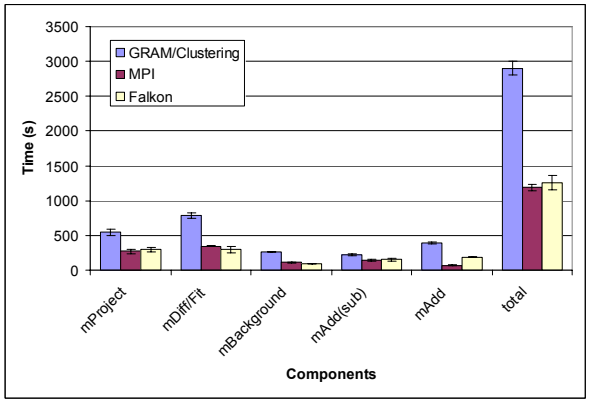

Figure 13: Execution time for Montage application

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.

6 Future Work

We plan to implement and evaluate enhancements, such as task pre-fetching, alternative technologies, data management, and three-tier architecture.

Pre-fetching: As is commonly done in manager-worker systems, executors can request new tasks before they complete execution of old tasks, thus overlapping communication and execution.

Technologies: Performance depends critically on the behavior of our task dispatch mechanisms; the number of messages needed to interact between the various components of the system; and the hardware, programming language, and compiler used. We implemented Falkon in Java and use the Sun JDK 1.4.2 to compile and run Falkon. We use the GT4 Java WS-Core to handle Web Services communications.

One potential optimization is to rewrite Falkon in C/C++, (using, for example, the Globus Toolkit C WS-Core). Another is to change internal communications between components to a custom TCP-based protocol. However, dispatch rates are adequate for applications studied to date, and the primary obstacle to scaling is likely to be data access, not task dispatch.

Data management: Many Swift applications read and write large amounts of data. Applications typically access data from a shared data repository (e.g., NFS, GPFS, GridFTP, web server). Thus, data access can become a bottleneck as applications scale. We expect that data caching, proactive data replication, and data-aware scheduling can offer significant performance improvements for applications that have locality in their data access patterns. We plan to implement data caching mechanisms in Falkon executors, which would allow executors to populate local caches with data the corresponding task would require.

In conjunction with data caching we may wish to implement a data-aware dispatcher. We will evaluate to what extent data aware dispatching reduces performance. A user can choose which dispatcher and executor to use for a specific application..

3-Tier Architecture: Falkon currently requires that the dispatcher and client can each send messages to the other. Thus, each must have at least one port open in their firewall. We have implemented a polling mechanism to bypass firewalls on executors or clients, but we loose this performance and scalability port open in the firewall on which it will accept WS due to the polling mechanism vs. the notification mechanisms. Note that the dispatcher is still required to receive messages from clients and executors.

Falkon also currently assumes that executors operate in a public IP space, so that the dispatcher can communicate with them directly. If (as is sometimes the case) a cluster is configured with a private IP space, to which only a head node has access, the Falkon dispatcher must run on that head node. This organization prevents the use of multiple such clusters. A potential solution to this problem is to introduce intermediate “forwarder” nodes that would act to pass messages between dispatcher and executors.

7 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. Bundling and piggybacking optimizations can reduce further per-task dispatch cost.

Microbenchmarks show that Falkon can achieve one to two orders of magnitude higher throughput (440 tasks/sec) when compared to other batch schedulers. It can sustain high throughput with up to 54,000 managed executors and can process 2,000,000 tasks over a two hour period, operating reliably even as the queue length grew to 1,300,000 tasks.

A "Falkon provider" allows applications coded to the Karajan workflow engine and the Swift parallel programming system to use Falkon with no modification. When using Swift and Falkon together, we demonstrated reductions in end-to-end run time by as much as 90% for applications from the astronomy and medical fields, when compared to the same applications run over batch schedulers.

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.