Harnessing Grid Resources to Enable the Dynamic Analysis of Large Astronomy Datasets

Ioan Raicu[1], Ian T. Foster[1,2,3]

[1]Department of Computer Science, University of Chicago, Chicago, IL, USA
[2]Computation Institute, University of Chicago, Chicago, IL, USA
[3]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA

iraicu@cs.uchicago.edu, foster@anl.gov

1 Introduction

Large datasets are being produced at a very fast pace in the astronomy domain. In principle, these datasets are most valuable if and only if they are made available to the entire community, which may have tens to thousands of members. The astronomy community will generally want to perform various analyses on these datasets to be able to extract new science and knowledge that will both justify the investment in the original acquisition of the datasets as well as provide a building block for other scientists and communities to build upon to further the general quest for knowledge.

Grid Computing has emerged as an important new field focusing on large-scale resource sharing and high-performance orientation. The Globus Toolkit, the “de facto standard” in Grid Computing, offers us much of the needed middleware infrastructure that is required to realize large scale distributed systems. We proposed to develop a collection of Web Services-based systems that use grid computing to federate large computing and storage resources for dynamic analysis of large datasets. We proposed to build a Globus Toolkit 4 based prototype named the “AstroPortal” that would support the “stacking” analysis on the Sloan Digital Sky Survey (SDSS). The stacking analysis is the summing of multiple regions of the sky, a function that can help both identify variable sources and detect faint objects. We proposed to deploy the AstroPortal on the TeraGrid distributed infrastructure and apply the stacking function to the SDSS DR5 dataset, which comprises more than 320 million objects dispersed over 1.5 million files, a total of 9 terabytes of data.

We claimed that our work with the AstroPortal would lead to interesting and innovative research work in three main areas: 1. *resource management* (efficient task dispatch, dynamic resource provisioning); 2. *data management* (data diffusion, data-aware scheduling); and 3. *applications* (performance and scalability). We have produced interesting and useful results at both the theoretical and practical level, and have even generalized our results. In generalizing our results, we have come to define a new paradigm, called Many-Task Computing (MTC) [20].

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 found in the scientific community to support many-task computing do not scale to today’s largest systems, due to 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 the middleware – Falkon [2, 4, 22] – to support the most demanding many-task computing applications at the largest scales.

Falkon, the 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 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. Micro-benchmarks have shown Falkon to achieve over 15K+ tasks/sec throughputs, scale to millions of queued tasks, execute billions of tasks per day, and scale to hundreds of thousands of processors. 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 (astronomy, physics, astrophysics, pharmaceuticals, bioinformatics, biometrics, neuroscience, medical imaging,

chemistry, climate modeling, economics, and data analytics) 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 believed impossible scales with high efficiency. We are grateful for the generous support of NASA GSRP program, which helped fund this research for three years, and helped produce 23 publications and proposals [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23].

Over the past year and a half, Falkon [2, 4] has seen wide deployment and usage across a variety of systems, from the TeraGrid [35], the SiCortex [36], the IBM Blue Gene/P [37], and the Sun Constellation [35]. Figure 1 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 1, although some applications were run prior to the history log repository being instantiated in late 2007.

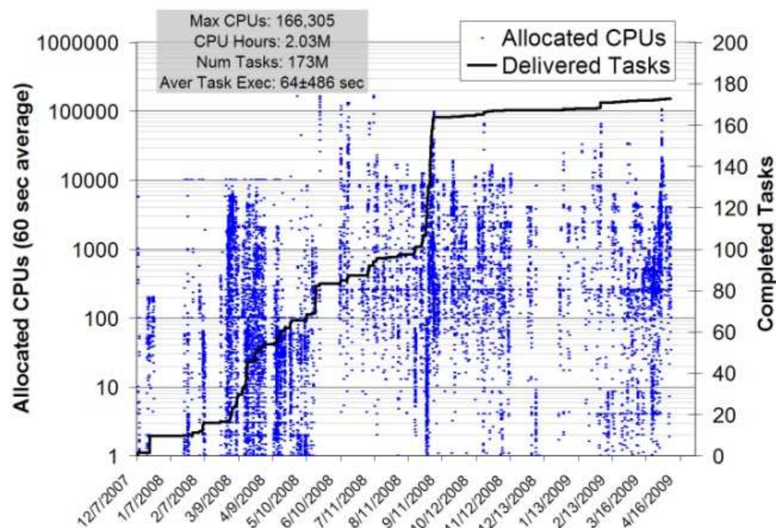

Figure 1: December 2007 – April 2009 plot of Falkon across various systems (ANL/UC TG 316 processor cluster, SiCortex 5832 processor machine, IBM Blue Gene/P 4K and 160K processor machines, and the Sun Constellation with 62K processors)

The rest of this report is organized as follows. We cover each of the three years of the fellowship, with the proposal for each year, as well as the results for that particular year. For the third year of the proposal, we went in more depth with the results, including results from other applications not specifically from astronomy, that have benefited from the research work generated by this fellowship and the AstroPortal to be exact. We finally conclude the report by outlining our contributions.

2 Year 1

The term "the Grid" denotes a distributed computing infrastructure for advanced science and engineering. Grid computing has emerged as an important new field, distinguished from conventional distributed computing by its focus on large-scale resource sharing, innovative applications, and high-performance orientation. [24]

The astronomy community has an abundance of imaging data (i.e. SDSS [57], GSC-II [60], 2MASS [61], POSS-II [62], etc) at its disposal which are essentially the "crown jewels"; however the terabytes of data makes the analysis of these datasets a very difficult process traditionally. Large astronomy datasets are generally very large (terabytes +) and contain many objects (100 million +) separated into many files (100,000+).

We propose to use grid computing as the main mechanism to enable the dynamic analysis of large astronomy datasets. There are five reasons why analyzing these large datasets is not trivial: 1. *large size of the datasets* (TB+ in size, 100M+ objects); 2. *large number of users* (1,000s); 3. *large amount of resources* needed to have adequate performance (potentially 1,000s of processors and 100s TB of rotating storage); 4. *dispersed geographic distribution of the users and resources*; and 5. *heterogeneity of the resources*.

The key question we will answer by the successful implementation of this proposal is: "*How can the analysis of large astronomy datasets be made a reality for the astronomy community using Grid resources?*" Our answer is the "AstroPortal", a science gateway to grid resources that is specifically tailored for the astronomy community.

Some of the interesting and innovative research work that will be the building blocks of the AstroPortal will be: 1. *resource provisioning* (advanced resource reservations, resource allocation, resource de-allocation, and resource migration); 2. *data management* (data location, data replication, and data caching); and 3. *distributed resource management* (coupling data and processing resources together efficiently, and distributed resource management for scalability).

The AstroPortal is a real implemented system that gives the astronomy community a new tool to advance their research and to open new doors to opportunities never before possible. At the same time, the building blocks of the AstroPortal have uncovered new approaches to resource and data management that are specifically tailored for the efficient and successful dynamic analysis of large scientific datasets.

2.1 AstroPortal Implementation & Evaluation

The AstroPortal implementation will be based on various components of the Globus Toolkit version 4 (GT4) [25], and it will be deployed in the TeraGrid [35]. Some of the GT4 components are: WS GRAM [25], MDS4 [26], RFT [25], GridFTP [27], RLS [25], DRS [25], and WS [25]. The implementation will be done in two versions focusing on different objectives: 1) AstroPortal functionality as a science gateway, along with the needed resource management support for astronomy analysis code to be efficiently run on large datasets; 2) it will focus on a distributed resource management design that will enhance scalability and performance of the AstroPortal.

The rest of Section 3 will assume the use of the SDSS DR4 [28] dataset as it will be the first supported dataset in our prototype to be deployed on the ANL/UC TeraGrid testbed.

2.1.1 AstroPortal Architecture

There are several components that make the building blocks of the AstroPortal (AP): 1) the AstroPortal Web Service (AP WS), 2) the AstroData Manager (AD), 3) the Astro Clients (AC) running on the compute nodes, and 4) the User Clients (UC). The communication between all these components would be using Web Services (WS). Furthermore, we will leverage GT4 functionality which offers persistent state storage for WS; the persistent state will make the AP WS more robust to failures as it will offer the alternative to continue execution of unfinished jobs after a system restart.

The AP WS and the AD are the two main components of the system where the resource management innovation needs to occur; furthermore, both of these components are rather generic, and with minor tuning, could be used in the analysis of other large non-astronomy related datasets. The AC and UC are specific to the astronomy community, and will offer the analysis and visualization functionality needed make the AP system useful to astronomers.

The AP WS is the centralized gateway for all UC to submit their analysis work into the grid. Once the AP WS is up through a basic bootstrapping mechanism, the AP WS would register itself with a well known MDS4 Index, so UC could dynamically find the location of the AP WS. The UC could use many existing tools offered by the SDSS / SkyServer [63] to find the location (the sky coordinates – {ra dec band}) of the interesting objects in question. The UC then packages the list of locations along with the analysis to be performed, and it is sent to the AP WS for processing as a job. Initially, the AP WS would make some advanced reservations (via GRAM) of some predefined set of resources for a predefined duration. New resources could be reserved dynamically to increase the performance of the AP under heavy loads, and resources could be de-allocated to a minimum under light loads. Upon the AP WS receiving the work from the UC, it places the list of locations in a user queue and spawns multiple threads to find (via RLS) the necessary data within the storage hierarchy. ACs use this data to perform the appropriate operation, and sends the results back to the AP WS. When the AP WS received the results from an entire job, it packages or aggregates them depending on the particular analysis performed, and sends the results back to the UC. For relatively large results, only the location of the results will be returned via WS, and the actual results will be retrievable via GridFTP for better efficiency.

The storage hierarchy is one of the key design choices that differentiates the AstroPortal from other related work. The storage hierarchy consists of 4 layers: remote data repository (REMOTE), TeraGrid GPFS (WAN), ANL/UC GPFS (LAN), and local storage (LOCAL). Each storage layer gets the data closer and closer to the computational resources making the analysis run faster. The AP WS could use RLS to maintain a coherent state between the replica location among the different layers (LOCAL, LAN, WAN, REMOTE). Ideally, as the data gradually flows in (from REMOTE, to WAN, to LAN, to LOCAL) as AC access the data, jobs would run faster over time. This would be true if the set of resources was static, however we are targeting a dynamic system which has a variable pool of resources. The REMOTE layer will offer persistent storage, the WAN GPFS and local GPFS (LAN) should

offer relatively persistent storage, but the LOCAL disk storage will be fairly dynamic, as worker resources will start-up and terminate regularly.

In order for the AP to reach that best case scenario performance, there would be a need for a worker resource to efficiently transfer its state (work queues and locally cached data) from one resource (i.e. node) to another. As the system is used, it is possible that this transferring of state take longer due to a growing local cache of data. This is high cost of transferring state is OK as long as it does not occur too often, but that means that the system will not be very dynamic and will not be able to respond to "hot-spots" of large number of queries for a short period of time without wasting resources. We believe some innovative resource migration mechanisms could help keep the LOCAL layer available for longer periods, and hence improving the likelihood that data is read from the fastest layer (LOCAL) during the analysis.

We envision that a natural evolution to the AP will be distribute the resource management decisions of the AP WS, offering a more scalable architecture! The majority of the intra-site communication remains unchanged, with the exception that the MDS4 Index need not be specifically associated with any particular site. Each AP WS from each site would register with the MDS4 Index; when users would query the MDS4 Index, users could pick a possible AP WS at random, or based on some simple metrics (i.e. AP WS load, latency from the AP WS to the user, etc) that MDS4 exposes to the users about each AP WS. The key to the enhanced performance is the ability to harness all the resources across various sites in the TG; the interaction between the various AP WS from each of the various sites is critical. Each AP WS would get some work from UC, and it would have a choice of completing it locally or forwarding the work to another site that might offer faster performance due to data locality, more available resources, faster hardware, etc.

2.1.2 Large Dataset Analysis Support

There are many different analysis/operations that the astronomy community can apply to astronomy datasets. One simple operation would be the GET operation, in which the input would be a list of locations that need to be retrieved, and the output would be the images corresponding to the input locations. The GET operation could be used if the user wanted to run some custom analysis not offered by the AP on a subset of the original dataset. Another operation could be MONTAGE, in which the input would be the coordinates to a rectangular area (4 set of coordinates), and the output would be an image that represented the entire rectangular area stitched together from smaller images. The MONTAGE operation could be useful for the visualization of the sky at different levels of detail. We will focus on the STACKING operation, in which the input would consist of a list of locations, and the output would be a single image corresponding to the stacked images. Stacking could be used to enhance faint objects that would otherwise have not been detected. In our initial prototype, we plan on supporting the GET operation and the STACKING operation. To the best of our knowledge, there is no system out there offering a STACKING like service for astronomy datasets. We do not plan to implement the MONTAGE operation since there currently exists a system (Montage [58]) that will be deployed on the TeraGrid as part of the NVO project.

2.1.3 Evaluation Methodology

We intend to thoroughly test the AstroPortal performance, scalability and robustness. Our initial evaluation will be conducted via DiPerF [29, 30, 31], a DIstributed PERformance testing Framework, aimed at simplifying and automating service performance evaluation.

The AP will be first deployed at the ANL/UC site in the TeraGrid, while the distributed AP will be deployed in the entire TG across eight different sites geographically distributed across the US. Our experiments will involve both very controlled experiments on dedicated resources within the TG, and more realistic scenario experiments with the UC running in another testbed, PlanetLab [32]. PlanetLab will offer real Internet conditions as the 500+ nodes are geographically distributed all over the world with relatively poor connectivity in comparison to the TG testbed.

2.2 Open Research Questions

We believe that there are at least three main areas with open research problems that the architecture design of the AstroPortal exposes. These areas are all in the broad context of resource management; they include: *resource provisioning, data management, and distributed resource management*.

Resource provisioning includes everything from advanced reservations, to resource allocation, to resource de-allocation issues in large scale systems. Different techniques and heuristics will apply for managing efficiently the set of resources depending on the problem we are addressing; some of the important things will be workload characteristics, number of users in total and number of concurrent users, data set size and distribution, computational intensive analysis, and I/O intensive analysis. The resource provisioning will be very important in order to achieve efficient use of existing resources, yet maintain a responsive and good performance system.

Data management: Data location, data caching, and data replication: Since one of our first two operations the AstroPortal will support is STACKING, we will focus on our motivation for the storage hierarchy described in Section 3.1, and the data management optimizations we hope to investigate. In a preliminary performance evaluation of the various data access methods, we found that there is a wide range of performance differences between the various different access methods. For example to complete 100K crops (needed for either the GET or the STACKING) on 100 nodes, the best case scenario is getting the data from the LOCAL layer which takes between at most 30 seconds. The next best performance is delivered when getting the data from the LAN layer, taking at most 200 seconds. The worst performance was the WAN layer, with times as high as 3000 seconds. Notice the difference in performance among the different layers in the storage hierarchy, which could open opportunities for good data access optimizations. There are some very interesting problems around **data management**, in which we have a very large data set that we want to break up among various sites, but also do some level of replication among the sites for improved performance. Furthermore, doing data movement based on past workloads and access patterns might prove to offer significant performance gains. We envision that the AstroData Manager will keep track of usage statistics on each object form the dataset, which will later be used to keep the most likely items in the fastest and smallest layer, optimizing the time to access the more popular data. Another significant challenge will be resource migration, in which our goal is to perform efficient state transfer among worker resources while maintaining a dynamic system.

Distributed Resource Management: The inter-site communication among the AstroPortal Web Service and its effects on the overall system performance is very interesting; work can be performed at the local site, or it could be delegated to another site that in theory could complete the work faster; the algorithms, the amount of state information, and the frequency of state information exchanges all contribute to how well and evenly the workload is spread across the various AP WS, which ultimately decides the response time that the user observes, and the aggregate throughput the entire distributed AstroPortal system can sustain. The successful implementation the distributed AP WS and the optimization of the use of both the data storage and the computational resources could lead to a scalable system supporting large numbers of concurrent users while providing very fast response times in comparison to traditional single server implementations. The use of the GT4 throughout the architecture will allow the system to interoperate with other system easily, and provide a standard method of accessing the system.

2.3 Results

We have made significant progress since our initial proposal. This section will first discuss the completed milestones, followed by the following short-term goals, the deliverables we expect to produce, and the dissemination of our results.

We initially proposed to build the AstroPortal, which would implicitly involve interesting and innovative research work in three main areas: 1) *resource provisioning, 2) data management, 3) distributed resource management.*

At this point we have developed a Web Services-based system, AstroPortal, that uses grid computing to federate large computing and storage resources for dynamic analysis of large datasets. We have deployed the AstroPortal on the TeraGrid distributed infrastructure and is now online in beta testing by our collaborator's group Alex Szalay at John Hopkins University.

As for the three main areas that we claimed to address in our work, we have implemented four basic building blocks to address them. 3DcacheGrid, Dynamic Distributed Data cache for Grid applications addresses the *data management.* CompuStore, a Data Aware Task Scheduler, addresses the *distributed resource management.* DRP, Dynamic Resource Provisioning, addresses *resource provisioning.* Finally, DeeF, Distributed execution environment Framework, is used to integrate all these three basic components into a unified execution environment that can be used to facilitate the ease of implementation of applications.

3 Year 2

3.1 Proposal

As a continuation to our Year 1 proposal, we would like to generalize our work from the AstroPortal even further beyond just the implementation of the basic components (3DcacheGrid, CompuStore, DRP, and DeeF). Although these basic building blocks should allow the implementation of many applications to be built with relatively little effort, we believe it would be valuable to define an abstract model that formally defines each basic component and its interaction with other components. This abstract model should allow us to explore the general problem space much more freely as we will break free of any application specific implementation or feature which might have influenced us when we implemented the basic building blocks and the AstroPortal.

The key observation we make is that as processing cycles become cheaper and data sets double in size every year, the main challenge for a rapid turnaround in the analysis of large datasets is the location of the data relative to the available computational resources; even with high capacity network interconnects, moving the data repeatedly to distant computational resources is becoming the bottleneck. There are large differences in data access speeds among the hierarchical storage systems normally found today in large distributed systems. Furthermore, data analysis workloads can be time varying in both their complexity and frequency, making both the computational and storage resource demands vary frequently.

Abstract Model: The analysis of large datasets normally follows a split/merge methodology, which includes an analysis query to be answered, which gets split down into independent tasks to be computed, after which the results from all the tasks are merged back into a single aggregated result. The hypothesis is that significant performance improvements can be obtained in the analysis of large dataset by leveraging information about data analysis workloads rather than individual data analysis tasks. We define workloads to be a complex query that can be decomposed into simpler tasks, or a set of queries that together answer some broader analysis questions. We believe it is feasible to allocate compute resources and caching storage resource that are relatively remote from the original data location, co-scheduled together to optimize the performance of entire data analysis workloads. Based on the split/merge methodology, we propose AMDASK, an Abstract Model for DAta-centric taSK farms, which defines the abstract model that allows us to study the stated hypothesis. Traditionally, task farms have been defined as a common parallel pattern which drives the computation of independent tasks, where a task is a self contained computation. The data-centric component of the abstract model emphasizes the central role data plays in the task farm model we are proposing, and the fact that the task farm is optimized to take advantage of data cache storage and data locality found in many large datasets and typical application workloads. Together, a data-centric task farm is defined as a common parallel pattern which drives the independent computational tasks taking into consideration the data locality in order to optimize the performance of the analysis of large datasets. This definition implies the integration of data semantics and application behavior in order to address critical challenges in the management of large scale datasets and the efficient execution of application workloads.

We intend to validate the AMDASK model through simulations. We expect the discrete event simulations to show the AMDASK model is both efficient and scalable given a wide range of simulation parameters. Once the model is validated, we will show that the current set of basic building blocks and AstroPortal application fits the model, as well as possibly implementing other applications on top of AMDASK in order to show the model's efficiency, effectiveness, scalability, and flexibility in practice.

Simulations: We will implement the AMDASK model in a discrete event simulation that will allow us to investigate a wider parameter space than we could in a real world implementation and deployment. We expect the simulations to both validate the AMDASK model and help us prove that the model is efficient and scalable given a wide range of simulation parameters (i.e. number of storage and computational resources, communication costs, management overhead, and workloads – including inter-arrival rates, query complexity, and data locality).

The simulations will specifically attempt to model a grid computing environment comprising of computational resources, storage resources, batch schedulers, various communication technologies, various types of applications, and workload models. We will perform careful and extensive empirical performance evaluations in order to create accurate input models to the simulator; the input models include 1) communication costs, 2) data management costs, 3) task scheduling costs, 4) storage access costs, and 5) workload models.

We expect to be able to scale simulations to more computational and storage resources than we could achieve in a real deployed system due to the availability of resources. Furthermore, assuming the input models to be correct, we should be able to accurately measure the end-to-end performance of various applications using a wide range of strategies for the various resource management components.

Applications: After showing that the defined basic building blocks (3DcacheGrid, CompuStore, DRP, and DeeF) and the AstroPortal fit the general abstract model, we intend to further pursue the identification and implementation of other applications to use the basic components based on the AMDASK model in order to prove both the effectiveness and the flexibility of the abstract model in practice. For each particular application, we also expect to quantify the efficiency and expected scalability based on the dataset sizes and typical workloads.

We have identified two such applications. The first is application is very similar to the "stacking" analysis and uses the same SDSS image dataset. This application is named "montage", which performs the stitching of many images in a contiguous portion of the sky to produce a single unified image. Another application we identified is from the

astro-physics domain which would utilize simulation data (as opposed to image data) from the Flash dataset. The applications that we have identified to fit the AMDASK model are volume rendering and vector visualization. The dataset is composed of 32 million files (1000 time steps times 32K files) taking up about 15TB of storage resources. The dataset contains both temporal and spatial locality, which should offer ample optimization opportunities in the data management component. More information can be found on the ASC / Alliances Center for Astrophysical Thermonuclear Flashes at their website at http://www.flash.uchicago.edu/website/home/.

3.2 Results

This section will only discuss the progress we have made on the Year 2 proposal. This section will first discuss the completed milestones, followed by the following short-term goals.

Our proposal which built upon the work on the AstroPortal and Falkon centered on two main areas: the 1) *definition of an abstract model for data-centric task farms,* and the *2) validation of the abstract model.* Our progress has been mostly in the formally defining the abstract model. A complete definition of the abstract model can be found in "Harnessing Grid Resources with Data-Centric Task Farms".

Base Definitions: A data-centric task farm has various components (i.e. computational resource where the tasks are to execute, storage resources where the data needed by the tasks is stored, etc). We formally defined 12 basic elements that are later used to derive relations regarding the model: 1) Persistent data stores, 2) Transient data stores, 3) Transient resources, 4) Data Objects, 5) Store Capacity, 6) Compute Speed, 7) Load, 8) Ideal Bandwidth, 9) Available Bandwidth, 10) Copy Time, 11) Tasks, and 12) Computational Resource State.

Execution Model: We also defined an execution model, to tie the relationships between the basic definitions defined prior. The execution model outlines the respective policies that control various parts of the execution model and how they relate to the definitions in the previous section. Each incoming task is dispatched to a transient resource, selected according to the *dispatch policy*. If a response is not received after a time determined by the *replay policy*, or a failed response is received, the task is re-dispatched according to the *dispatch policy*. A missing data object that is required by a task and does not exist on the transient data store is copied from transient or persistent data stores selected according to the *data fetch policy*. If necessary, existing data at a transient data store are discarded to make room for the new data, according to the *cache eviction policy*. Each computation is performed on the data objects found in a transient data store. When all computations are complete, the result is aggregated and returned; this aggregation of the results is assumed to be free to simplify the abstract model. Finally, we define a *resource acquisition policy* that decides when, how many, and for how long to acquire new transient computational and storage resources for. Similarly, we also define a *resource release policy* that decides when to release some acquired resources.

The Performance and Efficiency of the Abstract Model: We investigate when we can achieve good performance with this abstract model for data-centric task farms and under what assumptions. We define various costs (costs per task and average task execution time) and efficiency related metrics (efficiency, computational intensity, efficiency overheads). Furthermore, we explore the relationships between the different parameters in order to optimize efficiency.

Cost per task: We define the *cost per task* $\chi(\kappa)$ as follows: $\chi(\kappa) = \begin{cases} o(\kappa) + \mu(\kappa), & \delta \in \phi(\tau) \\ o(\kappa) + \mu(\kappa) + \zeta(\delta,\tau), & \delta \notin \phi(\tau) \end{cases}$

Average Task Execution Time: We define the *average task execution time,* B, as the summation of all the task execution times divided by the number of tasks; more formally, we have $\mathrm{B} = \frac{1}{|\mathrm{K}|}\sum_{k \in \mathrm{K}} \mu(\kappa)$.

Computational Intensity: Let A denote the *arrival rate of tasks*; we define the *computational intensity,* I, as follows: $\mathrm{I} = \mathrm{B} * \mathrm{A}$. If $\mathrm{I} = 1$, then all nodes are fully utilized; if $\mathrm{I} > 1$, tasks are arriving faster than they can be executed; finally, if $\mathrm{I} < 1$, then there are nodes that might be idle.

Workload Execution Time: We define the *workload execution time,* V, of our system as $V = \max\left(\frac{B}{|\mathrm{T}|}, \frac{1}{\mathrm{A}}\right) * |\mathrm{K}|$.

Workload Execution Time with Overhead: In general, the total execution time for a task $\kappa \in \mathrm{K}$ includes overheads, which reduced efficiency by a factor of $\frac{\mu(\kappa)}{\chi(\kappa)}$. We define the *workload execution time with overhead,* W, of our system as $W = \max\left(\frac{\mathrm{Y}}{|\mathrm{T}|}, \frac{1}{\mathrm{A}}\right) * |\mathrm{K}|$, where Y is the *average task execution time including overheads* defined as

$$Y = \begin{cases} \frac{1}{|\mathrm{K}|} \sum_{\kappa \in \mathrm{K}} [\mu(\kappa) + o(\kappa)], & \delta \in \phi(\tau), \delta \in \Omega \\ \frac{1}{|\mathrm{K}|} \sum_{\kappa \in \mathrm{K}} [\mu(\kappa) + o(\kappa) + \zeta(\delta, \tau)], & \delta \notin \phi(\tau), \delta \in \Omega \end{cases}.$$

Efficiency: We define the *efficiency,* E, of a particular workload as $\mathrm{E} = \frac{V}{W}$. The expanded version of efficiency is

$$\mathrm{E} = \frac{\max\left(\frac{B}{|\mathrm{T}|}, \frac{1}{\mathrm{A}}\right) * |\mathrm{K}|}{\max\left(\frac{\mathrm{Y}}{|\mathrm{T}|}, \frac{1}{\mathrm{A}}\right) * |\mathrm{K}|}$$

, which can be reduced to

$$E = \begin{cases} 1, & \frac{Y}{|T|} \le \frac{1}{A} \\ \max\left(\frac{B}{Y}, \frac{|\mathrm{T}|}{\mathrm{A} * Y}\right), & \frac{Y}{|T|} > \frac{1}{A} \end{cases}.$$

We claim that for the caching mechanisms to be effective in this model (i.e. the needed data objects to be found in transient data stores), the *aggregate capacity of our transient storage resources* T *is greater than our workload's working set,* Ω, (all data objects required by a sequence of tasks) *size*; formally, we can say $\sum_{\tau \in \mathrm{T}} \sigma(\tau) \ge |\Omega|$.

We also claim that we can obtain $\mathrm{E} > 0.5$ if $\mu(\kappa) > o(\kappa) + \zeta(\delta, \tau)$, where $\mu(\kappa)$, $o(\kappa)$, $\zeta(\delta, \tau)$ are the time to execute and dispatch the task $\kappa \in \mathrm{K}$, and copy the object δ to $\tau \in \mathrm{T}$, respectively.

Speedup: We define the *speedup,* S, of a particular workload as $S = E * |T|$.

Optimizing Efficiency: Having defined both efficiency and speedup, it is possible to maximize for either one, as efficiency normally monotonically decreases and speedup increases with more resources used. We can *optimize efficiency* by finding the smallest number of *transient compute/storage resources* $|\mathrm{T}|$ while we maximize speedup times efficiency.

4 Year 3

4.1 Proposal

As a continuation to our initial proposal, we would like to generalize our work further to allow a large class of applications to transparently use the mechanisms that allowed the AstroPortal to perform and scale so well. Those mechanisms have been implemented in Falkon, to support efficient task dispatch, dynamic resource provisioning, and data management through data diffusion. We plan on exploring the performance of data diffusion with more applications and workloads through the synergy we have created between Falkon and the Swift parallel programming system. We have integrated Falkon into the Karajan workflow engine, which in term is used by the Swift parallel programming system. Thus, Karajan and Swift applications can use Falkon without modification. We have already observe reductions in end-to-end run time by as much as 90% when compared to traditional approaches in which applications used batch schedulers directly by performing dynamic resource provisioning and providing applications with a lighter weight task dispatch mechanism. Swift has been applied to applications in the physical sciences, biological sciences, social sciences, humanities, computer science, and science education. We have successfully executed several applications (medical imaging, astronomy image analysis, molecular dynamics simulations) through Swift over Falkon (without data diffusion). We plan on investigating the performance benefits of data diffusion on these applications as well as others from bio-informatics, pharmaceuticals, and physics for our Year 3 Proposal. There is considerable work that needs to be done to interface the Swift system's data management capabilities to those of Falkon's data management capabilities in order for Swift applications to take advantage of the data diffusion from Falkon.

We also plan to evolve the Falkon architecture from the current 2-Tier architecture to a 3-Tier one. We are expecting that this architecture change would allow us to introduce more parallelism and distribution of the currently centralized management component in Falkon, and hence offer higher dispatch and execution rates than Falkon currently supports. We are pursuing this work with the goal to have Falkon run at considerably larger scales, such as those found on the latest IBM BlueGene/P (BG/P) that will be online in 2008 at Argonne National Laboratory. The work in porting Falkon to the BG/P will open new opportunities to applications that traditionally could not execute on the BG/P due to the lack of support of task farms. It will be crucial to test the limits of the 3-Tier architecture from a performance point of view to evaluate the appropriateness of Falkon on the BG/P which can scale to 10s of millions of processors (the current configuration will boast 128K CPU cores). Furthermore, we will also be working at simplifying the various components in Falkon, including the communication protocols that are internal to the system. We plan to implement the Executor in C (in addition to the one that is already implemented in Java), and offer a proprietary TCP-based communication protocol (as opposed to the existing Web Services protocol) between the Executors and the Dispatcher. This transition should allow Falkon to achieve higher performance due to the lighter weight communication protocol, and allow the Executor to be deployed on computer architectures that do not support Java, such as the IBM BlueGene.

4.2 Results

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, the 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 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.

4.2.1 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) [44] WS container, which provides authentication, message integrity, and message encryption mechanisms, via transport-level, conversation-level, or message-level security [45].

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 [25]. 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. [4] 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 2 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* [46]), 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 [47] 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 2, 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 [39]). 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 [39]. 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 [1, 21], 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.

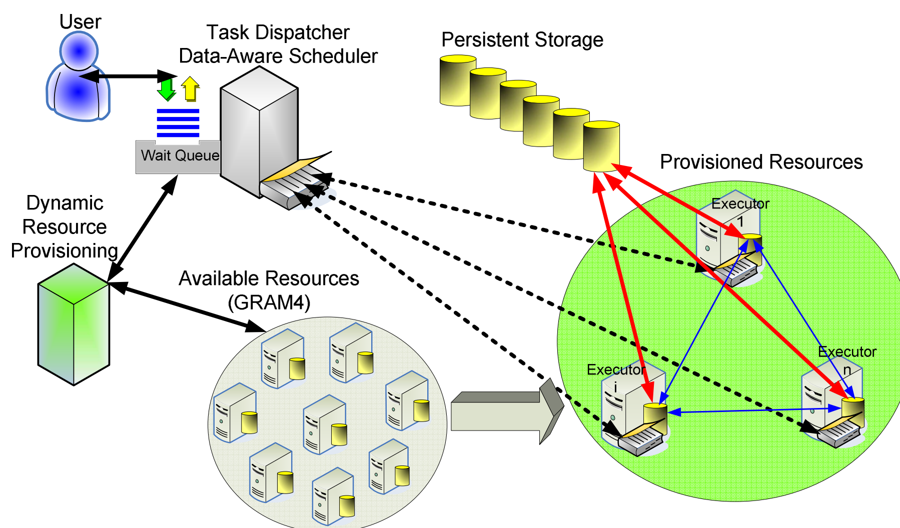

Figure 2: Architecture overview of Falkon extended with data diffusion (data management and data-aware scheduler)

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.

4.2.2 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. [44] 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.

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.

Distributed Falkon Architecture: The original Falkon architecture [4] 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 3 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.

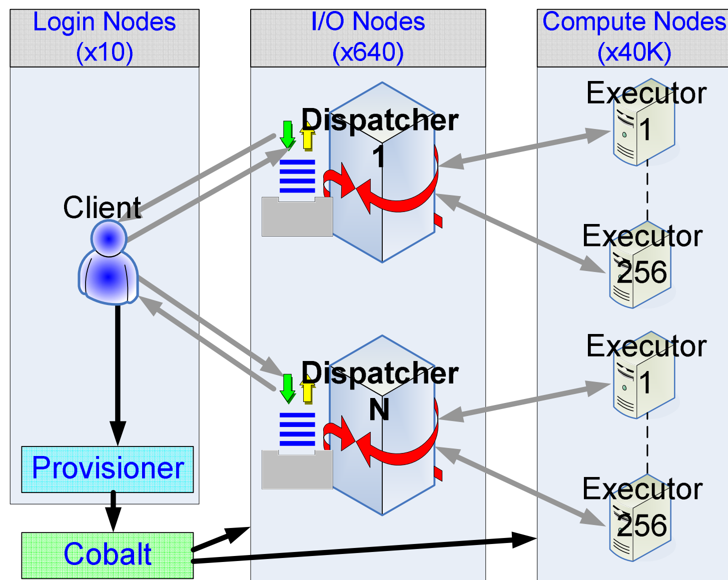

Figure 3: 3-Tier Architecture Overview

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 [33], 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.

4.3 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.

4.3.1 Falkon Task Dispatch Performance

One key component to achieving high utilization of large-scale systems is achieving high task dispatch and execute rates. In previous work [4] 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 4). 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.

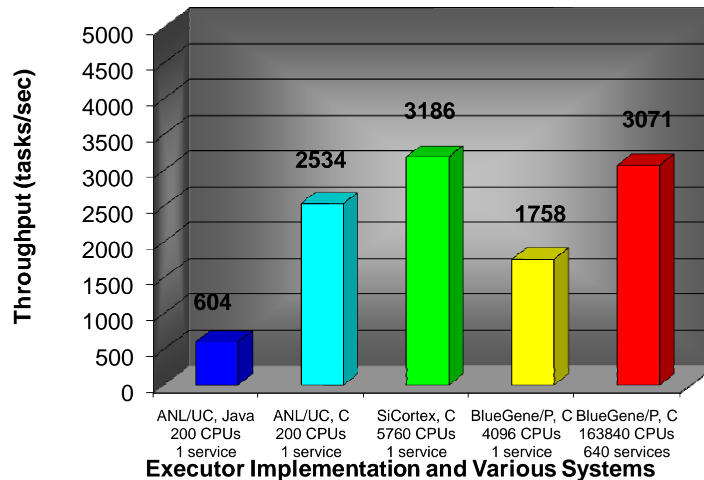

Figure 4: Task dispatch and execution throughput for trivial tasks with no I/O (sleep 0)

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 5 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 6 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.

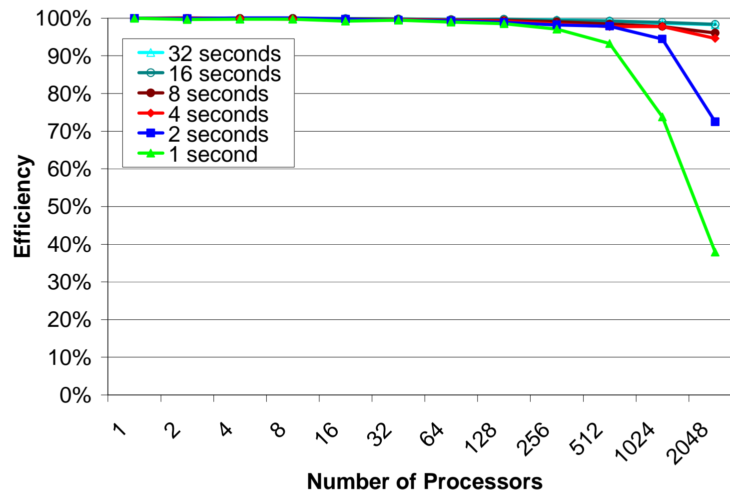

Figure 5: Efficiency graph for the Blue Gene/P for 1 to 2048 processors and task lengths from 1 to 32 seconds using a single dispatcher on a login node

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.

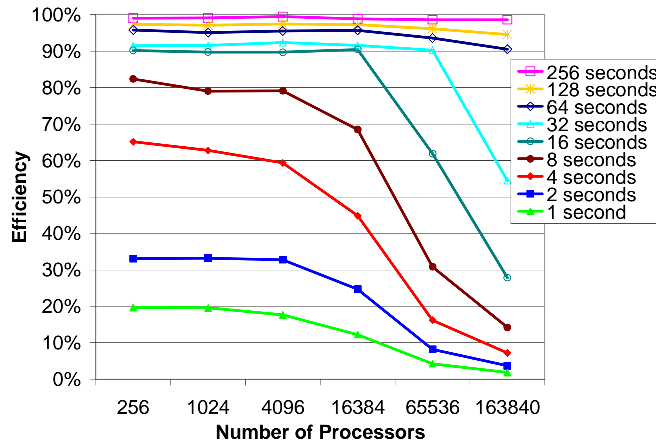

Figure 6: Efficiency graph for the Blue Gene/P for 256 to 160K processors and task lengths ranging from 1 to 256 seconds using N dispatchers with each dispatcher running on a separate I/O node

The experiments presented in Figure 4, Figure 5, and Figure 6 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 7 depicts the endurance test running one billion tasks (sleep 0) on 128 processors, 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 4. 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.

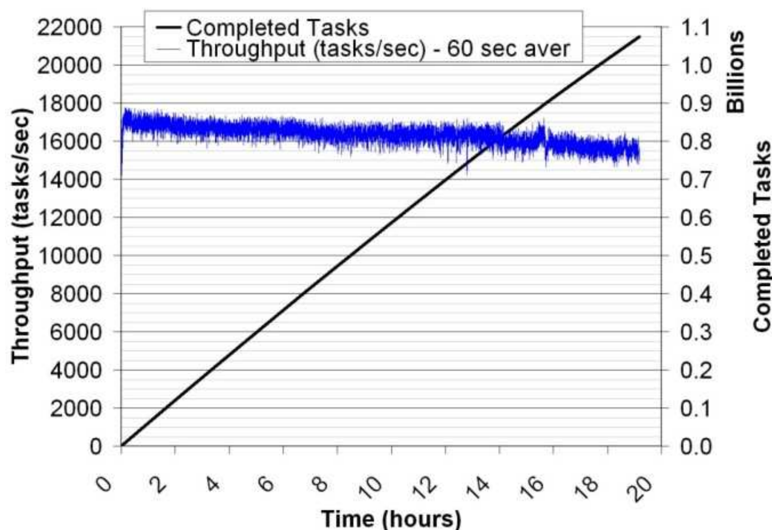

Figure 7: Endurance test with 1B tasks on 128 CPUs in ANL/UC cluster

4.3.2 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 [4], 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 [38].

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 [43] 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 in conjunction with Cobalt's HTC-mode support yielded a 0.29 task/sec throughput. 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. [41] also explored a similar space as we have, leveraging HTC-mode [43] 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 [42], 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).

4.3.3 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 [48] (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 [49] 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 [50].

4.3.3.1 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 8 shows the scheduler performance under different scheduling policies.

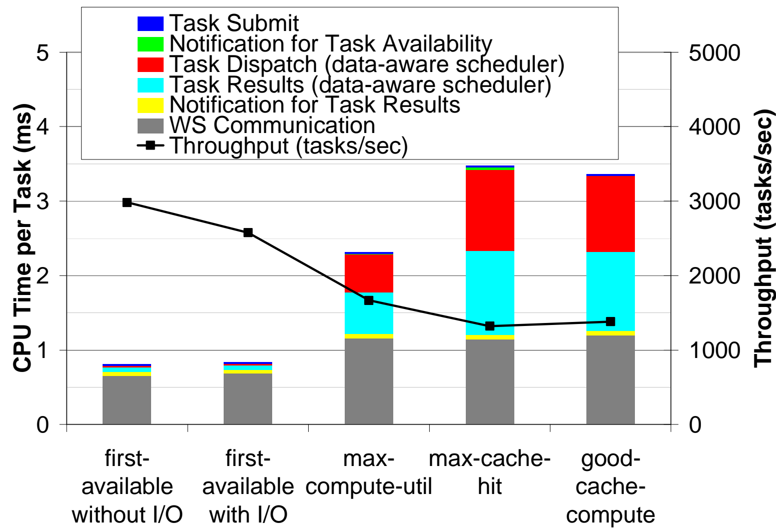

Figure 8: Data-aware scheduler performance and code profiling for the various scheduling policies

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.

4.3.3.2 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 9 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.

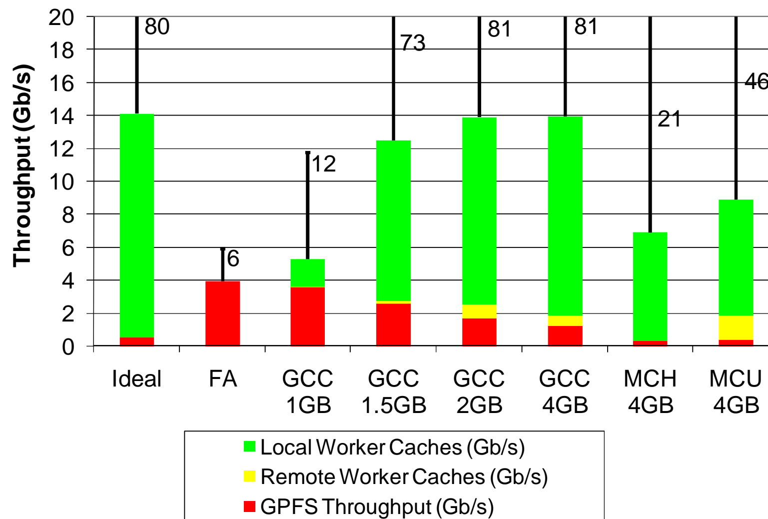

Figure 9: MI workload average and peak (99 percentile) throughput

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.

The response time (see Figure 10) 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.

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.

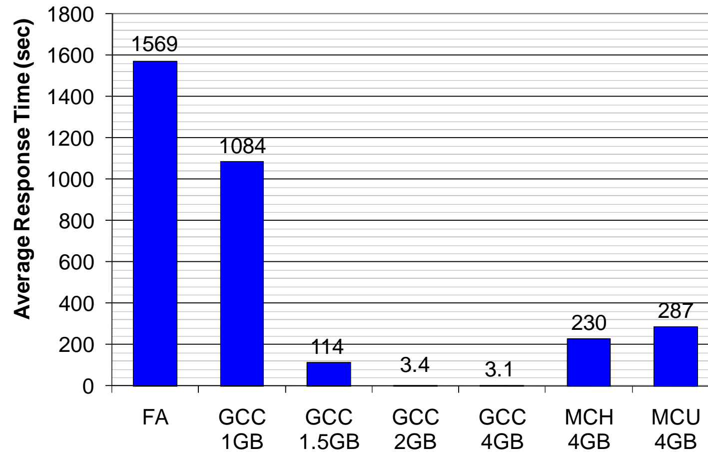

Figure 10: MI workload average response time

4.3.3.3 All-Pairs Workload Evaluation

In order to compare data diffusion with other related work, we implemented a common workload called All-Pairs (AP) [50]. This related work is part of the Chirp [51] 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. [**50**] 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 [50] 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 [50], 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 11).

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 [**50**] 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.

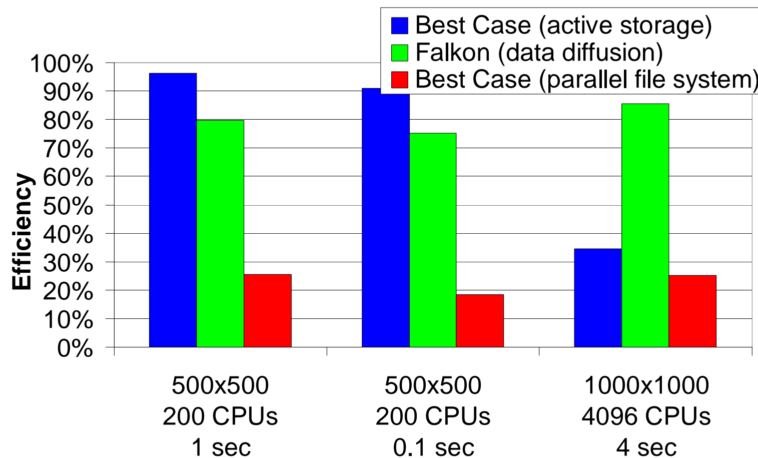

Figure 11: AP workload efficiency for 500x500 problem size on 200 processor cluster and 1000x1000 problem size on the Blue Gene/P supercomputer with 4096 processors

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.

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.

4.4 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 [13]. 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 runtime system, 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.

4.4.1 Functional Magnetic Resonance Imaging

We note that for each volume, each individual task in the fMRI [52] 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 12 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).

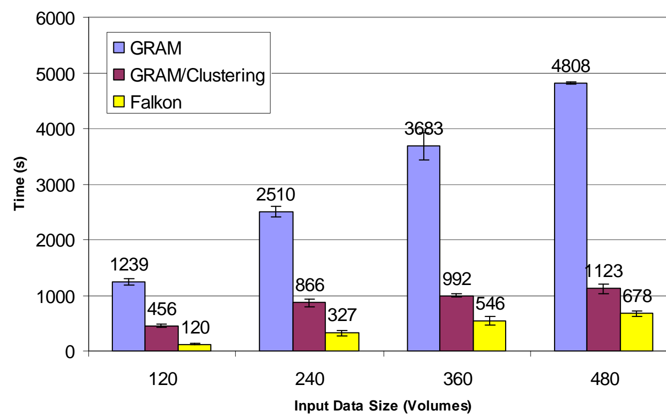

Figure 12 Execution Time for the fMRI Workflow

4.4.2 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 [53].

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 13 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 13 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.

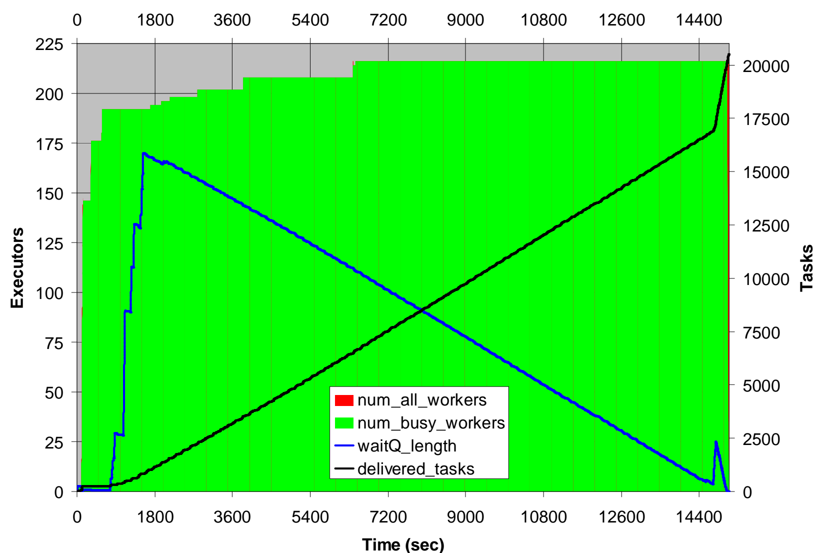

Figure 13: 244 Molecule MolDyn application; summary view showing executor's utilization in relation to the Falkon wait queue length as experiment time progressed

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 successfully run 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.

4.4.3 Molecular Dynamics: DOCK

The DOCK (molecular dynamics) application [54] deals with virtual screening of core metabolic targets against KEGG [55] 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 took 2.01 hours (see Figure 14). 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.

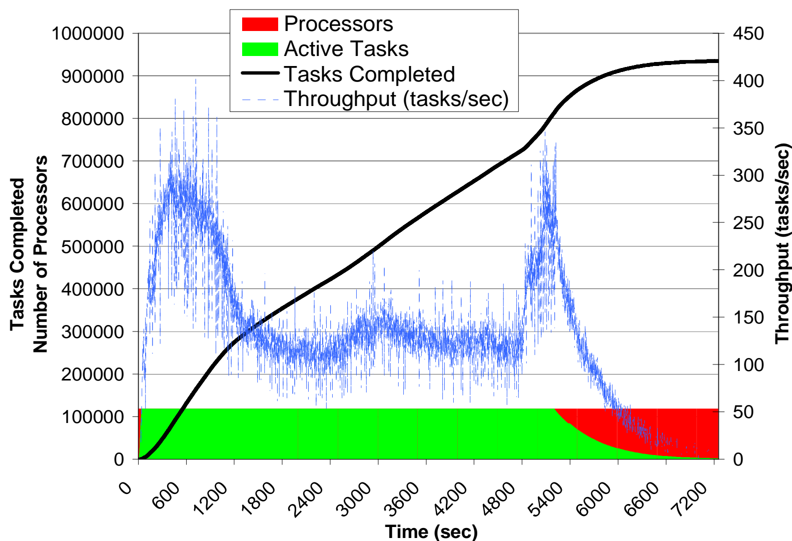

Figure 14: 934,803 DOCK5 runs on 118,784 CPU cores on Blue Gene/P

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.

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.

4.4.4 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.

4.4.5 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 [56]. 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 15). 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).

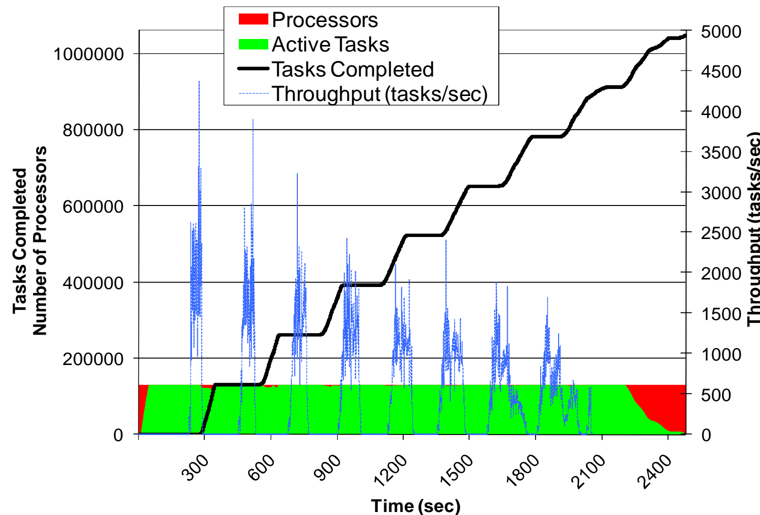

Figure 15: MARS application (summary view) on the Blue Gene/P; 1M tasks using 128K processor cores

4.4.6 Large-scale Astronomy Application Evaluation

We have implemented the AstroPortal [12, 5] 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 [57]. 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 16) 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.

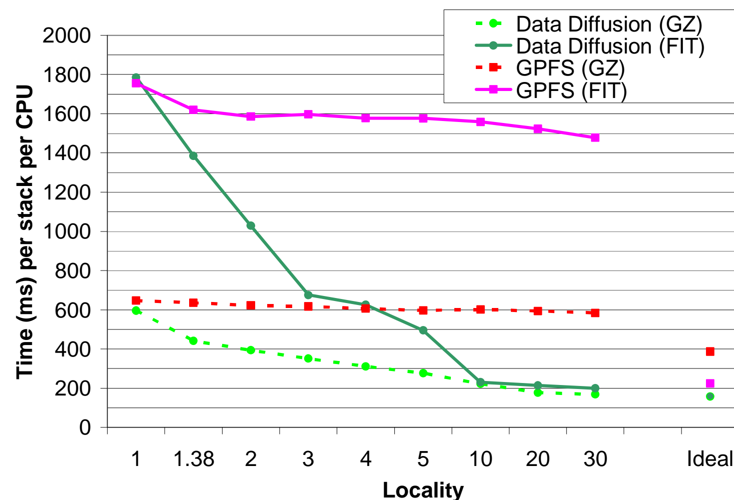

Figure 16: Performance of the stacking application using 128 CPUs for workloads with data locality ranging from 1 to 30, using data diffusion and GPFS

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.

4.4.7 Montage (Astronomy Domain)

The Montage [58] workflow demonstrated similar job execution time pattern as there were many small jobs involved. We show in Figure 17 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. [59] 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.

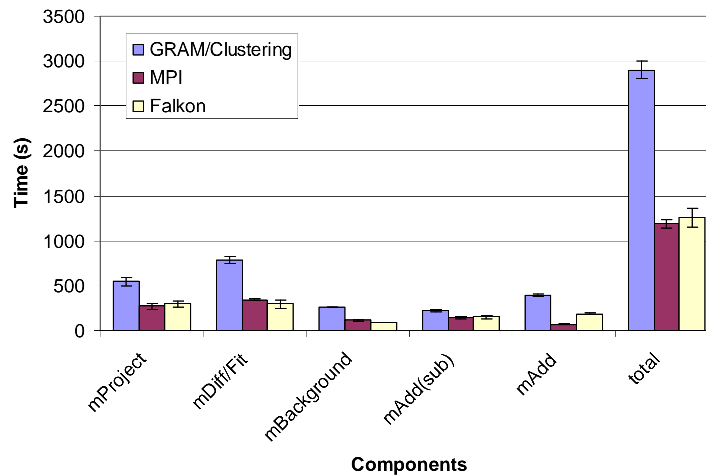

Figure 17: Execution Time for the Montage Workflow

4.4.8 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 [34] 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. [16] 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.

5 Contributions and Conclusions

We see the dynamic analysis of large datasets to be important due to the ever growing datasets that need to be accessed by larger and larger communities. Attempting to address the storage and computational problems separately (essentially forcing much data movement between computational and storage resources) will not scale to tomorrow's peta-scale datasets and will likely yield significant underutilization of the raw computational resources.

It has been argued that data intensive applications cannot be executed in grid environments because of the high costs of data movement. But if data analysis workloads have internal locality of reference, then it can be feasible to acquire and use even remote resources, as high initial data movement costs can be offset by many subsequent data analysis operations performed on that data. We envision "data diffusion" as a process in which data is stochastically moving around in the system, and that different applications can reach a dynamic equilibrium this way. One can think of a thermodynamic analogy of an optimizing strategy, in terms of energy required to move data around ("potential wells") and a "temperature" representing random external perturbations ("job submissions") and system failures. Our work proposes exactly such a stochastic optimizer.

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 have designed and implemented Falkon, a generalization of the initial prototype, the AstroPortal, to enable the rapid and efficient execution of many independent jobs on large compute clusters. Falkon combines three techniques to achieve this goal: (1) multi-level scheduling to enable dynamic resource provisioning; (2) a streamlined task dispatcher able to achieve order-of-magnitude higher task dispatch rates than conventional schedulers; and (3) performs data caching and uses a data-aware scheduler to co-locate computational and storage resources. Falkon has been deployed and tested in a wide range of environments, from 100 node clusters, to Grids (TeraGrid), to specialized machines (SiCortex with 5832 CPUs), to supercomputers (IBM BlueGene/P with 160K CPUs). Micro-benchmarks have shown Falkon to achieve over 15K+ tasks/sec throughputs, scale to millions of queued tasks, and to 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.

There are various fundamental research questions we have addressed through this work. They have centered on two main areas, data and compute resource management, and how they relate to particular workloads of data analysis on large datasets. We have explored a variety of applications from various domains, such as astronomy, astro-physics, medicine, chemistry, economics, bio-informatics, pharmaceuticals, physics, and analytics in order to show off the flexibility and effectiveness of Falkon and data diffusion on real world applications; much of the outreach in the wide range of scientific domains has been accomplished through the fantastic synergy that has been created between Falkon and the Swift parallel programming system.

Acknowledgements

This work was supported in part by the NASA Ames Research Center GSRP Grant Number NNA06CB89H and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. We also thank the NSF TeraGrid, the Computation Institute, and the Argonne National Laboratory ALCF for hosting many of the experiments reported in this dissertation.

References

[1] 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

[2] 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

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

[4] 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

[5] 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

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

[7] I. Raicu, I. Foster. "Harnessing Grid Resources to Enable the Dynamic Analysis of Large Astronomy Datasets", NASA, Ames Research Center, GSRP, February 2006

[8] I. Raicu, I. Foster. "Harnessing Grid Resources to Enable the Dynamic Analysis of Large Astronomy Datasets: Year 1 Status and Year 2 Proposal", NASA, Ames Research Center, GSRP, February 2007

[9] I. Raicu, I. Foster. "Harnessing Grid Resources to Enable the Dynamic Analysis of Large Astronomy Datasets: Year 2 Status and Year 3 Proposal", NASA, Ames Research Center, GSRP, February 2008

[10] 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

[11] Y. Zhao, I. Raicu, M. Hategan, M. Wilde, I. Foster. "Swift: Realizing Fast, Reliable, Large Scale Scientific Computation", under review

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

[13] 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

[14] I. Raicu, I. Foster. "Towards Data Intensive Many-Task Computing", under review as a book chapter in "Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management", IGI Global Publishers, 2009

[15] I. Raicu, Y. Zhao, I. Foster, M. Wilde, Z. Zhang, B. Clifford, M. Hategan, S. Kenny. "Managing and Executing Loosely Coupled Large Scale Applications on Clusters, Grids, and Supercomputers", Extended Abstract, GlobusWorld08, part of Open Source Grid and Cluster Conference 2008

[16] 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

[17] 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

[18] 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

[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] 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

[21] 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", under review at ACM HPDC09, 2009

[22] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster and M. Wilde. "Falkon: A Proposal for Project Globus Incubation", Globus Incubation Management Project, 2007

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

[24] I Foster, C Kesselman, S Tuecke, "*The Anatomy of the Grid*", International Supercomputing Applications, 2001.

[25] I Foster. "*A Globus Toolkit Primer*," 02/24/2005. http://www-unix.globus.org/toolkit/docs/development/3.9.5/key/GT4_Primer_0.6.pdf

[26] JM Schopf, I Raicu, L Pearlman, N Miller, C Kesselman, I Foster, M D'Arcy. "*Monitoring and Discovery in a Web Services Framework: Functionality and Performance of Globus Toolkit MDS4*", under review at IEEE HPDC 2006.

[27] B Allcock, J Bresnahan, R Kettimuthu, M Link, C Dumitrescu, I Raicu, I Foster. "*The Globus Striped GridFTP Framework and Server*", IEEE/ACM SC 2005.

[28] SDSS Data Release 4 (DR4), http://www.sdss.org/dr4/

[29] C Dumitrescu, I Raicu, M Ripeanu, I Foster. "*DiPerF: an automated DIstributed PERformance testing Framework*", IEEE/ACM GRID2004, Pittsburgh, PA, November 2004, pp 289 - 296

[30] I Raicu. "*A Performance Study of the Globus Toolkit® and Grid Services via DiPerF, an automated DIstributed PERformance testing Framework*", University of Chicago, Computer Science Department, MS Thesis, May 2005, Chicago, Illinois.

[31] I Raicu, C Dumitrescu, M Ripeanu, I Foster. "*The Design, Performance, and Use of DiPerF: An automated DIstributed PERformance testing Framework*", under review at Journal of Grid Computing.

[32] B Chun, D Culler, T Roscoe, A Bavier, L Peterson, M Wawrzoniak, and M Bowman, "*PlanetLab: An Overlay Testbed for Broad-Coverage Services*," ACM Computer Communications Review, vol. 33, no. 3, July 2003.

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

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

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

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

[37] 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

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

[39] 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

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

[42] 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

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

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

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

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

[47] 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

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

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

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

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

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

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

[54] 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

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

[56] 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

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

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

[59] 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

[60] GSC-II: Guide Star Catalog II, http://www-gsss.stsci.edu/gsc/GSChome.htm

[61] 2MASS: Two Micron All Sky Survey, http://irsa.ipac.caltech.edu/Missions/2mass.html

[62] POSS-II: Palomar Observatory Sky Survey, http://taltos.pha.jhu.edu/~rrg/science/dposs/dposs.html

[63] Sloan Digital Sky Survey / SkyServer, http://cas.sdss.org/astro/en/