Falkon: a Fast and Light-weight tasK executiON framework

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

[*]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

{iraicu,yongzh,catalind}@cs.uchicago.edu, {foster,wilde}@mcs.anl.gov

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 integrates (1) multi-level scheduling to separate resource acquisition (via, e.g., requests to batch schedulers) from task dispatch, and (2) a streamlined dispatcher. Falkon's integration of multi-level scheduling and streamlined dispatchers 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 (487 tasks/sec) and scalability (to 54,000 executors and 2,000,000 tasks processed in just 112 minutes) are one to two orders of magnitude better than other systems used in production Grids. Large-scale astronomy and medical applications executed under Falkon by the Swift parallel programming system achieve up to 90% reduction in end-to-end run time, relative to versions that execute tasks via separate scheduler submissions.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design – *Batch processing systems, Distributed systems, Hierarchical design, Interactive systems, Real-time systems and embedded systems.*

General Terms

Management, Performance, Design.

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 three 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. Third, the average wait time of grid jobs is higher in practice than the predictions from simulation-based research. [36] These factors can cause problems when dealing with application workloads that contain a large number of tasks.

One solution to this problem is to transform applications 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 [9] and Singh [10] create an embedded Condor pool by "gliding in" Condor workers to a compute cluster, while MyCluster [11] can embed both Condor pools and Sun Grid Engine (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:

1. Reducing task dispatch time by using a streamlined dispatcher that eliminates support for features such as multiple queues, priorities, accounting, etc.
2. 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 and scale to 54,000 executors. A synthetic application demonstrates the benefits of adaptive provisioning. Finally, results for two applications demonstrate that substantial speedups can be achieved for real scientific applications.

SC07 November 10-16, 2007, Reno, Nevada, USA
(c) 2007 ACM 978-1-59593-764-3/07/0011…$5.00

2. RELATED WORK

Full-featured local resource managers (LRMs) such as Condor [1, 15], Portable Batch System (PBS) [16], Load Sharing Facility (LSF) [17], and SGE [18] support client specification of resource requirements, data staging, process migration, check-pointing, accounting, and daemon fault recovery. Falkon, in contrast, is not a full-featured LRM: it focuses on efficient task dispatch and thus can omit some of these features in order 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, data staging).

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 (trivial) tasks in two hours, and has scaled to 54K managed executors with similarly high throughput. This test as well as other throughput tests achieving 487 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 et al. 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 GRAM4 [27]) 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 be dedicated to a single workload. Thus, Singh et al. find, in a simulation study [12], a reduction of about 50% in completion time, due to reduction in queue wait 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. Our work differs in its focus on resource provisioning on non-dedicated resources managed by LRMs.

3. FALKON ARCHITECTURE

Our description of the Falkon architecture encompasses execution model, communication protocol, performance enhancements, and information regarding ease of use of the Falkon API.

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 (up to some specified number of retries). 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 a *next-available* policy, which dispatches each task to the next available resource. We assume here that all data needed by a task is available in a shared file system. In the future, we will examine dispatch policies that take into account data locality.

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, *n* requests for a single resource, or a series of arithmetically or exponentially larger requests, or that use system functions to determine available resources. Due to space restrictions, in the experiments reported in this paper, we consider only the first policy ("all-at-once"), which allocates all needed resources in a single request.

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 queued tasks, release all resources" or "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 resource should release itself." Note that resource acquisition and release policies are typically not independent: in most batch schedulers, a set of resources allocated in a single request must all be de-allocated before the requested resources become free and ready to be used by the next allocation. Ideally, one must release all resources obtained in a single request at once, which requires a certain level of synchronization among the resources allocated within a single allocation. In the experiments reported in this paper, we used a distributed policy, releasing individual resources after a specified idle time was reached. In the future, we plan to improve our distributed policy by coordinating between all the resources allocated in a single request to de-allocate all at the same time.

3.2 Architecture

Falkon consists of a dispatcher, a provisioner, and zero or more executors (Figure 1). Figure 2 has the series of message exchanges that occur between the various Falkon components. As we describe the architecture and the components' interaction, we will denote the message numbers from Figure 2 in square braces; some messages have two numbers, denoting both a send and receive, while others have only a single number, denoting a simple send.

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 (solid lines in Figure 2), except that notifications are performed via a custom TCP-based protocol (dotted lines). The notification mechanism is implemented over TCP because when we first implemented the core Falkon components using GT3.9.5, the Globus Toolkit did not support brokered WS notifications. The recent GT4.0.5 release supports brokered notifications.

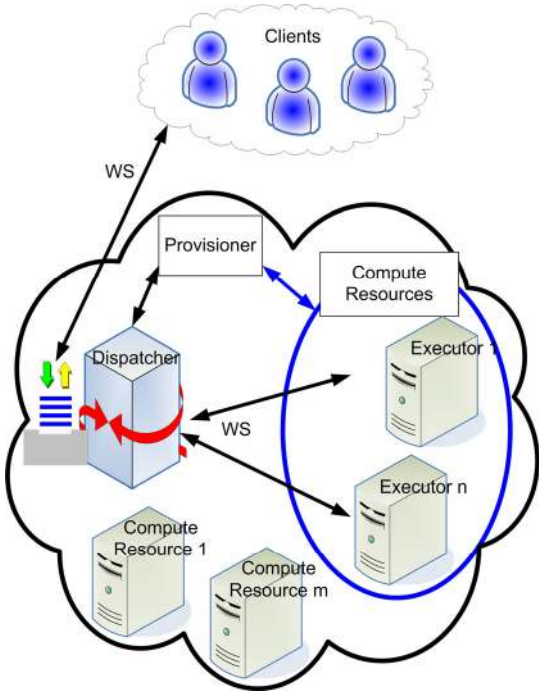

Figure 1: Falkon architecture overview

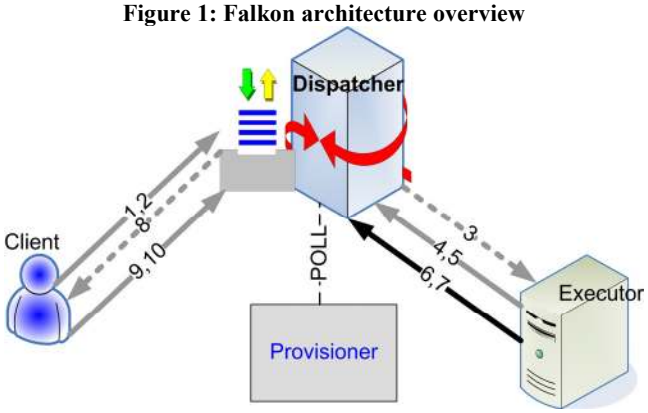

Figure 2: Falkon components and message exchange

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 {1,2}, monitor progress (or wait for notifications {8}), retrieve results {9,10}, and (finally) destroy the instance.

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). A shared notification engine among all the different queues 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 effectively 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 {POLL} and, based on policy, determines whether to create additional executors, and if so, how many, and for how long. Creation requests are issued via GRAM4 [27] to abstract LRM details.

A new **executor** registers with the dispatcher. Work is then supplied as follows: the dispatcher notifies the executor when work is available {3}; the executor requests work {4}; the dispatcher returns the task(s) {5}; the executor executes the supplied task(s) and returns results, including return code and optional standard output/error strings {6}; and the dispatcher acknowledges delivery {7}.

3.3 Push vs. Pull Model

We considered both a push and a pull model when designing the Dispatcher-Executor communication protocol. We explain here why we chose a hybrid push/pull model, where the push is a notification {3} and the pull is the get work {4}.

In a pull model, Executors request work from the Dispatcher. A "get work" request can be either blocking or non-blocking. A blocking request can provide better responsiveness than a non-blocking request (as it avoids polling), but requires that the Dispatcher maintain state for each Executor waiting for work. In the case of non-blocking requests, Executors must poll the Dispatcher periodically, which can reduce responsiveness and scalability. For example, we find that when using Web Services operations to communicate requests, a cluster with 500 Executors polling every second keeps Dispatcher CPU utilization at 100%. Thus, the polling interval must be increased for larger deployments, which reduces responsiveness accordingly. Additionally, the Dispatcher does not control the order and rate of Executor requests, which can hinder efficient scheduling due to the inability for the scheduler to decide the order dispatched tasks. Despite all these negative things about a pull model, there are two advantages: 1) it is friendly with firewalls, and 2) it simplifies the Dispatcher logic.

A push model assumes that the Dispatcher can initiate a communication with its Executors, which implies one of the following three implementation alternatives for the Executor:

1) It is implemented as a Web Service (as opposed to a simpler client that can only initiate WS communication). Thus, a WS

container must be deployed on every compute node (in the absence of a shared file system); this alternative has the largest footprint but is easy to implement.

2) It supports notifications. Here, we only need the client code plus a few libraries required for WS communications. This alternative has a medium-sized footprint with a medium implementation complexity (WS and notification).

3) It uses a custom communication protocol and can be both a server and a client. This approach only needs the libraries to support that protocol (e.g., TCP). It has the smallest footprint but requires the implementation of the custom protocol.

All three approaches have problems with firewalls, but we have not found this to be a big issue in deployments to date, as the Dispatcher and Executors are typically located within a single site in which firewalls are not an issue. We discuss in Section 6 how this problem could be addressed via a three-tier architecture that supports both cross-firewall communications and communications with Executors operating in a private IP space.

We decided to use alternative two, with medium footprint and medium implementation complexity. A notification simply identifies the resource key where the work can be picked up from at the Dispatcher, and then the Executor uses a WS call to request the corresponding work.

This hybrid pull/push model provides the following benefits: higher system responsiveness and efficiency relative to a pure push model; higher scalability relative to a pure pull model; medium size disk and memory footprint; more controllable throttling than a pure pull model; and the ability to implement more sophisticated (e.g., data-aware) schedulers.

3.4 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 (messages {6,7} from Figure 2).

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 two per task (one message from executor to dispatcher to deliver a result, and one associated response from dispatcher to executor to acknowledge receipt and provide a new task); these two messages make up a single WS call. Line shading in Figure 2 shows where bundling optimization can be used: black lines denote that the corresponding message occurs on a per-task basis, while grey lines denote that through bundling optimizations, the corresponding messages occur for a set of tasks.

3.5 Ease of Use

We modified the Swift parallel programming system by implementing a new provider to use Falkon for task dispatch. The Falkon provider has 840 lines of Java code, a value comparable to GRAM2 provider (850 lines), GRAM4 provider (517 lines), and the Condor provider (575 lines).

4. PERFORMANCE EVALUATION

Table 1 lists the platforms used in experiments. Latency between these systems was one to two milliseconds. We assume a one-to-one mapping between executors and processors in all experiments. Of the 162 nodes on TG_ANL_IA32 and TG_ANL_IA64, 128 were free for our 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 without Data Access

To determine maximum throughput, we measured performance running "sleep 0." We ran executors on TG_ANL_IA32 and TG_ANL_IA64, the dispatcher on UC_x64, and the client generating the workload on TP_UC_x64. As each node had two processors, we ran two executors per node, for a total of 256 executors. We measured Falkon throughput for short ("sleep 0") tasks both without any security and with GSISecureConversation that performs both authentication and encryption. We enabled two optimizations discussed below, namely client-dispatcher bundling and piggy-backing; however, every task is transmitted individually from dispatcher to an executor.

For purposes of comparison, we also tested GT4 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. We claim this to be the upper bound on Falkon throughput performance that can be achieved on the tested hardware (UC_x64), assuming that there is no task bundling between dispatcher and executors, and that each task is handled via a separate dispatch.

Figure 3 shows GT4 without security achieves 500 WS calls/sec; Falkon reaches 487 tasks/sec (without security) and 204 tasks/sec (with security). A single Falkon executor without and with security can handle 28 and 12 tasks/sec, respectively.

We also measured Condor and PBS performance on the same testbed, with nodes managed by PBS v2.1.8. To measure PBS throughput, we submitted 100 short tasks (sleep 0) and measured the time to completion on the 64 available nodes. The experiment took on average 224 seconds for 10 runs netting 0.45 tasks/sec.

As we did not have access to a dedicated Condor pool, we used MyCluster [11] to create a 64-node Condor v6.7.2 pool via PBS submissions. Once the 64 nodes were allocated from PBS and were available within MyCluster, we performed the same experiment, 100 short tasks over Condor. The total time was on average 203 seconds for 10 runs netting 0.49 tasks/sec. As far as we could tell, neither PBS nor Condor were using any security mechanisms between the various components within these systems. MyCluster does use authentication and authorization to setup the virtual cluster (a one time cost), but thereafter no security was used. It is also worth mentioning that we intentionally used a small number of tasks to test PBS and Condor as the achieved throughput drops as tasks accumulate in the wait queue, and our goal was to measure the best case scenario for their ability to dispatch and execute small tasks.

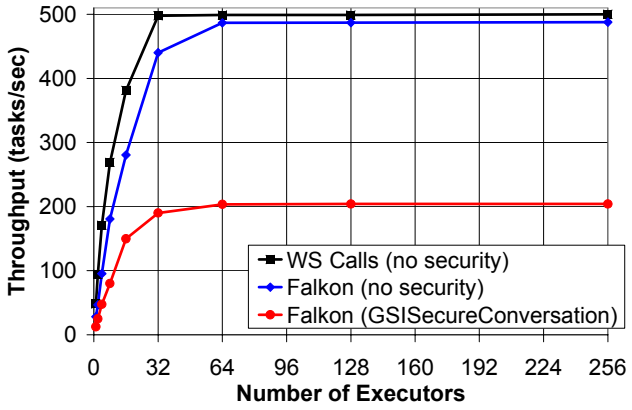

Figure 3: Throughput as function of executor count

There are newer versions of both Condor and PBS, and both systems can likely be configured for higher throughput. We do not know whether or not these experiments reflect performance with security enabled or not, and all the details regarding the hardware used; see Table 2 for details on the various hardware used and a summary of the reported throughputs. In summary, Falkon's throughput performance compares favorably to all, regardless of the security settings used be these other systems.

Table 2: Measured and cited throughput for Falkon, Condor, and PBS

System	Comments	Throughput (tasks/sec)
Falkon (no security)	Dual Xeon 3GHz w/ HT 2GB	487
Falkon (GSISecureConversation)	Dual Xeon 3GHz w/ HT 2GB	204
Condor (v6.7.2)	Dual Xeon 2.4GHz, 4GB	0.49
PBS (v2.1.8)	Dual Xeon 2.4GHz, 4GB	0.45
Condor (v6.7.2) [15]	Quad Xeon 3 GHz, 4GB	2
Condor (v6.8.2) [34]		0.42
Condor (v6.9.3) [34]		11
Condor-J2 [15]	Quad Xeon 3 GHz, 4GB	22
BOINC [19, 20]	Dual Xeon 2.4GHz, 2GB	93

4.2 Throughput with Data Access

Most tasks started via a system such as Falkon, Condor, or PBS will need to read and write data. A comprehensive evaluation of these systems' I/O performance is difficult because of the wide range of I/O architectures encountered in practical settings.

As a first step towards such an evaluation, we measured Falkon throughput with synthetic tasks that performed data staging as well as computation. We fixed the number of executors at 128 (64 nodes) and performed four sets of experiments in which, for varying data sizes from one byte to one GB, we varied (a) data location (on GPFS shared file system or the local disk of each compute node), and (b) whether tasks only read or both read and wrote the data. All experiments were performed without security.

Figure 4 shows our results. All scales are logarithmic. The solid lines denote throughput in tasks/sec and the dotted lines denote throughput in Mb/sec. Falkon maintained high task throughput (within a few percent of the peak 487 tasks/sec) for up to 1 MB data sizes (for GPFS read and LOCAL read+write) and up to 10 MB data size (for LOCAL read). For GPFS read+write, the best throughput Falkon could achieve was 150 tasks/sec, even with 1 byte data sizes. We attribute this result to the GPFS shared file system's inability to support many write operations from 128 concurrent processors. (The GPFS shared file system in our testbed has eight I/O nodes.)

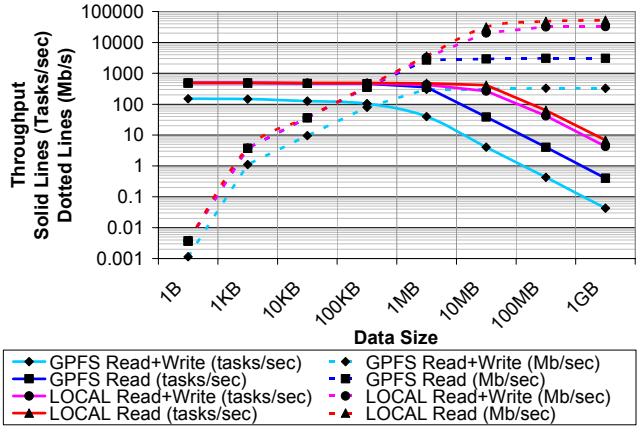

Figure 4: Throughput as a function of data size on 64 nodes

As data sizes increase, throughput (Mb/sec: dotted lines) plateaus at either 1 MB or 10 MB data sizes, depending on the experiment. GPFS read+write peaks at 326 Mb/sec, GPFS read at 3,067 Mb/sec, LOCAL read+write at 32,667 Mb/sec, and LOCAL read at 52,015 Mb/sec. With 1 GB data, throughput was 0.04 tasks/sec, 0.4 tasks/sec, 4.28 tasks/sec, and 6.81 tasks/sec, respectively.

We have not performed comparable experiments with the PBS and Condor systems considered earlier. However, as tasks started via these systems will access data via the same mechanisms as those evaluated here, we can expect that as the amount of data accesses increases, I/O costs will come to dominate and performance differences among the systems will become smaller.

More importantly, these results emphasize the importance of using local disk to cache data products written by one task and read by another on local disk—a feature supported by Falkon, although not evaluated here.

4.3 Bundling

It has been shown that real grid workloads comprise a large percentage of tasks submitted as batches of tasks. [37] In order to optimize the task submission performance, we propose to bundle many tasks together in each submission. We measured performance for a workload of "sleep 0" tasks as a function of

task bundle size. Figure 5 shows that performance increases from about 20 tasks/sec, without bundling, to a peak of almost 1500 tasks/sec, with bundling.

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 to attempt to remedy this limitation.

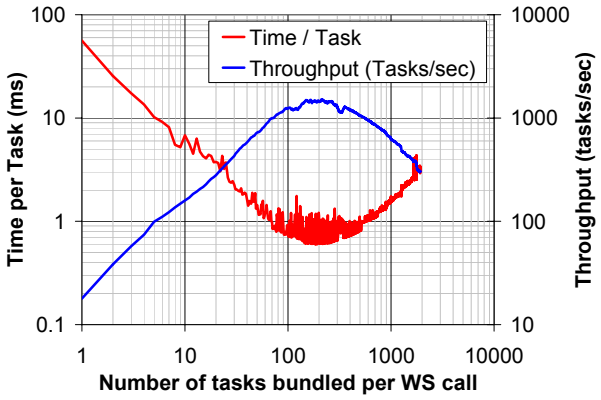

Figure 5: Bundling throughput and cost per task

4.4 Efficiency and Speedup

Figures 6 shows efficiency ($E_P=S_P/P$) as a function of number of processors (P) and task length; speedup is defined as $S_P=T_1/T_P$, where T_n is the execution time on n processors. These experiments were conducted on TG_ANL_IA32 and TG_ANL_IA64 with no security and with optimizations such as bundling and "piggy-backing" enabled.

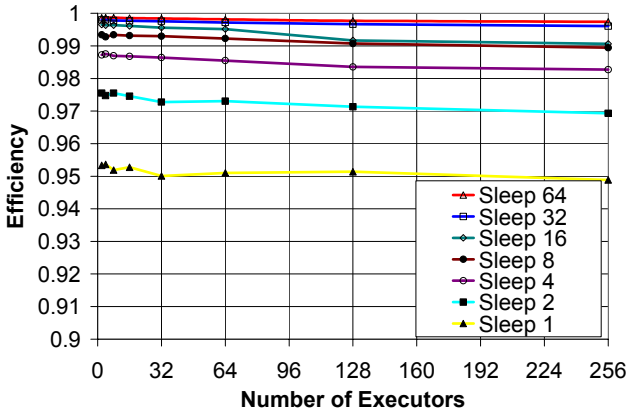

Figure 6: Efficiency for various task length and executors

We see that even with short (1 sec) tasks, we achieve high efficiencies (95% in the worst case with 256 executors). Note that there is typically less than 1% loss in efficiency as we increase from 1 executor to 256 executors. As we increase the number of executors beyond the maximum throughput we can sustain (487 executors with 1 sec long tasks, netting the 487 tasks/sec), the efficiency of the 1 sec tasks will start to drop as the Dispatcher's CPU utilization will be saturated. In the worst case (1 sec tasks), we achieve a speedup of 242 with 256 executors; with 64 sec tasks, the speedup is 255.5.

We performed two similar experiments on Condor and PBS to gain insight into how Falkon efficiency compared with that of other systems. We fixed the number of resources to 32 nodes and measured the time to complete 64 tasks of various lengths (ranging from 1 sec to 16384).

We see Falkon's efficiency to be 95% with 1 sec tasks and 99% with 8 sec tasks. In contrast, both PBS (v2.1.8) and Condor (v6.7.2) have an efficiency of less than 1% for 1 sec tasks and require about 1,200 sec tasks to get 90% efficiency and 3,600 sec tasks to get 95% efficiency. They only achieve 99% efficiency with 16,000 sec tasks.

As both the tested PBS and Condor versions that are in production on the TG_ANL_IA32 and TG_ANL_IA64 clusters are not the latest versions, we also derived the efficiency curve for Condor version 6.9.3, the latest development Condor version, which is claimed to have a throughput of 11 tasks/sec [34] (up from our measured 0.45~0.49 tasks/sec and the 2 tasks/sec reported by others [15]). Efficiency is much improved, reaching 90%, 95%, and 99% for task lengths of 50, 100, and 1000 secs. respectively.

The results in Figure 7 for Condor v6.9.3 are derived, not measured. We derived based on the achieved throughput cited in [34] of 11 tasks/sec for sleep 0 tasks. Essentially, we computed the per task overhead of 0.0909 seconds, which we could then add to the ideal time of each respective task length to get an estimated task execution time. With this execution time, we could compute speedup, which we then used to compute efficiency. Our derivation of efficiency is simplistic, but it allowed us to plot the likely efficiency of the latest development Condor code against the older production Condor code, the PBS production code, and Falkon. It should be noted that Figure 7 illustrates the efficiency of these systems for a relatively small set of resources (only 64 processors), and that the efficiency gap will likely only increase as the number of resources increases.

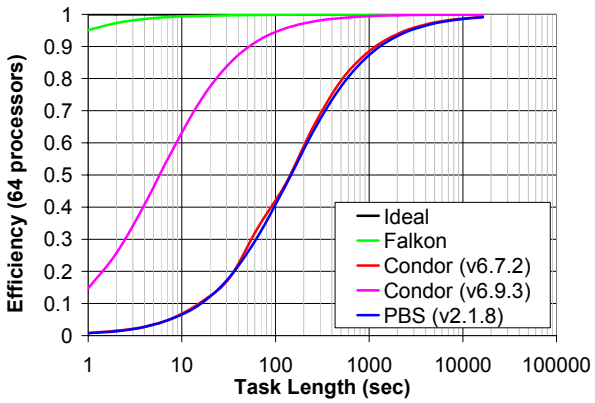

Figure 7: Efficiency of resource usage for varying task lengths on 64 processors comparing Falkon, Condor and PBS

4.5 Scalability

To test scalability and robustness, we performed experiments that pushed Falkon to its limits, both in terms of memory consumption and in terms of CPU utilization.

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 64 executors on 32 machines from TG_ANL_IA32 and ran the dispatcher on UC_x64 and the client on TP_UC_x64.

Figure 8 results show the entire run over 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 completions, and the solid blue line is the moving average (over 60 sample intervals, and thus 60 secs) of raw throughput. Average throughput was 298 tasks/sec. Note the slight increase of about 10~15 tasks/sec when the queue stopped growing, as the client finished submitting all two million tasks.

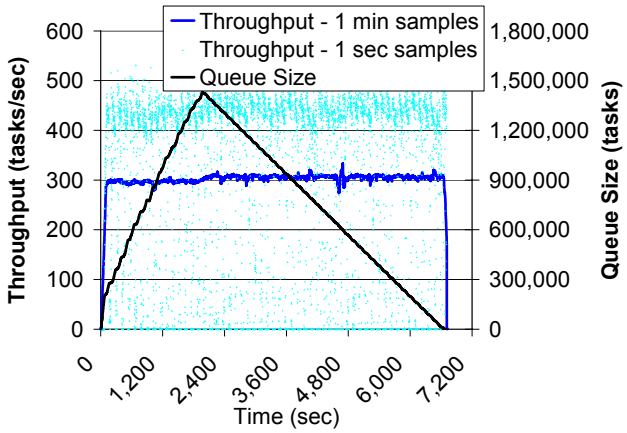

Figure 8: Long running test with 2M tasks

The graph shows the raw throughput samples (taken at 1 second intervals) to be between 400 and 500 tasks per second for the majority of the experiment, yet the moving average was around 300 tasks/sec. A close analysis shows frequent raw throughput samples at 0 tasks/sec, which we attribute to JVM garbage collection. We may be able to reduce this variation by configuring the JVM to garbage collect more frequently.

In a second experiment, we tested how many executors the dispatcher could handle. We did not have an available system large enough to test the limits of the Falkon implementation, and therefore we ran multiple executors on each physical machine emulating a larger number of virtual executors. Others have used this experimental method with success [15].

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 secs." 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 as each executor only processed one task each.

Figure 9 shows that the dispatch rate (green line) equals the submit rate. The black line shows the number of busy executors, which increases from 0 to 54K in 408 secs. As soon as the first task finishes after 480 secs (the task length), results start to be delivered to 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.

We also measured task overhead, by which we mean 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). Figure 10 shows per task overhead in millisecs for each task executed in the experiment of Figure 9, ordered by task start time.

We see that most overheads were below 200 ms, with just a few higher than that and a maximum of 1300 ms. (As we have 900 executors per physical machine, overhead is higher than normal as each thread gets only a fraction of the computer's resources.)

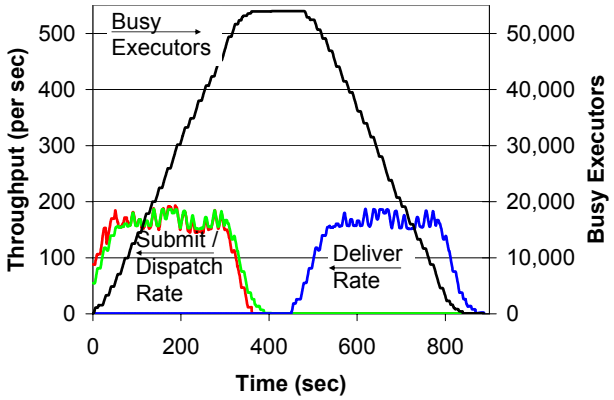

Figure 9: Falkon scalability with 54K executors

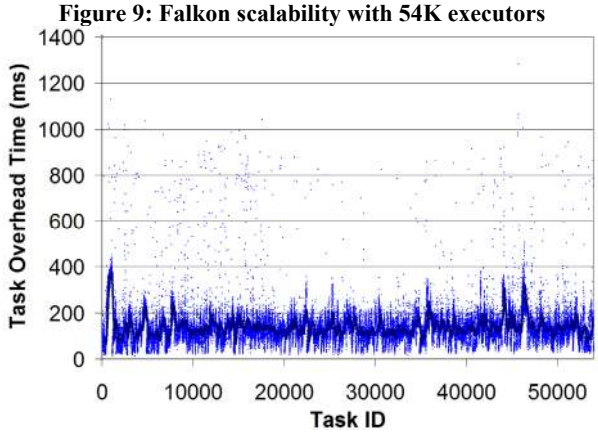

Figure 10: Task overhead with 54K executors

4.6 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 11 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 in stages 13 and 14, 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. We choose this workload to showcase and evaluate the flexibility of Falkon's dynamic resource provisioning, as it can adapt to varying resource requirements and task durations.

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:

- *GRAM4+PBS* (without Falkon): Each task was submitted as a separate GRAM4 task to 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.

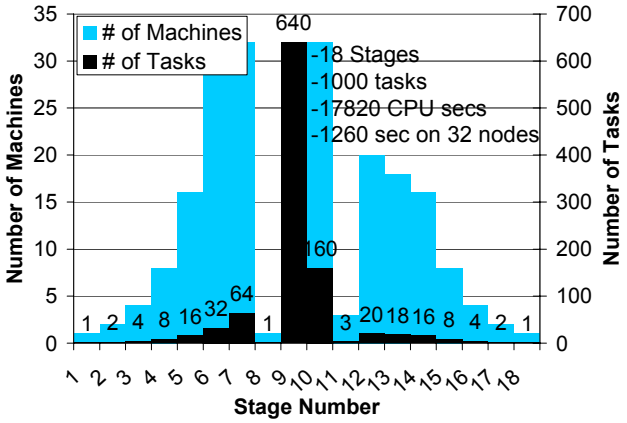

Figure 11: The 18-stage synthetic workload.

Table 3 gives, for each experiment, the average per-task queue time and execution time, and also the ratio exec_time/(exec_time+queue_time).

Table 3: Average per-task queue and execution times for synthetic workload

	GRAM4 +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 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 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.) GRAM4+PBS yields the worst performance, with only 8.5% on average, less than a third of ideal.

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. In contrast, GRAM4+PBS experiences a queue time of 611.1 secs: 15 times larger than the ideal. Also, note the execution time for Falkon with 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, GRAM4+PBS has 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 GRAM4 and PBS have, which further strengthens our argument that they are not suitable for short tasks.

Table 4 shows, for each strategy, the time to complete the 18 stages, resource utilization, execution efficiency, and number of resource allocations. We define resource utilization and execution efficiency as follows:

- $resource\_utilization = \frac{resources\_used}{resources\_used + resources\_wasted}$
- $exec\_efficiency = \frac{ideal\_time}{actual\_time}$

We define resources as "wasted," for the Falkon strategies, when they have been allocated and registered with the dispatcher, but are idle. For the GRAM4+PBS case, the "wasted" time is the difference between the measured and reported task execution time, where the reported task execution time is from the time GRAM4 sends a notification of the task becoming "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 execution.

The resources used are the same (17,820 CPU secs) for all cases, as we have fixed run times for all 1000 tasks. We expected GRAM4+PBS to not have any wasted resources, 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. The average execution time of 56.5 secs shows that GRAM4+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 GRAM4+PBS resource wastage yet worse.

Falkon with dynamic resource provisioning fairs better from the perspective of resource wastage. Falkon-15 has the fewest wasted resources (2032 secs) and Falkon-∞ the worst (22,940 CPU secs). The resource utilization shows the fraction of time the machines were executing tasks vs. idle. Due to its high resource wastage, GRAM4+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 GRAM4+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 GRAM4+PBS (~0.5/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 4: Summary of overall resource utilization and execution efficiency for the synthetic workload

	GRAM4 +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 illustrate how provisioning works in practice, we show in Figures 10 and 11 execution details for Falkon-15 and Falkon-180, respectively. These figures show the instantaneous number of allocated, registered, and active executors over time.

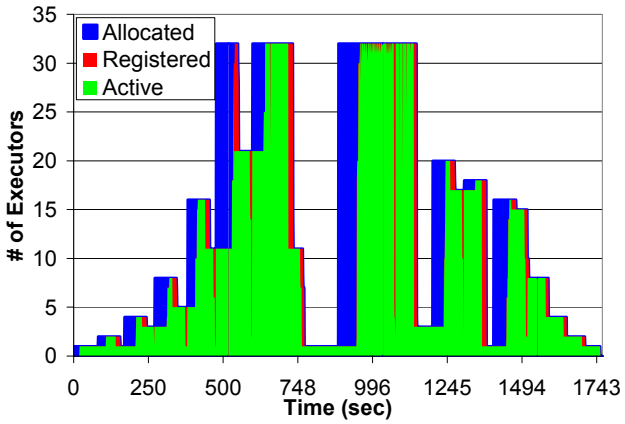

Figure 12: Synthetic workload for Falkon-15

Allocated (blue) are executors 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 second 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 utilized resources. We see that Falkon-15 has fewer idle resources (as they are released sooner) but spends more time acquiring resources and overall has a longer total execution time than Falkon-60.

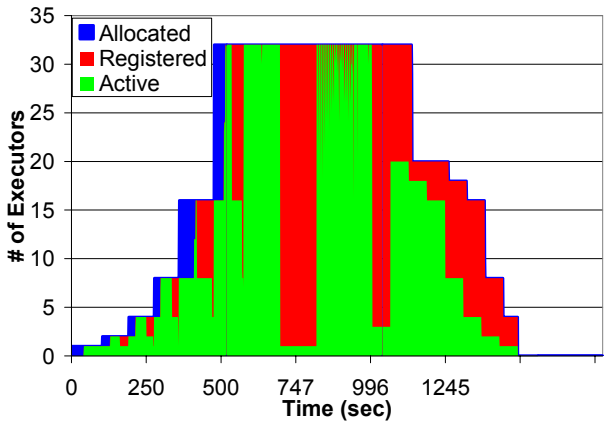

Figure 13: Synthetic workload for Falkon-180

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 [3]. Thus, Karajan and Swift applications can use Falkon without modification. We observe reductions in end-to-end run time by as much as 90% when compared to traditional approaches in which applications used batch schedulers directly.

Swift has been applied to applications in the 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 stages.

Table 5: Swift applications [3]; all could benefit from Falkon

Application	#Tasks/workflow	#Stages
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
SDSS: Stacking, AstroPortal	10Ks ~ 100Ks	2 ~ 4
MolDyn: Molecular Dynamics	1Ks ~ 20Ks	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 [22, 32].

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, from 120 volumes (480 tasks for the four stages) to 480 volumes (1960 tasks). Each task can run in a few seconds on an nodes in TG_ANL_IA64.

We compared three implementation approaches: task submission via GRAM4+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 TG_ANL_IA32 and application tasks on TG_ANL_IA64.

In Figure 14 we show execution times for the different approaches and for different problem sizes. Although GRAM4+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.

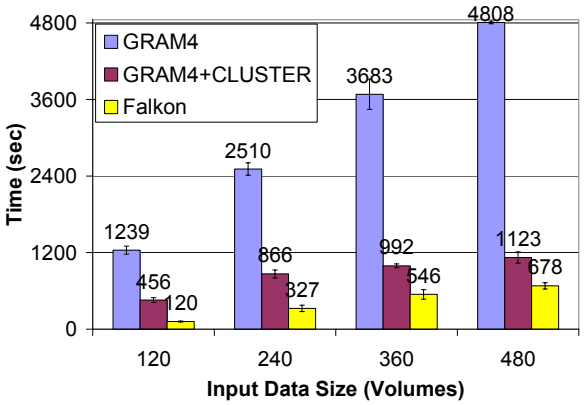

Figure 14: Execution Time for the fMRI Workflow

5.2 Montage Image Mosaic Composition

Our second application, Montage, generates large astronomical image mosaics by composing multiple small images [22, 32]. 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 487 input images and 2,200 overlapping image sections between them. The resulting task graph has many small tasks.

Figure 15 shows execution times for three versions of Montage: Swift with clustering, submitting via GRAM4+PBS; Swift submitting via Falkon; and an MPI version constructed by the Montage team. The second co-add step was only parallelized in the MPI version; thus, Falkon performs poorly in this step. Both the GRAM4 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 and DAGman [33]. They do not implement quite the same application: for example, they run two tasks (mOverlap and mImgtlb) in a 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) than 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.

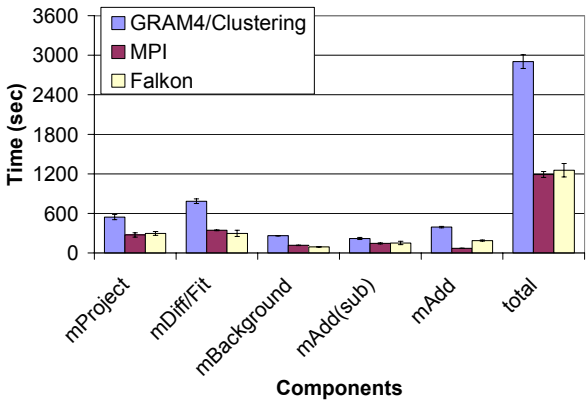

Figure 15: Execution time for Montage application

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, current dispatch rates approaching 500 tasks/sec are adequate for applications studied to date; we believe the primary obstacle to scaling applications (that have many small tasks) will likely be data access, and not task dispatch as it has been in the past.

Data management: Many Swift applications read and write large amounts of data. Applications typically access data from a shared data repository (e.g., via NFS, GPFS, GridFTP, or HTTP). Thus, data access can become a bottleneck as applications scale. We expect that data caching, proactive data replication [35], and data-aware scheduling can offer significant performance improvements for applications that exhibit locality in their data access patterns. We plan to implement data caching mechanisms in Falkon executors, so that executors can populate local caches with data that tasks require. We also plan to implement a data-aware dispatcher, and will evaluate the impact of data aware dispatching on both application performance and raw dispatch performance.

3-Tier Architecture: Falkon currently requires that the dispatcher and client be able to exchange messages. The two-way communication on different connections is an artifact of the

notification messages that are being sent from the dispatcher to both the clients and the executors. We have implemented a polling mechanism to bypass any firewall issues on executors or clients, but we lose performance and scalability due to polling overheads. Note that the dispatcher is still required to have a port open in the firewall, to accept WS 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 manager node has access, the Falkon dispatcher must run on that manager node. In such cases, Falkon cannot use multiple clusters. A potential solution to this problem is to introduce intermediate "forwarder" nodes that pass messages between dispatcher and executors.

Figure 16 shows such a 3-Tier architecture overview, which has a strong resemblance to a hierarchical structure. One or more *forwarders* receive tasks from a client. Both the client and the forwarder(s) may reside anywhere in a public IP space. Next, *dispatchers* are deployed on cluster manager nodes, which—if a cluster's compute nodes are in a private IP space—typically have both a public IP address and a private IP address. Finally, each dispatcher manages a disjoint set of *executors* that may run in either a private or public IP space. We are investigating this three-tier architecture with the goal of scaling Falkon to two or more orders of magnitude more executors, as will be required for future grids and for modern supercomputers, such as the IBM BlueGene/P, that may have 256,000 or more processors.

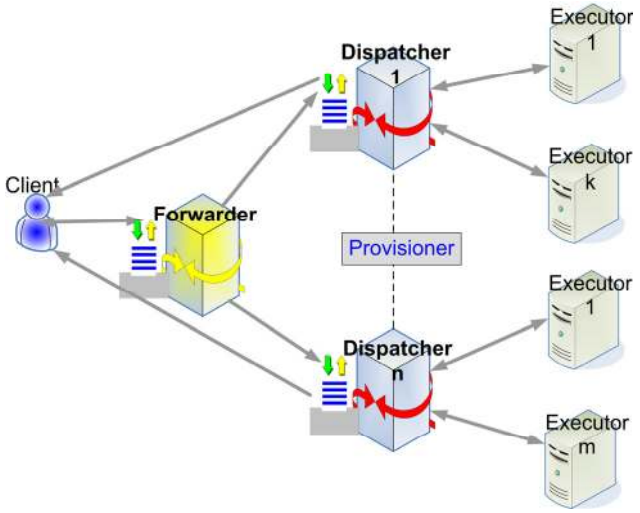

Figure 16: 3-Tier Architecture Overview

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.

We have designed Falkon, a Fast and Light-weight tasK executiON framework, 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 provisioner is responsible for allocating and de-allocating resources in response to changing 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 (487 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 in just 112 minutes, operating reliably even as the queue length grew to 1,500,000 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.

Falkon's novelty consist in its combination of a fast lightweight scheduling overlay on top of virtual clusters with the use of grid protocols for adaptive resource allocation. This approach allows us to achieve higher task throughput than previous systems, while also allowing applications to trade off system responsiveness, resource utilization, and execution efficiency.

8. ACKNOWLEDGMENTS

This work was supported in part by the NASA Ames Research Center GSRP Grant Number NNA06CB89H and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. We thank our collaborator Alex Szalay who inspired the Falkon architecture and implementation by proposing as a challenge problem a sky survey stacking service, whose primary requirement was to perform many small tasks in Grid environments. We also thank TeraGrid and the Computation Institute for hosting the experiments reported here.

9. 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, 2007.

[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", 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," IEEE International Symposium on High Performance Distributed Computing, 2000.

[6] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", International Journal of Supercomputer Applications, 15 (3). 200-222. 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, 2002.

[10] G. Singh, C. Kesselman, E. Deelman, "Optimizing Grid-Based Workflow Execution." Journal of Grid Computing, Volume 3(3-4), December 2005, pp. 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", USC ISI Technical Report 2006.

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

[14] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the Grid", 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, 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, 2004.

[20] D.P. Anderson, E. Korpela, R. Walton. "High-Performance Task Distribution for Volunteer Computing." IEEE 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." 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," 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 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC06), 2006.

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

[26] Catlett, C. et al., "TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications," HPC 2006.

[27] M. Feller, I. Foster, and S. Martin. "GT4 GRAM: A Functionality and Performance Study", TeraGrid Conference 2007.

[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, 2005.

[30] I. Raicu, I. Foster, A. Szalay. "Harnessing Grid Resources to Enable the Dynamic Analysis of Large Astronomy Datasets", IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC06), 2006.

[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." 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, 219-237.

[34] T. Tannenbaum. "Condor RoadMap", Condor Week 2007.

[35] K. Ranganathan, I. Foster, "Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids", Journal of Grid Computing, V1(1) 2003.

[36] A. Iosup, C. Dumitrescu, D.H.J. Epema, H. Li, L. Wolters, "How are Real Grids Used? The Analysis of Four Grid Traces and Its Implications", IEEE/ACM International Conference on Grid Computing (Grid), 2006.

[37] A. Iosup, M. Jan, O. Sonmez, and D.H.J. Epema, "The Characteristics and Performance of Groups of Jobs in Grids", EuroPar 2007.