Cluster Computing (2009)

Middleware Support for Many-Task Computing

Ioan Raicu • Ian Foster • Mike Wilde • Zhao Zhang • Kamil Iskra • Pete Beckman • Yong Zhao • Alex Szalay • Alok Choudhary • Philip Little • Christopher Moretti • Amitabh Chaudhary • Douglas Thain

Received: November 6th, 2009

Abstract Many-task computing aims to bridge the gap between two computing paradigms, high throughput computing and high performance computing. Many-task computing denotes high-performance computations comprising multiple distinct activities, coupled via file system operations. The aggregate number of tasks, quantity of computing, and volumes of data may be extremely large. Traditional techniques found in production systems in the scientific community to support many-task computing do not scale to today's largest systems, due to issues in local resource manager scalability and granularity, efficient utilization of the raw hardware, long wait queue times, and shared/parallel file system contention and scalability. To address these limitations, we adopted a "top-down" approach to building a middleware called Falkon, to support the most demanding many-task computing applications at the largest scales. Falkon (Fast and Light-weight tasK executiON framework) integrates (1) multi-level scheduling to enable dynamic resource provisioning and minimize wait queue times, (2) a streamlined task dispatcher able to achieve orders-of-magnitude higher task dispatch rates than conventional schedulers, and (3) data diffusion which performs data caching and uses a data-aware scheduler to co-locate computational and storage resources. Micro-benchmarks have shown Falkon to achieve over 15K+ tasks/sec throughputs, scale to hundreds of thousands of processors and to millions of queued tasks, and execute billions of tasks per day. Data diffusion has also shown to improve applications scalability and performance, with its ability to achieve hundreds of Gb/s I/O rates on modest sized clusters, with Tb/s I/O rates on the horizon. Falkon has shown orders of magnitude improvements in performance and scalability than traditional approaches to resource management across many diverse workloads and applications at scales of billions of tasks on hundreds of thousands of processors across clusters, specialized systems, Grids, and supercomputers. Falkon's performance and scalability have enabled a new class of applications called Many-Task Computing to operate at previously so-believed impossible scales with high efficiency.

I. Raicu*
Northwestern University, Evanston IL, USA
email: iraicu@eecs.northwestern.edu

I. Raicu
email: iraicu@eecs.northwestern.edu

I. Foster • M. Wilde • K. Iskra, P. Beckman
University of Chicago, Chicago IL, USA
Argonne National Laboratory, Argonne IL, USA

I. Foster
email: foster@anl.gov

M. Wilde
email: wilde@mcs.anl.gov

K. Iskra
email: iskra@mcs.anl.gov

P. Beckman
email: beckman@mcs.anl.gov

Z. Zhang
University of Chicago, Chicago IL, USA
email: zhaozhang@uchicago.edu

Y. Zhao
Microsoft, Redmond WA, USA
email: yozha@microsoft.com

A. Szalay
John Hopkins University, Baltimore MD, USA
email: szalay@jhu.edu

A. Choudhary
Northwestern University, Evanston IL, USA
email: choudhar@eecs.northwestern.edu

P. Little • C. Moretti • A. Chaudhary • D. Thain
University of Notre Dame, Notre Dame IN, USA

P. Little
email: plittle1@nd.edu

C. Moretti
email: cmoretti@cse.nd.edu

A. Chaudhary
email: achaudha@cse.nd.edu

D. Thain
email: dthain@nd.edu

1. Introduction

We want to enable the use of large-scale distributed systems for task-parallel applications, which are linked into useful workflows through the looser

task-coupling model of passing data via files between dependent tasks. This potentially larger class of task-parallel applications is precluded from leveraging the increasing power of modern parallel systems such as supercomputers (e.g. IBM Blue Gene/L [1] and Blue Gene/P [2]) due to the lack of efficient support in those systems for the "scripting" programming model [3]. With advances in e-Science and the growing complexity of scientific analyses, more scientists and researchers rely on various forms of scripting to automate end-to-end application processes involving task coordination, provenance tracking, and bookkeeping. Their approaches are typically based on a model of loosely coupled computation, in which data is exchanged among tasks via files, databases or XML documents, or a combination of these. Vast increases in data volume combined with the growing complexity of data analysis procedures and algorithms have rendered traditional manual exploration unfavorable as compared with modern high performance computing processes automated by scientific workflow systems. [4]

The problem space can be partitioned into four main categories (see Figure 1). 1) At the low end of the spectrum (low number of tasks and small input size), we have tightly coupled Message Passing Interface (MPI) applications (white). 2) As the data size increases, we move into the analytics category, such as data mining and analysis (blue); MapReduce [5] is an example for this category. 3) Keeping data size modest, but increasing the number of tasks moves us into the loosely coupled applications involving many tasks (yellow); Swift/Falkon [6, 7] and Pegasus/DAGMan [8] are examples of this category. 4) Finally, the combination of both many tasks and large datasets moves us into the data-intensive Many-Task Computing [9] category (green); examples are Swift/Falkon and data diffusion [10], Dryad [11], and Sawzall [12].

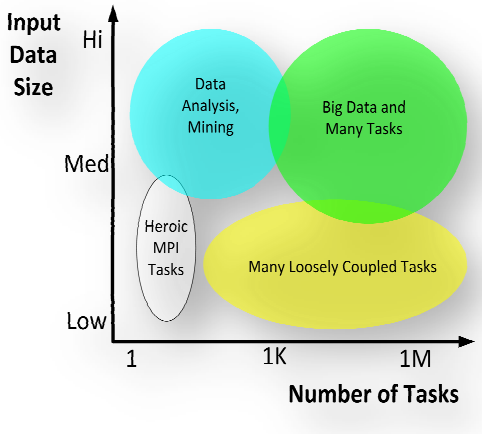


Figure 1: Problem types with respect to data size and number of tasks

High-performance computing can be classified in the category denoted by the white area. High-throughput computing [13] can be classified as a subset of the category denoted by the yellow area. Many-Task Computing [9] can be classified in the categories denoted by the yellow and green areas. This paper focuses on techniques to enable the support of many-task computing, including data-intensive many-task computing.

Clusters and Grids [14, 15] have been the preferred platform for loosely coupled applications that have been traditionally part of the high throughput computing class of applications, which are managed and executed through workflow systems or parallel programming systems. Various properties of a new emerging applications, such as large number of tasks (i.e. millions or more), relatively short per task execution times (i.e. seconds to minutes long), and data intensive tasks (i.e. tens of MB of I/O per CPU second of compute) have led to the definition of a new class of applications called Many-Task Computing [9]. MTC emphasizes on using larger number of computing resources over short periods of time to accomplish many computational tasks, where the primary metrics are in seconds (e.g., FLOPS, tasks/sec, IO/sec), while HTC requires large amounts of computing for long periods of time with the primary metrics being operations per month [13]. MTC applications are composed of many tasks (both independent and dependent) that can be individually scheduled on many computing resources across multiple administrative boundaries to achieve some larger application goal.

MTC denotes high-performance computations comprising multiple distinct activities, coupled via file system operations. Tasks may be small or large, uniprocessor or multiprocessor, compute-intensive or data-intensive. The set of tasks may be static or dynamic, homogeneous or heterogeneous, loosely coupled or tightly coupled. The aggregate number of tasks, quantity of computing, and volumes of data may be extremely large. The new term MTC draws attention to the many computations that are heterogeneous but not "happily" parallel.

Within the science domain, the data that needs to be processed generally grows faster than computational resources and their speed. The scientific community is facing an imminent flood of data expected from the next generation of experiments, simulations, sensors and satellites. Scientists are now attempting calculations requiring orders of magnitude more computing and communication than was possible only a few years ago. Moreover, in many currently planned and future experiments, they are also planning to generate several orders of magnitude more data than has been collected in the entire human history [16]. Many applications in the scientific computing generally use a shared infrastructure such as TeraGrid [17] and Open Science Grid [18], where data movement relies on shared or parallel file systems. The rate of increase in the number of processors per system is

outgrowing the rate of performance increase of parallel file systems, which requires rethinking existing data management techniques. Unfortunately, this trend will continue, as advanced multi-core and many-core processors will increase the number of processor cores one to two orders of magnitude over the next decade. [4] We believe that data locality is critical to the successful and efficient use of large distributed systems for data-intensive applications [19, 20] in the face of a growing gap between compute power and storage performance. Large scale data management needs to be a primary objective for any MTC-enabled middleware, to ensure data movement is minimized by intelligent data-aware scheduling.

Over the past year and a half, Falkon [21, 7] has seen wide deployment and usage across a variety of systems, from the TeraGrid [17], the SiCortex [22], the IBM Blue Gene/P [23], and the Sun Constellation [17]. Figure 2 shows plot of Falkon across these various systems from December 2007 – April 2009. Each blue dot represents a 60 second average of allocated processors, and the black line denotes the number of completed tasks. In summary, there were 166,305 peak concurrent processors, with 2 million CPU hours consumed and 173 million tasks for an average task execution time of 64 seconds and a standard deviation of 486 seconds. Many of the results presented here are represented in Figure 2, although some applications were run prior to the history log repository being instantiated in late 2007.

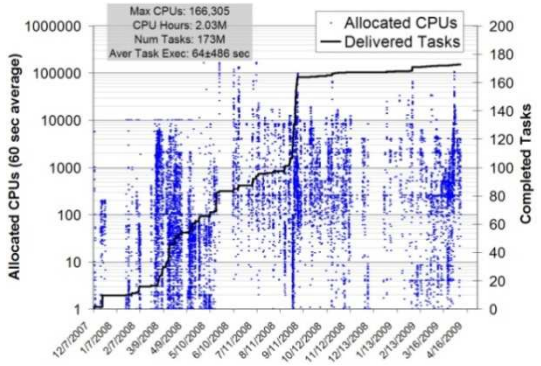


Figure 2: December 2007 – April 2009 plot of Falkon across various systems (ANL/UC TG 316 processor cluster, SiCortex 5832 processor machine, IBM Blue Gene/P 4K and 160K processor machines, and the Sun Constellation with 62K processors)

This paper is a culmination of a collection of papers [7, 9, 10, 19, 24, 25, 21, 26, 27, 28] dating back to 2006, and includes a deeper analysis of previous results as well as some new results. This paper explores the issues in building the middleware to support the many-task computing paradigm on large scale distributed systems. We have designed and implemented this middleware – Falkon – to enable the support of many-task computing on clusters, grids and supercomputers. Falkon addresses shortcomings in traditional resource management systems that support high-throughput and high-performance computing that are not efficient in supporting many-task computing. Falkon was designed to enable the rapid and efficient execution of many tasks on large scale systems, and integrate novel data management capabilities to extend data intensive applications scalability beyond that of traditional parallel file systems.

2. Related Work

As high throughput computing (HTC) is a subset of MTC, it is worth mentioning the various efforts in enabling HTC on large scale systems. Some of these systems are Condor [29, 30], Portable Batch System (PBS) [31], Load Sharing Facility (LSF) [32], SGE [33], MapReduce [5], Hadoop [34], and BOINC [35]. Full-featured local resource managers (LRMs) such as Condor, PBS, LSF, and SGE support client specification of resource requirements, data staging, process migration, check-pointing, accounting, and daemon fault recovery. Condor and glide-ins [36] are the original tools to enable HTC, but their emphasis on robustness and recoverability limits their efficiency for MTC applications in large-scale systems. We found that relaxing some constraints (e.g. recoverability) from the middleware and encouraging the end applications to implement these constraints has enabled significant improvements in middleware performance and efficiency at large scale, between two to four orders of magnitude better performance.

Multi-level scheduling has been applied at the OS level [37, 38] 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 resource provisioning to clusters via their work on Condor "glide-ins" [36]. 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. [39] embed a Condor pool in a batch-scheduled cluster, while MyCluster [40] 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 [41], a reduction of about 50% in completion time. However, because they rely on heavyweight schedulers to dispatch work to the virtual cluster, the per-task dispatch time remains high, and hence the wait queue times remain significantly higher than in the ideal case due to the schedulers' inability to push work out faster.

The BOINC "volunteer computing" system [35, 42] is known to scale well to large number of compute resources, but lacks support for data intensive applications due to the nature of the wide area network deployment BOINC typically has, as well as lack of support for "black box" applications. Although the performance of BOINC is significantly better than traditional resource managers, it is still one to two orders of magnitude slower than our proposed solution, running at about 100 jobs/sec compared to up to 3000 jobs/sec in our proposed solution.

On the IBM Blue Gene supercomputer, various works [43, 44] have leveraged the HTC-mode [45] support in Cobalt [46] scheduling system. These works have aimed at integrating their solution as much as possible in Cobalt; however, it is not clear that the current implementations will be able to support the largest MTC applications at the largest scales, as their performance is still one to two orders of magnitude slower than our proposed solution. Furthermore, these works only focus on compute resource management, and ignore data management altogether.

MapReduce (including Hadoop) is typically applied to a data model consisting of name/value pairs, processed at the programming language level. Its strengths are in its ability to spread the processing of a large dataset to thousands of processors with minimal expertise in distributed systems; however it often involves the development of custom filtering scripts and does not support "black box" application execution as is commonly found in MTC or HTC applications.

Swift [6, 47, 48] and Falkon [7] have been used to execute MTC applications on clusters, multi-site Grids (e.g., Open Science Grid [18], TeraGrid [17]), specialized large machines (SiCortex [22]), and supercomputers (e.g., Blue Gene/P [2]). Swift enables scientific workflows through a data-flow-based functional parallel programming model. It is a parallel scripting tool for rapid and reliable specification, execution, and management of large-scale science and engineering workflows. The runtime system in Swift relies on the CoG Karajan [49] workflow engine for efficient scheduling and load balancing, and it integrates with the Falkon light-weight task execution dispatcher. In this paper, we will focus on Falkon, the middleware we have designed and implemented to enable MTC on a wide range of systems from the average cluster to the largest supercomputers, and will also provide some details of the Swift system.

In summary, our proposed work in light-weight task dispatching and data management offers many orders of magnitude better performance and scalability than traditional resource management techniques, and it is changing the types of applications that can efficiently execute on large distributed resources. Once the capability of light-weight task dispatching and scalable data management was available, new applications emerged that needed to run at ever increasing scales. We have achieved these improvements by narrowing the focus of the resource management by not supporting various expensive features, and by relaxing other constraints from the resource management framework effectively pushing them to the application or the clients.

3. The Falkon Framework

To address the limitations of existing resource management systems in supporting many-task computing, we adopted a "top-down" approach to building the middleware – Falkon – to support the most demanding many-task computing applications at the largest scales. Falkon integrates (1) multi-level scheduling to enable dynamic resource provisioning and minimize wait queue times, (2) a streamlined task dispatcher able to achieve order-of-magnitude higher task dispatch rates than conventional schedulers, and (3) data diffusion which performs data caching and uses a data-aware scheduler to co-locate computational and storage resources. This section will describe each of these in detail.

3.1 Architecture Overview

Falkon consists of a dispatcher, a provisioner, and zero or more executors. 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 that notifications are performed via a custom TCP-based protocol. 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. Starting with GT4.0.5, there is support for brokered notifications.

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 (or wait for notifications), retrieve results, 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 allows the service to scale effectively on newer multicore and multiprocessor systems.

The dispatcher runs within a Globus Toolkit 4 (GT4) [50] WS container, which provides authentication, message integrity, and message encryption mechanisms, via transport-level, conversation-level, or message-level security [51].

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 determines whether to create additional executors, and if so, how many, and for how long. The provisioner supports both static and dynamic provisioning. Dynamic provisioning is supported through GRAM4 [52]. Static provisioning is supported by directly interfacing with LRMs; Falkon currently supports PBS, SGE and Cobalt.

A new **executor** registers with the dispatcher. Work is then supplied as follows: the dispatcher notifies the executor when work is available; the executor requests work; the dispatcher returns the task(s); the executor executes the supplied task(s) and returns the exit code and the optional standard output/error strings; and the dispatcher acknowledges delivery.

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. Another technique that can reduce message exchanges is to *piggy-back* new task dispatches when acknowledging result delivery. [7] Using both task bundling and piggy-backing, we can reduce the average number of message exchanges per task to be close to zero, by increasing the bundle size. In practice, we find that performance degrades for bundle sizes of greater than 300 tasks.

Figure 3 shows the Falkon architecture, including both the data management and data-aware scheduler components. Individual executors manage their own caches, using local eviction policies (e.g. *LRU* [53]), and communicate changes in cache content to the dispatcher. The scheduler sends tasks to compute nodes, along with the necessary information about where to find related input data. Initially, each executor fetches needed data from remote persistent storage. Subsequent accesses to the same data results in executors fetching data from other peer executors if the data is already cached elsewhere. The current implementation runs a GridFTP server [54] at each executor, which allows other executors to read data from its cache. This scheduling information are only hints, as remote cache state can change frequently and is not guaranteed to be 100% in sync with the global index. In the event that a data item is not found at any of the known cached locations, it attempts to retrieve the item from persistent storage; if this also fails, the respective task fails. In Figure 3, the black dotted lines represent the scheduler sending the task to the compute nodes, along with the necessary information about where to find input data. The red thick solid lines represent the ability for each executor to get data from remote persistent storage. The blue thin solid lines represent the ability for each storage resource to obtain cached data from another peer executor. We assume data follows the normal pattern found in scientific computing, which is to write-once/read-many (the same assumption as HDFS makes in the Hadoop system [34]). Thus, we avoid complicated and expensive cache coherence schemes other parallel file systems enforce.

To support data-aware scheduling, we implement a centralized index within the dispatcher that records the location of every cached data object; this is similar to the centralized NameNode in Hadoop's HDFS [34]. This index is maintained loosely coherent with the contents of the executor's caches via periodic update messages generated by the executors. In addition, each executor maintains a local index to record the location of its cached data objects. We believe that this hybrid architecture provides a good balance between latency to the data and good scalability. In previous work [10, 24], we offered a deeper analysis in the difference between a centralized index and a distributed one, and under what conditions a distributed index is preferred.

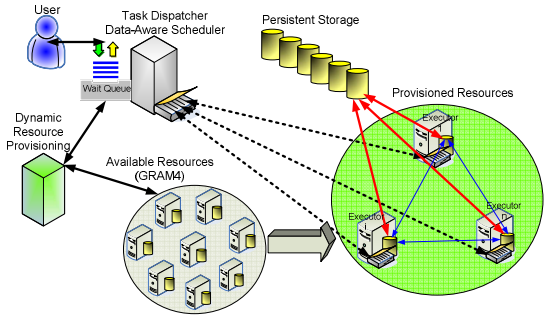


Figure 3: Architecture overview of Falkon extended with data diffusion (data management and data-aware scheduler)

We implement four dispatch policies: first-available (FA), max-cache-hit (MCH), max-compute-util (MCU), and good-cache-compute (GCC).

The FA policy ignores data location information when selecting an executor for a task; it simply chooses the first available executor, and provides the executor with no information concerning the location of data objects needed by the task. Thus, the executor must fetch all data needed by a task from persistent storage on every access. This policy is used for all experiments that do not use data diffusion.

The MCH policy uses information about data location to dispatch each task to the executor with the largest amount of data needed by that task. If that executor is busy, task dispatch is delayed until the executor becomes available. This strategy is expected to reduce data movement operations compared to first-cache-available and max-compute-util, but may lead to load imbalances where processor utilization will be sub optimal, if nodes frequently join and leave.

The MCU policy leverages data location information, attempting to maximize resource utilization even at the potential higher cost of data movement. It sends a task to an available executor, preferring executors with the most needed data locally.

The GCC policy is a hybrid MCH/MCU policy. The GCC policy sets a threshold on the minimum processor utilization to decide when to use MCH or MCU. We define processor utilization to be the number of processors with active tasks divided by the total number of processors allocated. MCU used a threshold of 100%, as it tried to keep all allocated processors utilized. We find that relaxing this threshold even slightly (e.g., 90%) works well in practice as it keeps processor utilization high and it gives the scheduler flexibility to improve cache hit rates significantly when compared to MCU alone.

3.2 Distributing the Falkon Architecture

Significant engineering efforts were needed to get Falkon to work on systems such as the Blue Gene/P efficiently at large scale. In order to improve Falkon's performance and scalability, we developed alternate implementation and distributed the Falkon architecture.

Alternative Implementations: Performance depends critically on the behavior of our task dispatch mechanisms. The initial Falkon implementation was 100% Java, and made use of GT4 Java WS-Core to handle Web Services communications. [50] The Java-only implementation works well in typical Linux clusters and Grids, but the lack of Java on the Blue Gene/L, Blue Gene/P, and SiCortex prompted us to re-implement some functionality in C. Table 1 has a summary of the differences between the two implementations.

In order to keep the implementation simple that would work on these specialized systems, we used a simple TCP-based protocol (to replace the prior WS-based protocol), internally between the dispatcher and the executor. We implemented a new component called TCPCore to handle the TCP-based communication protocol. TCPCore is a component to manage a pool of threads that lives in the same JVM as the Falkon dispatcher, and uses in-memory notifications and shared objects for communication. For performance reasons, we implemented persistent TCP sockets so connections can be reused across tasks.

Table 1: Feature comparison between the Java and C Executor implementations

Description	Java	C
Robustness	high	Medium
Security	GSITransport, GSIConversation, GSIMessageLevel	None could support SSL
Communication Protocol	WS-based	TCP-based
Error Recovery	yes	Yes
Lifetime Management	yes	No
Concurrent Tasks	yes	No
Push/Pull Model	PUSH notification based	PULL
Firewall	no	yes
NAT / Private Networks	no in general yes in certain cases	yes
Persistent Sockets	no - GT4.0 yes - GT4.2	yes
Performance	Medium~High 600~3700 tasks/s	High 1700~3200 tasks/s
Scalability	High ~ 54K CPUs	Medium ~ 10K CPUs
Portability	medium	high (needs recompile)
Data Caching	yes	no

Distributed Falkon Architecture: The original Falkon architecture [7] use a single dispatcher (running on one login node) to manage many executors (running on compute nodes). The architecture of the Blue Gene/P is hierarchical, in which there are 10 login nodes, 640 I/O nodes, and 40K compute nodes. This led us to the offloading of the dispatcher from one login node (quad-core 2.5GHz PPC) to the many I/O nodes (quad-core 0.85GHz PPC); Figure 4 shows the distribution of components on different parts of the Blue Gene/P.

Experiments show that a single dispatcher, when running on modern node with 4 to 8 cores at 2GHz+ and 2GB+ of memory, can handle thousands of tasks/sec and tens of thousands of executors. However, as we ramped up our experiments to 160K processors (each executor running on one processor), the centralized design began to show its limitations. One limitation (for scalability) was the

fact that our implementation maintained persistent sockets to all executors (two sockets per executor). With the current implementation, we had trouble scaling a single dispatcher to 160K executors (320K sockets). Another motivation for distributing the dispatcher was to reduce the load on login nodes. The system administrators of the Blue Gene/P did not approve of the high system utilization (both memory and processors) of a login node for extended periods of time when we were running intense workloads.

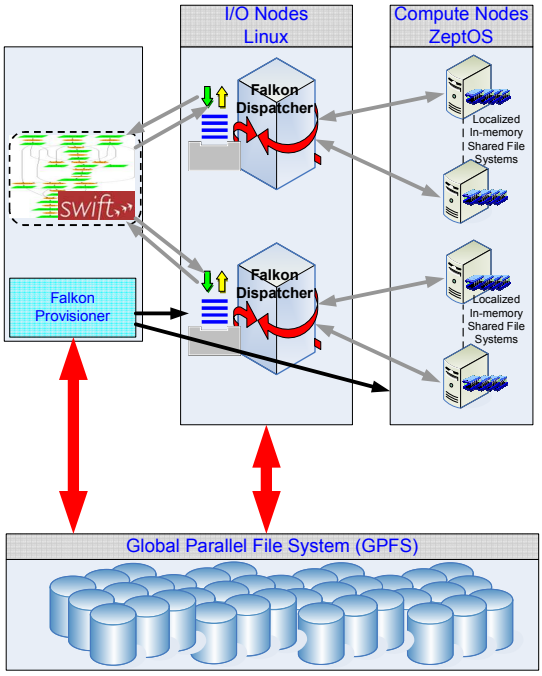


Figure 4: 3-Tier Architecture Overview

Our change in architecture from a centralized one to a distributed one allowed each dispatcher to manage a disjoint set of 256 executors, without requiring any inter-dispatcher communication. We did however had to implement additional client-side functionality to load balance task submission across many dispatchers, and to ensure that it did not overcommit tasks that could cause some dispatchers to be underutilized while others queued up tasks. Our new architecture allowed Falkon to scale to 160K processors while minimizing the load on the login nodes.

Reliability Issues at Large Scale: We discuss reliability only briefly here, to explain how our approach addresses this critical requirement. The Blue Gene/L has a mean-time-to-failure (MTBF) of 10 days [1], which can pose challenges for long-running applications. When running loosely coupled applications via Swift and Falkon, the failure of a single node only affects the task(s) that were being executed by the failed node at the time of the failure. I/O node failures only affect their respective psets (256 processors); these failures are identified by heartbeat messages or communication failures. Falkon has mechanisms to identify specific errors, and act upon them with specific actions. Most errors are generally passed back up to the application (Swift) to deal with them, but other (known) errors can be handled by Falkon directly by rescheduling the tasks. Falkon can suspend offending nodes if too many tasks fail in a short period of time. Swift maintains persistent state that allows it to restart a parallel application script from the point of failure, re-executing only uncompleted tasks. There is no need for explicit check-pointing as is the case with MPI applications; check-pointing occurs inherently with every task that completes and is communicated back to Swift.

3.3 Monitoring

In order to make visualizing the state of Falkon easier, we have formatted various Falkon logs to be printed in a specific format that can be read by the GKrellm [55] monitoring GUI to display real time state information. Figure 5 shows 1 million tasks (sleep 60) executed on 160K processors on the IBM Blue Gene/P supercomputer.

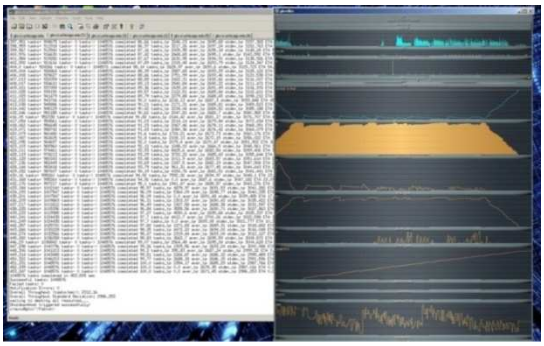


Figure 5: Monitoring via GKrellm while running 1M tasks on 160K processors

Overall, it took 453 seconds to complete 1M tasks, with an ideal time being 420 seconds, achieving 93% efficiency. To place this benchmark in context, of what an achievement it is to be able to run 1 million tasks in 7.5 minutes, others [56] have managed to run 1 million jobs in 6 months. Grant it that the 1 million jobs they referred to in [56] were real computations with real data, and not just “sleep 60” tasks, due to the large overheads of scheduling jobs through Condor [29] and other production local resource managers, running 1 million jobs, no matter how short they are, will likely still take on the order of days.

3.4 Ease of Use

The Swift parallel programming system already supported a wide variety of resource managers, such as GRAM, PBS, Condor, and others, through a concept called providers. Implementing a new provider specific for Falkon was a simple one day effort, consuming 840 lines of code. This is comparable to GRAM2 provider (850 lines), GRAM4 provider (517 lines), and the Condor provider (575 lines). For applications that are

already batch-scheduler aware, interfacing with Falkon does not pose a significant challenge. There is also a wide array of command line clients and scripts that can allow an application to interface with Falkon through loosely coupled scripts, rather than a JAVA API using web services.

4. Performance Evaluation

We use micro-benchmarks to determine performance characteristics and potential bottlenecks on systems with many cores. This section explores the dispatch performance, how it compares with other traditional LRMs, efficiency, and data diffusion effectiveness.

4.1 Falkon Task Dispatch Performance

One key component to achieving high utilization of large-scale systems is achieving high task dispatch and execution rates. In previous work [7] we reported that Falkon with a Java Executor and WS-based communication protocol achieves 487 tasks/sec in a Linux cluster (Argonne/Univ. of Chicago) with 256 CPUs, where each task was a "sleep 0" task with no I/O. We repeated the peak throughput experiment on a variety of systems (Argonne/Univ. of Chicago Linux cluster, SiCortex, and Blue Gene/P) for both versions of the executor (Java and C, WS-based and TCP-based respectively) at significantly larger scales (see Figure 6). We achieved 604 tasks/sec and 2534 tasks/sec for the Java and C Executors respectively (Linux cluster, 1 dispatcher, 200 CPUs), 3186 tasks/sec (SiCortex, 1 dispatcher, 5760 CPUs), 1758 tasks/sec (Blue Gene/P, 1 dispatcher, 4096 CPUs), and 3071 tasks/sec (Blue Gene/P, 640 dispatchers, 163840 CPUs). Note that the SiCortex and Blue Gene/P only support the C Executors. The throughput numbers that indicate "1 dispatcher" are tests done with the original centralized dispatcher running on a login node. The last throughput of 3071 tasks/sec was achieved with the dispatchers distributed over 640 I/O nodes, each managing 256 processors.

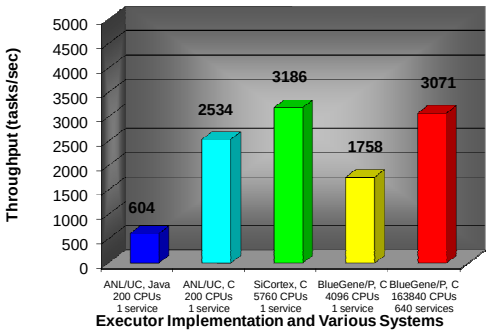


Figure 6: Task dispatch and execution throughput for trivial tasks with no I/O (sleep 0)

To better understand the performance achieved for different workloads, we measured performance as a function of task length. We made measurements in two different configurations: 1) 1 dispatcher up to 2K processors, and 2) N/256 dispatchers on up to N=160K processors, with 1 dispatcher managing 256 processors. We varied the task lengths from 1 second to 256 seconds (using sleep tasks with no I/O), and ran weak scaling workloads ranging from 2K tasks to 1M tasks (7 tasks per core).

Figure 7 investigates the effects of efficiency of 1 dispatcher running on a faster login node (quad core 2.5GHz PPC) at relatively small scales. With 4 second tasks, we can get high efficiency (95%+) across the board (up to the measured 2K processors). Figure 8 shows the efficiency with the distributed dispatchers on the slower I/O nodes (quad core 850 MHz PPC) at larger scales. It is interesting to notice that the same 4 second tasks that offered high efficiency in the single dispatcher configuration now achieves relatively poor efficiency, starting at 65% and dropping to 7% at 160K processors. This is due to both the extra costs associated with running the dispatcher on slower hardware, and the increasing need for high throughputs at large scales. If we consider the 160K processor case, based on our experiments, we need tasks to be at least 64 seconds long to get 90%+ efficiency. Adding I/O to each task will further increase the minimum task length in order to achieve high efficiency.

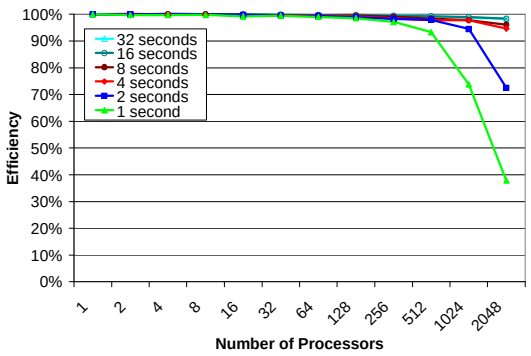


Figure 7: Efficiency graph for the Blue Gene/P for 1 to 2048 processors and task lengths from 1 to 32 seconds using a single dispatcher on a login node

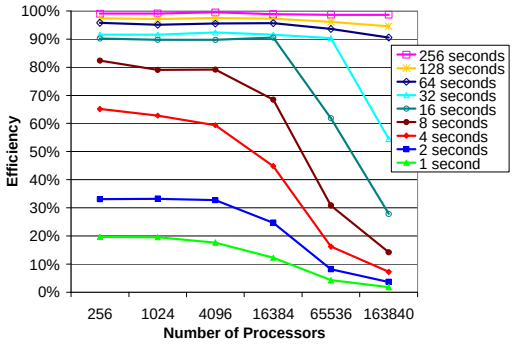


Figure 8: Efficiency graph for the Blue Gene/P for 256 to 160K processors and task lengths ranging from 1 to 256 seconds using N dispatchers with each dispatcher running on a separate I/O node

To summarize: distributing the Falkon dispatcher from a single (fast) login node to many (slow) I/O nodes has both advantages and disadvantages. The advantage is that we achieve good scalability to 160K processors, but at the cost of significantly worse efficiency at small scales (less than 4K processors) and short tasks (1 to 8 seconds). We believe both approaches are valid, depending on the application task execution distribution and scale of the application.

The experiments presented in Figure 6, Figure 7, and Figure 8 were conducted using one million tasks per run. We thought it would be worthwhile to conduct a larger scale experiment, with one billion tasks, to validate that the Falkon service can reliably run under heavy stress for prolonged periods of time. Figure 9 depicts the endurance test running one billion tasks (sleep 0) on 128 processors in a Linux cluster, which took 19.2 hours to complete. We ran the distributed version of the Falkon dispatcher using four instances on an 8-core server using bundling of 100, which allowed the aggregate throughput to be four times higher than that reported in Figure 6. Over the course of the experiment, the throughput decreased from 17K+ tasks/sec to just over 15K+ tasks/sec, with an average throughput of 15.6K tasks/sec. The loss in throughput is attributed to a memory leak in the client, which was making the free heap size smaller and smaller, and hence invoking the garbage collection more frequently. We estimated that 1.5 billion tasks would have been sufficient to exhaust the 1.5GB heap we had allocated the client, and the client would have likely failed at that point. Nevertheless, 1.5 billion tasks is larger than any application parameter space we have today, and is many orders of magnitude larger than what other systems support. The following sub-section attempts to compare and contrast the throughputs achieved between Falkon and other local resource managers.

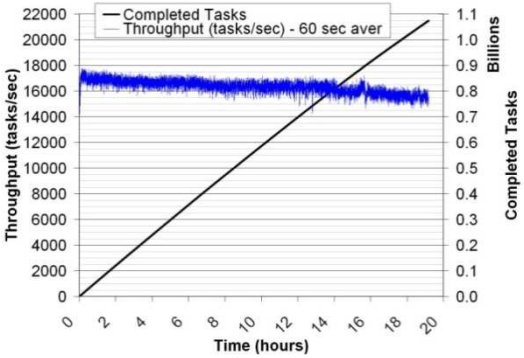


Figure 9: Endurance test with 1B tasks on 128 CPUs in ANL/UC cluster

4.2 Comparing Falkon to Other LRMs and Solutions

It is instructive to compare with task execution rates achieved by other local resource managers. In previous work [7], we measured Condor (v6.7.2, via MyCluster [40]) and PBS (v2.1.8) performance in a Linux environment (the same environment where we test Falkon and achieved 2534 tasks/sec throughputs). The throughputs we measured for PBS was 0.45 tasks/sec and for Condor was 0.49 tasks/sec; other studies in the literature have measured Condor's performance as high as 22 tasks/sec in a research prototype called Condor J2 [30].

We also tested the performance of Cobalt (the Blue Gene/P's LRM), which yielded a throughput of 0.037 tasks/sec; recall that Cobalt also lacks the support for single processor tasks, unless HTC-mode [45] is used. HTC-mode means that the termination of a process does not release the allocated resource and initiates a node reboot, after which the launcher program is used to launch the next application. There is still some management (which we implemented as part of Falkon) that needs to happen on the compute nodes, as exit codes from previous application invocations need to be persisted across reboots (e.g. to shared file system), sent back to the client, and have the ability to launch an arbitrary application from the launcher program. Running Falkon on the BlueGene/L in conjunction with Cobalt's HTC-mode support yielded a 0.29 task/sec throughput. The low throughput was attributed to the fact that nodes had to be rebooted across jobs, and node bootup was serialized in the Cobalt scheduler. We only investigated the performance of HTC-mode on the Blue Gene/L at small scales, as we realized that it will not be sufficient for MTC applications due to the high overhead of node reboots across tasks; we did not pursue it at larger scales, or on the Blue Gene/P.

Cope et al. [43] also explored a similar space as we have, leveraging HTC-mode [45] support in Cobalt on the Blue Gene/L. The authors had various experiments, which we tried to replicate for comparison reasons. The authors measured an overhead of 46.4±21.2 seconds for running 60 second tasks on 1 pset of 64 processors on the Blue Gene/L. In a similar experiment in running 64 second tasks on 1 pset of 256 processors on the Blue Gene/P, we achieve an overhead of 1.2±2.8 seconds, more than an order of magnitude better. Another comparison is the task startup time, which they measured to be on average about 25 seconds, but sometimes as high as 45 seconds; the startup times for tasks in our system are 0.8±2.7 seconds. Another comparison is average task load time by number of simultaneously submitted tasks on a single pset and executable image size of 8MB. The authors reported an average of 40~80 seconds for 32 simultaneous tasks on 32 compute nodes on the Blue Gene/L (1 pset, 64 CPUs). We measured our overheads of executing an 8MB binary to be 9.5±3.1 seconds on 64 compute nodes on the Blue Gene/P (1 pset, 256 CPUs).

Finally, Peter's et al. from IBM also recently published some performance numbers on the HTC-mode native support in Cobalt [44], which shows a similar one order of magnitude difference between HTC-mode on Blue Gene/L and our Falkon support for MTC workloads on the Blue Gene/P. For example, the authors reported a workload of 32K tasks on 8K processors and 32 dispatchers take 182.85 seconds to complete (an overhead of 5.58ms per task), but the same workload on the same number of processors using Falkon completed in 30.31 seconds with 32 dispatchers (an overhead of 0.92ms per task). Note that a similar workload of 1M tasks on 160K processors run by Falkon can be completed in as little as 368 seconds (0.35ms per task overheads).

4.3 Data Diffusion Performance

We measured the performance of the data-aware scheduler on various workloads, both with static and dynamic resource provisioning, and ran experiments on the ANL/UC TeraGrid [58] (up to 100 nodes, 200 processors). The Falkon service ran on an 8-core Xeon@2.33GHz, 2GB RAM, Java 1.5, 100Mb/s network, and 2 ms latency to the executors. The persistent storage was GPFS [59] with <1ms latency to executors.

We investigate three diverse workloads: Monotonically-Increasing (MI) and All-Pairs (AP). We use the MI workload to explore the dynamic resource provisioning support in data diffusion, and the various scheduling policies and cache sizes. We use the AP workload to compare data diffusion with active storage [60].

4.3.1 Data-Aware Scheduler Performance

In order to understand the performance of the data-aware scheduler, we developed several micro-benchmarks to test scheduler performance. We used the FA policy that performed no I/O as the baseline scheduler, and tested the various scheduling policies. We measured overall achieved throughput in terms of scheduling decisions per second and the breakdown of where time was spent inside the Falkon service. We conducted our experiments using 32 nodes; our workload consisted of 250K tasks, where each task accessed a random file (uniform distribution) from a dataset of 10K files of 1B in size each. We use files of 1 byte to measure the scheduling time and cache hit rates with minimal impact from the actual I/O performance of persistent storage and local disk. We compare the FA policy using no I/O (sleep 0), FA policy using GPFS, MCU policy, MCH policy, and GCC policy. The scheduling window size was set to 100X the number of nodes, or 3200. We also used 0.8 as the CPU utilization threshold in the GCC policy to determine when to switch between the MCH and MCU policies. Figure 10 shows the scheduler performance under different scheduling policies.

We see the throughput in terms of scheduling decisions per second range between 2981/sec (for FA without I/O) to as low as 1322/sec (for MCH). Note that for the FA policy, the cost of communication is significantly larger than the rest of the costs combined, including scheduling. The scheduling is quite inexpensive for this policy as it simply load balances across all workers. However, we see that with the data-aware policies, the scheduling costs (red and light blue areas) are significant.

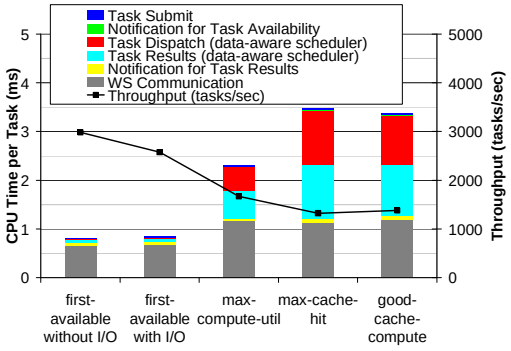


Figure 10: Data-aware scheduler performance and code profiling for the various scheduling policies

4.3.2 Monotonically Increasing Workload

We investigated the performance of the FA, MCH, MCU, and GCC policies, while also analyzing cache size effects by varying node cache size (1GB to 4GB). The MI workload has a high I/O to compute ratio (10MB:10ms). The dataset is 100GB large (10K x 10MB files). Each task reads one file chosen at random (uniform distribution) from the dataset, and computes for 10ms. The arrival rate is initially 1 task/sec and is increased by a factor of 1.3 every 60 seconds to a maximum of 1000 tasks/sec. The function varies arrival rate A from 1 to 1000 in 24 distinct intervals makes up 250K tasks and spans 1415 seconds; we chose a maximum arrival rate of 1000 tasks/sec as that was within the limits of the data-aware scheduler, and offered large aggregate I/O requirements at modest scales. This workload aims to explore a varying arrival rate under a systematic increase in task arrival rate, to explore the data-aware scheduler's ability to optimize data locality with an increasing demand.

The baseline experiment (FA policy) ran each task directly from GPFS, using dynamic resource provisioning. Aggregate throughput matches demand for arrival rates up to 59 tasks/sec, but remains flat at an average of 4.4Gb/s beyond that. The workload execution time was 5011 seconds, yielding 28% efficiency (ideal being 1415 seconds).

We ran the same workload with data diffusion with varying cache sizes per node (1GB to 4GB) using the GCC policy, optimizing cache hits while keeping processor utilization high (90%). The working set was 100GB, and with a per-node cache size of 1GB,

1.5GB, 2GB, and 4GB caches, we get aggregate cache sizes of 64GB, 96GB, 128GB, and 256GB. The 1GB and 1.5GB caches cannot fit the working set in cache, while the 2GB and 4GB cache can.

For the GCC policy with 1GB caches, throughput keeps up with demand better than the FA policy, up to 101 tasks/sec arrival rates (up from 59), at which point the throughput reached an average of 5.2Gb/s. Once the working set caching reaches a steady state, the throughput reaches 6.9Gb/s. The overall cache hit rate was 31%, resulting in a 57% higher throughput than GPFS. The workload execution time is reduced to 3762 seconds (from 5011 seconds), with 38% efficiency.

Increasing the cache size to 2GB (128GB aggregate), the aggregate throughput is close to the demand (up to the peak of 80Gb/s) for the entire experiment. We attribute this good performance to the ability to cache the entire working set and then schedule tasks to the nodes that have required data to achieve cache hit rates approaching 98%. With an execution time of 1436 seconds, efficiency was 98.5%.

Both the MCH and MCU policies performed significantly worse than GCC, due to them being too rigid and causing either unnecessary transfers over the network, or leaving processors idle. However, both MCH and MCU still managed to outperform the baseline FA policy.

Figure 11 summarizes the aggregate I/O throughput measured in each of the experiments conducted. We present in each case first, as the solid bars, the average throughput achieved from start to finish, partitioned among local cache, remote cache, and GPFS, and second, as a black line, the "peak" (actually 99th percentile) throughput achieved during the execution. The second metric is interesting because of the progressive increase in job submission rate: it may be viewed as a measure of how far a particular method can go in keeping up with user demands.

We see that the FA policy had the lowest average throughput of 4Gb/s, compared to between 5.3Gb/s and 13.9Gb/s for data diffusion (GCC, MCH, and MCU with various cache sizes), and 14.1Gb/s for the ideal case. In addition to having higher average throughputs, data diffusion also achieved significantly throughputs towards the end of the experiment (the black bar) when the arrival rates are highest, as high as 81Gb/s as opposed to 6Gb/s for the FA policy.

Note also that GPFS file system load (the red portion of the bars) is significantly lower with data diffusion than for the GPFS-only experiments (FA); in the worst case, with 1GB caches where the working set did not fit in cache, the load on GPFS is still high with 3.6Gb/s due to all the cache misses, while FA tests had 4Gb/s load. However, as the cache sizes increased and the working set fit in cache, the load on GPFS became as low as 0.4Gb/s; similarly, the network load was considerably lower, with the highest values of 1.5Gb/s for the MCU policy, and less than 1Gb/s for the other policies.

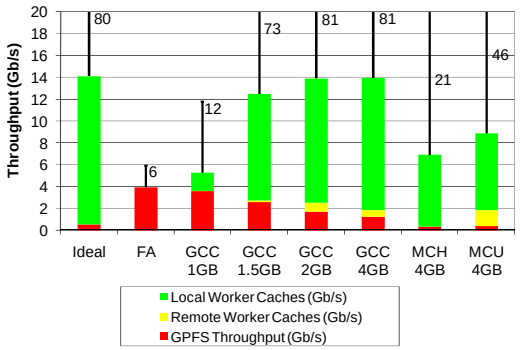


Figure 11: MI workload average and peak (99 percentile) throughput

The response time (see Figure 12) is probably one of the most important metrics interactive applications. *Average Response Time (AR_i)* is the end-to-end time from task submission to task completion notification for task *i*; $AR_i = WQ_i+TK_i+D_i$, where WQ_i is the wait queue time, TK_i is the task execution time, and D_i is the delivery time to deliver the result.

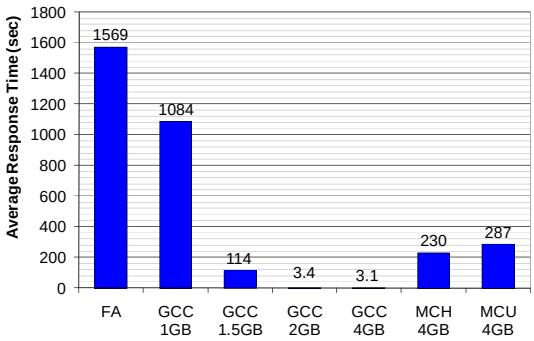


Figure 12: MI workload average response time

We see a significant different between the best data diffusion response time (3.1 seconds per task) to the worst data diffusion (1084 seconds) and the worst GPFS (1870 seconds). That is over 500X difference between the data diffusion GCC policy and the FA policy response time. A principal factor influencing the average response time is the time tasks spend in the Falkon wait queue. In the worst (FA) case, the queue length grew to over 200K tasks as the allocated resources could not keep up with the arrival rate. In contrast, the best (GCC with 4GB caches) case only queued up 7K tasks at its peak. The ability to keep the wait queue short allowed data diffusion to keep average response times low (3.1 seconds), making it a better for interactive workloads.

4.3.3 All-Pairs Workload Evaluation

In order to compare data diffusion with other related work, we implemented a common workload called All-Pairs (AP) [60]. This related work is part of the

Chirp [61] project. We call the All-Pairs use of Chirp *active storage*. Chirp has several contributions, such as delivering an implementation that behaves like a file system and maintains most of the semantics of a shared filesystem, and offers efficient distribution of datasets via a spanning tree making Chirp ideal in scenarios with a slow and high latency data source. However, Chirp does not address data-aware scheduling, so when used by All-Pairs, it typically distributes an entire application working data set to each compute node local disk prior to the application running. This requirement hinders active storage from scaling as well as data diffusion, making large working sets that do not fit on each compute node local disk difficult to handle, and producing potentially unnecessary transfers of data. Data diffusion only transfers the minimum data needed per job.

Variations of the AP problem occur in many applications. For example when we want to understand the behavior of a new function F on sets A and B, or to learn the covariance of sets A and B on a standard inner product F. [60] The AP problem is easy to express in terms of two nested for loops over some parameter space. This regular structure also enables the optimization of its data access operations.

Thain et al [60] conducted experiments with biometrics and data mining workloads using Chirp. The most data-intensive workload was where each function executed for 1 second to compare two 12MB items, for an I/O to compute ratio of 24MB:1000ms. At the largest scale (50 nodes and 500x500 problem size), we measured an efficiency of 60% for the active storage implementation, and 3% for the demand paging (to be compared to the GPFS performance we cite). These experiments were conducted in a campus wide heterogeneous cluster with nodes at risk for suspension, network connectivity of 100Mb/s between nodes, and a shared file system rated at 100Mb/s from which the dataset needed to be transferred to the compute nodes.

Due to differences in our testing environments, a direct comparison is difficult, but we compute the best case for active storage as defined in [60], and compare the data diffusion performance against this best case. Our environment has 100 nodes (200 processors) which are dedicated for the duration of the allocation, with 1Gb/s network connectivity between nodes, and a parallel file system (GPFS) rated at 8Gb/s. For the 500x500 workload, data diffusion achieves a throughput that is 80% of the best case of all data accesses occurring to local disk (see Figure 13).

We computed the best case for active storage to be 96%, however in practice, based on the efficiency of the 50 node case from previous work [60] which achieved 60% efficiency, we believe the 100 node case would not perform significantly better than the 80% efficiency of data diffusion. Running the same workload through Falkon directly against a parallel file system achieves only 26% of the ideal throughput.

In order to push data diffusion harder, we made the workload 10X more data-intensive by reducing the compute time from 1 second to 0.1 seconds, yielding a I/O to compute ratio of 24MB:100ms. For this workload, the throughput steadily increased to about 55Gb/s as more local cache hits occurred. We found extremely few cache misses, which indicates the high data locality of the AP workload. Data diffusion achieved 75% efficiency. Active storage and data diffusion transferred similar amounts of data over the network (1536GB for active storage and 1528GB for data diffusion with 0.1 sec compute time and 1698GB with the 1 sec compute time workload) and to/from the parallel file system (12GB for active storage and 62GB and 34GB for data diffusion for the 0.1 sec and 1 sec compute time workloads respectively). The similarities in bandwidth usage manifested themselves in similar efficiencies, 75% for data diffusion and 91% for the best case active storage.

In order to explore larger scale scenarios, we emulated (ran the entire Falkon stack on 200 processors with multiple executors per processor and emulated the data transfers) an IBM Blue Gene/P. We configured the Blue Gene/P with 4096 processors, 2GB caches per node, 1Gb/s network connectivity, and a 64Gb/s parallel file system. We also increased the problem size to 1000x1000 (1M tasks), and set the I/O to compute ratios to 24MB:4sec (each processor on the Blue Gene/P is about ¼ the speed of those in our 100 node cluster). On the emulated Blue Gene/P, we achieved an efficiency of 86%. The throughputs steadily increased up to 180Gb/s (of a theoretical upper bound of 187Gb/s). It is possible that our emulation was optimistic due to a simplistic modeling of the Torus network, however it shows that the scheduler scales well to 4K processors and is able to do 870 scheduling decisions per second to complete 1M tasks in 1150 seconds. The best case active storage yielded only 35% efficiency. We justify the lower efficiency of the active storage due to the significant time that is spent to distribute the 24GB dataset to 1K nodes via the spanning tree. Active storage used 12.3TB of network bandwidth (node-to-node communication) and 24GB of parallel file system bandwidth, while data diffusion used 4.7TB of network bandwidth, and 384GB of parallel file system bandwidth.

In reality, the best case active storage would require cache sizes of at least 24GB to fit the 1000x1000 problem size, while the existing 2GB cache sizes for the Blue Gene/P would only be sufficient for an 83X83 problem. This comparison is not only emulated, but also hypothetical. Nevertheless, it is

interesting to see the significant difference in efficiency between data diffusion and active storage at this larger scale.

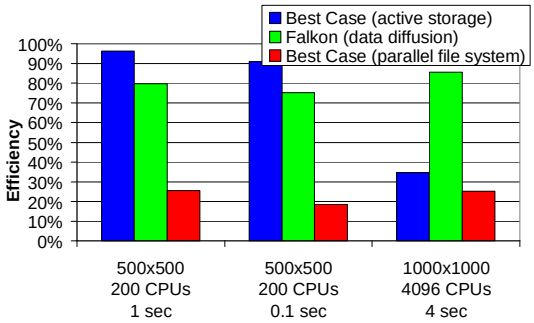


Figure 13: AP workload efficiency for 500x500 problem size on 200 processor cluster and 1000x1000 problem size on the Blue Gene/P supercomputer with 4096 processors

Our comparison between data diffusion and active storage fundamentally boils down to a comparison of pushing data versus pulling data. The active storage implementation pushes all the needed data for a workload to all nodes via a spanning tree. With data diffusion, nodes pull only the files immediately needed for a task, creating an incremental spanning forest (analogous to a spanning tree, but one that supports cycles) at runtime that has links to both the parent node and to any other arbitrary node or persistent storage. We measured data diffusion to perform comparably to active storage on our 200 processor cluster, but differences exist between the two approaches. Data diffusion is more dependent on having a well balanced persistent storage for the amount of computing power, but can scale to larger number of nodes due to the more selective nature of data distribution. Furthermore, data diffusion only needs to fit the per task working set in local caches, rather than an entire workload working set as is the case for active storage.

5. Applications

We have found many real applications that are a better fit for MTC than HTC or HPC. Their characteristics include having a large number of small parallel jobs, a common pattern in many scientific applications [6]. They also use files (instead of messages, as in MPI) for intra-processor communication, which tends to make these applications data intensive.

We have identified various loosely coupled applications from many domains as potential good candidates that have these characteristics to show examples of many-task computing applications. These applications cover a wide range of domains, from astronomy, physics, astrophysics, pharmaceuticals, bioinformatics, biometrics, neuroscience, medical imaging, chemistry, climate modeling, economics, and data analytics. They often involve many tasks, ranging from tens of thousands to billions of tasks, and have a large variance of task execution times ranging from hundreds of milliseconds to hours. Furthermore, each task is involved in multiple reads and writes to and from files, which can range in size from kilobytes to gigabytes. These characteristics made traditional resource management techniques found in HTC inefficient; also, although some of these applications could be coded as HPC applications, due to the wide variance of the arrival rate of tasks from many users, an HPC implementation would also yield poor utilization. Furthermore, the data intensive nature of these applications can quickly saturate parallel file systems at even modest computing scales.

Many of the applications presented in this section were executed via the Swift parallel programming system [6], which in turn used Falkon, although some applications are coded directly against the Falkon APIs. All these applications pose significant challenges to traditional resource management found in HPC and HTC, from both job management and storage management perspective, and are in critical need of MTC enabled middleware. This section discusses these applications in more details, and explores their performance scalability across a wide range of systems, such as clusters, grids, and supercomputers.

5.1 Functional Magnetic Resonance Imaging

We note that for each volume, each individual task in the fMRI [62] workflow required just a few seconds on an ANL_TG cluster node, so it is quite inefficient to schedule each job over GRAM and PBS, since the overhead of GRAM job submission and PBS resource allocation is large compared with the short execution time. In Figure 14 we show the execution time for different input data sizes for the fMRI workflow.

We submitted from UC_SUBMIT to ANL_TG and measured the turnaround time for the workflows. A 120-volume input (each volume consists of an image file of around 200KB and a header file of a few hundred bytes) involves 480 computations for the four stages, whereas the 480-volume input has 1960 computation tasks. The GRAM+PBS submission had low throughput although it could have potentially used all the available nodes on the site (62 nodes to be exact, as we only used the IA64 nodes). We can however bundle small jobs together using the clustering mechanism in Swift, and we show the execution time was reduced by up to 4 times (jobs were bundled into roughly 8 groups, as the grouping of jobs was a dynamic process) with GRAM and clustering, as the overhead was amortized by the bundled jobs. The Falkon execution service (with 8 worker nodes) however further cuts down the execution time by 40-70%, as each job was dispatched efficiently to the workers. We carefully chose the bundle size for the clustering

so that the clustered jobs only required 8 nodes to execute. This choice allowed us to compare GRAM/Clustering against Falkon, which used 8 nodes, fairly. We also experimented with different bundle sizes for the 120-volume run, but the overall variations for groups of 4, 6 and 10 were not significant (within 10% of the total execution time for the 8 groups).

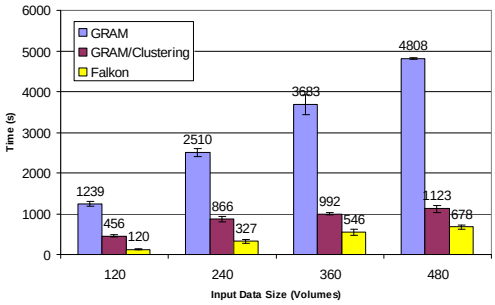


Figure 14 Execution Time for the fMRI Workflow

5.2 MolDyn (Chemistry Domain)

The goal of this molecular dynamics (MolDyn) application is to optimize and automate the computational workflow that can be used to generate the necessary parameters and other input files for calculating the solvation free energy of ligands, and can also be extended to protein-ligand binding energy. Solvation free energy is an important quantity in Computational Chemistry with a variety of applications, especially in drug discovery and design. The accurate prediction of solvation free energies of small molecules in water is still a largely unsolved problem, which is mainly due to the complex nature of the water-solute interactions. In the study, a library of 244 neutral ligands is chosen for free energy perturbation calculations. This library contains compounds with various chemical functional groups. Also, the absolute free energies of solvation for these compounds are known experimentally, and will serve as a tool to benchmark our calculations. All the structures were obtained from the NIST Chemistry WebBook database [63].

Our experiment performed a 244 molecule run, which is composed of 20497 jobs that should take less than 957.3 CPU hours to complete; in practice, it takes even less as some job executions are shared between molecules. Figure 15 shows the resource utilization in relation to Falkon queue length as the experiment progressed. We see that as resources were acquired (using the dynamic resource provisioning, starting with 0 CPUs and ending with 216 CPUs at the peak), the CPU utilization was near perfect (green means utilized, red mean idle) with the exception of the end of the experiment as the last few jobs completed (the last 43 seconds). Figure 15 shows the same information on a per task basis. The entire experiment with the exception of the last 43 seconds consumed 866.33 CPU hours and wasted 0.09 CPU hours (99.98971% efficiency); if we include the last 43 seconds as the experiment was winding down, the workflow consumed 867.1 CPU hours and it wasted 1.78 CPU hours, with a final efficiency of 99.7949013%. The experiment completed in 15091 seconds on a maximum of 216 processors, which results in a speedup of 206.9; note the average number of processors for the entire experiment was 207.26 CPUs, so the speedup of 206.9 reflects the 99.79% computed efficiency.

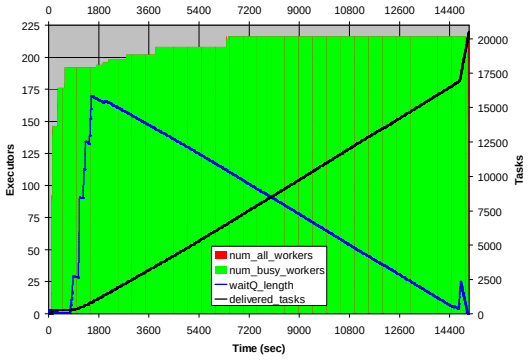


Figure 15: 244 Molecule MolDyn application; summary view showing executor's utilization in relation to the Falkon wait queue length as experiment time progressed

It is worth comparing the performance we obtained for MolDyn using Falkon with that of MolDyn over traditional GRAM/PBS. Due to reliability issues (with GRAM and PBS) when submitting 20K jobs over the course of hours, we were not able to successfullyfinish the same 244 molecule run over GRAM/PBS. We therefore tried to do some smaller experiments, in the hopes that it would increase the probability of doing a successful run. We tried several runs with 50 molecules (4201 of jobs for the 50 molecule run, instead of 20497 jobs for the 244 molecule run); the best execution times we were able to achieve for the 50 molecule runs with GRAM/PBS (on the same testbed) took 25292 seconds. We achieved a speedup of only 25.3X compared to 206.9X when using Falkon on the same workflow and the same Grid site in a similar state.

We explain this drastic difference mostly due to the typical job duration (~200 seconds) and the submission rate throttling of 1/5 jobs per second; with 200 second jobs, the most concurrent jobs we could expect was 40. Increasing the submission rate throttle resulted in GRAM/PBS gateway instability, or even causing it to stop functioning. Furthermore, each node was only using a single processor of the dual processors available on the compute nodes due to the local site PBS policy that allocates each job an entire (dual processor) machine and does not allow other jobs to run on allocated machines; it is left up to the application to fully utilize the entire machine, through multi-threading, or by invoking several different jobs to run in parallel on the same machine.

This is a great example of the benefits of having the flexibility to set queue policies per application, which is impractical to do in real-world deployed systems.

5.3 Molecular Dynamics: DOCK

The DOCK (molecular dynamics) application [64] deals with virtual screening of core metabolic targets against KEGG [65] compounds and drugs. DOCK6 addresses the problem of "docking" molecules to each other. In general, "docking" is the identification of the low-energy binding modes of a small molecule, or ligand, within the active site of a macromolecule, or receptor, whose structure is known. A compound that interacts strongly with a receptor (such as a protein molecule) associated with a disease may inhibit its function and thus act as a beneficial drug. Development of antibiotic and anticancer drugs is a process fraught with dead ends. Each dead end costs potentially millions of dollars, wasted years and lives. Computational screening of protein drug targets helps researchers prioritize targets and determine leads for drug candidates.

The goal of this project was to 1) validate our ability to approximate the binding mechanism of the protein's natural ligand (a.k.a compound that binds), 2) determine key interaction pairings of chemical functional groups from different compounds with the protein's amino acid residues, 3) study the correlation between a natural ligand that is similar to other compounds and its binding affinity with the protein's binding pocket, and 4) prioritize the proteins for further study.

Running a workload consisting of 934,803 molecules on 116K CPU cores using Falkon took 2.01 hours (see Figure 16). The per-task execution time was quite varied with a minimum of 1 second, a maximum of 5030 seconds, and a mean of 713±560 seconds. The two-hour run has a sustained utilization of 99.6% (first 5700 seconds of experiment) and an overall utilization of 78% (due to the tail end of the experiment). Note that we had allocated 128K CPUs, but only 116K CPUs registered successfully and were available for the application run; this was due to GPFS contention in bootstrapping Falkon on 32 racks, and was fixed in later large runs by moving the Falkon framework to RAM before starting, and by pre-creating log directories on GPFS to avoid lock contention. We have made dozens on runs at 32 and 40 rack scales, and we have not encountered this specific problem since.

Despite the loosely coupled nature of this application, our preliminary results show that the DOCK application performs and scales well to nearly full scale (116K of 160K CPUs). The excellent scalability (99.7% efficiency when compared to the same workload at 64K CPUs) was achieved only after careful consideration was taken to avoid the shared file system, which included the caching of the multi-megabyte application binaries, and the caching of 35MB of static input data that would have otherwise been read from the shared file system for each job. Note that each job still had some minimal read and write operations to the shared file system, but they were on the order of 10s of KB (only at the start and end of computations), with the majority of the computations being in the 100s of seconds, with an average of 713 seconds.

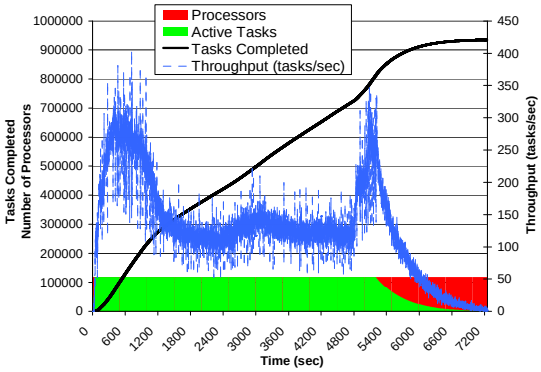


Figure 16: 934,803 DOCK5 runs on 118,784 CPU cores on Blue Gene/P

These computations are, however, just the beginning of a much larger computational pipeline that will screen millions of compounds and tens of thousands of proteins. The downstream stages use even more computationally intensive and sophisticated programs that provide for more accurate binding affinities by allowing for the protein residues to be flexible and the water molecules to be explicitly modeled. Computational screening, which is relatively inexpensive, cannot replace the wet lab assays, but can significantly reduce the number of dead ends by providing more qualified protein targets and leads. To grasp the magnitude of this application, the largest run we made of 934,803 tasks we performed represents only 0.09% of the search space (1 billion runs) being considered by the scientists we are working with; simple calculations project a search over the entire parameter space to need 20,938 CPU years, the equivalent of 48 days on the 160K-core Blue Gene/P. This is a large problem that cannot be solved in a reasonable amount of time without a supercomputer scale resource. Our loosely coupled approach holds great promise for making this problem tractable and manageable on today's largest supercomputers.

5.4 Production Runs in Drug Design

We have been working extensively with a group of researchers at the Midwest Center for Structural Genomics at Argonne National Laboratory, who have adopted Falkon and use it in their daily production runs in modeling three-dimensional protein structures towards drug design. Since proteins with similar structures tend to behave in similar ways, the team compares the modeled

structures to known proteins in order to predict their functions – a computationally intensive task.

As the Protein Data Bank expands exponentially, it becomes more difficult to coax desktop machines to do the types of analysis required. They turned to Falkon as a way to utilize their existing software applications on increasingly large machines, such as the IBM Blue Gene/P supercomputer with 160K processors. "Falkon has allowed us to ask bigger questions and perform experiments on a scale never before attempted — or even thought possible," said Andrew Binkowski, one of the main researchers involved in performing the productions runs. "This is the difference between comparing a newly determined protein structure to a family of related proteins versus comparing it to the entire protein universe." The team has done all of this using existing software packages that were not designed for high-throughput computing or many-task computing, and used Falkon to coordinate and drive the execution of many loosely-coupled computations that are treated as "black boxes" without any application-specific code modifications.

Over the course of 7 months (09/08 – 04/09), this group managed to run 2 million production jobs consuming 170K CPU hours with a minimum of 256 concurrent processors, an average of 8192 processors, and a maximum of 51200 concurrent processors; the average per job execution time was 310 seconds, with a standard deviation of 335 seconds.

5.5 Economic Modeling: MARS

We also evaluated MARS (Macro Analysis of Refinery Systems), an economic modeling application for petroleum refining developed by D. Hanson and J. Laitner at Argonne [66]. This modeling code performs a fast but broad-based simulation of the economic and environmental parameters of petroleum refining, covering over 20 primary & secondary refinery processes. MARS analyzes the processing stages for six grades of crude oil (from low-sulfur light to high-sulfur very-heavy and synthetic crude), as well as processes for upgrading heavy oils and oil sands. It includes eight major refinery products including gasoline, diesel and jet fuel, and evaluates ranges of product shares. It models the economic and environmental impacts of the consumption of natural gas, the production and use of hydrogen, and coal-to-liquids co-production, and seeks to provide insights into how refineries can become more efficient through the capture of waste energy.

While MARS analyzes this large number of processes and variables, it does so at a coarse level without involving intensive numerics. It consists of about 16K lines of C code, and can process many internal model execution iterations, with a range from 0.5 seconds (1 internal iteration) to hours (many thousands of internal iterations) of Blue Gene/P CPU time. Using the power of the Blue Gene/P we can perform detailed multi-variable parameter studies of the behavior of all aspects of petroleum refining covered by MARS.

As a larger and more complex test, we performed a 2D parameter sweep to explore the sensitivity of the investment required to maintain production capacity over a 4-decade span on variations in the diesel production yields from low sulfur light crude and medium sulfur heavy crude oils. This mimics one possible segment of the many complex multivariate parameter studies that become possible with ample computing power. A single MARS model execution involves an application binary of 0.5MB, static input data of 15KB, 2 floating point input variables and a single floating point output variable. The average micro-task execution time is 0.454 seconds. To scale this efficiently, we performed task-batching of 600 model runs into a single task, yielding a workload with 4KB of input and 4KB of output data, and an average execution time of 271 seconds.

We executed a workload with 600 million model runs (1M tasks) on 128K processors on the Blue Gene/P (see Figure 17). The experiment consumed 9.3 CPU years and took 2483 seconds to complete. Even at this large scale, the per task execution times were quite deterministic with an average of 280±10 seconds; this means that most processors would start and stop executing tasks at about the same time, which produces the peaks in task completion rates (blue line) that are as high as 4000 tasks/sec. As a comparison, a 1 processor experiment using a small part of the same workload had an average of 271±0.3 seconds; this yielded an efficiency of 97% with a speedup of 126,892 (ideal speedup being 130,816).

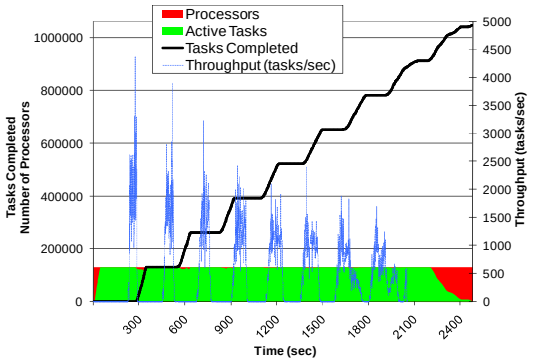


Figure 17: MARS application (summary view) on the Blue Gene/P; 1M tasks using 128K processor cores

5.6 Large-scale Astronomy Application Evaluation

We have implemented the AstroPortal [67, 68] which performs the "stacking" of image cutouts from different parts of the sky. This function can help to statistically detect objects too faint otherwise. Astronomical image collections usually

cover an area of sky several times (in different wavebands, different times, etc). On the other hand, there are large differences in the sensitivities of different observations: objects detected in one band are often too faint to be seen in another survey. In such cases we still would like to see whether these objects can be detected, even in a statistical fashion. There has been a growing interest to re-project each image to a common set of pixel planes, then stacking images. The stacking improves the signal to noise, and after coadding a large number of images, there will be a detectable signal to measure the average brightness/shape etc of these objects. While this has been done for years manually for a small number of pointing fields, performing this task on wide areas of sky in a systematic way has not yet been done. It is also expected that the detection of much fainter sources (e.g., unusual objects such as transients) can be obtained from stacked images than can be detected in any individual image.

Astronomical surveys produce terabytes of data, and contain millions of objects. For example, the SDSS DR5 dataset has 320M objects in 9TB of images [69]. To construct realistic workloads, we identified the interesting objects (for a quasar search) from SDSS DR5. The working set we constructed consisted of 771,725 objects in 558,500 files, where each file was either 2MB compressed or 6MB uncompressed, resulting in a total of 1.1TB compressed and 3.35TB uncompressed. From this working set, various workloads were defined, with certain data locality characteristics, varying from the lowest locality of 1 (i.e., 1-1 mapping between objects and files) to the highest locality of 30 (i.e., each file contained an average of 30 objects).

The AstroPortal was tested on the ANL/UC TeraGrid site, with up to 128 processors. The experiments investigate the performance and scalability of the stacking code in four configurations: 1) Data Diffusion (GZ), 2) Data Diffusion (FIT), 3) GPFS (GZ), and 4) GPFS (FIT). At the start of each experiment, all data is present only on the persistent storage system (GPFS). For data diffusion we use the MCU policy and cached data on local nodes. For the GPFS experiments we use the FA policy and perform no caching. GZ indicates that the image data is in compressed format while FIT indicates that the image data is uncompressed.

Data diffusion can make its largest impact on larger scale deployments, and hence we ran a series of experiments to capture the performance at a larger scale (128 processors) as we vary the data locality. We investigated the data-aware scheduler's ability to exploit the data locality found in the various workloads and its ability to direct tasks to computers on which needed data was cached. We found that the data-aware scheduler can get within 90% of the ideal cache hit ratios in all cases.

The following experiment (Figure 18) offers a detailed view of the performance (time per stack per processor) of the stacking application as we vary the locality. The last data point in each case represents ideal performance when running on a single node. Note that although the GPFS results show improvements as locality increases, the results are far from ideal. However, we see data diffusion gets close to the ideal as locality increases beyond 10.

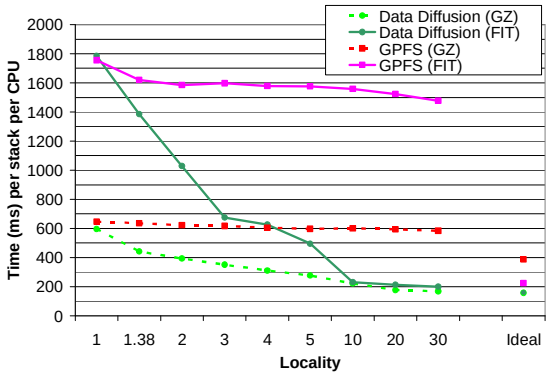


Figure 18: Performance of the stacking application using 128 CPUs for workloads with data locality ranging from 1 to 30, using data diffusion and GPFS

Using data diffusion, we achieve an aggregated I/O throughput of 39Gb/s with high data locality, a significantly higher rate than with GPFS, which tops out at 4Gb/s. These results show the decreased load on shared infrastructure (i.e., GPFS), which ultimately gives data diffusion better scalability.

5.7 Montage (Astronomy Domain)

The Montage [70] workflow demonstrated similar job execution time pattern as there were many small jobs involved. We show in Figure 19 the comparison of the workflow execution time using Swift with clustering over GRAM, Swift over Falkon, and MPI. The Montage application code we used for clustering and Falkon are the same. The code for the MPI runs is derived from the same set of source code, with the addition of data partitioning and inter-processor communication, so when multiple processors are allocated, each would process part of the input datasets, and combine the outputs if necessary. The MPI execution was well balanced across multiple processors, as the processing for each image was similar and the image sizes did not vary much. All three approaches needed to go over PBS to request for computation nodes, we used 16 nodes for Falkon and MPI, and also configured the clustering for GRAM to be around 16 groups.

The workflow had twelve stages, and we only show the parallel stages and the total execution time in the figure (the serial stages ran on a single node, and the difference of running them across the three approaches was small, so we only included them in the total time for comparison purposes). The workflow produced a 3x3 square degree mosaic around galaxy M16, where there were about 440

input images (2MB each), and 2,200 overlappings between them. There were two *mAdd* stages because we divided the region into subsets, co-added images in each subset, and then co-added the subsets together into a final mosaic. We can observe that the Falkon execution service performed close to the MPI execution, which indicated that jobs were dispatched efficiently to the 16 workers. The GRAM execution with clustering enabled still did not perform as well as the other two, mainly due to PBS queuing overhead. It is worth noting that the last stage *mAdd* was parallelized in the MPI version, but not for the version for GRAM or Falkon, and hence the big difference in execution time between Falkon and MPI, and the source of the major difference in the entire run between MPI and Falkon.

Katz et al. [71] have also created a task-graph implementation of the Montage code, using Pegasus. They did not implement quite the same application as us: for example, they ran mOverlap and mImgtbl on the portal rather than on compute nodes, and they omitted the final *mAdd* phase. Thus direct comparison with Swift over Falkon is difficult. However, if we omit the final *mAdd* phase from the comparison, Swift over Falkon is then about 5% faster than MPI, and thus also faster than the Pegasus approach, as they claimed that MPI execution time was the lower bound for them. The reasons that Swift over Falkon performs better are that MPI incurs initialization and aggregation processes, which involve multi-processor communications, for each of the parallel stages, where Falkon acquires resource at one time and then the communications in dispatching tasks from the Falkon service to workers have been kept minimum (only 2 message exchanges for each job dispatch). The Pegasus approach used Condor's glide-in mechanism, where Condor is still a heavy-weight scheduler compared with Falkon.

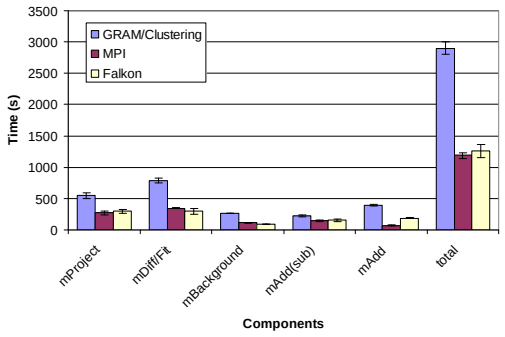


Figure 19: Execution Time for the Montage Workflow

5.8 Data Analytics: Sort and WordCount

Many programming models and frameworks have been introduced to abstract away the management details of running applications in distributed environments. MapReduce [5] is regarded as a power-leveler that solves computation problems using brutal-force resources. It provides a simple programming model and powerful runtime system for processing large datasets. The model is based on two key functions: "map" and "reduce", and the runtime system automatically partitions input data and schedules the execution of programs in a large cluster of commodity machines. MapReduce has been applied to document processing problems (e.g. distributed indexing, sorting, clustering).

Applications that can be implemented in MapReduce are a subset of those that can be implemented in Swift due to the more generic programming model found in Swift. Contrasting Swift and Hadoop are interesting as it could potentially attract new users and applications to systems which traditionally were not considered.

We compared two benchmarks, Sort and WordCount, and tested them at different scales and with different datasets. [72] The testbed consisted of a 270 processor cluster (TeraPort at UChicago). Hadoop (the MapReduce implementation from Yahoo!) was configured to use Hadoop Distributed File System (HDFS), while Swift used Global Parallel File System (GPFS). We found Swift offered comparable performance with Hadoop, a surprising finding due to the choice of benchmarks which favored the MapReduce model. In Sorting over a range of small to large files, Swift execution times were on average 38% higher when compared to Hadoop. However, for WordCount, Swift execution times were on average 75% lower.

Our experience with Swift and Hadoop indicate that the file systems (GPFS and Hadoop) are the main bottlenecks as applications scale; HDFS is more scalable than GPFS, but it still has problems with small files, and it requires applications be modified. There are current efforts in Falkon to enable Swift to operate over local disks rather than shared file systems and to cache data across jobs, which would in turn offers comparable scalability and performance to HDFS without the added requirements of modifying applications.

6. Future Work and Conclusions

Clusters with 62K processor cores (e.g., TACC Sun Constellation System, Ranger), Grids (e.g., TeraGrid with over a dozen sites and 161K processors), and supercomputers with 160K processors (e.g., IBM Blue Gene/P) are now available to the scientific community. These large HPC systems are considered efficient at executing tightly coupled parallel jobs within a particular machine using MPI to achieve inter-process communication. We proposed using HPC systems for loosely-coupled applications, which involve the execution of independent, sequential jobs that can be individually scheduled, and using files for inter-process communication.

We believe that there is more to HPC than tightly coupled MPI, and more to HTC than embarrassingly parallel long running jobs. Like HPC applications, and science itself, applications are becoming increasingly complex opening new doors for many opportunities to apply HPC in new ways if we broaden our perspective. We hope this paper leaves the broader community with a stronger appreciation of the fact that applications that are not tightly coupled MPI are not necessarily embarrassingly parallel. Some have just so many simple tasks that managing them is hard. Applications that operate on or produce large amounts of data need sophisticated data management in order to scale. There exist applications that involve many tasks, each composed of tightly coupled MPI tasks. Loosely coupled applications often have dependencies among tasks, and typically use files for inter-process communication. Efficient support for these sorts of applications on existing large scale systems, including future ones (e.g. Blue Gene/Q [73] and Blue Water supercomputers) involves substantial technical challenges and will have big impact on science.

This paper has shown good support for MTC on a variety of resources from clusters, grids, and supercomputers through the use of Swift and Falkon. Furthermore, we have taken the first steps to address data-intensive MTC by offloading much of the I/O away from parallel file systems and into the network, making full utilization of caches (both on disk and in memory) and the full network bandwidth of commodity networks (e.g. gigabit Ethernet) as well as proprietary and more exotic networks (Torus, Tree, and Infiniband).

We argue that data locality is critical to the successful and efficient use of large distributed systems for data-intensive applications, where the threshold of what constitutes a data-intensive application is lowered every year as the performance gap between processing power and storage performance widens. Large scale data management is the next major road block that must be addressed in a general way, to ensure data movement is minimized by intelligent data-aware scheduling both among distributed computing sites, and among compute nodes. Storage systems design should shift from being decoupled from the computing resources, as is commonly found in today's large-scale systems. Storage systems must be co-located among the compute resources, and make full use of all resources at their disposal, from memory, solid state storage, spinning disk, and network interconnects, giving them unprecedented high aggregate bandwidth to supply to an ever growing demand for data-intensive applications at the largest scales. We believe this shift in large-scale architecture design will lead to improving application performance and scalability for the most demanding data intensive applications as system scales continue to increase according to Moore's Law.

In future work, we will develop both the theoretical and practical aspects of building efficient and scalable support for both compute-intensive and data-intensive MTC. To achieve this, we envision building a new distributed data-aware execution fabric that scales to at least millions of processors and petabytes of storage, and will support HPC, MTC, and HTC workloads concurrently and efficiently. Clients will be able to submit computational jobs into the execution fabric by submitting to any compute node (as opposed to submitting to single point of failure gateway nodes), the fabric will guarantee that jobs will execute at least once, and that it will optimize the data movement in order to maximize processor utilization and minimize data transfer costs. The execution fabric will be elastic in which nodes will be able to join and leave dynamically, and data will be automatically replicated throughout the distributed system for both redundancy and performance. We will employ a variety of semantic for the data access patterns, from full POSIX compliance for generality, to relaxed semantics (e.g. eventual consistency on data modifications, write-once read-many data access patterns) to avoid consistency issues and increase scalability. Achieving this level of scalability and transparency will allow the data-aware execution fabric to revolutionize the types of applications that can be supported at petascale and future exascale levels.

Acknowledgements

This work was supported in part by the NASA Ames Research Center GSRP Grant Number NNA06CB89H and by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. This research was also supported in part by the National Science Foundation through TeraGrid resources provided by UC/ANL.

References

[1] A. Gara, et al. "Overview of the Blue Gene/L system architecture", IBM Journal of Research and Development 49(2/3), 2005

[2] IBM BlueGene/P, http://www.research.ibm.com/bluegene/, 2008

[3] J. Ousterhout, "Scripting: Higher Level Programming for the 21st Century", IEEE Computer, March 1998

[4] Y. Zhao, I. Raicu, I. Foster. "Scientific Workflow Systems for 21st Century e-Science, New Bottle or New Wine?" IEEE Workshop on Scientific Workflows 2008

[5] J. Dean, S. Ghemawat. "MapReduce: Simplified data processing on large clusters." USENIX OSDI04, 2004

[6] 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

[7] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, M. Wilde. "Falkon: a Fast and Light-weight tasK executiON framework", IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC07), 2007

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

[9] I. Raicu, I. Foster, Y. Zhao. "Many-Task Computing for Grids and Supercomputers", IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08) 2008

[10] I. Raicu, Y. Zhao, I. Foster, A. Szalay. "Accelerating Large-Scale Data Exploration through Data Diffusion," ACM International Workshop on Data-Aware Distributed Computing 2008

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," European Conference on Computer Systems, 2007

[12] R. Pike, S. Dorward, R. Griesemer, S. Quinlan. "Interpreting the Data: Parallel Analysis with Sawzall," Scientific Programming Journal, Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure 13(4), pp. 227-298, 2005

[13] M. Livny, J. Basney, R. Raman, T. Tannenbaum. "Mechanisms for High Throughput Computing," SPEEDUP Journal 1(1), 1997

[14] I. Foster and C. Kesselman, Eds., "The Grid: Blueprint for a Future Computing Infrastructure", "Chapter 2: Computational Grids." Morgan Kaufmann Publishers, 1999

[15] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid", International Journal of Supercomputer Applications, 2001

[16] T. Hey, A. Trefethen. "The data deluge: an e-sicence perspective", Gid Computing: Making the Global Infrastructure a Reality, 2003

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

[18] Open Science Grid (OSG), http://www.opensciencegrid.org/, 2008

[19] A. Szalay, A. Bunn, J. Gray, I. Foster, I. Raicu. "The Importance of Data Locality in Distributed Computing Applications", NSF Workflow Workshop 2006

[20] J. Gray. "Distributed Computing Economics", Technical Report MSR-TR-2003-24, Microsoft Research, Microsoft Corp., 2003

[21] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, B. Clifford. "Towards Loosely-Coupled Programming on Petascale Systems", IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SuperComputing/SC08), 2008

[22] SiCortex, http://www.sicortex.com/, 2008

[23] IBM Blue Gene team, "Overview of the IBM Blue Gene/P Project". IBM Journal of Research and Development, vol. 52, no. 1/2, pp. 199-220, Jan/Mar 2008

[24] I. Raicu, I. Foster, Y. Zhao, P. Little, C. Moretti, A. Chaudhary, D. Thain. "The Quest for Scalable Support of Data Intensive Workloads in Distributed Systems", ACM HPDC09, 2009

[25] I. Raicu, Y. Zhao, I. Foster, A. Szalay. "A Data Diffusion Approach to Large-scale Scientific Exploration," Microsoft eScience Workshop at RENCI 2007

[26] I. Raicu. "Harnessing Grid Resources with Data-Centric Task Farms", Technical Report, University of Chicago, 2007

[27] Z. Zhang, A. Espinosa, K. Iskra, I. Raicu, I. Foster, M. Wilde. "Design and Evaluation of a Collective I/O Model for Loosely-coupled Petascale Programming", IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08) 2008

[28] I. Raicu, I. Foster, Y. Zhao, A. Szalay, P. Little, C. Moretti, A. Chaudhary, D. Thain. "Towards Data Intensive Many-Task Computing", book chapter in Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management, IGI Global Publishers, 2009

[29] D. Thain, T. Tannenbaum, M. Livny, "Distributed Computing in Practice: The Condor Experience" Concurrency and Computation: Practice and Experience 17 (2-4), pp. 323-356, 2005

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

[31] 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, Linux Showcase & Conference, 2000

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

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

[34] A. Bialecki, M. Cafarella, D. Cutting, O. O’Malley. “Hadoop: A Framework for Running Applications on Large Clusters Built of Commodity Hardware,” http://lucene.apache.org/hadoop/, 2005

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

[36] J. Frey, T. Tannenbaum, I. Foster, M. Frey, S. Tuecke. “Condor-G: A Computation Management Agent for Multi-Institutional Grids,” Cluster Computing, 2002

[37] 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

[38] 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, 1999

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

[40] 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

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

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

[43] J. Cope, et al. “High Throughput Grid Computing with an IBM Blue Gene/L,” Cluster 2007

[44] A. Peters, A. King, T. Budnik, P. McCarthy, P. Michaud, M. Mundy, J. Sexton, G. Stewart. “Asynchronous Task Dispatch for High Throughput Computing for the eServer IBM Blue Gene® Supercomputer,” Parallel and Distributed Processing (IPDPS), 2008

[45] IBM Coorporation. “High-Throughput Computing (HTC) Paradigm,” IBM System Blue Gene Solution: Blue Gene/P Application Development, IBM RedBooks, 2008

[46] N. Desai. “Cobalt: An Open Source Platform for HPC System Software Research,” Edinburgh BG/L System Software Workshop, 2005

[47] Y. Zhao, I. Raicu, I. Foster, M. Hategan, V. Nefedova, M. Wilde. “Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments”, Book chapter in Grid Computing Research Progress, Nova Publisher 2008

[48] “Swift Workflow System”: www.ci.uchicago.edu/swift, 2008

[49] G.v. Laszewski, M. Hategan, D. Kodeboyina. “Java CoG Kit Workflow,” in I.J. Taylor, E. Deelman, D.B. Gannon, and M. Shields, eds., Workflows for eScience, pp. 340-356, 2007

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

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

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

[53] S. Podlipnig, L. Böszörmenyi. “A survey of Web cache replacement strategies”, ACM Computing Surveys (CSUR), Volume 35 , Issue 4, Pages: 374 – 398, 2003

[54] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, I. Foster. “The Globus Striped GridFTP Framework and Server”, ACM/IEEE SC05, 2005

[55] GKrellM. http://members.dslextreme.com/users/billw/gkrellm/gkrellm.html, 2008

[56] E. Walker, D.J. Earl, M.W. Deem. "How to Run a Million Jobs in Six Months on the NSF TeraGrid", TeraGrid Conference 2007

[57] I. Raicu, C. Dumitrescu, I. Foster. "Dynamic Resource Provisioning in Grid Environments", TeraGrid Conference 2007

[58] ANL/UC TeraGrid Site Details, http://www.uc.teragrid.org/tg-docs/tg-tech-sum.html, 2007

[59] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," FAST 2002

[60] C. Moretti, J. Bulosan, D. Thain, and P. Flynn. "All-Pairs: An Abstraction for Data-Intensive Cloud Computing", IPDPS 2008

[61] D. Thain, C. Moretti, and J. Hemmes, "Chirp: A Practical Global File system for Cluster and Grid Computing", Journal of Grid Computing, Springer 2008

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

[63] NIST Chemistry WebBook Database, http://webbook.nist.gov/chemistry/, 2008

[64] D.T. Moustakas et al. "Development and Validation of a Modular, Extensible Docking Program: DOCK 5," J. Comput. Aided Mol. Des. 20, pp. 601-619, 2006

[65] KEGG's Ligand Database: http://www.genome.ad.jp/kegg/ligand.html, 2008

[66] D. Hanson. "Enhancing Technology Representations within the Stanford Energy Modeling Forum (EMF) Climate Economic Models," Energy and Economic Policy Models: A Reexamination of Fundamentals, 2006

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

[68] 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

[69] SDSS: Sloan Digital Sky Survey, http://www.sdss.org/, 2008

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

[71] D. Katz, G. Berriman, E. Deelman, J. Good, J. Jacob, C. Kesselman, A. Laity, T. Prince, G. Singh, M. Su. A Comparison of Two Methods for Building Astronomical Image Mosaics on a Grid, Proceedings of the 7th Workshop on High Performance Scientific and Engineering Computing (HPSEC-05), 2005

[72] Q.T. Pham, A.S. Balkir, J. Tie, I. Foster, M. Wilde, I. Raicu. "Data Intensive Scalable Computing on TeraGrid: A Comparison of MapReduce and Swift", TeraGrid Conference (TG08) 2008

[73] R. Stevens. "The LLNL/ANL/IBM Collaboration to Develop BG/P and BG/Q," DOE ASCAC Report, 2006

Dr. Ioan Raicu is a NSF/CRA Computation Innovation Fellow at Northwestern University, in the Department of Electrical Engineering and Computer Science. Ioan holds a Ph.D. in Computer Science from University of Chicago under the guidance of Dr. Ian Foster. He is a 3-year award winner of the GSRP Fellowship from NASA Ames Research Center. His research work and interests are in the general area of distributed systems. His dissertation work focused on a new paradigm Many-Task Computing (MTC), which aims to bridge the gap between two predominant paradigms from distributed systems, High-Throughput Computing (HTC) and High-Performance Computing (HPC). He is particularly interested in efficient task dispatch and execution systems, resource provisioning, data management, scheduling, and performance evaluations in distributed systems. His work has been funded by the NASA Ames Research Center Graduate Student Research Program, the DOE Office of Advanced Scientific Computing Research, and most recently by the prestigious NSF/CRA CIFellows program.

Dr. Ian Foster is the Associate Division Director and a Senior Scientist in the Mathematics and Computer Science Division at Argonne National Laboratory, where he leads the Distributed Systems Laboratory, and he is an Arthur Holly Compton Professor in the Department of Computer Science at the University of Chicago. He is also involved with both the Open Grid Forum and with the Globus Alliance as an open source strategist. In 2006, he was appointed director of the Computation Institute, a joint project between the University of Chicago, and Argonne. An earlier project, Strand, received the British Computer Society Award for technical innovation. His research resulted in the development of techniques, tools and algorithms for high-performance distributed computing and parallel computing. As a result he is denoted as "the father of the Grid". Foster led research and development of software for the I-WAY wide-area distributed computing experiment, which connected supercomputers, databases and other high-end resources at 17 sites across North America in 1995. His own labs, the Distributed Systems Laboratory is the nexus of the multi-institute Globus Project, a research and development effort that encourages collaborative computing by providing advances necessary for engineering, business and other fields. Furthermore the Computation Institute addresses many of the most challenging computational and communications problems facing Grid implementations today. In 2004, he founded Univa Corporation, which was merged with United Devices in 2007 and operate under the name Univa UD. Foster's honors include the Lovelace Medal of the British Computer Society, the Gordon Bell Prize for high-performance computing (2001), as well as others. He was elected Fellow of the American Association for the Advancement of Science in 2003. Dr. Foster also serves as PI or Co-PI on projects connected to the DOE global change program, the National Computational Science Alliance, the NASA Information Power Grid project, the NSF Grid Physics Network, GRIDS Center, and International Virtual Data Grid Laboratory projects, and other DOE and NSF programs. His research is supported by DOE, NSF, NASA, Microsoft, and IBM.

Mike Wilde is a Software Architect in the Distributed Systems Laboratory in the Math and Computer Science Division at Argonne National Laboratory. He is also a Fellow at the Computation Institute at the University of Chicago. His research interests include parallel programming, parallel scripting languages, data provenance, scientific and engineering computing. He leads the Swift parallel scripting language project supported by NSF, and is deeply involved with the Falkon light-weight task execution framework.

Zhao Zhang is a PhD student in the Distributed Systems Laboratory in the Department of Computer Science at University of Chicago. His research interests are in supercomputers, grid computing, and cloud computing. His research focus is on supercomputing data management.

Dr. Kamil Iskra is an Assistant Computer Scientist in the Mathematics and Computer Science Division at Argonne National Laboratory. He has worked on various projects from Parallel Virtual File System (PVFS), ZeptoOS,

I/O Forwarding Scalability Layer (IOFSL), and Petascale I/O Characterization Tools (Darshan).

Dr. Peter Beckman is the director of the Leadership Computing Facility at the U.S. Department of Energy's Argonne National Laboratory. The Leadership Computing Facility operates the Argonne Leadership Computing Facility (ALCF), which is home to one of the world's fastest computers for open science, the Blue Gene/P, and is part of the U.S. Department of Energy's (DOE) effort to provide leadership-class computing resources to the scientific community. Beckman also leads Argonne's exascale computing strategic initiative and has previously served as the ALCF's chief architect and project director. He has worked in systems software for parallel computing, operating systems and Grid computing for 20 years. After receiving a Ph.D. degree in computer science from Indiana University in 1993, he helped create the Extreme Computing Laboratory at Indiana University. In 1997, Beckman joined the Advanced Computing Laboratory (ACL) at Los Alamos National Laboratory, where he founded the ACL's Linux cluster team and organized the Extreme Linux series of workshops and activities that helped catalyze the high-performance Linux computing cluster community. Beckman has also worked in industry, founding a research laboratory in 2000 in Santa Fe sponsored by Turbolinux Inc., which developed the world's first dynamic provisioning system for large clusters and data centers. The following year, he became vice president of Turbolinux's worldwide engineering efforts, managing development offices in the US, Japan, China, Korea and Slovenia. Beckman joined Argonne in 2002. As Director of Engineering for the TeraGrid, he designed and deployed the world's most advanced Grid system for linking production HPC computing for the National Science Foundation. After the TeraGrid became fully operational, he started research teams focusing on petascale high-performance operating systems, fault tolerance, system software and the SPRUCE urgent computing framework, which supports running critical high-performance applications at many of the nation's supercomputer centers.

Dr. Yong Zhao obtained his Ph.D. in Computer Science from The University of Chicago under Dr. Ian Foster's supervision, and was a key developer of the GriPhyN Virtual Data System (VDS), a data and workflow management system for data-intensive science collaborations. VDS plays a fundamental role in various Data Grid projects such as iVDGL (International Virtual Data Grid Laboratory), PPDG (Partical Physics Data Grid), OSG (Open Science Grid) etc. The system has been applied to scientific applications in various disciplines such as the high energy physics experiments CMS and ATLAS, the astrophysics project Sloan Digital Sky Survey, the QuarkNet science education project, and various Neuroscience and bioinformatics projects. He also designed and developed the Swift system, a programming tool for fast, scalable and reliable loosely-coupled parallel computation. He has also been actively involved in the Falkon project, a lightweight task execution framework for high throughput computing. He is now working at Microsoft on Business Intelligence projects that leverage large scale storage and computing infrastructures for Web analytics and behavior targeting.

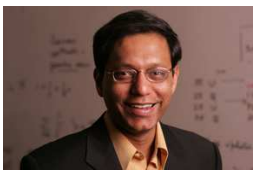
Dr. Alex Szalay is a professor in the Department of Physics and Astronomy of the Johns Hopkins University. His interests are theoretical astrophysics and galaxy formation. His research includes: Multicolor Properties of Galaxies, Galaxy Evolution, the Large Scale Power Spectrum of Fluctuations, Gravitational Lensing, and Pattern recognition and Classification Problems.

Dr. Alok Choudhary is the chair and Professor of Electrical Engineering and Computer Science Department. He is also a Professor of Marketing and Technology Industry Management at Kellogg School of Management at Northwestern University. From 1989 to 1996, he was an a faculty in the ECE department at Syracuse University. Alok Choudhary received his Ph.D. from University of Illinois, Urbana-Champaign, in Electrical and Computer Engineering, in 1989, an M.S. from University of Massachusetts, Amherst, in 1986, and B.E. (Hons.) from Birla Institue of Technology and Science, Pilani, India in 1982. Dr. Choudhary was a co-founder of Accelchip Inc. and was its Vice President for Research and Technology from 2000-2002. He received the National Science Foundation's Young Investigator Award in 1993, an IEEE Engineering Foundation award, an IBM Faculty Development award, and an Intel research council award. Choudhary has published more than 250 papers in

various journals and conferences. He has also written a book and several book chapters on the above topics. Choudhary serves on the editorial boards of IEEE Transactions on Parallel and Distributed Systems, Journal of Parallel and Distributed Systems and International Journal of High Performance Computing and Networking. He has served as a consultant to many companies and as well as on their technical advisory boards. His research interests are high-performance computing and communication systems, power aware systems, computer architecture, high-performance I/O systems and software and their applications in many domains including information processing (e.g., data mining, CRM, BI) and scientific computing (e.g., scientific discoveries). Furthermore, his interests lie in the design and evaluation of architectures and software systems (from system software such as runtime systems to compilers), high-performance servers, high-performance databases and input-output and software protection/security.

Philip Little is a PhD student in the Computer Science and Engineering at the University of Notre Dame. His research interests include online and randomized algorithms.

Christopher Moretti is a PhD candidate in the Computer Science and Engineering at the University of Notre Dame. His research interests are in distributed computing abstractions for scientific computing.



Dr. Amitabh Chaudhary is an Assistant Professor of Computer Science and Engineering at the University of Notre Dame. His research interests are in online algorithms, online learning, spatial data structures, and graph theory.

Dr. Douglas Thain is an Assistant Professor of Computer Science and Engineering at the University of Notre Dame. He directs the Cooperative Computing Lab, am enterprise that connects computer scientists with researchers in other scientific fields to solve new problems on large distributed systems. He received a PhD in Computer Sciences from the University of Wisconsin, where he contributed to the Condor distributed computing project. He received a BS in Physics from the University of Minnesota. Prof. Thain received an NSF CAREER award in 2006.