THE UNIVERSITY OF CHICAGO

MANY-TASK COMPUTING: BRIDGING THE GAP BETWEEN HIGH-THROUGHPUT COMPUTING AND HIGH-PERFORMANCE COMPUTING

A DISSERTATION SUBMITTED TO

THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES

IN CANDIDACY FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

IOAN RAICU

CHICAGO, ILLINOIS

MARCH 2009

Abstract

Many-task computing aims to bridge the gap between two computing paradigms, high-throughput computing and high-performance computing. Many-task computing is reminiscent to high-throughput computing, but it differs in the emphasis of using many computing resources over short periods of time to accomplish many computational tasks, where the primary metrics are measured in seconds (e.g. tasks per second, I/O per second), as opposed to operations per month (e.g. jobs per month). Many-task computing denotes high-performance computations comprising of multiple distinct activities, coupled via file system operations. Tasks may be small or large, uniprocessor or multiprocessor, compute-intensive or data-intensive. The set of tasks may be static or dynamic, homogeneous or heterogeneous, loosely coupled or tightly coupled. The aggregate number of tasks, quantity of computing, and volumes of data may be extremely large. Many-task computing includes loosely coupled applications that are generally communication-intensive but not naturally expressed using message passing interface commonly found in high-performance computing, drawing attention to the many computations that are heterogeneous but not "happily" parallel.

This dissertation explores fundamental issues in defining the many-task computing paradigm, as well as theoretical and practical issues in supporting both compute and data intensive many-task computing on large scale systems. We have defined an abstract model for data diffusion – an approach to supporting data-intensive many-task computing, have defined data-aware scheduling policies with heuristics to optimize real world performance, and developed a competitive online caching eviction policy. We also designed and implemented the necessary middleware – Falkon – to enable the support of many-task computing on clusters, grids and supercomputers. Falkon, a Fast and Light-weight tasK executiON framework, addresses shortcomings in traditional resource management systems that support high throughput and high performance computing that are not suitable or efficient at supporting many-task computing applications.

Falkon was designed to enable the rapid and efficient execution of many tasks on large scale systems (i.e. through multi-level scheduling and streamlined distributed task dispatching), and integrate novel data management capabilities (i.e. data diffusion which uses data caching and data-aware scheduling to exploit data locality) to extend data intensive applications scalability well beyond that of traditional shared or parallel file systems. As the size of scientific data sets and the resources required for their analysis increase, data locality becomes crucial to the efficient use of large scale distributed systems for data-intensive many-task computing. We propose a “data diffusion” approach that acquires compute and storage resources dynamically, replicates data in response to demand, and schedules computations close to data. As demand increases, more resources are acquired, allowing faster response to subsequent requests that refer to the same data, and as demand drops, resources are released. This approach provides the benefits of dedicated hardware without the associated high costs, depending on workload and resource characteristics.

Micro-benchmarks have shown Falkon to achieve over 15K+ tasks/sec throughputs, scale to millions of queued tasks, to execute billions of tasks per day, and achieve hundreds of Gb/s I/O rates. Falkon has shown orders of magnitude improvements in performance and scalability across many diverse workloads (e.g heterogeneous tasks from milliseconds to hours long, compute/data intensive, varying arrival rates) and applications (e.g. astronomy, medicine, chemistry, molecular dynamics, economic modeling, and data analytics) at scales of billions of tasks on hundreds of thousands of processors across Grids (e.g. TeraGrid) and supercomputers (e.g. IBM Blue Gene/P and Sun Constellation).

Committee Members

Dr. Ian Foster

Arthur Holly Compton Distinguished Service Professor, CS, University of Chicago

Director, Computation Institute, University of Chicago

Associate Director, MCS, Argonne National Laboratory

Email: foster@cs.uchicago.edu, foster@mcs.anl.gov

Rick Stevens

Professor, CS, University of Chicago

Senior Fellow, Computation Institute, University of Chicago

Associate Laboratory Director, CELS, Argonne National Laboratory

Email: stevens@cs.uchicago.edu, stevens@anl.gov

Dr. Alex Szalay

Alumni Centennial Professor, Dept. of Physics and Astronomy, Johns Hopkins Univ.

Email: szalay@jhu.edu

Acknowledgements

This dissertation was made possible through the patience and guidance of my advisor, Ian Foster. My only hope is that a small part of Ian's vision of the future of computing has brushed off on me, and that my dissertation work ends up being a solid foundation for many others.

Special thanks go to my wife Daniela and my two boys, Johnny, and Lucas, who have been by my side all these years – this doctorate would not have been possible without your love, support, and understanding!

Many thanks for many specific contributions from various people. Alex Szalay inspired much of this work by proposing a challenging problem (e.g. the sky survey stacking service) whose primary requirement was to perform many small tasks in a distributed environment. I have had countless discussions on how to best support data intensive computing with Yong Zhao, Mike Wilde, Matei Ripeanu, Adriana Iamnitchi, Tevfik Kosar, and Doug Thain. Yong Zhao, Ben Clifford, Mihael Hategan, and Mike Wilde have made significant contributions to Falkon and its integration with the Swift system. Zhao Zhang has helped with porting Falkon to the IBM Blue Gene/P. Mike Kubal, Rick Stevens, Mathew Cohoon, and Fangfang Xia have helped considerably with the DOCK application, as well as to Don Hanson and J. Laitner with the MARS application. Catalin Dumitrescu has helped with the early work on the implementation of Falkon, its testing, and deploying on Amazon's Elastic Cloud, as well Kate Keahey and Tim Freeman have helped to configure and deploy virtual clusters. I sincerely thank all these people, who have contributed to the ideas, implementations, and publications making up my dissertation.

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.

Table of Contents

List of Figures

List of Tables

List of Equations

1 Introduction

Many-task computing aims to bridge the gap between two computing paradigms, high throughput computing and high performance computing. Many-task computing is reminiscent to high throughput computing, but it differs in the emphasis of using many computing resources over short periods of time to accomplish many computational tasks (i.e. including both dependent and independent tasks), where the primary metrics are measured in seconds (e.g. FLOPS, tasks/sec, MB/s I/O rates), as opposed to operations (e.g. jobs) per month. Many-task computing denotes high-performance computations comprising multiple distinct activities, coupled via file system operations or message passing. Tasks may be small or large, uniprocessor or multiprocessor, compute-intensive or data-intensive. The set of tasks may be static or dynamic, homogeneous or heterogeneous, loosely coupled or tightly coupled. The aggregate number of tasks, quantity of computing, and volumes of data may be extremely large. Many-task computing includes loosely coupled applications that are generally communication-intensive but not naturally expressed using standard message passing interface commonly found in high performance computing, drawing attention to the many computations that are heterogeneous but not "happily" parallel.

Many-task computing has taken shape from a fusion of many publications over the past several years [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23] and various workshops and sessions [24, 25]. The definitions and positions of this chapter have been published in [20].

1.1 Defining Many-Task Computing

We want to enable the use of large-scale distributed systems for task-parallel applications, which are linked into useful workflows through the looser task-coupling model of passing data via files between dependent tasks. This potentially larger class of task-parallel applications is precluded from leveraging the increasing power of modern parallel systems such as supercomputers (e.g. IBM Blue Gene/L [26] and Blue Gene/P [27]) because the lack of efficient support in those systems for the "scripting" programming model [28]. With advances in e-Science and the growing complexity of scientific analyses, more scientists and researchers rely on various forms of scripting to automate end-to-end application processes involving task coordination, provenance tracking, and bookkeeping. Their approaches are typically based on a model of loosely coupled computation, in which data is exchanged among tasks via files, databases or XML documents, or a combination of these. Vast increases in data volume combined with the growing complexity of data analysis procedures and algorithms have rendered traditional manual processing and exploration unfavorable as compared with modern high performance computing processes automated by scientific workflow systems. [18]

The problem space can be partitioned into four main categories (Figure 1 and Figure 2). 1) At the low end of the spectrum (low number of tasks and small input size), we have tightly coupled MPI [29] applications (white). 2) As the data size increases, we move into the analytics category, such as data mining and analysis (blue); MapReduce [30] is an example for this category. 3) Keeping data size modest, but increasing the number of tasks moves us into the loosely coupled applications involving many tasks (yellow); Swift/Falkon [13, 4] and Pegasus/DAGMan [31] are examples of this category. 4) Finally, the combination of both many tasks and large datasets moves us into the data-intensive many-task computing category (green); examples of this category are Swift/Falkon and data diffusion [1], Dryad [32], and Sawzall [33].

High performance computing can be considered to be part of the first category (denoted by the white area). High throughput computing [34] can be considered to be a subset of the third category (denoted by the yellow area). Many-Task Computing can be considered as part of categories three and four (denoted by the yellow and green areas). This chapter focuses on defining many-task computing, and the challenges that arise as datasets and computing systems are growing exponentially.

Clusters and Grids have been the preferred platform for loosely coupled applications that have been traditionally part of the high throughput computing class of applications, which are managed and executed through workflow systems or parallel programming systems.

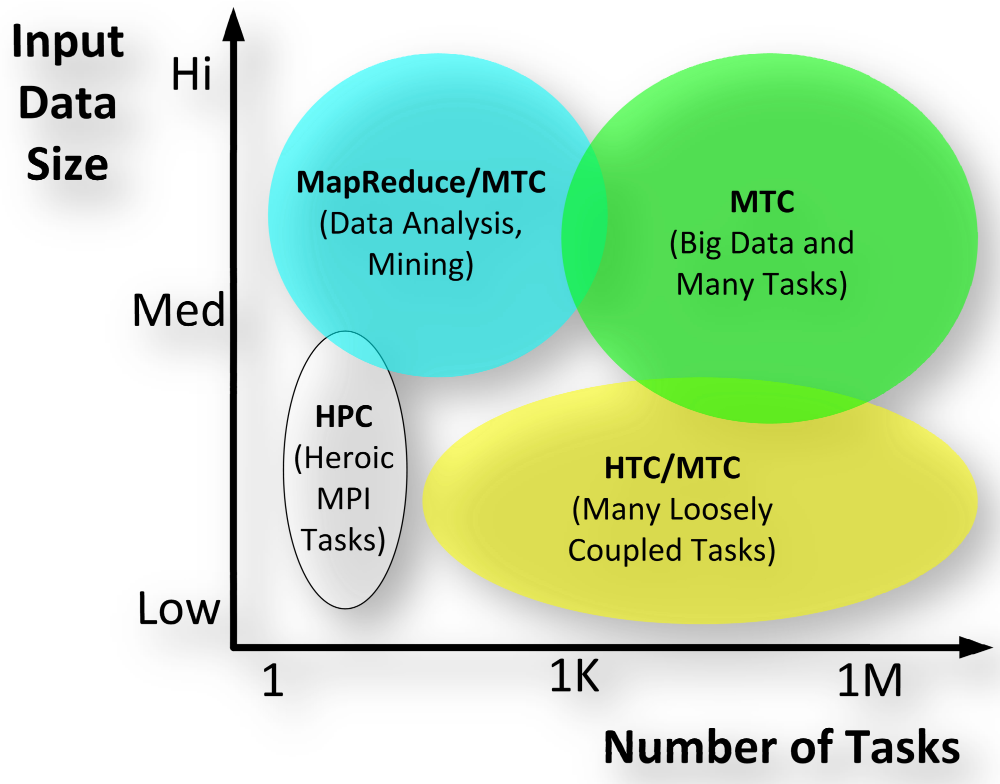


Figure 1: Problem types with respect to data size and number of tasks

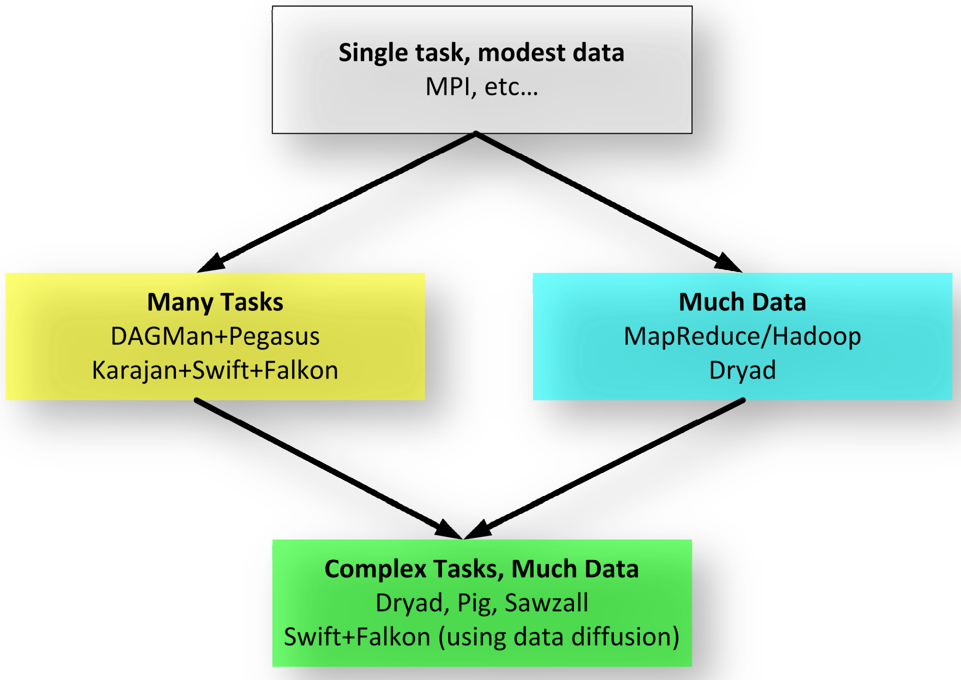


Figure 2: An incomplete and simplistic view of programming models and tools

Various properties of a new emerging applications, such as large number of tasks (i.e. millions or more), relatively short per task execution times (i.e. seconds to minutes long), and data intensive tasks (i.e. tens of MB of I/O per CPU second) have lead to the definition of a new class of applications called Many-Task Computing. MTC emphasizes on using much large numbers of computing resources over short periods of time to accomplish many computational tasks, where the primary metrics are in seconds (e.g., FLOPS, tasks/sec, MB/sec I/O rates), while HTC requires large amounts of computing for long periods of time with the primary metrics being operations per month [34]. MTC applications are composed of many tasks (both independent and dependent tasks) that can be individually scheduled on many different computing resources across multiple administrative boundaries to achieve some larger application goal.

MTC denotes high-performance computations comprising multiple distinct activities, coupled via file system operations or message passing. Tasks may be small or large, uniprocessor or multiprocessor, compute-intensive or data-intensive. The set of tasks may be static or dynamic, homogeneous or heterogeneous, loosely coupled or tightly coupled. The aggregate number of tasks, quantity of computing, and volumes of data may be extremely large. Is MTC really different enough to justify coining a new term? There are certainly other choices we could have used instead, such as multiple program multiple data (MPMD), high throughput computing, workflows, capacity computing, or embarrassingly parallel.

MPMD is a variant of Flynn's original taxonomy [35], used to denote computations in which several programs each operate on different data at the same time. MPMD can be contrasted with SPMD, in which multiple instances of the same program each execute on different processors, operating on different data. MPMD lacks the emphasis that a set of tasks can vary dynamically. High throughput computing [34], a term coined by Miron Livny within the Condor project [36], to contrast workloads for which the key metric is not floating-point operations per second (as in high performance computing) but “per month or year.” MTC applications are often just as concerned with performance as is the most demanding HPC application; they just don't happen to be SPMD programs. The term “workflow” was first used to denote sequences of tasks in business processes, but the term is sometimes used to denote any computation in which control and data passes from one “task” to another. We find it often used to describe many-task computations (or MPMD, HTC, MTC, etc.), making its use too general. “Embarrassingly parallel computing” is used to denote parallel computations in which each individual (often identical) task can execute without any significant communication with other tasks or with a file system. Some MTC applications will be simple and embarrassingly parallel, but others will be extremely complex and communication-intensive, interacting with other tasks and shared file-systems.

Is “many-task computing” a useful distinction? Perhaps we could simply have said “applications that are communication-intensive but are not naturally expressed in MPI”. Through the new term MTC, we are drawing attention to the many computations that are heterogeneous but not “happily” parallel.

1.2 MTC for Clusters, Grids, and Supercomputers

We claim that MTC applies to traditional HTC environments such as clusters and Grids, assuming appropriate support in the middleware, but also to supercomputers. Emerging petascale computing systems, such as IBM's Blue Gene/P [27], incorporate high-speed, low-latency interconnects and other features designed to support tightly coupled parallel computations. Most of the applications run on these computers have a single program multiple data (SMPD) structure, and are commonly implemented by using the Message Passing Interface (MPI) [29] to achieve the needed inter-process communication. I believe that MTC is a viable paradigm for supercomputers. As the computing and storage scales increase, the set of problems that must be overcome to make MTC practical (ensuring good efficiency and utilization at large-scale) are enlarged. These challenges include local resource manager scalability and granularity, efficient utilization of the raw hardware, shared file system contention and scalability, reliability at scale, application scalability, and understanding the limitations of the HPC systems in order to identify promising and scientifically valuable MTC applications.

One could ask, why use petascale systems for problems that might work well on terascale systems? We point out that petascale scale systems are more than just many processors with large peak petaflop ratings. They normally come well balanced, with proprietary, high-speed, and low-latency network interconnects to give tightly-coupled applications that use MPI good opportunities to scale well at full system scales. Even IBM has proposed in their internal project Kittyhawk [37] that the Blue Gene/P can be used to run non-traditional workloads (e.g. HTC).

Four factors motivate the support of MTC applications on petascale HPC systems.

1) *The I/O subsystem of petascale systems offers unique capabilities needed by MTC applications.* For example, collective I/O operations [17] could be implemented to use the specialized high-bandwidth and low-latency interconnects; we show preliminary results for collective I/O in Section 6.6. MTC applications could be composed of individual tasks that are themselves parallel programs, many tasks operating on the same input data, and tasks that need considerable communication among themselves. Furthermore, the aggregate shared file system performance of a supercomputer can be potentially larger than that found in a distributed infrastructure (i.e., Grid), with data rates in the 10GB+/s range, rather than the more typical 0.1GB/s to 1GB/s range of most Grid sites.

2) *The cost to manage and run on petascale systems like the Blue Gene/P is less than that of conventional clusters or Grids.* [37] For example, a single 13.9 TF Blue Gene/P rack draws 40 kilowatts, for 0.35 GF/watt. Two other systems that get good compute power per watt consumed are the SiCortex with 0.32 GF/watt and the Blue Gene/L with 0.23 GF/watt. In contrast, the average power consumption of the Top500 systems is 0.12 GF/watt [38]. Furthermore, we also argue that it is more cost effective to manage one large system in one physical location, rather than many smaller systems in geographically distributed locations.

3) *Large-scale systems inevitably have utilization issues.* Hence it is desirable to have a community of users who can leverage traditional back-filling strategies to run loosely coupled applications on idle portions of petascale systems.

4) *Perhaps most important, some applications are so demanding that only petascale systems have enough compute power to get results in a reasonable timeframe, or to leverage new opportunities.* With petascale processing of ordinary applications, it becomes possible to perform vast computations quickly, thus answering questions in a timeframe that can make a quantitative difference in addressing significant scientific challenges or responding to emergencies.

1.3 The Data Deluge Challenge and the Growing Storage/Compute Gap

Within the science domain, the data that needs to be processed generally grows faster than computational resources and their speed. The scientific community is facing an imminent flood of data expected from the next generation of experiments, simulations, sensors and satellites. Scientists are now attempting calculations requiring orders of magnitude more computing and communication than was possible only a few years ago. Moreover, in many currently planned and future experiments, they are also planning to generate several orders of magnitude more data than has been collected in the entire human history [39].

For instance, in the astronomy domain the Sloan Digital Sky Survey [40] has datasets that exceed 10 terabytes in size. They can reach up to 100 terabytes or even petabytes if we consider multiple surveys and the time dimension. In physics, the CMS detector being built to run at CERN's Large Hadron Collider [41] is expected to generate over a petabyte of data per year. In the bioinformatics domain, the rate of growth of DNA databases such as GenBank [42] and European Molecular Biology Laboratory (EMBL)

[43] has been following an exponential trend, with a doubling time estimated to be 9-12 months.

To enable the storage and analysis of large quantities of data and to achieve rapid turnaround, data needs to be distributed over thousands to tens of thousands of compute nodes. In such circumstances, data locality is crucial to the successful and efficient use of large scale distributed systems for data-intensive applications [19]. Scientific computing is generally executed on a shared infrastructure such as TeraGrid [44], Open Science Grid [45], and dedicated clusters, where data movement relies on shared or parallel file systems that are known bottlenecks for data intensive operations. If data analysis workloads have locality of reference, then it is feasible to cache and replicate data at each individual compute node, as high initial data movement costs can be offset by many subsequent data operations performed on cached data [1].

The rate of increase in the number of processors per system is outgrowing the rate of performance increase of parallel file systems, which requires rethinking existing data management techniques. For example, a cluster that was placed in service in 2002 with 316 processors has a parallel file system (i.e. GPFS [46]) rated at 1GB/s, yielding 3.2MB/s per processor of bandwidth. The second largest open science supercomputer, the IBM Blue Gene/P from Argonne National Laboratory, has 160K processors, and a parallel file system (i.e. also GPFS) rated at 8GB/s, yielding a mere 0.05MB/s per processor. That is a 65X reduction in bandwidth between a system from 2002 and one from 2008. Unfortunately, this trend is not bound to stop, as advances multi-core and many-core processors will increase the number of processor cores one to two orders of magnitude over the next decade.

We argue that in such circumstances, data locality is critical to the successful and efficient use of large distributed systems for data-intensive applications [19, 47]. Large scale data management needs to be a primary objective for any middleware targeting to support MTC workloads, to ensure data movement is minimized by intelligent data-aware scheduling both among distributed computing sites (assuming that each site has a local area network shared storage infrastructure), and among compute nodes (assuming that data can be stored on compute nodes' local disk and/or memory).

1.4 Middleware Support for MTC

As high throughput computing (HTC) is a subset of MTC, it is worth mentioning the various efforts in enabling HTC on large scale systems. Some of these systems are Condor [36], MapReduce [30], Hadoop [48], and BOINC [49].

Condor and glide-ins [50] are the original tools to enable HTC, but their emphasis on robustness and recoverability limits their efficiency for MTC applications in large-scale systems. We found that relaxing some constraints (e.g. recoverability) from the middleware and encouraging the end applications to implement these constraints has enabled significant improvements in middleware performance and efficiency at large scale.

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

BOINC is known to scale well to large number of compute resources, but lacks support for data intensive applications due to the nature of the wide area network deployment BOINC typically has, as well as lack of support for "black box" applications.

On the IBM Blue Gene supercomputer, various works [51, 52] have leveraged the HTC-mode [53] support in Cobalt [54] scheduling system. These works have aimed at integrating their solution as much as possible in Cobalt; however, it is not clear that the current implementations will be able to support the largest MTC applications at the largest scales. Furthermore, these works focus on compute resource management, and ignore data management altogether.

Swift [13, 55] and Falkon [4] have been used to execute MTC applications on clusters, multi-site Grids (e.g., Open Science Grid [45], TeraGrid [44]), specialized large machines (SiCortex [56]), and supercomputers (e.g., Blue Gene/P [27]). Swift enables scientific workflows through a data-flow-based functional parallel programming model. It is a parallel scripting tool for rapid and reliable specification, execution, and management of large-scale science and engineering workflows. The runtime system in Swift relies on the CoG Karajan [57] workflow engine for efficient scheduling and load balancing, and it integrates with the Falkon light-weight task execution dispatcher for optimized task throughput and efficiency, as well as improved data management capabilities to ensure good scalability with increasing compute resources. Large-scale applications from many domains (e.g., astronomy [4, 58], medicine [4, 59, 60], chemistry [10], molecular dynamics [61], economics [62, 63], and data analytics [16]) have been run at scales of up to millions of tasks on up to hundreds of thousands of processors. This chapter focuses on defining MTC, as well the theory and practice to enable MTC a wide range of systems from the average cluster to the largest supercomputers. The Falkon middleware represents the practical aspects of enabling MTC workloads on these systems.

1.5 MTC Applications

We have found many 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.

While we can push hundreds or even thousands of such small jobs via GRAM to a traditional local resource manager (e.g. PBS [64], Condor [50], SGE [65]), the achieved utilization of a modest to large resource set will be poor due to high queuing and dispatching overheads, which ultimately results in low job throughput. A common technique to amortize the costs of the local resource management is to “cluster” multiple jobs into a single larger job. Although this lowers the per job overhead, it is best suited when the set of jobs to be executed are homogenous in execution times, or accurate execution time information is available prior to the job execution; with heterogeneous job execution times, it is hard to maintain good load balancing of the underlying resource, causing low resource utilization. We claim that it is not enough to "cluster" jobs, and that the middleware that manages jobs (or tasks used synonymously throughout) must be streamlined and made as light-weight as possible to allow applications with heterogonous execution times to execute without "clustering" with high efficiency.

Traditionally, scientific applications rely on centralized server(s) (e.g. shared file system, parallel file systems, ftp server, web server, gridftp server) making these applications scalability limited depending on how data/meta-data intensive they are. MTC applications are often data and/or meta-data intensive, as each job requires at least one input file and one output file, and can sometimes involve many files per job. In order to support MTC applications in general, additional data management techniques are needed to handle both data intensive and meta-data intensive applications. These data management techniques need to make good utilization of the full network bandwidth of large scale systems, which is a function of the number of nodes and networking technology employed, as opposed to the relatively small number of storage servers that are behind a parallel file system or GridFTP server.

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.

Astronomy: One of the first applications that motivated much of this work was called the "AstroPortal" [12], which offered a stacking service of astronomy images from the Sloan Digital Sky Survey (SDSS) dataset using grid resources. 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. This application involved the SDSS dataset [40] (currently at 10TB with over 300 million objects, but these datasets could be petabytes in size if we consider multiple surveys in both time and space) [18], many tasks ranging from 10K to millions of tasks, each requiring 100ms to seconds of compute and 100KB to MB of input and output data.

Astronomy: Another related application in astronomy is MONTAGE [66, 58], a national virtual observatory project [67] that stitches tiles of images of the sky from various sky surveys (e.g. SDSS [40], etc) into a photorealistic single image. Execution times per task range in the 100ms to 10s of seconds, and each task involves multiple input images and at least one image output. This application is both compute intensive and data intensive, and has been run as both a HTC (using Pegasus/DAGMan [31], and Condor [36]) and a HPC (using MPI) application, but we found its scalability to be limited when run under HTC or HPC.

Astrophysics: Another application is from astrophysics, which analyzes the Flash turbulence dataset (simulation data) [68] from various perspectives, using 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 resource, and contains both temporal and spatial locality. In the physics domain, the CMS detector being built to run at CERN's Large Hadron Collider [41] is expected to generate over a petabyte of data per year. Supporting applications that can perform a wide range of analysis of the LHC data will require novel support for data intensive applications.

Economic Modeling: An application from the economic modeling domain that we have investigated as a good MTC candidate is Macro Analysis of Refinery Systems (MARS) [62], which studies economic model sensitivity to various parameters. MARS 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. Other economic modeling applications perform numerical optimization to determine optimal resource assignment in energy problems. This application is challenging as the parameter space is extremely large, which can produce millions, even billions of individual tasks, each with a relatively short execution time of only seconds long.

Pharmaceutical Domain: In the pharmaceutical domain, there are applications that screen KEGG [69] compounds and drugs against important metabolic protein targets using DOCK6 [61] to simulate the "docking" of small molecules, or ligands, to the "active sites" of large macromolecules of known structure called "receptors". 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. The economic and health benefits of speeding drug development by rapidly screening for promising compounds and eliminating costly dead-ends is significant in terms of both resources and human life. The parameter space is quite large, totaling to more than one billion computations that have a large variance of execution times from seconds to hours, with an average of 10 minutes. The entire parameter space would require over 22,600 CPU years, or over 50 days on a 160K processor Blue Gene/P supercomputer [27]. This application is challenging as there many tasks, each task has a wide range of execution times with little to no prior knowledge about its execution time, and involves significant I/O for each computation as the compounds are typically stored in a database (i.e. 10s to 100s of MB large) and must be read completely per computation.

Chemistry: Another application in the same domain is OOPS [70], which aims to predict protein structure and recognize docking partners. In chemistry, specifically in molecular dynamics, we have an application MolDyn whose goal is to calculate the solvation free energy of ligands and protein-ligand binding energy, with structures obtained from the NIST Chemistry WebBook database. [71] Solvation free energy is an important quantity in Computational Chemistry with a variety of applications, especially in drug discovery and design. These applications have similar characteristics as the DOCK application previously discussed.

Bioinformatics: In bioinformatics, Basic Local Alignment Search Tool (BLAST), is a family of tools for comparing primary biological sequence information (e.g. amino-acid sequences of proteins, nucleotides of DNA sequences). A BLAST search enables one to compare a query sequence with a library or database of sequences, and identify library sequences that resemble the query sequence above a certain threshold. [72]. Although the BLAST codes have been implemented in both HTC and HPC, they are often both data and compute intensive, requiring multi-GB databases to be read for each comparison (or kept in memory if possible), and each comparison can be done within minutes on a modern processor-core. MTC and its support for data intensive applications is critical in scaling BLAST on large scale resources with thousands to hundreds of thousands of processors.

Neuroscience Domain: In the neuroscience domain, we have the Computational Neuroscience Applications Research Infrastructure (CNARI), which aims to manage neuroscience tools and the heterogeneous compute resources on which they can enable large-scale computational projects in the neuroscience community. The analysis includes the aphasia study, structural equation modeling, and general use of R for various data analysis. [73] The application workloads involve many tasks, relatively short on the order of seconds, and each task containing many small input and output files making the application meta-data intensive at large scale.

Cognitive Neuroscience: The fMRI application is a workflow from the cognitive neuroscience domain with a four-step pipeline, which includes Automated Image Registration (AIR), Preprocessing and stats from NIH and FMRIB (AFNI and FSL), and Statistical Parametric Mapping (SPM2) [59]. An fMRI Run is a series of brain scans called volumes, with a Volume containing a 3D image of a volumetric slice of a brain image, which is represented by an Image and a Header. Each volume can contain hundreds to thousands of images, and with multiple patients, the number of individual analysis tasks can quickly grow. Task execution times were only seconds long, and the input and output files ranged from kilobytes to megabytes in size. This application could run as an HTC one at small scales, but needs MTC support to scale up.

Data Analytics: Data analytics and data mining is a large field that covers many different applications. Here, we outline several applications that fit MTC well. One example is the analysis of log data from millions computations. Another set of applications are ones commonly found in the MapReduce [30] paradigm, namely "sort" and "word count". Both of these applications are essential to World Wide Web search engines, and are challenging at medium to large scale due to their data intensive nature.

All three applications involve many tasks, many input files (or many disjoint sub-sections of few files), and are data intensive.

Data Mining: Another set of applications that perform data analysis can be classified in the “All-Pairs” class of applications. These applications aim to understand the behavior of a function on two sets, or to learn the covariance of these sets on a standard inner product. Two common applications in All-Pairs are data mining and biometrics. Data mining is the study of extracting meaning from large data sets; one phase of knowledge discovery is reacting to bias or other noise within a set. Different classifiers work better or worse for varying data, and hence it is important to explore many different classifiers in order to be able to determine which classifier is best for that type of noise on a particular distribution of the validation set.

Biometrics: Biometrics aims to identifying humans from measurements of the body (e.g. photos of the face, recordings of the voice, and measurements of body structure). A recognition algorithm may be thought of as a function that accepts two images (e.g. face) as input and outputs a number between zero and one indicating the similarity between the two input images. The application would then compare all images of a database and create a scoring matrix which can later be easily searched to retrieve the most similar images. These All-Pairs applications are extremely challenging as the number of tasks can rapidly grow in the millions and billions, with each task being hundreds of milliseconds to tens of seconds, with multi-megabyte input data per task.

MPI Ensembles: Finally, another class of applications is managing an ensemble of MPI applications. One example is from the climate modeling domain, which has been studying climate trends and predicting global warming [74], is already implemented as an HPC MPI application. However, the current climate models could be run as ensemble runs (many concurrent MPI applications) to quantify climate model uncertainty. This is challenging in large scale systems such as supercomputers (a typical resource such models would execute on), as the local resource managers (e.g. Cobalt) favor large jobs and have policy against running many jobs at the same time (i.e. where many is more than single digit number of jobs per user).

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 support as the scale of these resources grows. We discuss these applications in more details in Chapter 7, and explore their performance scalability across a wide range of systems, such as clusters, grids, and supercomputers.

1.6 Conclusions

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

We believe that there is more to HPC than tightly coupled MPI, and more to HTC than embarrassingly parallel long running jobs. Like HPC applications, and science itself, applications are becoming increasingly complex opening new doors for many opportunities to apply HPC in new ways if we broaden our perspective. We hope this dissertation leaves the broader community with a stronger appreciation of the fact that applications that are not tightly coupled MPI are not necessarily embarrassingly parallel. Some have just so many simple tasks that managing them is hard. Applications that operate on or produce large amounts of data need sophisticated data management in order to scale. There exist applications that involve many tasks, each composed of tightly coupled MPI tasks. Loosely coupled applications often have dependencies among tasks, and typically use files for inter-process communication. Efficient support for these sorts of applications on existing large scale systems, including future ones (e.g. Blue Gene/Q [75] and Blue Water supercomputers) involves substantial technical challenges and will have big impact on science.

We have already shown good support for MTC on a variety of resources from clusters, grids, and supercomputers through our work on Swift [10, 13, 55] and Falkon [2, 4]. Furthermore, we have taken the first steps to address data-intensive MTC by offloading much of the I/O away from parallel file systems and into the network, making full utilization of caches (both on disk and in memory) and the full network bandwidth of commodity networks (e.g. gigabit Ethernet) as well as proprietary and more exotic networks (Torus, Tree, and Infiniband). [1, 3, 17]

1.7 Dissertation Roadmap

The rest of this dissertation is organized in several chapters, each with their unique contributions. Chapter 2 is titled "Background Information and Related Work", which covers essential background information on the topics of clusters, grids, supercomputers, high throughput computing, high performance computing, and testbeds used throughout the dissertation. The chapter also discusses related work in three main areas, local resource management, resource provisioning, and data management. Chapter 3 is "Multi-Level Scheduling and Streamlined Task Dispatching", which covers the practical foundation of this dissertation in the form of Falkon, a Fast and Light-weight tasK executiON framework. Chapter 4 titled "Distributing the Falkon Architecture to Support Petascale Systems" extends the work from Chapter 3 by distributing the Falkon architecture and showing that Falkon can scale to 160K processors on the IBM Blue Gene/P. Chapter 5, "Dynamic Resource Provisioning", covers the techniques needed to allow applications resource usage to adapt to dynamic and varying workloads for both resource efficiency and application performance reasons. Chapter 6 titled "Towards Data Intensive Many-Task Computing with Data Diffusion" covers work and results aimed to support data-intensive MTC, and shows how data locality can be leveraged in order to achieve significantly better performance and scalability; this chapter also covers support for data-intensive applications on the IBM Blue Gene/P supercomputer with collective I/O primitives to enable efficient distribution of input data files to computing nodes and gathering of output results from them. Chapter 7, titled "Accelerating Scientific Applications on Clusters, Grids, and Supercomputers", showcases the end-product of this entire dissertation, the wide range of applications that have been run using the proposed techniques, and what improvements they have achieved in both performance and scalability. Chapter 8 concludes this dissertation with a summary of our contributions, concluding statements, and a discussion of my future research directions.

2 Background Information and Related Work

This chapter covers essential background information on the topics of clusters, grids, supercomputers, high throughput computing, high performance computing, and testbeds used throughout the dissertation. This chapter also discusses related work in three main areas: 1) local resource management, 2) resource provisioning, and 3) data management.

2.1 Clusters, Grids, and Supercomputers

A computer cluster is a collection of computers, connected together by some networking fabric, and is composed of commodity processors, network interconnects, and operating systems. Clusters are usually aimed to improve performance and availability over that of a single computer; furthermore, clusters are typically more cost-effective than a single computer of comparable speed or availability. Middleware such as MPI allows cluster computing to be portable to a wide variety of cluster architectures, operating systems, and networking offering high performance computing over commodity hardware. [76] High throughput computing [34] has also seen good success on clusters, as the needed computations are more loosely coupled and most scientists can be satisfied by commodity CPUs and memory, essentially making high efficiency not playing a major role.

Grids tend to be composed of multiple clusters, and are typically loosely coupled, heterogeneous, and geographically dispersed. The term “the Grid” was coined in the mid 1990s to denote a proposed distributed computing infrastructure for advanced science and engineering [77]. Foster et al. describes the definition of Grid Computing to be about large scale resource sharing, innovative applications, and high performance computing. It is meant to offer flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources, namely virtual organizations. Some examples of grids are TeraGrid [44], Open Science Grid (OSG) [45], and Enabling Grids for E-sciencE (EGEE) [78]. Some of the major grid middleware are the Globus Toolkit [79] and Unicore [80]. Grids have also been used for both HPC and HTC, just as clusters have; HPC is more challenging for grids as resources can be geographically distributed which can increases latency significantly between nodes, but it can still be done effectively with careful tuning for some HPC applications.

A supercomputer is a highly-tuned computer clusters using commodity processors combined with custom network interconnects and typically customized operating system. The term supercomputer is rather fluid, as today's supercomputer usually ends up being an ordinary computer within a decade. Figure 3 shows the Top500 [38] trends in the fastest supercomputers for the past fifteen years, and projecting out for the next decade. We have currently surpassed the petaflop/s rating, and it is predicted that we will surpass an exaflop/s within the next decade. Much of the predicted increase computing power comes from the prediction of increasing the number of cores per processor (see Figure 4), which is expected to be in the hundreds to thousands within a decade [81]. Until recently, supercomputers have been strictly HPC systems, but more recently they have gained support for HTC as well.

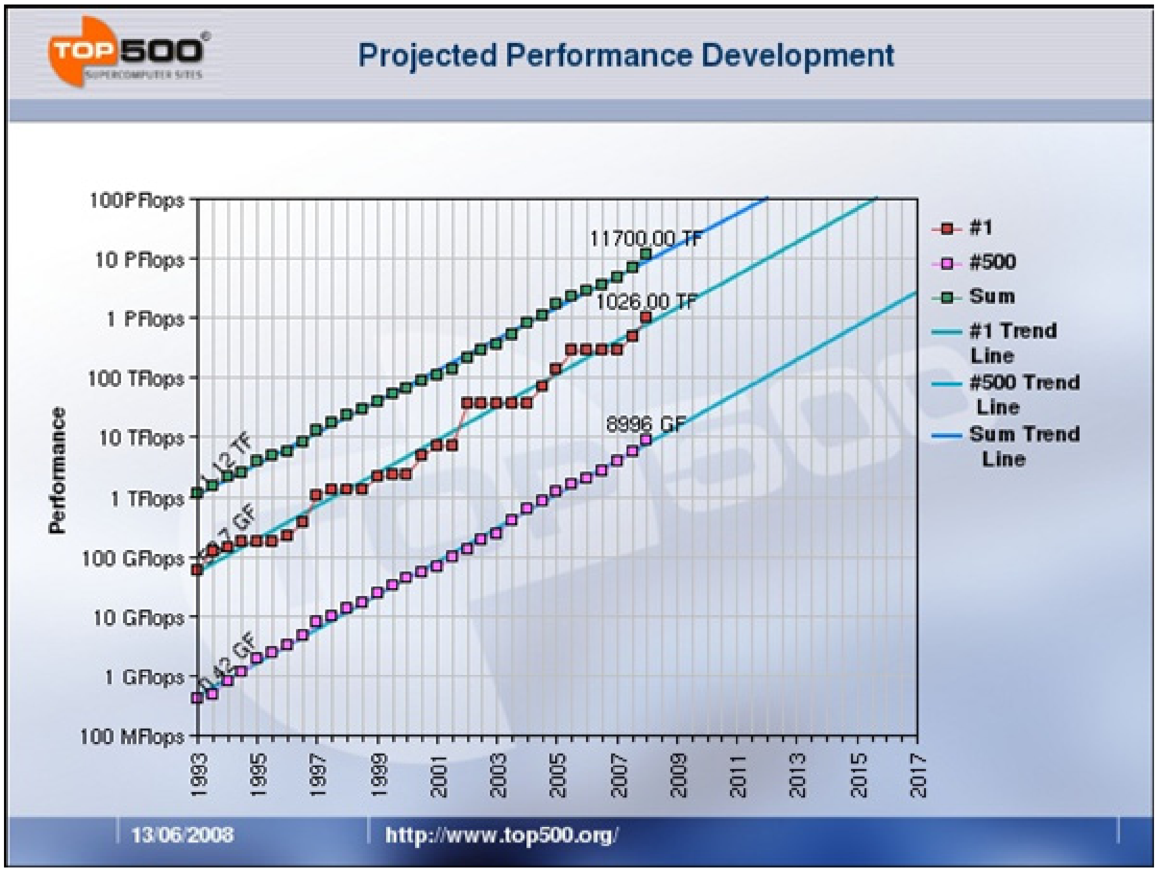


Figure 3: Projected Performance of Top500, November 2008

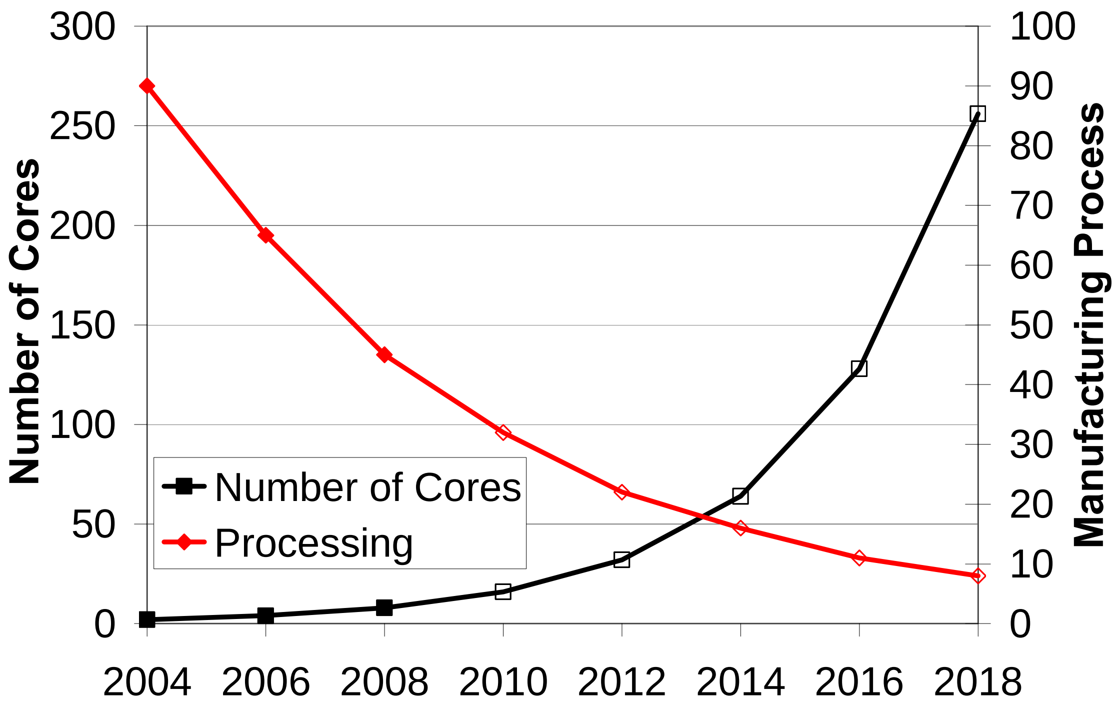


Figure 4: Projected number of cores per processor, November 2007

With the work presented in this dissertation, we have shown that many-task computing [20] is also a viable computing model for all these platforms, from clusters, grids, and supercomputers, and in fact, many more applications naturally fit in MTC than HTC, and without having to rewrite them in HPC.

2.2 Testbeds

The experiments conducted in this dissertation were completed over the span of several years, on a range of systems from single machines, small clusters, to grids, to specialized systems, to supercomputers. This sub-section describes all these systems for easy reference for the rest of the dissertation where we discuss experimental results.

One of the testbeds was the ANL/UC Linux cluster (a 128 node cluster from the TeraGrid), which consisted of dual Xeon 2.4 GHz CPUs or dual Itanium 1.3GHz CPUs with 4GB of memory and 1Gb/s network connectivity. We also used two other systems: 1) VIPER.CI was a dual Xeon 3GHz with HT (4 hardware threads), 2GB of RAM, Linux kernel 2.6, and 1Gb/s network connectivity; and 2) GTO.CI was a dual Xeon 2.33 GHz with quad cores each (8-cores), 6GB of RAM, Linux kernel 2.6, and 1Gb/s network connectivity. Both machines were located at University of Chicago and had a network latency of less than 2 ms to and from the resources it managed that were housed at Argonne National Laboratory (ANL).

ANL also acquired a new 6 TFlop machine named the SiCortex [56] (see Figure 5); it has 6-core processors for a total of 5832-cores each running at 500 MHz, has a total of 4TB of memory, and runs a standard Linux environment with kernel 2.6. The system is connected to a NFS shared file system which is only served by one server, and can sustain only about 320 Mb/s read performance. A PVFS shared file system is also planned that will increase the read performance to 12,000 Mb/s, but that was not available to us during out testing. All experiment on the SiCortex were performed using the NFS shared file system, the only available shared file system at the time of the experiments.

The IBM Blue Gene/P Supercomputer [27, 75] (named Intrepid and hosted at Argonne National Laboratory) has quad-core processors with a total of 160K cores. The Blue Gene/P is rated at 557TF Rmax (450TF Rpeak) with 160K PPC450 processors running at 850MHz, with a total of 80 TB of main memory. The Blue Gene/P GPFS is rated at 8GB/s. In our experiments, we use an alpha version of Argonne's Linux-based ZeptoOS [82] compute node kernel. The BG/P architecture overview is depicted in Figure 6.

Figure 5: SiCortex Model 5832

The full BG/P at ANL is over 500 TFlops with 160K PPC450 processors running at 850MHz, with a total of 80 TB of main memory. The system architecture is designed to scale to 3 PFlops for a total of 4.3 million processors. The BG/P has both GPFS and PVFS file systems available; the GPFS we tested was rated at 8GB/s I/O rates. Experiments involving the BG/P were done on both the reference pre-production BG/P that had 4096 processors, and the full production 160K processor system, and using the GPFS parallel file system.

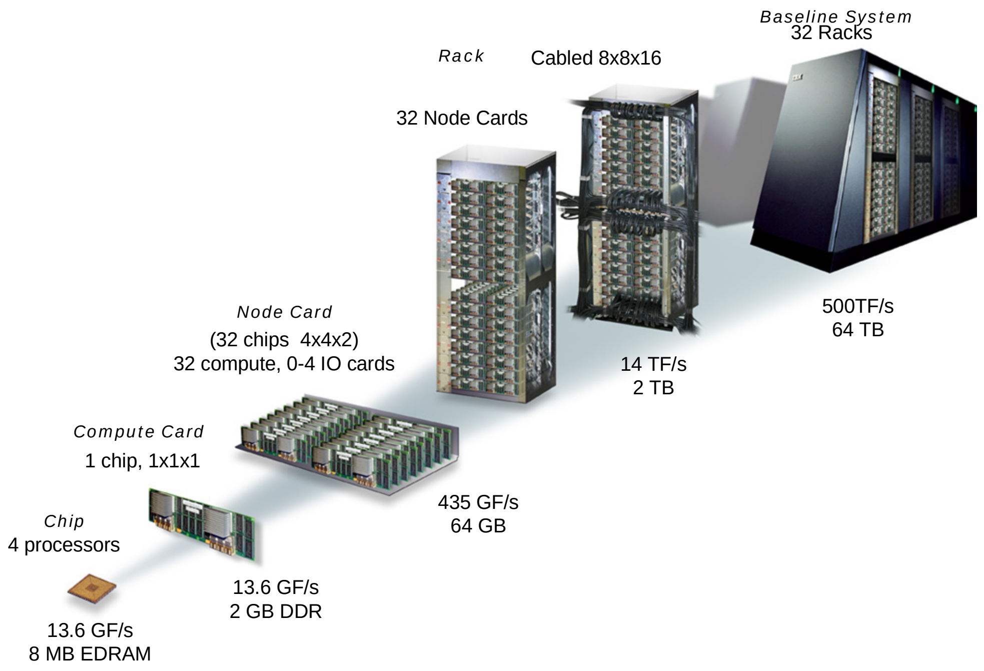


Figure 6: BG/P Architecture Overview

The various systems we used in this dissertation are outlined in Table 1.

Table 1: Summary of testbeds used in this dissertation

Name	NodesCPUs	CPU Type Speed	RAM	File System Peak	Operating System
BG/P.4K	1024 4096	PPC450 0.85GHz	2TB	GPFS 775Mb/s	Linux (ZeptOS)
BG/P.160K	40960 163840	PPC450 0.85GHz	80TB	GPFS 64Gb/s	Linux (ZeptOS)
BG/P.Login	8 32	PPC 2.5GHz	32GB	GPFS 775Mb/s	Linux Kernel 2.6
SiCortex	972 5832	MIPS64 0.5GHz	3.5TB	NFS 320Mb/s	Linux Kernel 2.6
ANL/UC	98 196	Xeon 2.4GHz	0.4TB	GPFS 3.4Gb/s	Linux Kernel 2.4
	62 124	Itanium 1.3GHz	0.25TB	GPFS 3.4Gb/s	Linux Kernel 2.4
VIPER.CI	1 2	Xeon 3GHz	2GB	Local 800Mb/s	Linux Kernel 2.6
GTO.CI	1 8	Xeon 2.3GHz	6GB	Local 800Mb/s	Linux Kernel 2.6

2.3 Local Resource Management

Full-featured local resource managers (LRMs) such as Condor [36, 83], Portable Batch System (PBS) [64], Load Sharing Facility (LSF) [84], and Sun Grid Engine (SGE) [65] support client specification of resource requirements, data staging, process migration, check-pointing, accounting, and daemon fault recovery. In contrast, we have focused on efficient task dispatch and thus can omit some of these features in order to streamline task submission. This narrow focus is possible as we can rely on certain functions (e.g., accounting) on other LRMs and others (e.g., recovery, data staging) on client applications.

The BOINC "volunteer computing" system [49, 85] has a similar architecture to that of Falkon. BOINC's database-driven task dispatcher is estimated to be capable of dispatching 8.8M tasks per day to 400K workers. This estimate is based on extrapolating from smaller synthetic benchmarks of CPU and I/O overhead, on the task distributor only, for the execution of 100K tasks. We have achieved throughputs in excess of 15K tasks/sec, in comparison to 101 tasks/sec delivered by BOINC.

Due to the only recent availability of parallel systems with 100K cores or more for open science research, and the even scarcer experience or success in loosely coupled programming at this scale, we find that there is little existing work with which we can compare, although we found two papers that explored a similar space of high throughput computing on large-scale supercomputers, such as the IBM Blue Gene/L [26]. Cope et al. aimed at integrating their solution as much as possible in the Cobalt scheduling system (as opposed to bringing in another system such as Falkon); their implementation was on the Blue Gene/L using the HTC-mode [53] support in Cobalt, and the majority of the performance study was done at a small scale (64 nodes, 128 processors). [51] In summary, Cope's results were at least one order of magnitude worse at small scales, and the performance gap would only increase with larger scale tests as their approach has higher overheads (i.e. nodes reboot after each task, in contrast with simply forking another process). Furthermore, Peter's et al. from IBM also recently published some performance numbers on the HTC-mode [52] native support in Cobalt [54], which shows a similar one order of magnitude difference between HTC-mode on Blue Gene/L. We will show detailed comparisons on similar benchmarks between this work (Cope at al. [51] and Peter et al. [52]) and our own work on the Blue Gene/P supercomputer.

Task farming is a general concept that has been applied on a wide range of systems, and is loosely related to resource management and high throughput computing in which many individual jobs can be scheduled to a processor farm. The Blue Gene supercomputer is one example which has defined and implemented task farms in order to implement parallelism in some applications. [86] Casanova et al. addresses basic scheduling strategies for task farming in Grids [87]; they acknowledge the difficulties that arise in scheduling task farms in dynamic and heterogeneous systems, but do little to address these problems. M. Danelutto argues the inefficiencies of task farms in heterogeneous and unreliable environments; he proposes various adaptive task farm implementation strategies [88] to address these classical inefficiencies found in task farms. H. Gonzalez-Velez argues for similar inefficiencies for task farms due to heterogeneity typically found in Grids. He claims that the dynamicity of Grids also leads to sub-optimal task farm solutions. He proposes an adaptive skeletal task farm for Grids [89] which take into account predictions of network bandwidth, network latency, and processor availability. Heymann et al. investigates scheduling strategies that dynamically measures the execution times of tasks and uses this information to dynamically adjust the number of workers to achieve a desirable efficiency, minimizing the impact in loss of speedup. [90] Petrou et al. show how scheduling speculative tasks in a compute farm [91] can significantly reduce the visible response time. The basic idea is that a typical end use would submit more work than he really needed in the hopes of allowing the scheduler ample opportunities to schedule work before the end user needed to retrieve the results. We believe that this model of scheduling would work only in a lightly loaded compute farm, which is not the norm in today's deployed Grids.

In summary, our proposed work in light-weight task dispatching offers many orders of magnitude better performance and scalability than traditional resource management techniques, which is changing the types of applications that can efficiently execute on large scale distributed resources; this was one of the reasons that led us to defining many-task computing, as once the capability of light-weight task dispatching was available, a whole new set of applications emerged that could make use of this capability to efficiently run at large scale. We have achieved these improvements by narrowing the focus of the resource management by not supporting various expensive features, and by relaxing other constraints from the resource management framework effectively pushing them to the application or the clients. Furthermore, we differentiate our work from previous work on task farming by addressing data-aware scheduling to optimize the raw compute and storage resources utilization.

2.4 Resource Provisioning

Multi-level scheduling has been applied at the OS level [92, 93] to provide faster scheduling for groups of tasks for a specific user or purpose by employing an overlay that does lightweight scheduling within a heavier-weight container of resources: e.g., threads within a process or pre-allocated thread group.

Frey and his colleagues pioneered the application of resource provisioning to clusters via their work on Condor "glide-ins" [50]. Requests to a batch scheduler (submitted, for example, via Globus GRAM) create Condor "startd" processes, which then register with a Condor resource manager that runs independently of the batch scheduler. Others have also used this technique. For example, Mehta et al. [94] embed a Condor pool in a batch-scheduled cluster, while MyCluster [95] creates "personal clusters" running Condor or SGE. Such "virtual clusters" can dedicated to a single workload; thus, Singh et al. find, in a simulation study [96], a reduction of about 50% in completion time. However, because they rely on heavyweight schedulers to dispatch work to the virtual cluster, the per-task dispatch time remains high, and hence the wait queue times are likely to remain significantly higher than in the ideal case due to the schedulers' inability to push work out faster.

In a different space, Bresnahan et al. [97] describe a multi-level scheduling architecture specialized for the dynamic allocation of compute cluster bandwidth. A modified Globus GridFTP server varies the number of GridFTP data movers as server load changes.

Appleby et al. [98] were one of several groups to explore *dynamic resource provisioning within a data center*. Ramakrishnan et al. [99] also address *adaptive resource provisioning* with a focus primarily on resource sharing and container level resource management.

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

2.5 Data Management

There has been much work in the general space of data management in distributed systems over the last decade. The Globus Toolkit includes two components (Replica Location Service [100] and Data Replication Service [101]) that can be used to build data management services for Grid environments. Data management on its own is useful, but not as useful as it could be if it were to be coupled with compute resource management as well. Ranganathan et al. used simulation studies [102] to show that proactive data replication can improve application performance. The Stork [103] scheduler seeks to improve performance and reliability when batch scheduling by explicitly scheduling data placement operations. While Stork can be used with other system components to coschedule CPU and storage resources, there is no attempt to retain nodes and harness data locality in data access patterns between tasks.

The GFarm [104, 105] team implemented a data-aware scheduler in Gfarm using an LSF scheduler plugin [106, 107]. Their performance results are for a small system in comparison to our own results and offer relatively slow performance (6 nodes, 300 jobs, 900 MB input files, 0.1–0.2 jobs/sec, and 90MB/s to 180MB/s data rates); furthermore, the papers present no evidence that their system scales. In contrast, we have tested our proposed data diffusion with 200 processors, 2M jobs, input data ranging from 1B to 1GB per job, working sets of up to 1TB, workflows exceeding 1000 jobs/sec, and data rates exceeding 9GB/s.

The NoW [108] project aimed to create massively parallel processors (MPPs) from a collection of networked workstations; NoW has its similarities with the Falkon task dispatching framework, but it differs in the level of implementation, Falkon being higher-level (i.e. cluster local resource manager) and NoW being lower-level (i.e. OS). The proceeding River [109] project aimed to address specific challenges in running data intensive applications via a data-flow system, specifically focusing on database operations (e.g. selects, sorts, joins). The Swift [13, 10] parallel programming system, which can use Falkon as an underlying execution system, is a general purpose parallel programming system that is data-flow based, and has all the constructs of modern programming languages (e.g. variables, functions, loops, conditional statements). One of the limitations of River is that one of its key enabling concepts, graduated declustering (GD), requires data to be fully replicated throughout the entire cluster. This indicates that their scheduling policies are simpler than those found in data diffusion, as all data can be found everywhere; this assumption also incurs extra costs for replication, and has large wastage in large-scale systems. River is a subset of the combination of Swift, Falkon and data diffusion.

BigTable [110], Google File System (GFS) [111], MapReduce [30], and Hadoop [48] couple data and computing resources to accelerate data-intensive applications. However, these systems all assume a dedicated set of resources, in which a system configuration dictates nodes with roles (i.e., clients, servers) at startup, and there is no support to increase or decrease the ratio between client and servers based on load; note that upon failures, nodes can be dynamically removed from these systems, but this is done for system maintenance, not to optimize performance or costs. This is a critical difference, as these systems are typically installed by a system administrator and operate on dedicated clusters. Our work (Falkon and data diffusion) works on batch-scheduled distributed resources (such as those found in clusters and Grids used by the scientific community) which are shared by many users. Although MapReduce/Hadoop systems can also be shared by many users, nodes are shared by all users and data can be stored or retrieved from any node in the cluster at any time. In batch scheduled systems, sharing is done through abstraction called jobs which are bound to some number of dedicated nodes at provisioning time. Users can only access nodes that are provisioned to them, and when nodes are released there are no assumptions on the preservation of node local state (i.e. local disk and ram). Data diffusion supports dynamic resource provisioning by allocating resources from batch-scheduled systems when demand is high, and releasing them when demand is low, which efficiently handles workloads which have much variance over time. The tight coupling of execution engine (MapReduce, Hadoop) and file system (GFS, HDFS) means that scientific applications must be modified, to use these underlying non-POSIX compliant filesystems to read and write files. Data diffusion coupled with the Swift parallel programming system [13, 10] can enable the use of data diffusion without any modifications to scientific applications, which typically rely on POSIX compliant file systems. Furthermore, through the use of Swift's check-pointing at a per task level, failed application runs (synonymous with a job for MapReduce/Hadoop) can be restarted from the point they previously failed; although tasks can be retried in MapReduce/Hadoop, a failed task can render the entire MapReduce job failed. It is also worth mentioning that data replication in data diffusion occurs implicitly due to demand (e.g. popularity of a data item), while in Hadoop it is an explicit parameter that must be tuned per application. We believe Swift and data diffusion is a more generic solution for scientific applications and is better suited for batch-scheduled clusters and grids.

Two systems that often compare themselves with MapReduce and GFS are Sphere [112] and Sector [113]. Sphere is designed to be used with the Sector Storage Cloud, and implements certain specialized, but commonly occurring, distributed computing operations. For example, the MapReduce programming model is a subset of the Sphere programming model, as the Map and Reduce functions could be any arbitrary functions in Sphere. Sector is the underlying storage cloud that provides persistent storage for the data required by Sphere and manages the data for Sphere operations. Sphere is analogous to Swift, and Sector is analogous with data diffusion, although they each differ considerably. For example, Swift is a general purpose parallel programming system, and the programming model of both MapReduce and Sphere are a subset of the Swift programming model. Data diffusion and Sector are quite similar in function, both providing the underlying data management for Falkon and Sphere, respectively. However, Falkon and data diffusion has been mostly tested within LANs, while Sector seems to be targeting WANs. Data diffusion has been architected to run in non-dedicated environments, where the resource pool (both storage and compute) varies based on load, provisioning resources on-demand, and releasing them when they are idle. Sector seems to be running on dedicated resources, and only handles decreasing the resource pool due to failures. Another important difference between Swift running over Falkon and data diffusion, as opposed to Sphere running over Sector, is the capability to run "black box" applications on distributed resources without any need to modify legacy applications, and access to files are done over POSIX read and write operations. Sphere and Sector seem to take the approach of MapReduce, in which applications are modified to support the read and write operations of applications.

Our work is motivated by the potential to improve application performance and even enable the ease of implementation of certain applications that would otherwise be difficult to implement with adequate performance. This sub-section covers an overview of a broad range of systems used to perform analysis on large datasets. The DIAL project that is part of the PPDG/ATLAS project focuses on the distributed interactive analysis of large datasets [114]. Chervenak et al. developed the Earth System Grid-I prototype to analyze climate simulation data using data Grid technologies [115]. The Mobius project developed a sub-project DataCutter for distributed processing of large datasets [116]. A database oriented view for data analysis is taken in the design of GridDB, a data-centric overlay for the scientific Grid [117]. Olson et al. discusses Grid service requirements for interactive analysis of large datasets [118]. Several large projects (Mobius [119] and ATLAS [120]) implemented their own data management systems to aid in the respective application's implementations.

With respect to provable performance results, several online competitive algorithms are known for a variety of problems in scheduling (see [141] for a survey) and for some problems in caching (see [142] for a survey), but there are none, to the best of our knowledge, that combine the two. The closest problem in caching is the two weight paging problem [143]; it allows for different page costs but assumes a single cache.

In summary, we have seen very little work that tries to combine data management and compute management to harness data locality down to the node level, and to do this in a dynamic environment that has the capability to expand and contract its resource pool. Data aware scheduling has typically been done at the site level (within Grids), or perhaps rack level (for MapReduce and Hadoop), but no work has addressed data-aware scheduling down to the node or processor core level. Exploiting data locality in access patterns is the key to enabling scalable storage systems to efficiently scale to petascale systems and beyond. Furthermore, most of other work lack the assumption that Grid systems are managed by batch schedulers, which can complicate the deployment of permanent data management infrastructure such as Google's GFS (or Hadoop's HDFS) and the GFarm file system, making them impractical to be operated in a non-dedicated environment at the user level. Another assumption of batch scheduled systems is the ability to run “black box” applications, an assumption that is not true for systems such as MapReduce, Hadoop, or Sphere.

Much of our work is pushing the limits of the traditional scientific computing environments which heavily rely on parallel file systems for application runtimes, which are generally separated from the compute resources via a high speed network. Our work strives to make better use of the local resources found on most compute nodes (i.e. local memory and disk) and to minimize the reliance on shared infrastructure (i.e. parallel file systems) that can hamper performance and scalability of data-intensive applications at scale.

3 Multi-Level Scheduling and Streamlined Task Dispatching

To enable the rapid execution of many tasks on compute clusters, we have developed Falkon, a Fast and Light-weight tasK executiON framework. Falkon integrates (1) multi-level scheduling to separate resource acquisition (via, e.g., requests to batch schedulers) from task dispatch, and (2) a streamlined dispatcher. Falkon's integration of multi-level scheduling and streamlined dispatchers delivers performance not provided by any other system. We describe Falkon architecture and implementation, and present performance results for both microbenchmarks and applications. Microbenchmarks show that Falkon throughput and scalability are one to two orders of magnitude better than other systems used in production Grids.

The results presented in this chapter have been published in [4].

3.1 Overview

Many interesting computations can be expressed conveniently as data-driven task graphs, in which individual tasks wait for input to be available, perform computation, and produce output. Systems such as DAGMan [36], Karajan [55], Swift [13], and VDS [121] support this model. These systems have all been used to encode and execute thousands of individual tasks.

In such task graphs, as well as in the popular master-worker model [122], many tasks may be logically executable at once. Such tasks may be dispatched to a parallel compute cluster or (via the use of grid protocols [123]) to many such clusters. The batch schedulers used to manage such clusters receive individual tasks, dispatch them to idle processors, and notify clients when execution is complete.

This strategy of dispatching tasks directly to batch schedulers has three disadvantages. First, because a typical batch scheduler provides rich functionality (e.g., multiple queues, flexible task dispatch policies, accounting, per-task resource limits), the time required to dispatch a task can be large—30 secs or more—and the aggregate throughput relatively low (perhaps two tasks/sec). Second, while batch schedulers may support different queues and policies, the policies implemented in a particular instantiation may not be optimized for many tasks. For example, a scheduler may allow only a modest number of concurrent submissions for a single user. Third, the average wait time of grid jobs is higher in practice than the predictions from simulation-based research. [124] These factors can cause problems when dealing with application workloads that contain a large number of tasks.

One solution to this problem is to transform applications to reduce the number of tasks. However, such transformations can be complex and/or may place a burden on the user. Another approach is to employ multi-level scheduling [92, 93]. A first-level request to a batch scheduler allocates resources to which a second-level scheduler dispatches tasks. The second-level scheduler can implement specialized support for task graph applications. Frey [50] and Singh [125] create an embedded Condor pool by "gliding in" Condor workers to a compute cluster, while MyCluster [95] can embed both Condor pools and Sun Grid Engine (SGE) clusters. Singh et al. [96, 94] report 50% reductions in execution time relative to a single-level approach.

We seek to achieve further improvements by:

1. Reducing task dispatch time by using a streamlined dispatcher that eliminates support for features such as multiple queues, priorities, accounting, etc.

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

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

Microbenchmarks show that Falkon can process millions of task and scale to 54,000 executors. A synthetic application demonstrates the benefits of adaptive provisioning. Finally, results for two applications demonstrate that substantial speedups can be achieved for real scientific applications.

3.2 Falkon Architecture

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

3.2.1 Execution Model

Each task is dispatched to a computational resource, selected according to the *dispatch policy*. If a response is not received after a time determined by the *replay policy*, or a failed response is received, the task is re-dispatched according to the dispatch policy (up to some specified number of retries). The *resource acquisition policy* determines when and for how long to acquire new resources, and how many resources to acquire. The *resource release policy* determines when to release resources.

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

Resource acquisition policy. This policy determines the number of resources, *n*, to acquire; the length of time for which resources should be requested; and the request(s) to generate to LRM(s) to acquire those resources. We have implemented five strategies that variously generate a single request for *n* resources, *n* requests for a single resource, or a series of arithmetically or exponentially larger requests, or that use system functions to determine available resources. The experiments reported in this chapter only considered the first policy ("all-at-once"), which allocates all needed resources in a single request.

Resource release policy. We distinguish between centralized and distributed resource release policies. In a *centralized* policy, decisions are made based on state information available at a central location. For example: “if there are no queued tasks, release all resources” or “if the number of queued tasks is less than *q*, release a resource.” In a *distributed* policy, decisions are made at individual resources based on state information available at the resource. For example: “if the resource has been idle for time *t*, the resource should release itself.” Note that resource acquisition and release policies are typically not independent: in most batch schedulers, a set of resources allocated in a single request must all be de-allocated before the requested resources become free and ready to be used by the next allocation. Ideally, one must release all resources obtained in a single request at once, which requires a certain level of synchronization among the resources allocated within a single allocation. In the experiments reported in this chapter, we used a distributed policy, releasing individual resources after a specified idle time was reached. In the future, we plan to improve our distributed policy by coordinating between all the resources allocated in a single request to de-allocate all at the same time.

3.2.2 Architecture

Falkon consists of a dispatcher, a provisioner, and zero or more executors (Figure 7). Figure 8 has the series of message exchanges that occur between the various Falkon components. As we describe the architecture and the components' interaction, we will denote the message numbers from Figure 8 in square braces; some messages have two numbers, denoting both a send and receive, while others have only a single number, denoting a simple send.

The dispatcher accepts tasks from clients and implements the dispatch policy. The provisioner implements the resource acquisition policy. Executors run tasks received from the dispatcher. Components communicate via Web Services (WS) messages (solid lines in Figure 8), except that notifications are performed via a custom TCP-based protocol (dotted lines). The notification mechanism is implemented over TCP because when we first implemented the core Falkon components using GT3.9.5, the Globus Toolkit did not support brokered WS notifications. The recent GT4.0.5 release supports brokered notifications.

The **dispatcher** implements the factory/instance pattern, providing a *create instance* operation to allow a clean separation among different clients. To access the dispatcher, a client first requests creation of a new instance, for which is returned a unique endpoint reference (EPR). The client then uses that EPR to submit tasks {1,2}, monitor progress (or wait for notifications {8}), retrieve results {9,10}, and (finally) destroy the instance.

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

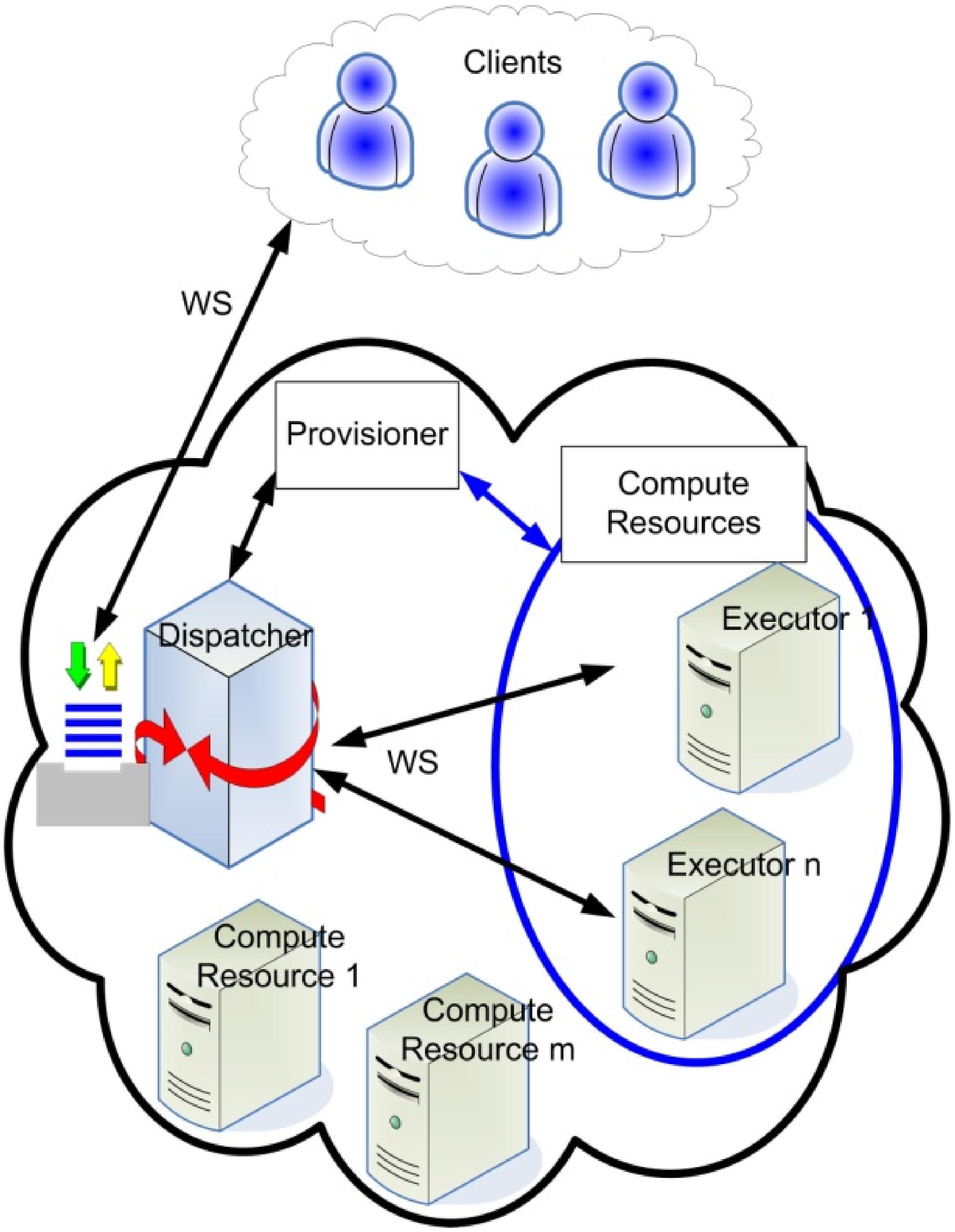


Figure 7: Falkon architecture overview

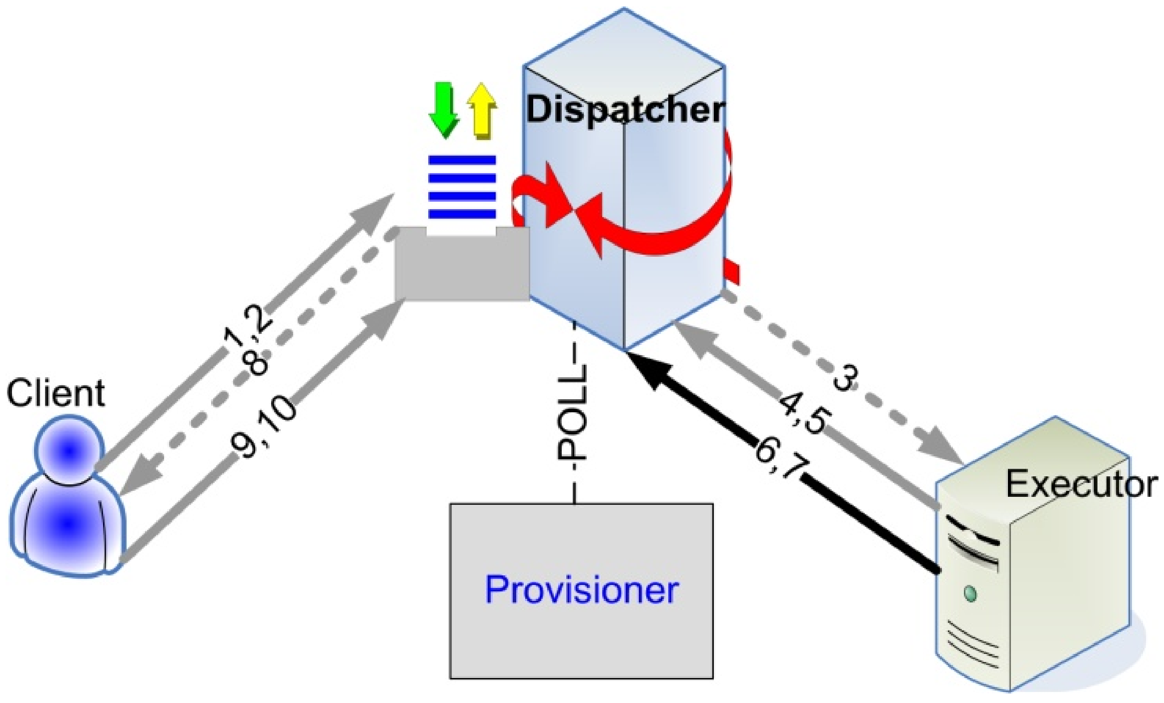


Figure 8: Falkon components and message exchange

The dispatcher runs within a Globus Toolkit 4 (GT4) [79] WS container, which provides authentication, message integrity, and message encryption mechanisms, via transport-level, conversation-level, or message-level security [126].

The **provisioner** is responsible for creating and destroying executors. It is initialized by the dispatcher with information about the state to be monitored and how to access it; the rule(s) under which the provisioner should create/destroy executors; the location of the executor code; bounds on the number of executors to be created; bounds on the time for which executors should be created; and the allowed idle time before executors are destroyed.

The provisioner periodically monitors dispatcher state {POLL} and, based on policy, determines whether to create additional executors, and if so, how many, and for how long. Creation requests are issued via GRAM4 [127] to abstract LRM details.

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

3.2.3 Push vs. Pull Model

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

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

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

1. It is implemented as a Web Service (as opposed to a simpler client that can only initiate WS communication). Thus, a WS container must be deployed on every compute node (in the absence of a shared file system); this alternative has the largest footprint but is easy to implement.
2. It supports notifications. Here, we only need the client code plus a few libraries required for WS communications. This alternative has a medium-sized footprint with a medium implementation complexity (WS and notification).
3. It uses a custom communication protocol and can be both a server and a client. This approach only needs the libraries to support that protocol (e.g., TCP). It has the smallest footprint but requires the implementation of the custom protocol.

All three approaches have problems with firewalls, but we have not found this to be a big issue in deployments to date, as the Dispatcher and Executors are typically located within a single site in which firewalls are not an issue. Chapter 4 addresses this problem via a three-tier architecture that supports both cross-firewall communications and communications with Executors operating in a private IP space.

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

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

3.2.4 Performance Enhancements

Communication costs can be reduced by *task bundling* between client and dispatcher and/or dispatcher and executors. In the latter case, problems can arise if task sizes vary and one executor gets assigned many large tasks, although that problem can be addressed by having clients assign each task an estimated runtime. We use client-dispatcher bundling in experiments described below, but (lacking runtime estimates) not dispatcher-executor bundling. Another technique that can reduce message exchanges is to *piggy-back* new task dispatches when acknowledging result delivery (messages {6,7} from Figure 8).

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

With client-dispatcher bundling and piggy-backing alone, we can reduce the number of messages to two per task (one message from executor to dispatcher to deliver a result, and one associated response from dispatcher to executor to acknowledge receipt and provide a new task); these two messages make up a single WS call. Line shading in Figure 8 shows where bundling optimization can be used: black lines denote that the corresponding message occurs on a per-task basis, while grey lines denote that through bundling optimizations, the corresponding messages occur for a set of tasks.

3.2.5 Ease of Use

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

3.3 Performance Evaluation

Table 2 lists the platforms used in experiments, in addition to those from Section 2.2.

Table 2: Platform descriptions

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

Latency between these systems was one to two milliseconds. We assume a one-to-one mapping between executors and processors in all experiments. Of the 162 nodes on TG_ANL_IA32 and TG_ANL_IA64, 128 were free for our experiments.

3.3.1 Throughput without Data Access

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

For purposes of comparison, we also tested GT4 performance with all security disabled. We created a simple service that incremented a counter for each WS call made to a counter service, and measured the number of WS calls per second that could be achieved from a varying number of machines. We claim this to be the upper bound on Falkon throughput performance that can be achieved on the tested hardware (UC_x64), assuming that there is no task bundling between dispatcher and executors, and that each task is handled via a separate dispatch.

Figure 9 shows GT4 without security achieves 500 WS calls/sec; Falkon reaches 487 tasks/sec (without security) and 204 tasks/sec (with security). A single Falkon executor without and with security can handle 28 and 12 tasks/sec, respectively.

We also measured Condor and PBS performance on the same testbed, with nodes managed by PBS v2.1.8. To measure PBS throughput, we submitted 100 short tasks (sleep 0) and measured the time to completion on the 64 available nodes. The experiment took on average 224 seconds for 10 runs netting 0.45 tasks/sec. As we did not have access to a dedicated Condor pool, we used MyCluster [95] to create a 64-node Condor v6.7.2 pool via PBS submissions. Once the 64 nodes were allocated from PBS and were available within MyCluster, we performed the same experiment, 100 short tasks over Condor. The total time was on average 203 seconds for 10 runs netting 0.49 tasks/sec. As far as we could tell, neither PBS nor Condor were using any security mechanisms between the various components within these systems. MyCluster does use authentication and authorization to setup the virtual cluster (a one time cost), but thereafter no security was used. It is also worth mentioning that we intentionally used a small number of tasks to test PBS and Condor as the achieved throughput drops as tasks accumulate in the wait queue, and our goal was to measure the best case scenario for their ability to dispatch and execute small tasks.

There are newer versions of both Condor and PBS, and both systems can likely be configured for higher throughput. We do not know whether or not these experiments reflect performance with security enabled or not, and all the details regarding the hardware used; see Table 3 for details on the various hardware used and a summary of the reported throughputs. In summary, Falkon's throughput performance compares favorably to all, regardless of the security settings used be these other systems.

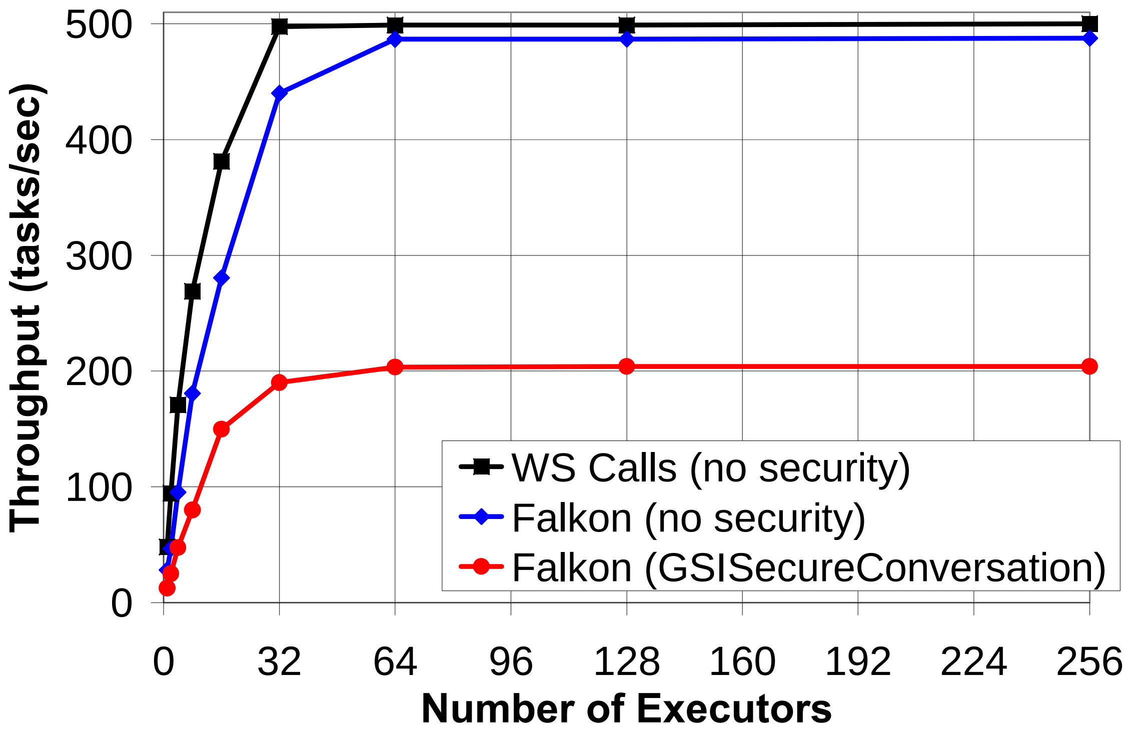


Figure 9: Throughput as function of executor count

Table 3: Measured and cited throughput for Falkon, Condor, and PBS

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

3.3.2 Throughput with Data Access

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

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

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

As data sizes increase, throughput (Mb/sec: dotted lines) plateaus at either 1 MB or 10 MB data sizes, depending on the experiment. GPFS read+write peaks at 326 Mb/sec, GPFS read at 3,067 Mb/sec, LOCAL read+write at 32,667 Mb/sec, and LOCAL read at

52,015 Mb/sec. With 1 GB data, throughput was 0.04 tasks/sec, 0.4 tasks/sec, 4.28 tasks/sec, and 6.81 tasks/sec, respectively.

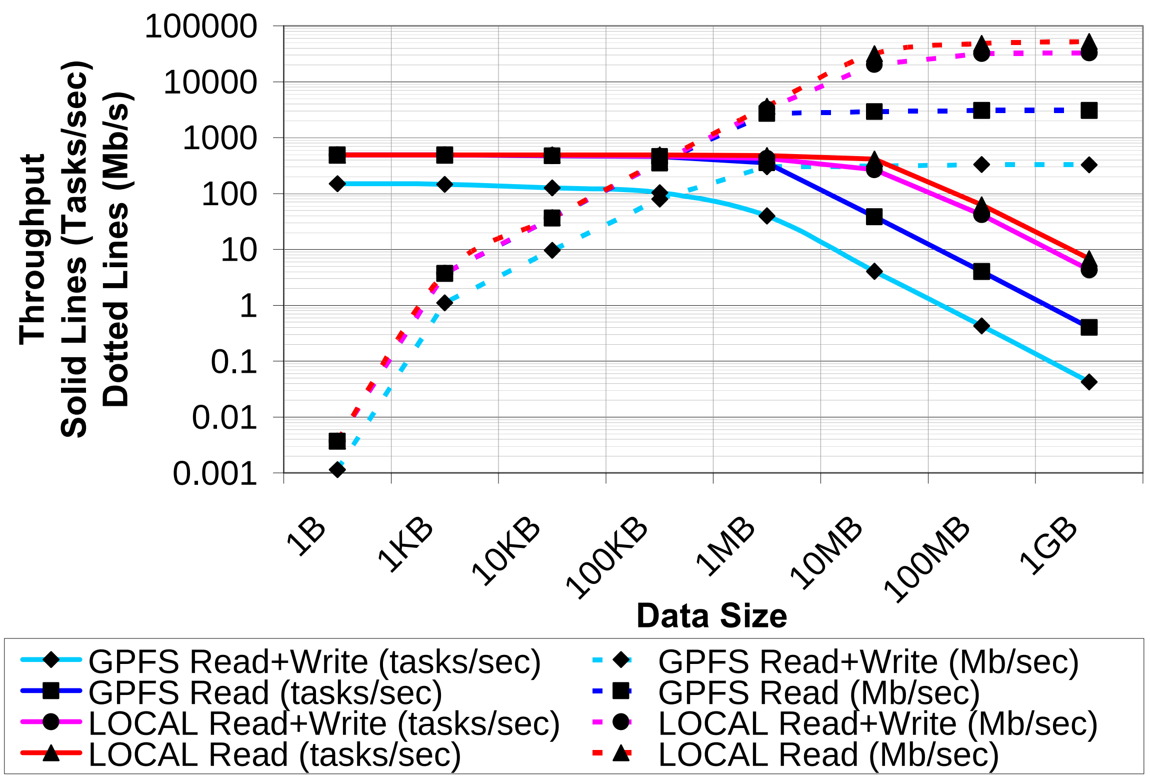


Figure 10: Throughput as a function of data size on 64 nodes

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

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

3.3.3 Bundling

It has been shown that real grid workloads comprise a large percentage of tasks submitted as batches of tasks. [128] In order to optimize the task submission performance, we propose to bundle many tasks together in each submission. We measured performance for a workload of "sleep 0" tasks as a function of task bundle size. Figure 11 shows that performance increases from about 20 tasks/sec, without bundling, to a peak of almost 1500 tasks/sec, with bundling.

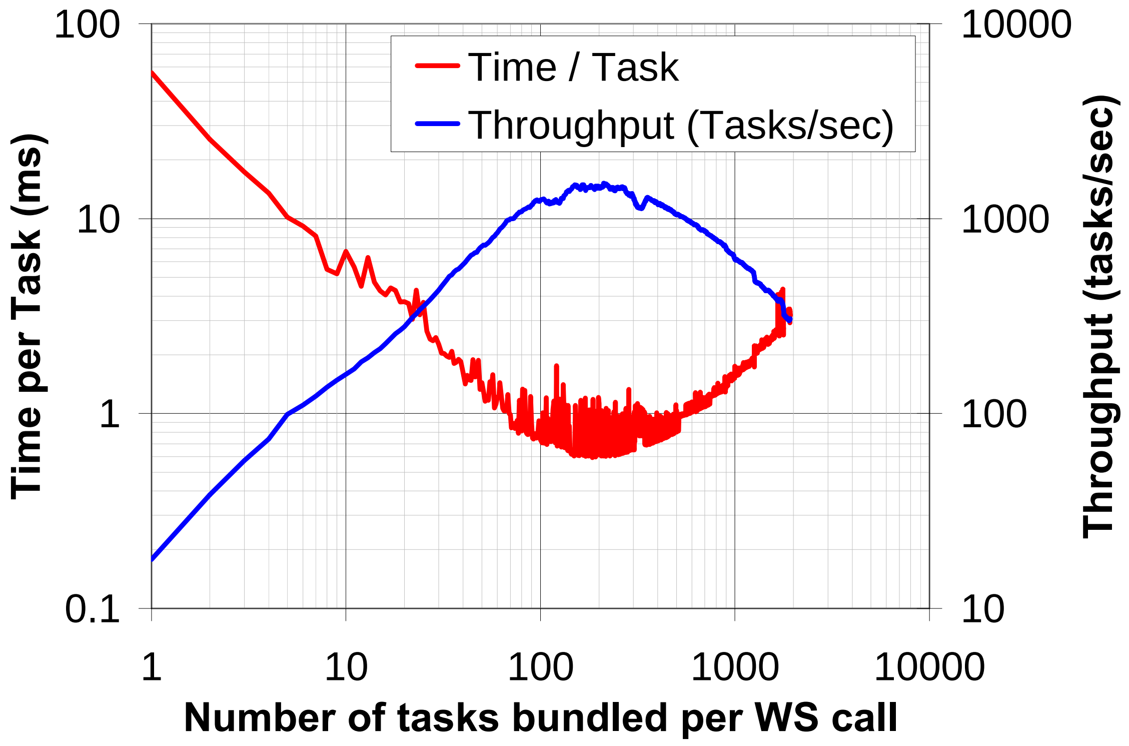


Figure 11: Bundling throughput and cost per task

Performance decreases after around 300 tasks per bundle. We attribute this drop to the array data structure implementation in the Axis software that GT4 uses to handle XML serialization and de-serialization. (Axis implements the array data-structure used to store the representation of the bundled tasks as a grow-able array, copying to a new bigger array each time its size increases.) We will investigate this inefficiency to attempt to remedy this limitation.

3.3.4 Efficiency and Speedup

Figure 12 shows efficiency ($E_P=S_P/P$) as a function of number of processors (P) and task length; speedup is defined as $S_P=T_1/T_P$, where T_n is the execution time on n processors. These experiments were conducted on TG_ANL_IA32 and TG_ANL_IA64 with no security and with optimizations such as bundling and "piggy-backing" enabled.

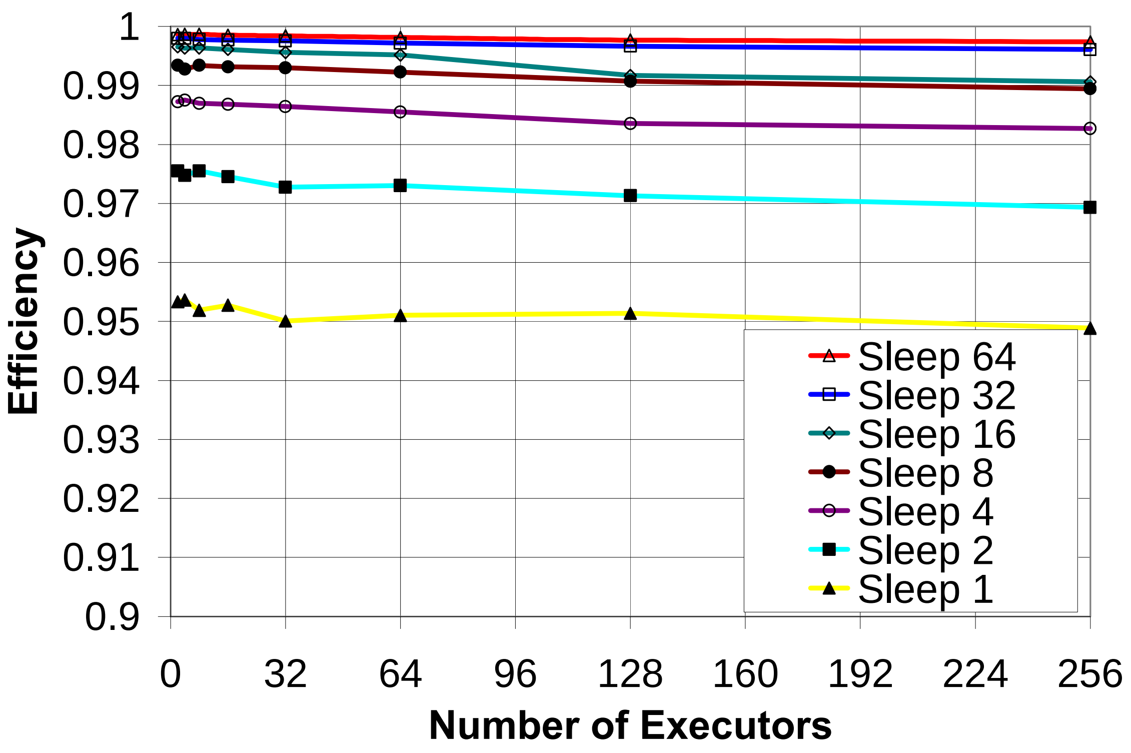


Figure 12: Efficiency for various task length and executors

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

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

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

As both the tested PBS and Condor versions that are in production on the TG_ANL_IA32 and TG_ANL_IA64 clusters are not the latest versions, we also derived the efficiency curve for Condor version 6.9.3, the latest development Condor version, which is claimed to have a throughput of 11 tasks/sec [129] (up from our measured 0.45~0.49 tasks/sec and the 2 tasks/sec reported by others [83]). Efficiency is much improved, reaching 90%, 95%, and 99% for task lengths of 50, 100, and 1000 secs. respectively.

The results in Figure 13 for Condor v6.9.3 are derived, not measured. We derived based on the achieved throughput cited in [129] of 11 tasks/sec for sleep 0 tasks. Essentially, we computed the per task overhead of 0.0909 seconds, which we could then add to the ideal time of each respective task length to get an estimated task execution time. With this execution time, we could compute speedup, which we then used to compute efficiency. Our derivation of efficiency is simplistic, but it allowed us to plot the likely efficiency of the latest development Condor code against the older production Condor code, the PBS production code, and Falkon. It should be noted that Figure 13 illustrates the efficiency of these systems for a relatively small set of resources (only 64 processors), and that the efficiency gap will likely only increase as the number of resources increases.

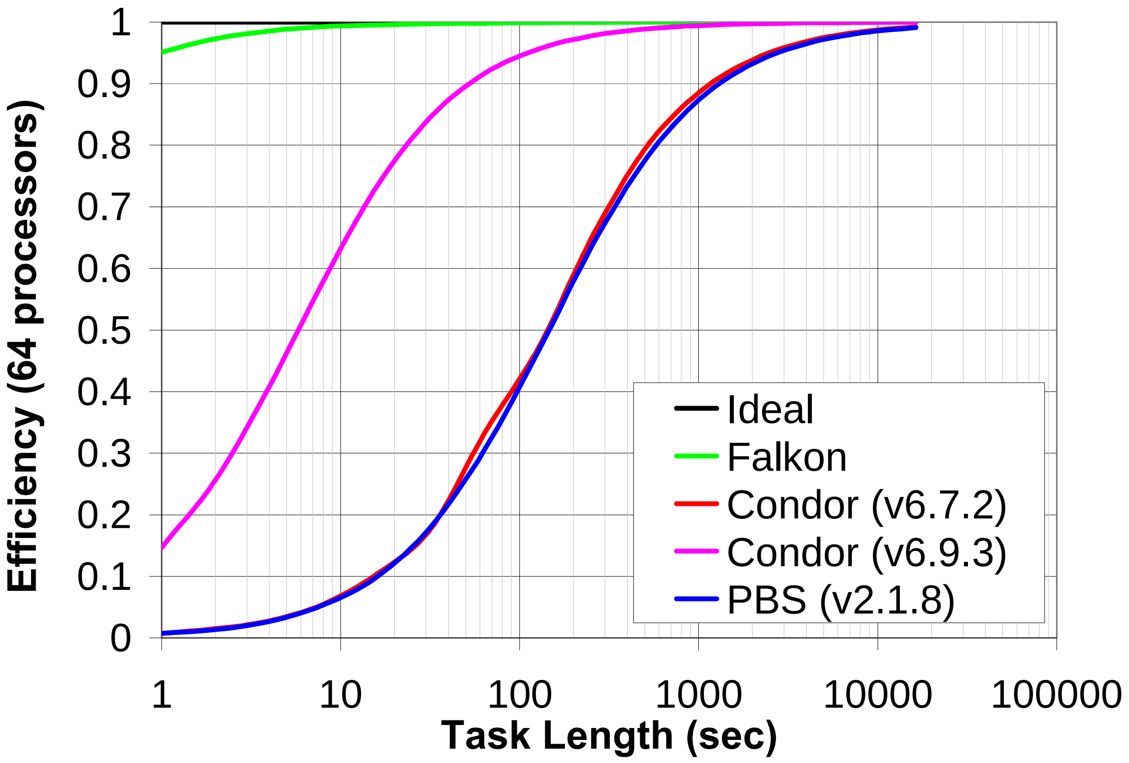


Figure 13: Efficiency of resource usage for varying task lengths on 64 processors comparing Falkon, Condor and PBS

3.3.5 Scalability

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

Our first experiment studies Falkon's behavior as the task queue increases in length. We constructed a client that submits two million "sleep 0" tasks to a dispatcher configured with a Java heap size set to 1.5GB. We created 64 executors on 32 machines from TG_ANL_IA32 and ran the dispatcher on UC_x64 and the client on TP_UC_x64.

Figure 14 results show the entire run over time. The solid black line is the instantaneous queue length, the light blue dots are raw samples (once per sec) of achieved throughput in terms of task completions, and the solid blue line is the moving average (over 60 sample intervals, and thus 60 secs) of raw throughput. Average throughput was 298 tasks/sec. Note the slight increase of about 10~15 tasks/sec when the queue stopped growing, as the client finished submitting all two million tasks.

The graph shows the raw throughput samples (taken at 1 second intervals) to be between 400 and 500 tasks per second for the majority of the experiment, yet the moving average was around 300 tasks/sec. A close analysis shows frequent raw throughput samples at 0 tasks/sec, which we attribute to JVM garbage collection. We may be able to reduce this variation by configuring the JVM to garbage collect more frequently. These results are from an early version of our implementation, which has been improved significantly. Performance results from the latest version of Falkon can be found in Chapter 4.

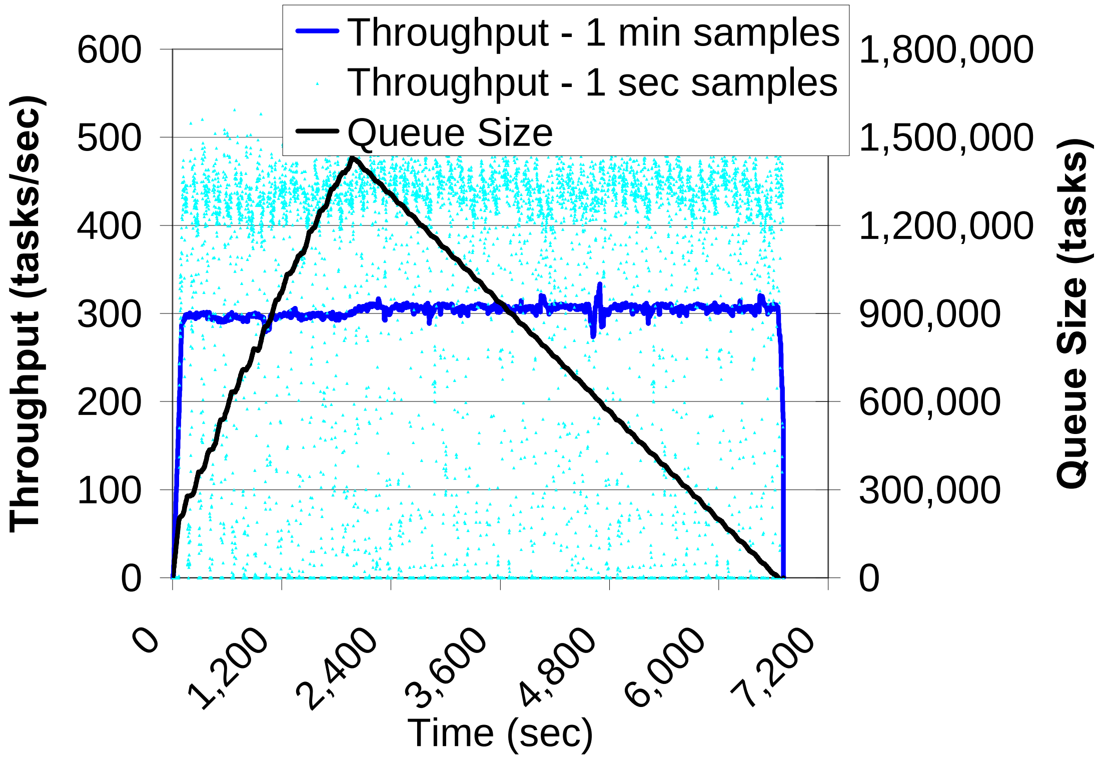


Figure 14: Long running test with 2M tasks

In a second experiment, we tested how many executors the dispatcher could handle. We did not have an available system large enough to test the limits of the Falkon implementation, and therefore we ran multiple executors on each physical machine emulating a larger number of virtual executors. Others have used this experimental method with success [83].

We performed our experiment on TP_UC_x64, on which we configured one dispatcher machine, one client machine, and 60 machines to run executors. We ran 900 executors (split over four JVMs) on each machine, for a total of 900x60=54,000 executors. Once we started up the system and all 54K executors registered and were ready to receive work, we started the experiment consisting of 54K tasks of "sleep 480 secs." For this experiment, we disabled all security, and only enabled bundling between the client and the dispatcher. Note that piggy-backing would have made no difference as each executor only processed one task each. More recently, we have been able to run Falkon at extremely large scale, on 160K executors on 160K real processors on the IBM Blue Gene/P supercomputer; more on these results can be found in Chapter 4.

Figure 15 shows that the dispatch rate (green line) equals the submit rate. The black line shows the number of busy executors, which increases from 0 to 54K in 408 secs. As soon as the first task finishes after 480 secs (the task length), results start to be delivered to the client at about the same rate as they were submitted and dispatched. Overall throughput (including ramp up and ramp down time) was about 60 tasks/sec.

We also measured task overhead, by which we mean the time it takes an executor to create a thread to handle the task, pick up a task via one WS call, perform an Java exec on the specified command ("sleep 480"), and send the result (the exit return code) back via one WS call, minus 480 secs (the task run time). Figure 16 shows per task overhead in millisecs for each task executed in the experiment of Figure 15, ordered by task start time.

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

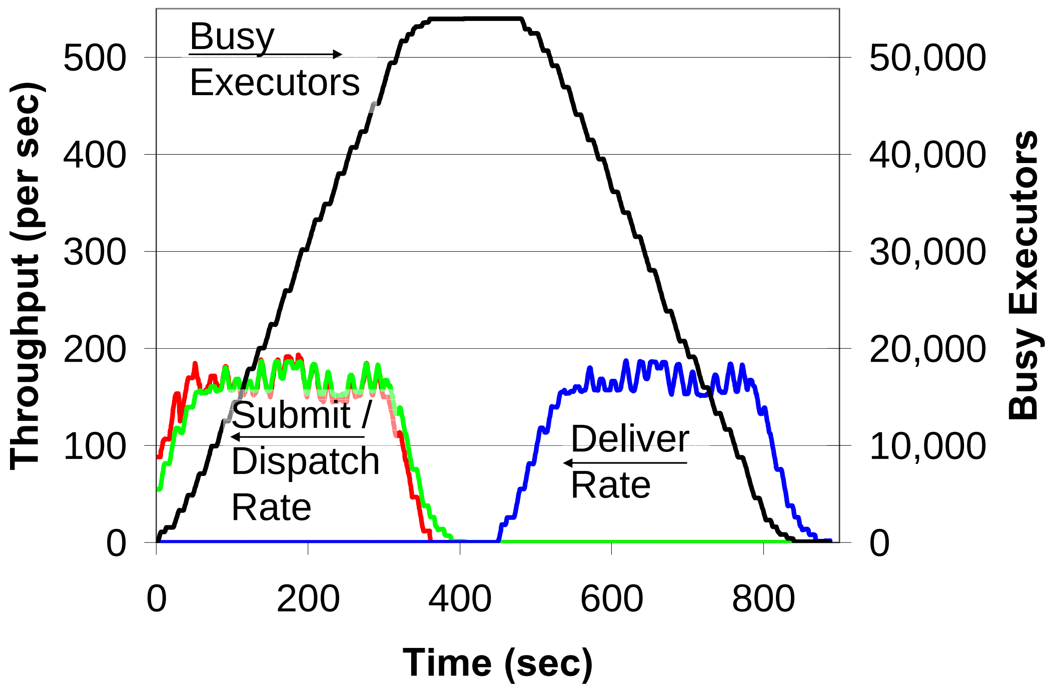


Figure 15: Falkon scalability with 54K executors

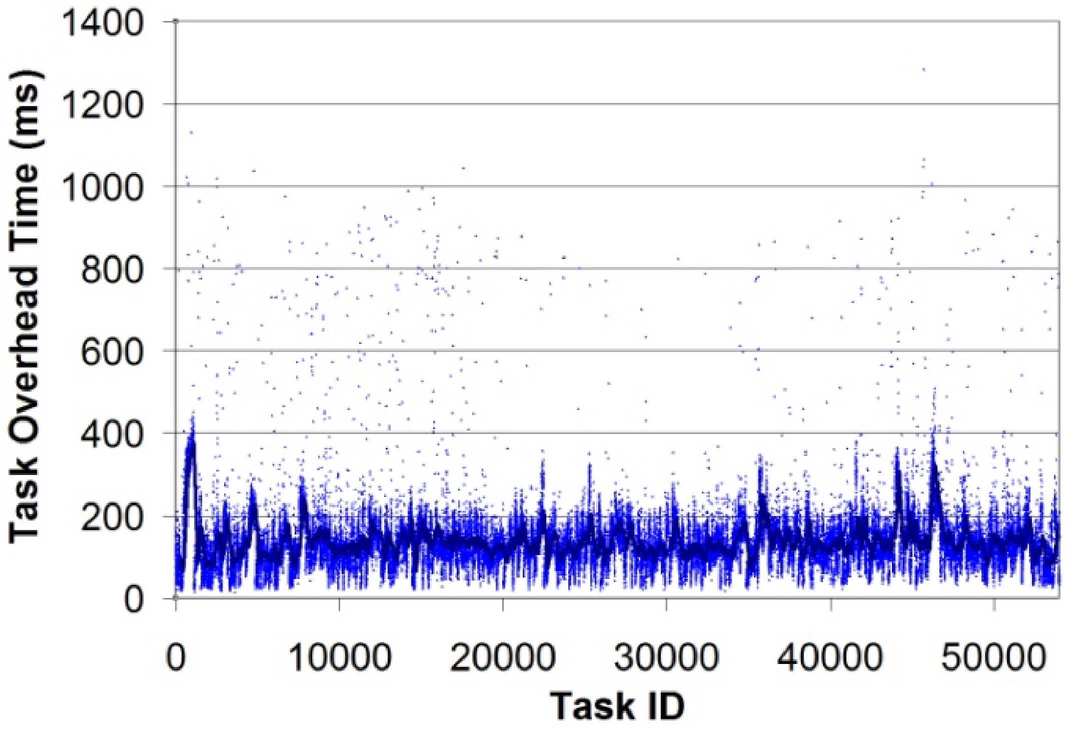


Figure 16: Task overhead with 54K executors

3.4 Conclusion

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

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

Microbenchmarks show that Falkon can achieve one to two orders of magnitude higher throughput (487 tasks/sec) when compared to other batch schedulers. It can sustain high throughput with up to 54,000 managed executors and can process 2,000,000 tasks in 112 minutes, operating reliably with queue lengths exceeding 1,500,000 tasks.

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

4 Distributing the Falkon Architecture to Support Petascale Systems

We have extended the Falkon lightweight task execution framework to make loosely coupled programming on petascale systems a practical and useful programming model. This work studies and measures the performance factors involved in applying this approach to enable the utilization of petascale systems by a broader user community, and with greater ease. Our work enables the execution of highly parallel computations composed of loosely coupled serial jobs with no modifications to the respective applications. This approach allows new—and potentially far larger—class of applications to leverage petascale systems, such as the IBM Blue Gene/P supercomputer. We present the challenges of I/O performance encountered in making this model practical, and show results using both micro-benchmarks and real applications from two domains, economic energy modeling and molecular dynamics. Our benchmarks show that we can scale up to 160K processor cores with high efficiency, and can achieve thousands of tasks/sec sustained execution rates.

The results presented in this chapter have been published in [2].

4.1 Overview

Emerging petascale computing systems, such as IBM's Blue Gene/P [27], incorporate high-speed, low-latency interconnects and other features designed to support tightly coupled parallel computations. The majority of the applications run on these computers have a single program multiple data (SMPD) structure, and are commonly implemented by using the Message Passing Interface (MPI) to achieve the needed inter-process communication.

We want to enable the use of these systems for task-parallel applications, which are linked into useful workflows through the looser task-coupling model of passing data via files between dependent tasks. This potentially larger class of task-parallel applications is precluded from leveraging the increasing power of modern parallel systems due to the lack of efficient support in those systems for the "scripting" programming model. With advances in e-Science and the growing complexity of scientific analyses, more scientists and researchers rely on various forms of scripting to automate end-to-end application processes involving task coordination, provenance tracking, and bookkeeping. Their approaches are typically based on a model of loosely coupled computation, in which data is exchanged among tasks via files, databases or XML documents, or a combination of these. Furthermore, with technological advances in both scientific instrumentation and simulation, the volume of scientific datasets is growing exponentially. This vast increase in data volume combined with the growing complexity of data analysis procedures and algorithms have rendered traditional manual processing and exploration unfavorable as compared with modern high performance computing processes automated by scientific workflow systems. [18]

We claim that MTC applications can be executed efficiently on today's supercomputers; this chapter provides empirical evidence to prove our hypothesis. The chapter also describes the set of problems that must be overcome to make loosely coupled programming practical on emerging petascale architectures: local resource manager scalability and granularity, efficient utilization of the raw hardware, shared file system contention, and application scalability. We address these problems, and identify the remaining challenges that need to be overcome to make loosely coupled supercomputing a practical reality. Through our work, we have enabled a Blue Gene/P to efficiently support loosely coupled parallel programming without any modifications to the respective applications (except for recompilation), enabling the same applications that execute in a distributed grid environment to be run efficiently on a supercomputer. The Blue Gene/P that we refer to is the new IBM Blue Gene/P supercomputer (also known as Intrepid) at the U.S. Department of Energy's Argonne National Laboratory (ANL), which is ranked number 3 in the Top500 rankings [38] with 160K processor-cores with a Rpeak of 557 TF and Rmax of 450 TF. Throughout this chapter, we will use the term Blue Gene/P to denote the specific Blue Gene/P named Intrepid from ANL.

We validate our hypothesis by testing and measuring two systems, Swift [13, 55] and Falkon [4], which have been used to execute large-scale loosely coupled applications on clusters and grids. We present results for both micro-benchmarks and real applications executed on the Blue Gene/P. Micro-benchmarks show that we can scale to 160K processor-cores with high efficiency, and can achieve sustained execution rates of thousands of tasks per second. We also investigated two applications from different domains, economic energy modeling and molecular dynamics, and show excellent application scalability, speedup and efficiency as they scale to 128K cores. Note that for the remainder of this chapter, we will use the terms processors, CPUs, and cores interchangeably to mean the same thing.

4.2 Requirements and Implementation

The contribution of this work is the ability to enable a new class of applications—large-scale loosely coupled—to efficiently execute on petascale systems which are traditionally HPC systems. This is accomplished primarily through three mechanisms: 1) multi-level scheduling, 2) efficient task dispatch, and 3) extensive use of caching to avoid shared infrastructure (e.g. file systems and interconnects).

Multi-level scheduling is essential on a system such as the Blue Gene/P because the local resource manager (LRM, Cobalt [54]) works at a granularity of psets [130], rather than individual computing nodes or processor cores. On the Blue Gene/P, a pset is a group of 64 quad-core compute nodes and one I/O node. Psets must be allocated in their entirety to user application jobs by the LRM, which imposes the constraint that the applications must make use of all 256 cores. Tightly coupled MPI applications are well suited for this constraint, but loosely coupled applications generally have many single processor jobs, each with possibly unique executables and parameters. Naively running such applications on the Blue Gene/P using the system's Cobalt LRM would yield a utilization of 1/256. We use multi-level scheduling to allocate compute resources from Cobalt at the pset granularity, and then make these resources available to applications at a single processor core granularity. Using this multi-level scheduling mechanism, we are able to launch a unique application, or the same application with unique arguments, on each core, and to launch such tasks repetitively throughout the allocation period. This capability is made possible through Falkon [4] and its resource provisioning mechanisms.

A related obstacle to loosely coupled programming when using the native Blue Gene/P LRM is the overhead of scheduling and starting resources. The Blue Gene/P compute nodes are powered off when not in use and must be booted when allocated to a job. As the compute nodes do not have local disks, the boot up process involves reading the lightweight IBM compute node kernel (or Linux-based ZeptoOS [82]) kernel image from a shared file system, which can be expensive if compute nodes are allocated and de-allocated frequently. Using multi-level scheduling allows this high initial cost to be amortized over many jobs, reducing it to an insignificant overhead. With the use of multi-level scheduling, executing a job is reduced to its bare and lightweight essentials: loading the application into memory, executing it, and returning its exit code – a process that can occur in milliseconds. We contrast this with the cost to reboot compute nodes, which is on the order of multiple seconds (for a single node) and can be as high as a thousand seconds in the case of 40K nodes all booting at the same time (see Figure 19).

The second mechanism that enables loosely coupled applications to be executed on the Blue Gene/P is a streamlined task submission framework (Falkon [4]). Falkon relies on LRMs for many functions (e.g., reservation, policy-based scheduling, accounting) and client frameworks such as workflow systems or distributed scripting systems for others (e.g., recovery, data staging, job dependency management). This specialization allows it to achieve several orders of magnitude higher performance (3773 tasks/sec in a Linux cluster environment, 3057 tasks/sec on the SiCortex, and 3071 tasks/sec on the Blue Gene/P—compared to 0.5 to 22 jobs per second for traditional LRMs such as Condor [36] and PBS [64]) [4]. These high throughputs are critical in running large number of tasks on many processors as efficiently as possible. For example, running many 60-second tasks on 160K processors on the Blue Gene/P requires that we sustain an average throughput of 2730 tasks/sec; considering the best LRM performance of 22 tasks/sec [83], we would need 2 hour long tasks to get reasonable efficiency.

The third mechanism we employ for enabling loosely coupled applications to execute efficiently on the Blue Gene/P is extensive caching of application data to allow better application scalability by avoiding shared file systems. As workflow systems frequently employ files as the primary communication medium between data-dependent jobs, having efficient mechanisms to read and write files is critical. The compute nodes on the Blue Gene/P do not have local disks, but they have both a shared file system (GPFS [46]) and local file system implemented in RAM ("ramdisk"). We make extensive use of the ramdisk local file system, to cache files such as application scripts and binary executables, static input data which is constant across many jobs running an application, and in some cases, output data from the application until enough data is collected to allow efficient writes to the shared file system. We found that naively executing applications directly against GPFS yielded unacceptably poor performance, but with successive levels of caching we were able to increase the execution efficiency to within a few percent of ideal.

The caching we refer to in this chapter is a different mechanism than the data diffusion described in [3, 19, 1], which is covered in Chapter 6. Data diffusion deals with dynamic data caching and replication, as well as with data-aware scheduling. Due to the network topology of the Blue Gene/P, and the architecture changes in which we distributed the Falkon dispatcher, where compute nodes are grouped into private networks per pset (in groups of 256 CPUs), we have not been able to use data diffusion in its current form on the Blue Gene/P. Despite the simple caching scheme we have employed on the Blue Gene/P, it has proved to be quite effective in scaling applications up to 128K processors (while the same applications and workloads didn't scale well beyond 8K processors). Note that caching is done completely automated, via a wrapper script that we facilitate to the application.

4.2.1 Swift and Falkon

To harness a wide array of loosely coupled applications that have already been implemented and executed in clusters and grids, we build upon the Swift [13, 10] and Falkon [4] systems. Swift enables scientific workflows through a data-flow-based functional parallel programming model. It is a parallel scripting tool for rapid and reliable specification, execution, and management of large-scale science and engineering workflows. The runtime system in Swift relies on the CoG Karajan [57] workflow engine for efficient scheduling and load balancing, and it integrates with the Falkon light-weight task execution dispatcher for optimized task throughput and efficiency.

Swift and Falkon have been used in a variety of environments from clusters, to multi-site Grids (e.g., Open Science Grid [45], TeraGrid [44]), to specialized large machines (SiCortex [56]), to supercomputers (e.g., Blue Gene/P [27]). Large-scale applications from many domains (e.g., astronomy [58, 4], medicine [59, 4, 60], chemistry [10], molecular dynamics [61], and economics [62, 63]) have been run at scales of up to millions of tasks on up to hundreds of thousands of processors.

4.2.2 Implementation Details

Significant engineering efforts were needed to get Falkon and Swift to work on systems such as the Blue Gene/P. This section discusses extensions we made to both systems, and the problems and bottlenecks they addressed.

Static Resource Provisioning: When using static resource provisioning, applications requests a number of processors for a fixed duration directly from the Cobalt LRM. For example, the following command "falkon-start-bgp-ram.sh prod 1024 60" submits a single job to Cobalt to the "prod" queue and asks for 1024 nodes (4096 processors) for 60 minutes; once the job goes into a running state and the Falkon framework is bootstrapped, applications interact directly with Falkon to submit single processor tasks for the duration of the allocation.

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. [79] The Java-only implementation works well in typical Linux clusters and Grids, but the lack of Java on the Blue Gene/L, Blue Gene/P, and SiCortex prompted us to re-implement some functionality in C. Table 4 has a summary of the differences between the two implementations.

Table 4: Feature comparison between the Java and C Executor implementations

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

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. In the future, we will consider switching the TCP-based proprietary protocol back to a WS-based one for the C implementation. It was not sufficient to change the worker implementation, as the service required corresponding revisions. In addition to the existing support for WS-based protocol, we implemented a new component called "TCPCore" to handle the TCP-based communication protocol (see Figure 17).

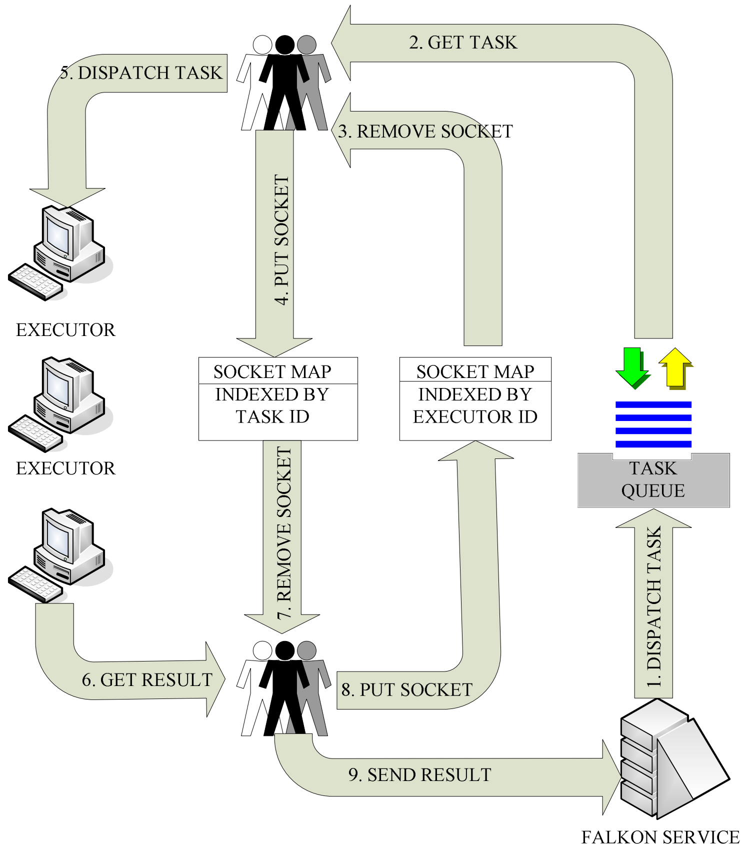


Figure 17: The TCPCore overview, replacing the GT4 WS-Core component

TCPCore is a component to manage a pool of threads that lives in the same JVM as the Falkon service, and uses in-memory notifications and shared objects to communicate with the Falkon service. In order to make the protocol as efficient as possible, we implemented persistent TCP sockets (which are stored in a hash table based on executor ID or task ID, depending on what state the task is in).

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 18 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 MTC applications.

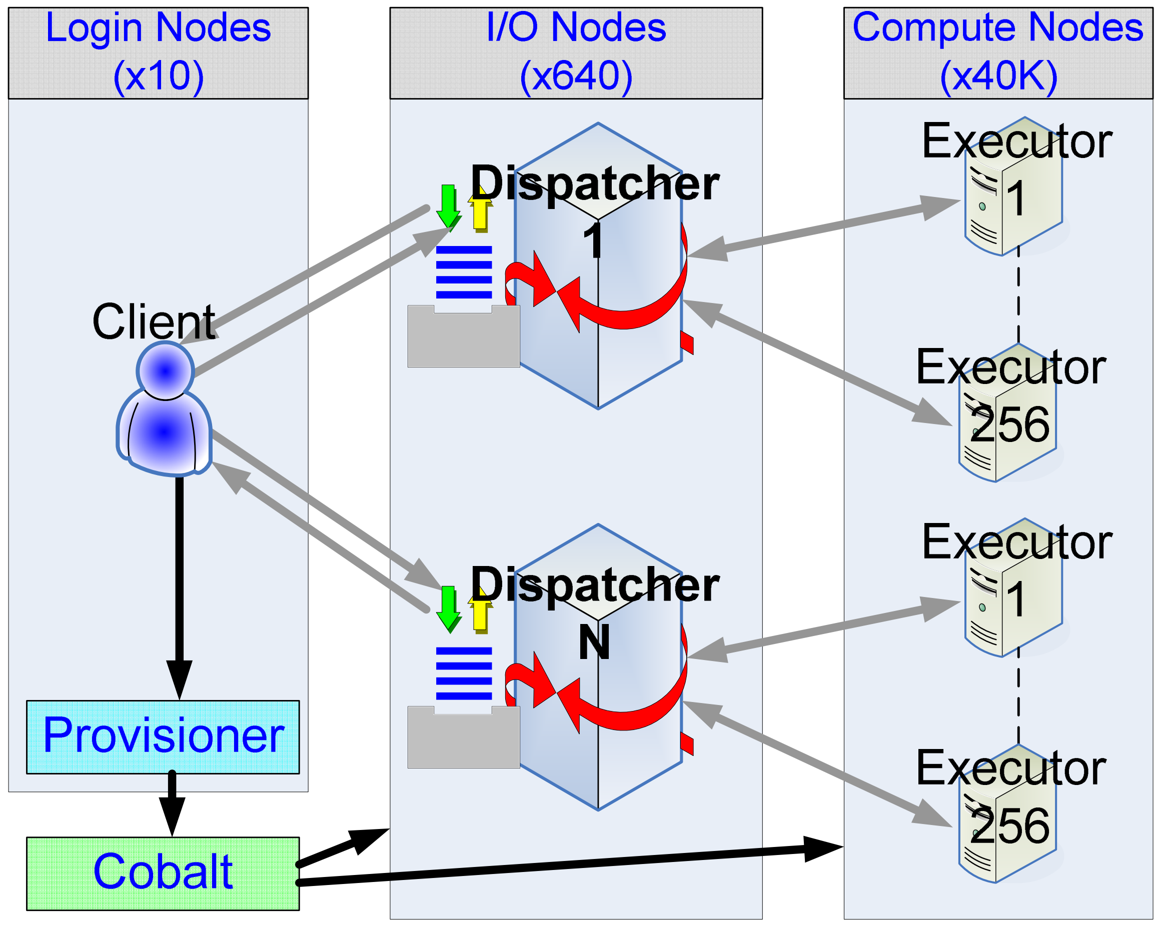


Figure 18: 3-Tier Architecture Overview

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. The most challenging architecture change was the additional client-side functionality to communicate and 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 solved both our scalability problems to 160K processors and in reducing 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 [26], 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 Micro-Benchmarks Performance

We use micro-benchmarks to determine performance characteristics and potential bottlenecks on systems with many cores. We measure startup costs, task dispatch rates, and costs for various file system operations (read, read+write, invoking scripts, mkdir, etc) on the shared file systems (GPFS) that we use when running large-scale applications.

4.3.1 Startup Costs

Our first micro-benchmark attempts to capture the incremental costs involved (see Figure 19) in variously booting the Blue Gene/P at various scales (red), starting the Falkon framework (green), and initializing the Falkon framework so it is ready to process tasks (blue). On a single pset (256 processors), it takes 125 seconds to prepare Falkon to process the first task; on the full 160K processors, it takes 1326 seconds. At the smallest scale, starting and initializing the Falkon framework constitutes 31% of the total time, but at large scales, the boot time starts to dominate and on 160K nodes the Falkon framework takes only 17% of total time.

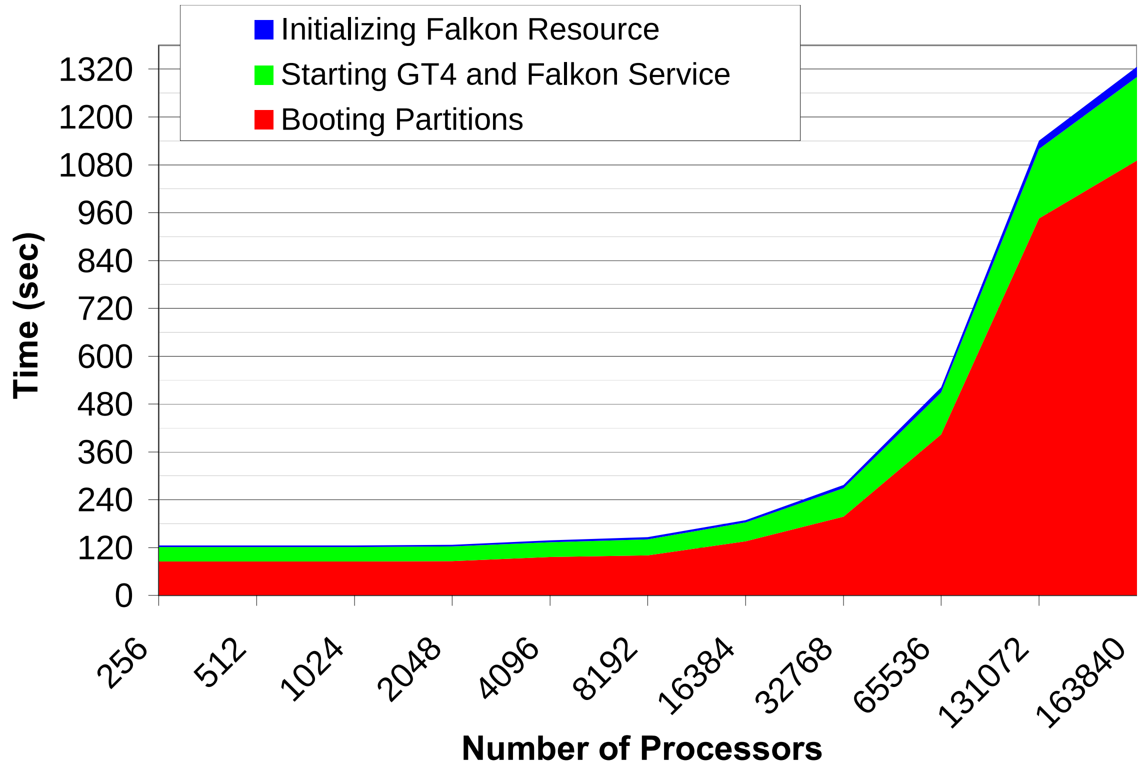


Figure 19: Startup costs in booting the Blue Gene/P, starting the Falkon framework, and initializing Falkon

We examine where the 1090 seconds is spent when booting ZeptOS on 160K nodes. The largest part of this time (708 seconds) is spent mounting GPFS. The next big block of time is the sending of the kernels and ramdisks to the compute and I/O nodes, which takes 213 seconds. Mounting NFS takes 55 seconds. Starting various services from NFS, such as SSH, takes 85 seconds. These costs account for over 97% of the 1090 seconds required to boot the Blue Gene/P.

4.3.2 Falkon Task Dispatch Performance

One key component to achieving high utilization of large-scale systems is achieving high task dispatch and execute rates. Figure 20 has a summary of the dispatch rates Falkon achieved on various testbeds. In Chapter 3, we reported that Falkon with a Java Executor and WS-based communication protocol achieves 487 tasks/sec in a Linux cluster (ANL/UC) 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 (ANL/UC Linux cluster, SiCortex, and Blue Gene/P) for both versions of the executor (Java and C, WS-based and TCP-base respectively) at significantly larger scales. We achieved 2534 tasks/sec (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). The throughput numbers that indicate “1 dispatcher” are tests done with the original centralized architecture. The last throughput of 3071 tasks/sec was achieved with the dispatchers distributed over 640 I/O nodes, each managing 256 processors.

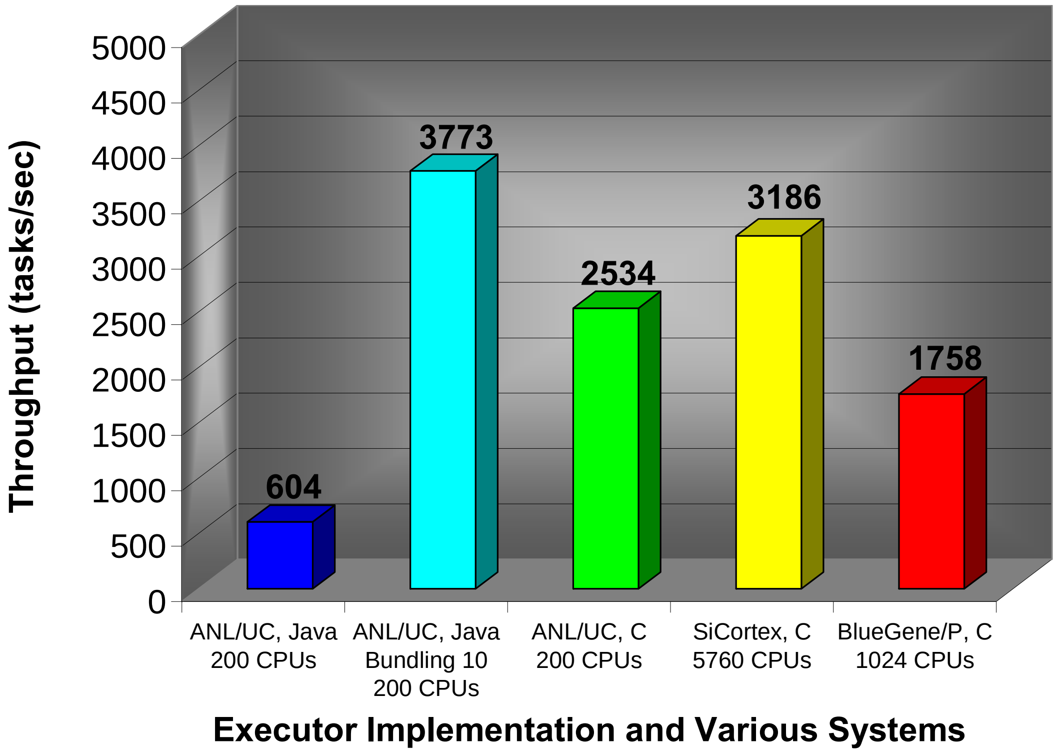
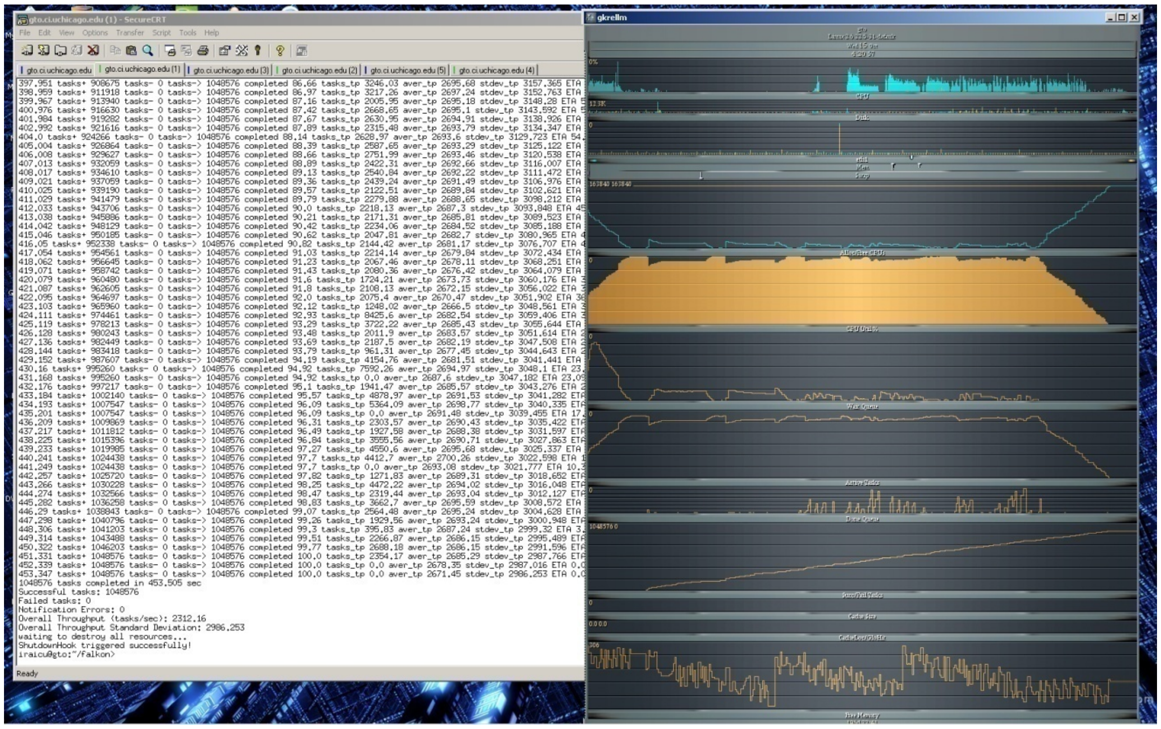


Figure 20: Task dispatch and execution throughput for trivial tasks with no I/O (sleep 0)

In order to make visualizing the state of Falkon easier, we have formatted various Falkon logs to be printed in a specific format that can be read by the GKrellm [131] monitoring GUI to display real time state information. Figure 21 shows 1 million tasks (sleep 60) executed on 160K processors on the IBM Blue Gene/P supercomputer. Overall, it took 453 seconds to complete 1M tasks, with an ideal time being 420 seconds, achieving 93% efficiency. To place this benchmark in context, of what an achievement it is to be able to run 1 million tasks in 7.5 minutes, others [132] have managed to run 1 million jobs in 6 months. Grant it that the 1 million jobs they referred to in [132] were real computations with real data, and not just “sleep 60” tasks, due to the large overheads of scheduling jobs through Condor [36] and other production local resource managers, running 1 million jobs, no matter how short they are, will likely still take on the order of days (not minutes as is the case with Falkon).



```
397,951 tasks+ 908675 tasks- 0 tasks-> 1048576 completed 86.66 tasks_tp 3246.03 aver_tp 2695.68 stdev_tp 3157.365 ETA
398,959 tasks+ 911918 tasks- 0 tasks-> 1048576 completed 86.97 tasks_tp 3217.26 aver_tp 2697.24 stdev_tp 3152.763 ETA
399,967 tasks+ 913940 tasks- 0 tasks-> 1048576 completed 87.16 tasks_tp 2005.95 aver_tp 2695.18 stdev_tp 3148.28 ETA 5
400,976 tasks+ 916630 tasks- 0 tasks-> 1048576 completed 87.42 tasks_tp 2668.65 aver_tp 2695.1 stdev_tp 3143.592 ETA 5
401,984 tasks+ 919282 tasks- 0 tasks-> 1048576 completed 87.67 tasks_tp 2630.95 aver_tp 2694.91 stdev_tp 3138.926 ETA
402,992 tasks+ 921616 tasks- 0 tasks-> 1048576 completed 87.89 tasks_tp 2315.48 aver_tp 2693.79 stdev_tp 3134.347 ETA
404.0 tasks+ 924266 tasks- 0 tasks-> 1048576 completed 88.14 tasks_tp 2628.97 aver_tp 2693.6 stdev_tp 3129.723 ETA 54.
405,004 tasks+ 926864 tasks- 0 tasks-> 1048576 completed 88.39 tasks_tp 2587.65 aver_tp 2693.29 stdev_tp 3125.122 ETA
406,008 tasks+ 929627 tasks- 0 tasks-> 1048576 completed 88.66 tasks_tp 2751.99 aver_tp 2693.46 stdev_tp 3120.538 ETA
407,013 tasks+ 932059 tasks- 0 tasks-> 1048576 completed 88.89 tasks_tp 2422.31 aver_tp 2692.66 stdev_tp 3116.007 ETA
408,017 tasks+ 934610 tasks- 0 tasks-> 1048576 completed 89.13 tasks_tp 2540.84 aver_tp 2692.22 stdev_tp 3111.472 ETA
409,021 tasks+ 937059 tasks- 0 tasks-> 1048576 completed 89.36 tasks_tp 2439.24 aver_tp 2691.49 stdev_tp 3106.976 ETA
410,025 tasks+ 939190 tasks- 0 tasks-> 1048576 completed 89.57 tasks_tp 2122.51 aver_tp 2689.84 stdev_tp 3102.621 ETA
411,029 tasks+ 941479 tasks- 0 tasks-> 1048576 completed 89.79 tasks_tp 2279.88 aver_tp 2688.65 stdev_tp 3098.212 ETA
412,033 tasks+ 943706 tasks- 0 tasks-> 1048576 completed 90.0 tasks_tp 2218.13 aver_tp 2687.3 stdev_tp 3093.848 ETA 45
413,038 tasks+ 945886 tasks- 0 tasks-> 1048576 completed 90.21 tasks_tp 2171.31 aver_tp 2685.81 stdev_tp 3089.523 ETA
414,042 tasks+ 948129 tasks- 0 tasks-> 1048576 completed 90.42 tasks_tp 2234.06 aver_tp 2684.52 stdev_tp 3085.188 ETA
415,046 tasks+ 950185 tasks- 0 tasks-> 1048576 completed 90.62 tasks_tp 2047.81 aver_tp 2682.7 stdev_tp 3080.965 ETA 4
416,05 tasks+ 952338 tasks- 0 tasks-> 1048576 completed 90.82 tasks_tp 2144.42 aver_tp 2681.17 stdev_tp 3076.707 ETA 4
417,054 tasks+ 954561 tasks- 0 tasks-> 1048576 completed 91.03 tasks_tp 2214.14 aver_tp 2679.84 stdev_tp 3072.434 ETA
418,062 tasks+ 956645 tasks- 0 tasks-> 1048576 completed 91.23 tasks_tp 2067.46 aver_tp 2678.11 stdev_tp 3068.251 ETA
419,071 tasks+ 958742 tasks- 0 tasks-> 1048576 completed 91.43 tasks_tp 2080.36 aver_tp 2676.42 stdev_tp 3064.079 ETA
420,079 tasks+ 960480 tasks- 0 tasks-> 1048576 completed 91.6 tasks_tp 1724.21 aver_tp 2673.73 stdev_tp 3060.176 ETA 3
421,087 tasks+ 962605 tasks- 0 tasks-> 1048576 completed 91.8 tasks_tp 2108.13 aver_tp 2672.15 stdev_tp 3056.022 ETA 3
422,095 tasks+ 964697 tasks- 0 tasks-> 1048576 completed 92.0 tasks_tp 2075.4 aver_tp 2670.47 stdev_tp 3051.902 ETA 36
423,103 tasks+ 965960 tasks- 0 tasks-> 1048576 completed 92.12 tasks_tp 1248.02 aver_tp 2666.5 stdev_tp 3048.561 ETA 3
424,111 tasks+ 974461 tasks- 0 tasks-> 1048576 completed 92.93 tasks_tp 8425.6 aver_tp 2682.54 stdev_tp 3059.406 ETA 3
425,119 tasks+ 978213 tasks- 0 tasks-> 1048576 completed 93.29 tasks_tp 3722.22 aver_tp 2685.43 stdev_tp 3055.644 ETA
426,128 tasks+ 980243 tasks- 0 tasks-> 1048576 completed 93.48 tasks_tp 2011.9 aver_tp 2683.57 stdev_tp 3051.614 ETA 2
427,136 tasks+ 982449 tasks- 0 tasks-> 1048576 completed 93.69 tasks_tp 2187.5 aver_tp 2682.19 stdev_tp 3047.508 ETA 2
428,144 tasks+ 983418 tasks- 0 tasks-> 1048576 completed 93.79 tasks_tp 961.31 aver_tp 2677.45 stdev_tp 3044.643 ETA 2
429,152 tasks+ 987607 tasks- 0 tasks-> 1048576 completed 94.19 tasks_tp 4154.76 aver_tp 2681.51 stdev_tp 3041.441 ETA
430,16 tasks+ 995260 tasks- 0 tasks-> 1048576 completed 94.92 tasks_tp 7592.26 aver_tp 2694.97 stdev_tp 3048.1 ETA 23.
431,168 tasks+ 995260 tasks- 0 tasks-> 1048576 completed 94.92 tasks_tp 0.0 aver_tp 2687.6 stdev_tp 3047.182 ETA 23.09
432,176 tasks+ 997217 tasks- 0 tasks-> 1048576 completed 95.1 tasks_tp 1941.47 aver_tp 2685.57 stdev_tp 3043.276 ETA 2
433,184 tasks+ 1002140 tasks- 0 tasks-> 1048576 completed 95.57 tasks_tp 4878.97 aver_tp 2691.53 stdev_tp 3041.282 ETA
434,193 tasks+ 1007547 tasks- 0 tasks-> 1048576 completed 96.09 tasks_tp 5364.09 aver_tp 2698.77 stdev_tp 3040.335 ETA
435,201 tasks+ 1007547 tasks- 0 tasks-> 1048576 completed 96.09 tasks_tp 0.0 aver_tp 2691.48 stdev_tp 3039.455 ETA 17.
436,209 tasks+ 1009869 tasks- 0 tasks-> 1048576 completed 96.31 tasks_tp 2303.57 aver_tp 2690.43 stdev_tp 3035.422 ETA
437,217 tasks+ 1011812 tasks- 0 tasks-> 1048576 completed 96.49 tasks_tp 1927.58 aver_tp 2688.38 stdev_tp 3031.597 ETA
438,225 tasks+ 1015396 tasks- 0 tasks-> 1048576 completed 96.84 tasks_tp 3555.56 aver_tp 2690.71 stdev_tp 3027.863 ETA
439,233 tasks+ 1019985 tasks- 0 tasks-> 1048576 completed 97.27 tasks_tp 4550.6 aver_tp 2695.68 stdev_tp 3025.337 ETA
440,241 tasks+ 1024438 tasks- 0 tasks-> 1048576 completed 97.7 tasks_tp 4412.7 aver_tp 2700.26 stdev_tp 3022.598 ETA 1
441,249 tasks+ 1024438 tasks- 0 tasks-> 1048576 completed 97.7 tasks_tp 0.0 aver_tp 2693.08 stdev_tp 3021.777 ETA 10.3
442,257 tasks+ 1025720 tasks- 0 tasks-> 1048576 completed 97.82 tasks_tp 1271.83 aver_tp 2689.31 stdev_tp 3018.652 ETA
443,266 tasks+ 1030228 tasks- 0 tasks-> 1048576 completed 98.25 tasks_tp 4472.22 aver_tp 2694.02 stdev_tp 3016.048 ETA
444,274 tasks+ 1032566 tasks- 0 tasks-> 1048576 completed 98.47 tasks_tp 2319.44 aver_tp 2693.04 stdev_tp 3012.127 ETA
445,282 tasks+ 1036258 tasks- 0 tasks-> 1048576 completed 98.83 tasks_tp 3662.7 aver_tp 2695.59 stdev_tp 3008.572 ETA
446,29 tasks+ 1038843 tasks- 0 tasks-> 1048576 completed 99.07 tasks_tp 2564.48 aver_tp 2695.24 stdev_tp 3004.628 ETA
447,298 tasks+ 1040796 tasks- 0 tasks-> 1048576 completed 99.26 tasks_tp 1929.56 aver_tp 2693.24 stdev_tp 3000.948 ETA
448,306 tasks+ 1041203 tasks- 0 tasks-> 1048576 completed 99.3 tasks_tp 395.83 aver_tp 2687.24 stdev_tp 2999.32 ETA 3.
449,314 tasks+ 1043488 tasks- 0 tasks-> 1048576 completed 99.51 tasks_tp 2266.87 aver_tp 2686.15 stdev_tp 2995.489 ETA
450,322 tasks+ 1046203 tasks- 0 tasks-> 1048576 completed 99.77 tasks_tp 2688.18 aver_tp 2686.15 stdev_tp 2991.596 ETA
451,331 tasks+ 1048576 tasks- 0 tasks-> 1048576 completed 100.0 tasks_tp 2354.17 aver_tp 2685.29 stdev_tp 2987.766 ETA
452,339 tasks+ 1048576 tasks- 0 tasks-> 1048576 completed 100.0 tasks_tp 0.0 aver_tp 2678.35 stdev_tp 2987.016 ETA 0.0
453,347 tasks+ 1048576 tasks- 0 tasks-> 1048576 completed 100.0 tasks_tp 0.0 aver_tp 2671.45 stdev_tp 2986.253 ETA 0.0
1048576 tasks completed in 453.505 sec
Successful tasks: 1048576
Failed tasks: 0
Notification Errors: 0
Overall Throughput (tasks/sec): 2312.16
Overall Throughput Standard Deviation: 2986.253
waiting to destroy all resources...
ShutdownHook triggered successfully!
iraicu@gto:~/falkon>
```

Ready

Figure 21: Monitoring via GKrellm while running 1M tasks on 160K processors

The experiments presented in Figure 20 and Figure 21 were conducted using one million tasks. 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 22 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 20. 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.

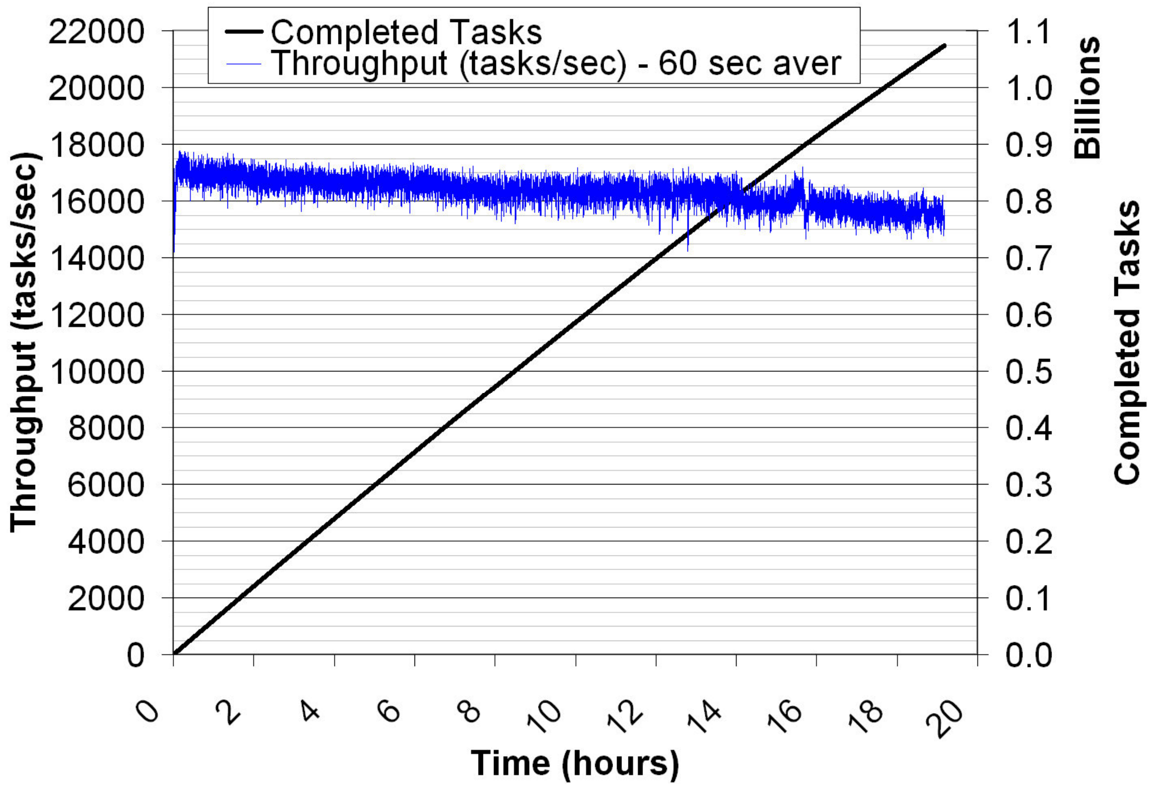


Figure 22: Endurance test with 1B tasks on 128 CPUs in ANL/UC cluster

4.3.2.1 Comparing Falkon to Other LRMs and Solutions

It is instructive to compare with task execution rates achieved by other local resource managers. In Chapter 3, we measured Condor (v6.7.2, via MyCluster [95]) 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 [83].

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 [53] 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.

As we covered in the related work section, Cope et al. [51] also explored a similar space as we have, leveraging HTC-mode [53] 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 (tasks return immediately, so the reported run time shows overhead). 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). Since these times include the time it took to cache the binary in ramdisk, we believe these numbers will remain relatively stable (within an order of magnitude) as we scale up to full 160K processors. Note that the work by Cope et al. is based on Cobalt's HTC-mode [53], which implies that they perform a node reboot for every task, while we simply fork the application as a separate process for each task.

Finally, Peter's et al. from IBM also recently published some performance numbers on the HTC-mode native support in Cobalt [52], 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 368 seconds, which translates to 0.35ms per task overheads.

4.3.2.2 Efficiency and Speedup

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 and 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 workloads ranging from 1K tasks to 1M tasks (depending on the task lengths, to ensure that the experiments completed in a reasonable amount of time).

Figure 23 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 24 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.

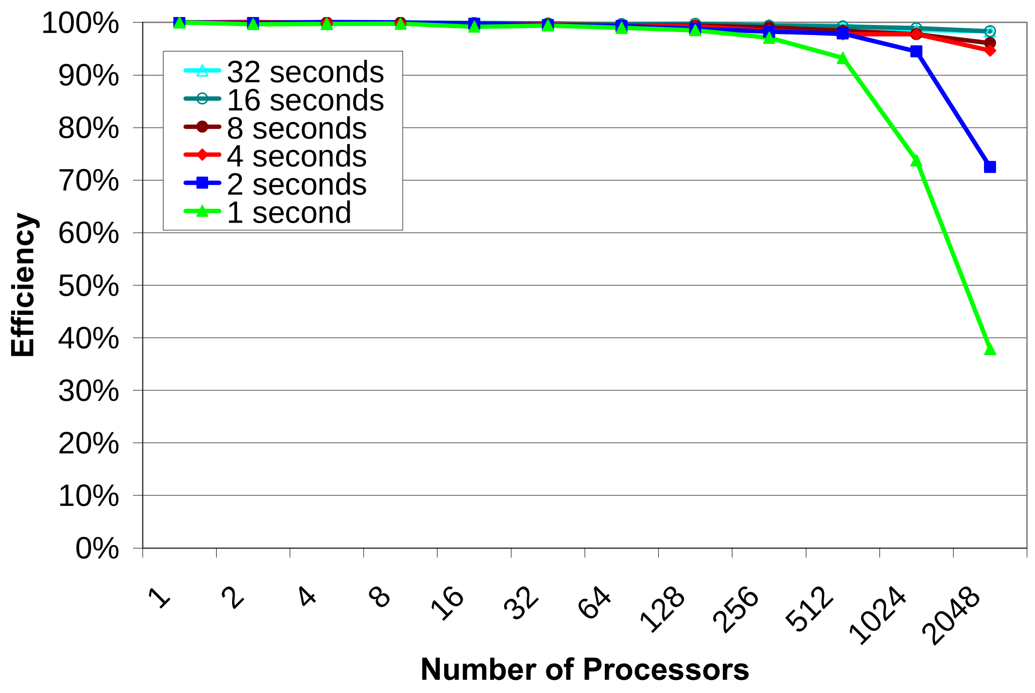


Figure 23: Efficiency graph for the Blue Gene/P for 1 to 2048 processors and task lengths from 1 to 32 seconds using a single dispatcher on a login node

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.

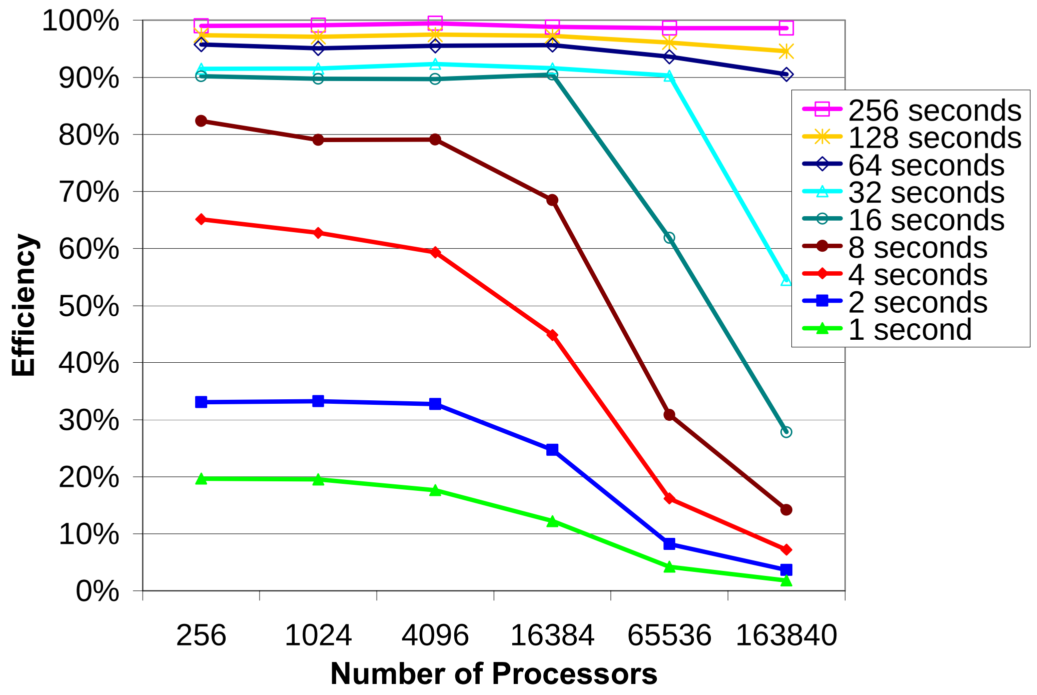


Figure 24: Efficiency graph for the Blue Gene/P for 256 to 160K processors and task lengths ranging from 1 to 256 seconds using N dispatchers with each dispatcher running on a separate I/O node

4.3.2.3 Dispatch Processing Overheads

In trying to understand the various costs leading to the throughputs achieved in Figure 20, Figure 25 profiles the service code, and breaks down the CPU time by code block. This test was done on the VIPER.CI and the ANL/UC Linux cluster with 200 CPUs, with throughputs reaching 487 tasks/sec and 1021 tasks/sec for the Java and C implementations respectively. A significant portion of the CPU time is spent in communication (WS and/or TCP). With bundling (not shown in Figure 25), the communication costs are reduced to 1.2 ms (down from 4.2 ms), as well as other costs. Our conclusion is that the peak throughput for small tasks can be increased by both adding faster processors, more processor cores to the service host, and reducing the communication costs by lighter weight protocols or by bundling where possible.

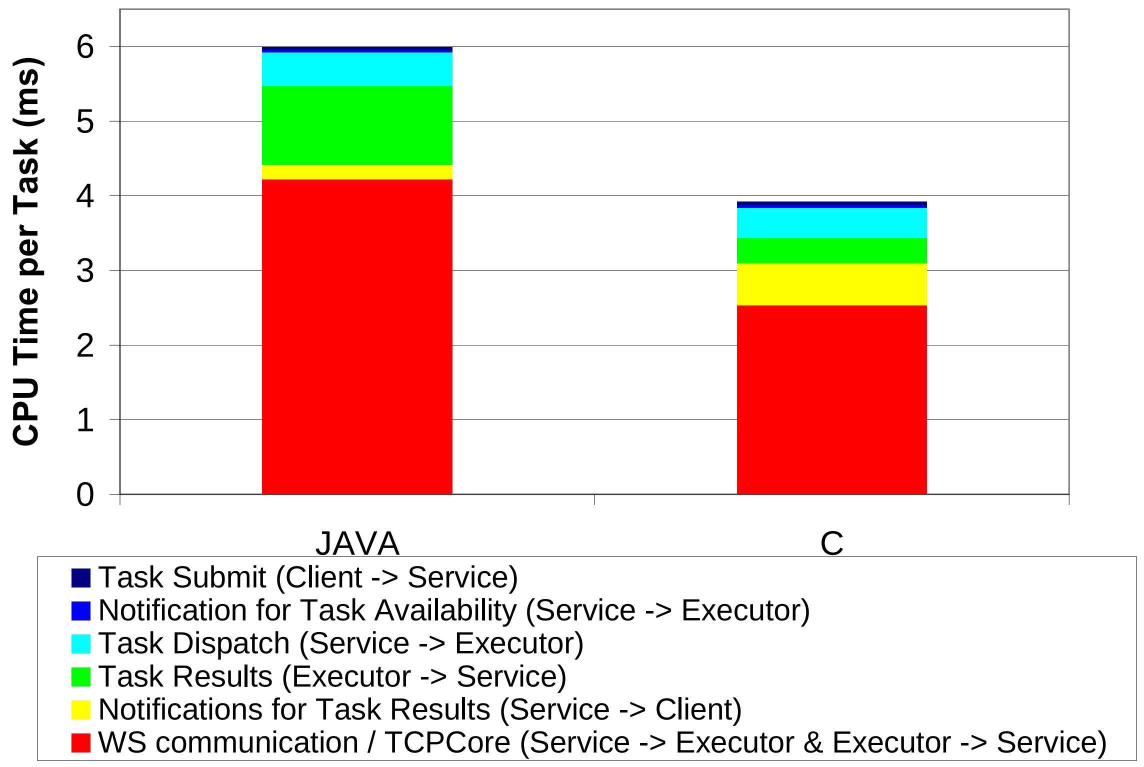


Figure 25: Falkon profiling comparing the Java and C implementation on VIPER.CI (dual Xeon 3GHz w/ HT)

4.3.2.4 Dispatch Network Overheads

The previous several experiments all investigated the throughput and efficiency of executing tasks which had a small and compact description. For example, the task "/bin/sleep 0" requires only 12 bytes of information. The following experiment (Figure 26) investigates how the throughput is affected by increasing the task description size. For this experiment, we compose 4 different tasks, "/bin/echo 'string'", where string is replaced with a different length string to make the task description 10B, 100B, 1KB, and 10KB. We ran this experiment on the SiCortex with 1002 CPUs and the service on GTO.CI, and processed 100K tasks for each case.

We see the throughput with 10B tasks is similar to that of sleep 0 tasks on 5760 CPUs with a throughput of 3184 tasks/sec. When the task size is increased to 100B, 1KB, and 10KB, the throughput is reduced to 3011, 2001, and 662 tasks/sec respectively. To better understand the throughput reduction, we also measured the network level traffic that the service experienced during the experiments. We observed that the aggregate throughput (both received and sent on a full duplex 100Mb/s network link) increases from 2.9MB/s to 14.4MB/s as we vary the task size from 10B to 10KB.

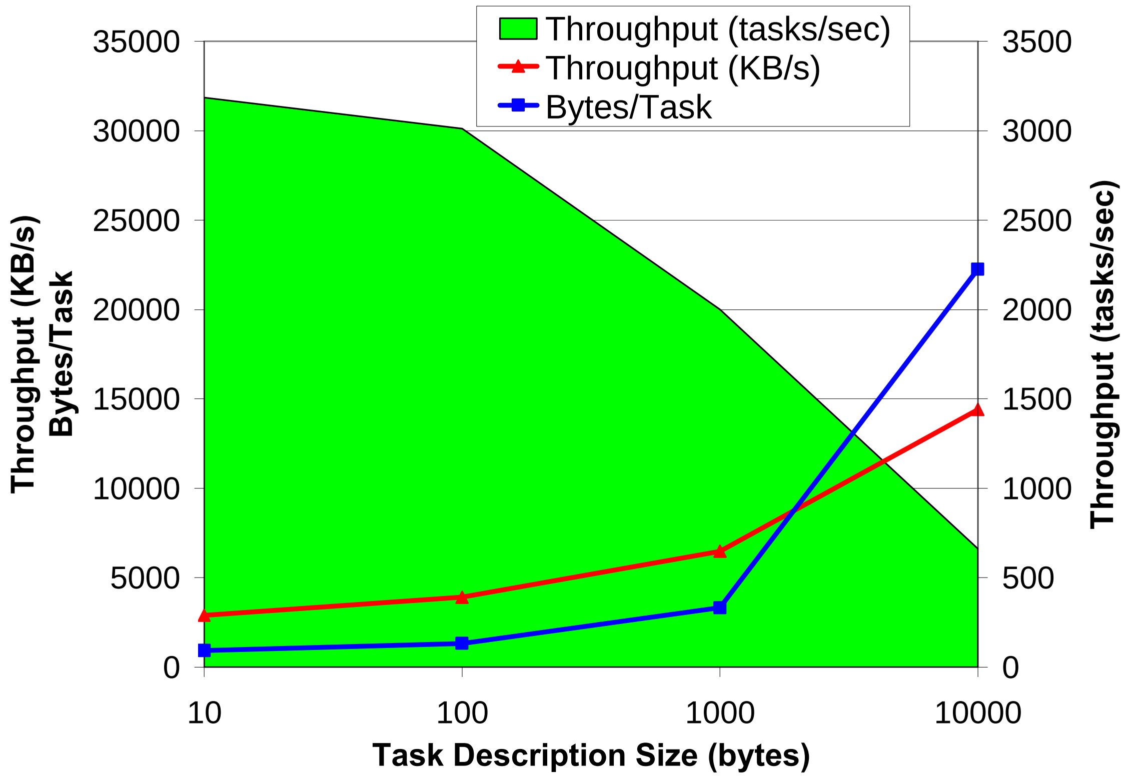


Figure 26: Task description size on the SiCortex and 1K CPUs

The bytes/task varies from 934 bytes to 22.3 KB for the 10B to 10KB tasks. The formula to compute the bytes per task is 2*task_size + overhead of TCP-based protocol (including TCP/IP headers) + overhead of WS-based submission protocol (including XML, SOAP, HTTP, and TCP/IP) + notifications of results from executors back to the service, and from the service to the user. We need to double the task size since the service first receives the task description from the user (or application), and then dispatches it to the remote executor. Only a brief notification with the task ID and exit code of the application is sent back. We might assume that the overhead is 934 – 2*10 = 914 bytes, but from looking at the 10KB tasks, we see that the overhead is 22.3KB – 2*10KB = 2.3KB (higher than 0.9KB). We measured the number of TCP packets to be 7.36 packets/task (10B tasks) and 28.67 packets/task (10KB tasks). The difference in TCP overhead 853 bytes (with 40 byte headers for TCP/IP, 28.67*40 - 7.36*40) explains most of the difference. We suspect that the remainder of the difference (513 bytes) is due to extra overhead in XML/SOAP/HTTP when submitting the tasks.

4.3.3 Shared File System Performance

Another key component to getting high utilization and efficiency on large-scale systems is to understand the shared resources well, and to make sure that the compute-to-I/O ratio is appropriate. This sub-section discusses the shared file system performance of the Blue Gene/P. This is important as many MTC applications use files for inter-process communication, and these files are typically transferred from one node to another through the shared file system. Future work will remove this bottleneck, by using TCP pipes, MPI messages, or data diffusion [3, 1]. We conducted (see Figure 27) several experiments with various data sizes (1KB to 10MB) on a varying number of processors (4 to 16K); we conducted both read-only tests (dotted lines) and read+write tests (solid lines).

At 16K processors, we were not able to saturated GPFS – note the throughput lines never plateau. GPFS is configured with 16 I/O servers, each with 10Gb/s network connectivity, and can sustain 8GB/s aggregate I/O rates. We were able to achieve 4.4GB/s read rates, and 1.3GB/s read+write rates with 10MB files and 16K processors (we used the Linux "dd" utility to read or read+write data in 128KB blocks). We made our measurements in a production system, where the majority (90%+) of the system was in use by other applications, which might have been using the shared file system as well, influencing the results from this micro-benchmark.

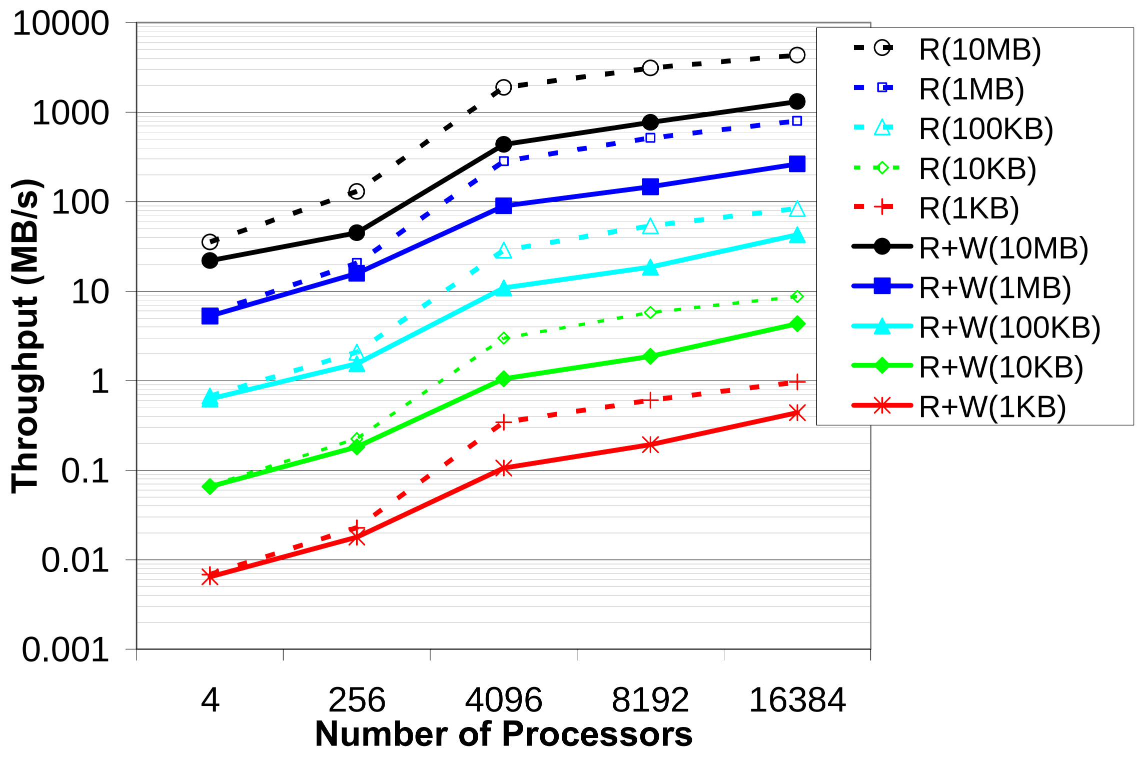


Figure 27: GPFS Throughput in MB/s measured through Falkon on various file sizes (1KB-10MB) and number of processors (4-16384)

It is also important to understand how operation costs scale with increasing number of processors (see Figure 28). We tested file and directory creation in two scenarios; when all files or directories are created in the same directory (single dir), when each file or directory is created in a unique pre-created directory (across many dirs). We also investigate the costs to invoke a script from GPFS. Finally, we measure the Falkon overhead (from a client perspective) of executing a trivial task with no I/O (sleep 0). Both the file and directory create when performed in the same directory are expensive operations as we scale up the number of processors; for example, at 16K processors, it takes (on average) 404 seconds to create a file, and 1217 seconds to create a directory.

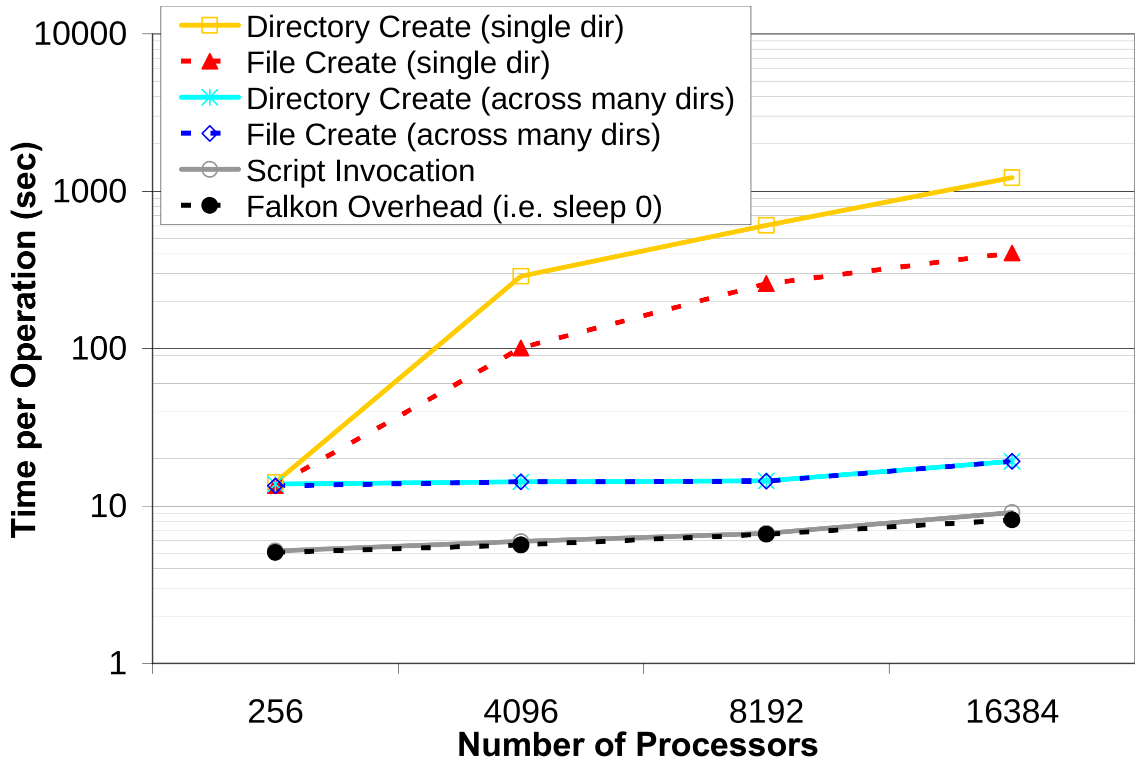


Figure 28: Time per operation (mkdir, touch, script execution) on GPFS on various number of processors (256-16384)

These overheads translate to an aggregate throughput of 40 file creates per second and 13 directory creates per second. At these rates, 160K processors would require 68 and 210 minutes to create 160K files or directories. In contrast, when each file or directory create take place in a unique directory, performance is significantly improved; at small scales (256 processors), a file/directory create (in a unique directory) only takes

8 seconds longer than a basic task with no I/O; at large scales (16K processors), the overhead grows to 11 seconds. We conclude that I/O writes should be split over many directories, to avoid lock contention within GPFS from concurrent writers. These times reflect the costs of creating a file or directory when all processors perform the operation concurrently; many applications have a wide range of task lengths, and read/write operations only occur at the beginning and/or end of a task (as is the case with our caching mechanism), the time per operation will be notably less due to the natural staggering of I/O calls.

4.3.4 Running applications through Swift

The results presented in these last several sections are from a static workload processed directly with Falkon. Swift on the other hand can be used to make the workload more dynamic, reliable, and provide a natural flow from the results of this application to the input of the following stage in a more complex workflow. Swift incurs its own overheads in addition to what Falkon experiences when running these applications. These overheads include 1) managing the data (staging data in and out, copying data from its original location to a workflow-specific location, and back from the workflow directory to the result archival location), 2) creating per-task working directories from the compute nodes (via mkdir on the shared file system), and 3) creation and tracking of several status logs files for each task.

We ran a 16K task workload for the MARS application 144 model runs batched per task (65 second tasks) on 2048 CPUs which yielded an end-to-end efficiency of 20% with the default Swift settings and implementation. We investigated the main bottlenecks, and they seemed to be shared file system related. We applied three distinct optimizations to the Swift wrapper script: 1) the placement of temporary directories in local ramdisk rather than the shared filesystem; 2) copies the input data to the local ramdisk of the compute node for each job execution; and 3) creates the per job logs on local ramdisk and only copies them at the completion of each job (rather than appending a file on shared file system at each job status change). These optimizations allowed us to increase the efficiency from 20% to 70% on 2048 processors for the MARS application. We will be working to narrow the gap between the efficiencies found when running Swift and those when running Falkon alone, and hope to get the Swift efficiencies up in the 90% range while reaching larger experiments involving 100K~1M tasks on 10K~160K CPUs.

4.4 Characterizing MTC Applications for PetaScale Systems

Based on our experience with the Blue Gene/P at 160K CPU scale (nearly 0.5 petaflop Rpeak) and its shared file system (GPFS, rated at 8GB/s), we identify the following characteristics that define MTC applications that are most suitable for peta-scale systems:

- number of tasks >> number of CPUs
- average task execution time > O(60 sec) with minimal I/O to achieve 90%+ efficiency
- 1 sec of compute per processor core per 5KB~50KB of I/O to achieve 90%+ efficiency

The main bottleneck we found was the shared file system. GPFS is used throughout our system, from booting the compute nodes and I/O nodes, starting the Falkon dispatcher and executors, starting the applications, and reading and writing data for the applications. Assuming a large enough application, the startup costs (e.g. 1326 seconds to bootstrap and be ready to process the first task at 160K processors) can be amortized to an insignificant value. We offloaded the shared file system to in-memory operations by caching the Falkon middleware, the applications binaries, and the static input data needed by the applications in memory, so repeated use could be handled completely from memory. We found that the three applications we worked with all had poor write access patterns, in which many small line-buffered writes in the range of 100s of bytes were performed throughout the task execution. When 160K CPUs are all doing these small I/O calls concurrently, it can slow down the shared file system to a crawl, or even worse, crash it. The solution was to read dynamic input data from shared file system into memory in bulk (e.g., dd with block sizes of 128KB), let applications interact with their input and output files directly in memory, and write dynamic output data from memory to shared file system in bulk (e.g., dd, merge many output files into a single tar archive).

4.5 Conclusions

Clusters with 50K+ processor cores (e.g., TACC Sun Constellation System, Ranger), Grids (e.g., TeraGrid) with a dozen sites and 100K+ processors, and supercomputers with 160K and 200K processors (e.g., IBM Blue Gene/P and Blue Gene/L respectively) are now available to the scientific community. This chapter focused on the ability to manage and execute large-scale applications on petascale class systems. Large clusters and supercomputers have traditionally been HPC systems, as they are efficient at executing tightly coupled parallel jobs within a particular machine with low-latency interconnects, typically using MPI to achieve the needed inter-process communication. On the other hand, Grids have been the preferred platform for more loosely coupled applications that tend to be managed and executed through workflow or parallel programming systems. We have characterized these loosely coupled applications as many-task computing, which generally involve the execution of independent, sequential jobs that can be individually scheduled on many different computing resources across multiple administrative boundaries, and often use files for inter-process communication.

Our work shows that today's existing HPC systems are a viable platform to host MTC applications. We identified challenges in running large-scale loosely coupled applications on petascale systems, which can hamper the efficiency and utilization of these large-scale machines. These challenges vary from local resource manager scalability and granularity, efficient utilization of the raw hardware, shared file system contention and scalability, reliability at scale, application scalability, and understanding the limitations of the HPC systems in order to identify promising and scientifically valuable MTC applications. This chapter presented new research and implementations in scaling loosely coupled applications micro-benchmarks up to 160K processors.

5 Dynamic Resource Provisioning

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

The results presented in this chapter have been published in [4, 6].

5.1 Overview

Chapter 3 introduced the concept of dynamic resource provisioning, but it mainly focused on streamlined task dispatching. This chapter will focus on dynamic resource provisioning, and offer results to show its effectiveness.

We seek to achieve improvements by:

- Using an adaptive provisioner to acquire and/or release resources as application demand varies
- Reducing average queue wait times by amortizing high overhead of resource allocation over the execution of many tasks

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

5.2 Architecture and Implementation

5.2.1 Execution Model

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

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

The first strategy, *Optimal***,** assumes that we can query the resource manager to determine the maximum number of resources available to us. We then simply request that number if it is less than *n*, and request *n* otherwise. This policy has not been implemented due to the fact that this feature is not the common case among LRMs and what they expose to applications. The TeraGrid [44] for example is a case where such information is available, but it is done via batch queue prediction mechanisms which can be used to statistically predict the number of resources that could be allocated in a relatively short period of time.

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

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

5.2.2 Architecture

As illustrated in Figure 29, our dynamic resource provisioning (DRP) system comprises: (1) user(s); (2) a DRP Utilizing Application (i.e. a Web Service such as

Falkon); (3) the Provisioner; (4) a Resource Manager (i.e. GRAM, Condor, PBS, etc); and (5) a resource pool. The interaction between these various components is as follows.

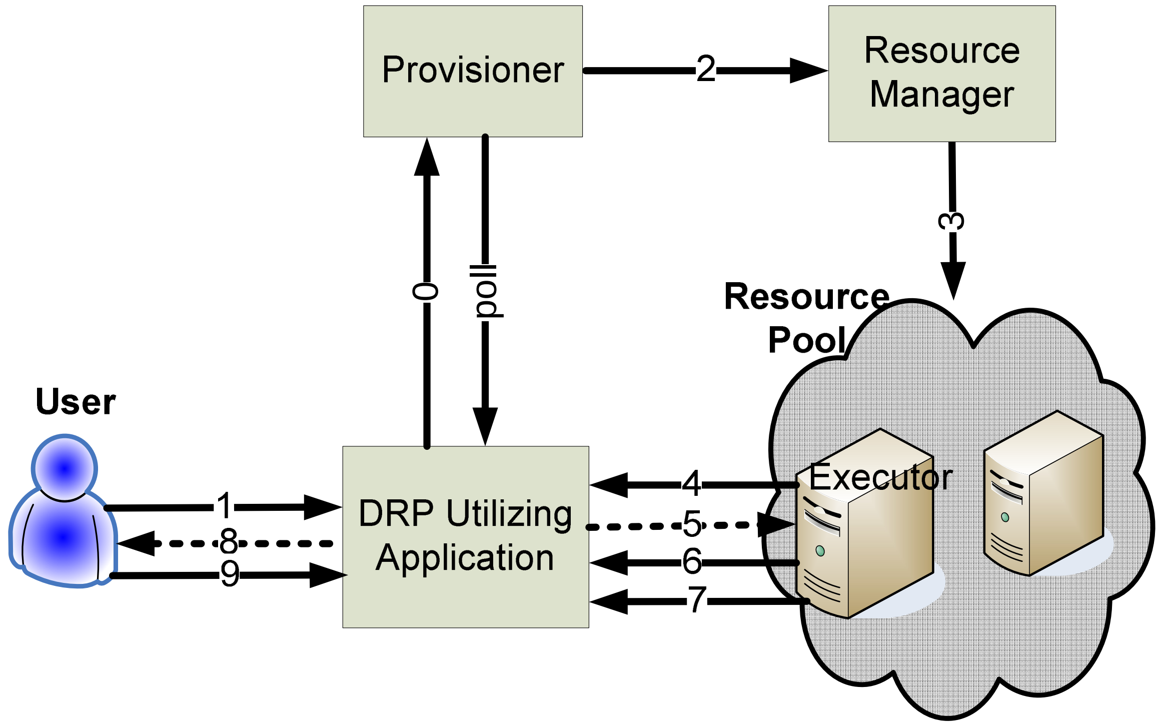


Figure 29: Dynamic Resource Provisioning Architecture

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

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

At first sight, this seems to be relatively complex and likely to add overhead to the execution of application work. It should be noted that while these executors are available (which is dictated by the resource de-allocation policies), any subsequent work requests from the user can simply use the same resources that have already been allocated (according to the resource allocation policy), without the need to go through the entire allocation process. After the initial phases in which resources are allocated and executors are started, a high volume of work broken down into many smaller tasks can essentially be performed using just messages 1, 5, 6, 7, 8, and 9. With some optimizations (task bundling and piggy-backing [4]), these messages can be reduced on average to only message 7 and 8 per task.

5.2.3 Provisioning in Falkon

Falkon, a Fast and Light-weight tasK executiON framework, provides a system for scheduling and dispatching of independent tasks to a set of executors. Integrating the provisioning mechanisms proposed in this chapter into Falkon gives Falkon expanded capabilities to dynamically deploy and run executors across multiple Grid sites based on Falkon's load (i.e. wait queue length). Falkon consists of a dispatcher and zero or more executors (Figure 7); the provisioner is added as a third component that acts as a mediator between he dispatcher and the Grid resources on which the executors are to run on. The dispatcher accepts tasks from clients and implements the dispatch policy. The provisioner implements the resource acquisition policy. Executors run tasks received from the dispatcher. Components communicate via Web Services (WS) messages, except for notifications are performed via a custom TCP-based protocol.

The **dispatcher** implements the factory/instance pattern, providing a *create instance* operation to allow a clean separation among different clients. To access the dispatcher, a client first requests creation of a new instance, for which is returned a unique endpoint reference (EPR). The client then uses that EPR to submit tasks, monitor progress, retrieve results, and (finally) destroy the instance. Each instance can be thought of as a separate instantiation of the dispatcher, maintaining its own task queue and related state. The dispatcher runs within a Globus Toolkit 4 (GT4) [79] WS container, which provides authentication, message integrity, and message encryption mechanisms, via transport-level, conversation-level, or message-level security [126].

The **provisioner** is responsible for creating and destroying executors. It is initialized by the dispatcher with information about the state to be monitored and how to access it; the rule(s) under which the provisioner should create/destroy executors; the location of the executor code; bounds on the number of executors to be created; bounds on the time for which executors should be created; and the allowed idle time before executors are destroyed. The provisioner periodically monitors dispatcher state and, based on the supplied rules, determines whether to create additional executors, and if so, how many, and for how long. Creation requests are issued via GRAM4 [127], to abstract away LRM details.

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

5.3 Performance Evaluation

This subsection investigates the performance of dynamic resource provisioning with respect to various allocation policies and de-allocation policies.

5.3.1 Allocation Policies

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

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

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

In a more realistic setting, sites are rarely idle, and hence some resource requests will end up waiting in the local resource manager's queue. To explore this case, we consider a scenario in which the site has only 47 resources available until the 160 second mark, at which point availability increases to 48; these results are shown in dotted lines in Figure 30. Thus, each strategy has their last resource request held in the wait queue until the 160 second mark. Those last requests are for 1, 3, 17, and 48 resources for One-at-a-time, Additive, Exponential, and All-at-once, respectively. On one extreme, the 1-at-a-time strategy manages to allocate 47 resources and has only 1 resource in the waiting queue; the other extreme, the all-at-once strategy has all 48 resources in the waiting queue waiting for a single resource to free up before it can process the entire request. This is evidence of the back-filling strategies of the local resource manager. Therefore, the all-at-once is now the worst overall, being over 60 CPU minutes behind One-at-a-time and Exponential, and almost 90 CPU minutes behind Additive. Note that these lags in accumulated CPU time will remain until the resources begin to de-allocate, at which time the strategies that received their resources later will also hang on to the resources later; in the end, all strategies should get the same accumulated CPU time eventually.

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

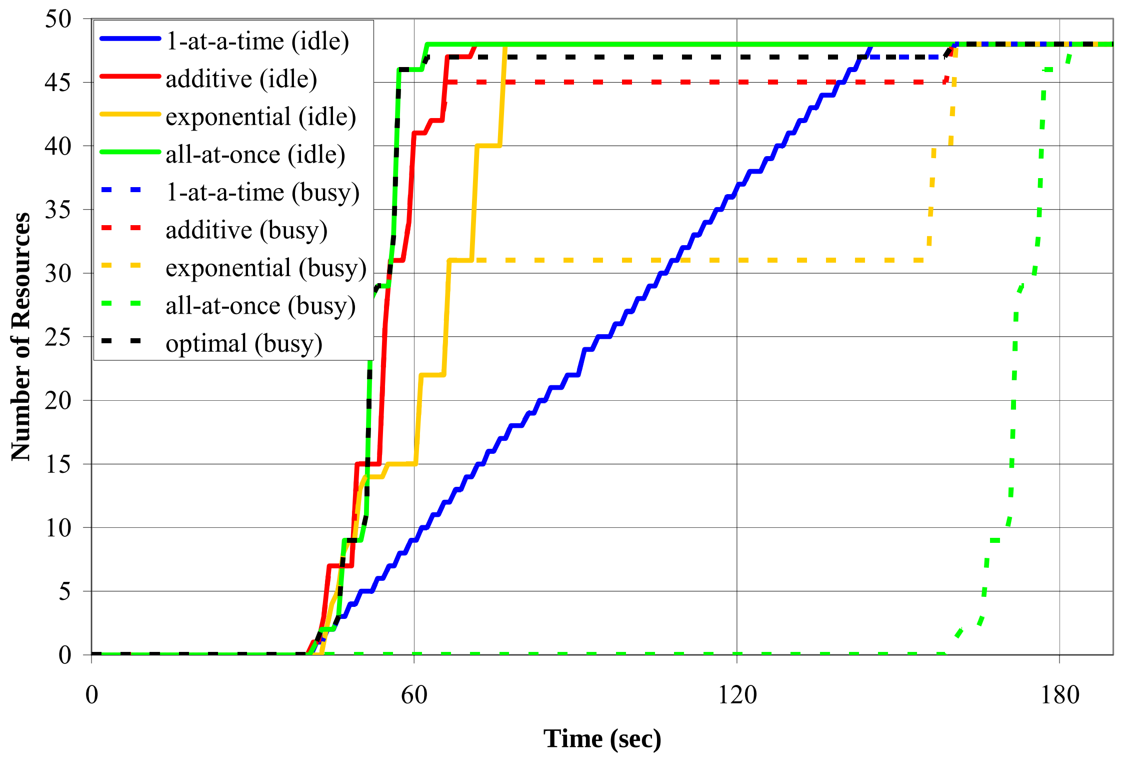


Figure 30: Provisioning latency in acquiring 48 resources for various strategies; the solid lines represent the time to acquire the resources in an idle system, while the dotted lines is the time to acquire the resources in a busy system

Table 5: Accumulated CPU time in seconds after 180 seconds in both an idle and busy system

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

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

5.3.2 De-Allocation Policies

To study provisioner performance, we constructed a synthetic 18-stage workload, in which the numbers of tasks and task lengths vary between stages. Figure 31 shows the number of tasks per stage and the number of machines needed per stage if each task is mapped to a separate machine (up to a maximum of 32 machines). Note the exponential ramp up in the number of tasks for the first few stages, a sudden drop at stage 8, and a sudden surge of many tasks in stages 9 and 10, another drop in stage 11, a modest increase in stage 12, followed by a linear decrease in stages 13 and 14, and finally an exponential decrease until the last stage has only a single task. All tasks run for 60 secs except those in stages 8, 9, and 10, which run for 120, 6, and 12 secs, respectively. In total, the 18 stages have 1,000 tasks, summing to 17,820 CPU secs, and can complete in an ideal time of 1,260 secs on 32 machines. We choose this workload to showcase and evaluate the flexibility of Falkon's dynamic resource provisioning, as it can adapt to varying resource requirements and task durations.

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

- *GRAM4+PBS* (without Falkon): Each task was submitted as a separate GRAM4 task to PBS, without imposing any hard limits on the number of machines to use; there were about 100 machines available for this experiment.
- *Falkon-15, Falkon-60, Falkon-120, Falkon-180*: Falkon configured to use a minimum of zero and a maximum of 32 machines; the allocation policy we used was *all-at-once*, and the resource release policy idle time was set to 15, 60, 120, and 180 secs (to give four separate experiments).
- *Falkon-∞:* Falkon, with the provisioner configured to retain a full 32 machines for one hour.

Table 6 gives, for each experiment, the average per-task queue time and execution time, and also the ratio exec_time/(exec_time+queue_time). The queue_time includes time waiting for the provisioner to acquire nodes, time spent starting executors, and time tasks spend in the dispatcher queue. We see that the ratio improves from 17% to 28.7% as the idle time setting increases from 15 to 180 secs; for Falkon-∞, it reaches 29.2%, a value close to the ideal of 29.7%. (The ideal is less than 100% because several stages have more than 32 tasks, which means tasks must be queued when running, as we do here, on 32 machines.) GRAM4+PBS yields the worst performance, with only 8.5% on average, less than a third of ideal.

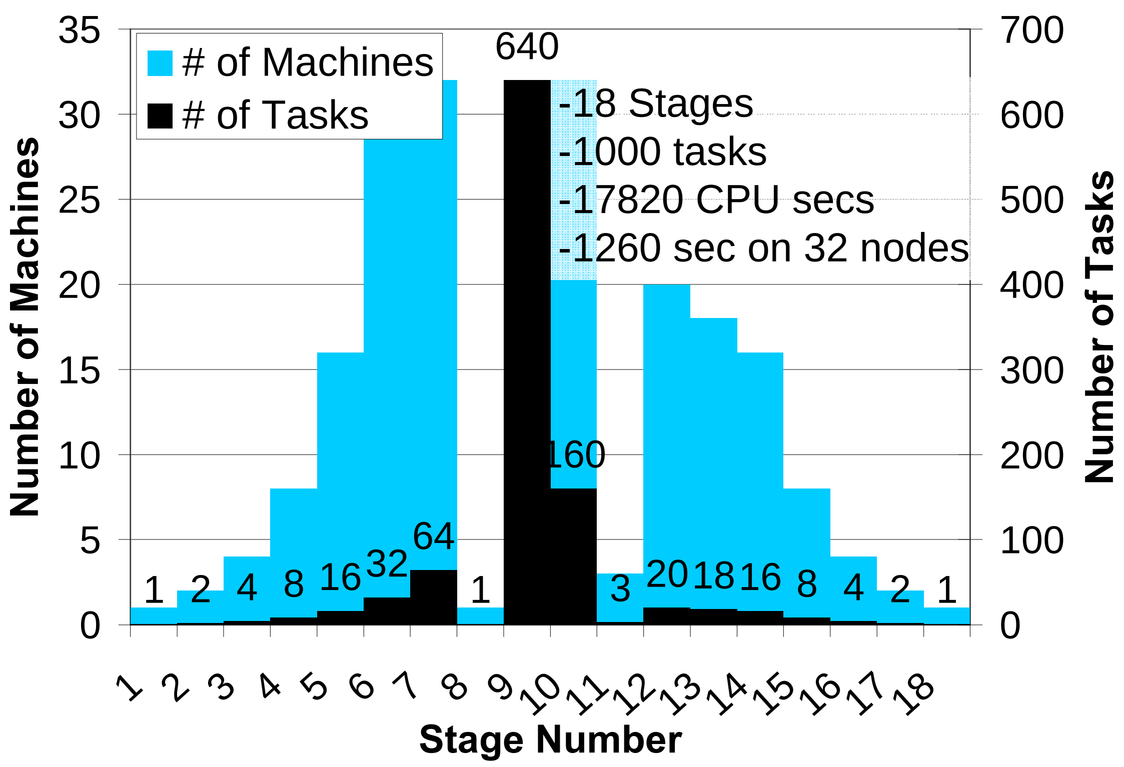


Figure 31: Dynamic Resource Provisioning: 18-stage synthetic workload.

Table 6: Average per-task queue and execution times for synthetic workload

	GRAM4 +PBS	Falkon -15	Falkon -60	Falkon -120	Falkon -180	Falkon -∞	Ideal (32 nodes)
Queue Time (sec)	611.1	87.3	83.9	74.7	44.4	43.5	42.2
Execution Time (sec)	56.5	17.9	17.9	17.9	17.9	17.9	17.8
Execution Time %	8.5%	17.0%	17.6%	19.3%	28.7%	29.2%	29.7%

The average per-task queue times range from a near optimal 43.5 secs (42.2 secs is ideal) to as high as 87.3 secs, more than double the ideal. In contrast, GRAM4+PBS experiences a queue time of 611.1 secs: 15 times larger than the ideal. Also, note the execution time for Falkon with resource provisioning (both static and dynamic) is the same across all the experiments, and is within 100 ms of ideal (which essentially accounts for the dispatch cost and delivering the result); in contrast, GRAM4+PBS has an average execution time of 56.5 secs, significantly larger than the ideal time. This large difference in execution time is attributed to the large per task overhead GRAM4 and PBS have, which further strengthens our argument that they are not suitable for short tasks.

Table 7 shows, for each strategy, the time to complete the 18 stages, resource utilization, execution efficiency, and number of resource allocations. We define resource utilization and execution efficiency as follows:

$$resource\_utilization = \frac{resources\_used}{resources\_used + resources\_wasted}$$

Equation 1: Resource utilization

$$exec\_efficiency = \frac{ideal\_time}{actual\_time}$$

Equation 2: Execution efficiency

We define resources as “wasted,” for the Falkon strategies, when they have been allocated and registered with the dispatcher, but are idle. For the GRAM4+PBS case, the “wasted” time is the difference between the measured and reported task execution time, where the reported task execution time is from the time GRAM4 sends a notification of the task becoming “Active”—meaning that PBS has taken the task off the wait queue and placed into the active queue assigned to some physical machine—to the time the state changes to “Done,” at which point the task has finished execution.

The resources used are the same (17,820 CPU secs) for all cases, as we have fixed run times for all 1000 tasks. We expected GRAM4+PBS to not have any wasted resources, as each machine is released after one task is run; in reality, the measured execution times were longer than the actual task execution times, and hence the resources wasted was high in this case: 41,040 secs over the entire experiment. The average execution time of 56.5 secs shows that GRAM4+PBS is slower than Falkon in dispatching the task to the remote machine, preparing the remote machine to execute the task, and cleaning up and releasing the machine. Note that the reception of the "Done" state change in GRAM4 does not imply that the utilized machine is ready to receive another task—PBS takes even longer to make the machine available again for more work, which makes GRAM4+PBS resource wastage yet worse.

Falkon with dynamic resource provisioning fairs better from the perspective of resource wastage. Falkon-15 has the fewest wasted resources (2032 secs) and Falkon-∞ the worst (22,940 CPU secs). The resource utilization shows the fraction of time the machines were executing tasks vs. idle. Due to its high resource wastage, GRAM4+PBS achieves a utilization of only 30%, while Falkon-15 reaches 89%. Falkon-∞ is 44%. Notice that as the resource utilization increases, so does the time to complete—as we assume that the provisioner has no foresight regarding future needs, delays are incurred allocating machines previously de-allocated due to a shorter idle time setting. Note the number of resource allocations (GRAM4 calls requesting resources) for each experiment, ranging from 1000 allocations for GRAM4+PBS to less than 11 for Falkon with provisioning. For Falkon-∞, the number of resource allocations is zero, since machines were provisioned prior to the experiment starting, and that time is not included in the time to complete the workload.

If we had used a different allocation policy (e.g., one-at-a-time), the Falkon results would have been less close to ideal, as the number of resource allocations would have grown significantly. The relatively slow handling of such requests by GRAM4+PBS (~0.5/sec on TG_ANL_IA32 and TG_ANL_IA64) would have delayed executor startup and thus increased the time tasks spend in the queue waiting to be dispatched.

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

Table 7: Resource utilization and execution efficiency summary

	GRAM4 +PBS	Falkon -15	Falkon -60	Falkon -120	Falkon -180	Falkon -∞	Ideal (32 nodes)
Time to complete (sec)	**4904**	**1754**	**1680**	**1507**	**1484**	**1276**	**1260**
Resouce Utilization	30%	89%	75%	65%	59%	44%	100%
Execution Efficiency	26%	72%	75%	84%	85%	99%	100%
Resource Allocations	1000	11	9	7	6	0	0

To illustrate how provisioning works in practice, we show in Figure 32, Figure 33, Figure 34, Figure 35, and Figure 36 the execution details for Falkon-15, Falkon-60, Falkon-120, Falkon-180, and Falkon-inf respectively. The number following Falkon (e.g. 15, 60, 120, 180, inf) denote the allowable number of seconds a processor was allowed to be idle before it would de-allocate it to avoid wasting resources. These figures show the instantaneous number of allocated, registered, and active executors over time.

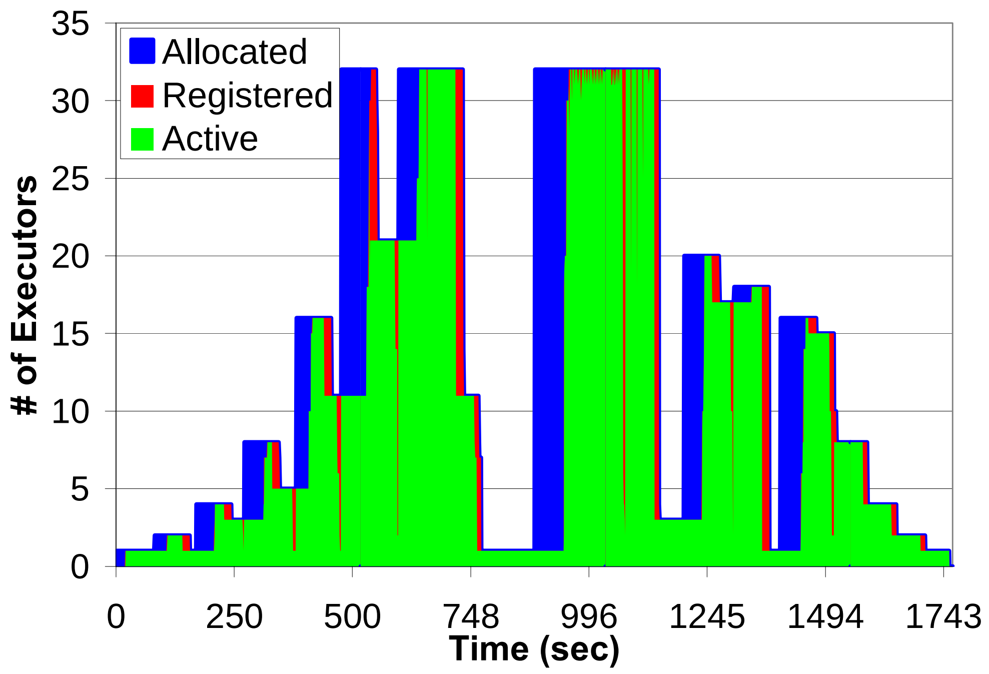


Figure 32: Dynamic Resource Provisioning: Synthetic workload for Falkon-15

Allocated (blue) are executors for which creation and registration are in progress. Creation and registration time can vary between 5 and 65 secs, depending on when a creation request is submitted relative to the PBS scheduler polling loop, which we believe occurs at 60 second intervals. JVM startup time and registration generally consume less than five secs. Registered executors (red) are ready to process tasks, but are not active. Finally, active executors (green) are actively processing tasks. In summary, blue is startup cost, red is wasted resources, and green is utilized resources. We see that Falkon-15 has fewer idle resources (as they are released sooner) but spends more time acquiring resources and overall has a longer total execution time than Falkon-60.

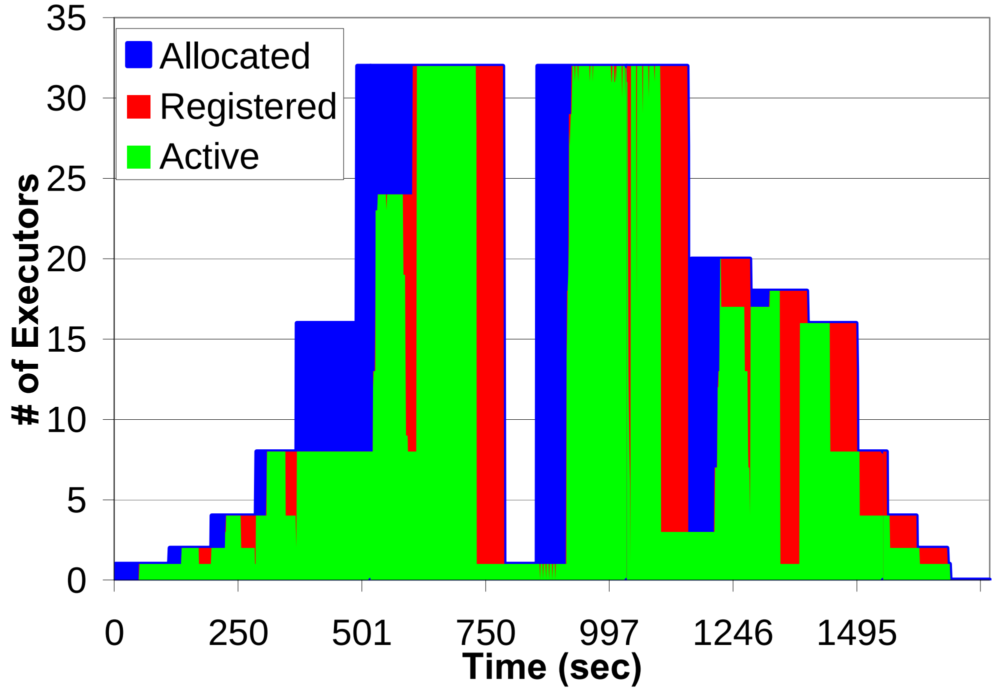


Figure 33: Dynamic Resource Provisioning: Synthetic workload for Falkon-60

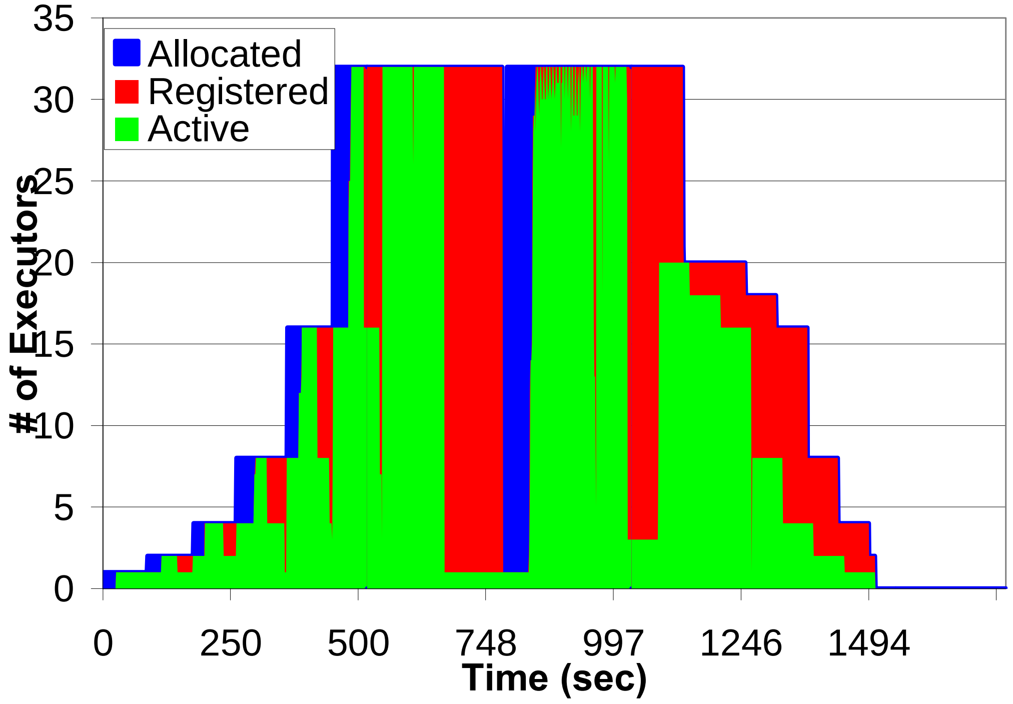


Figure 34: Dynamic Resource Provisioning: Synthetic workload for Falkon-120

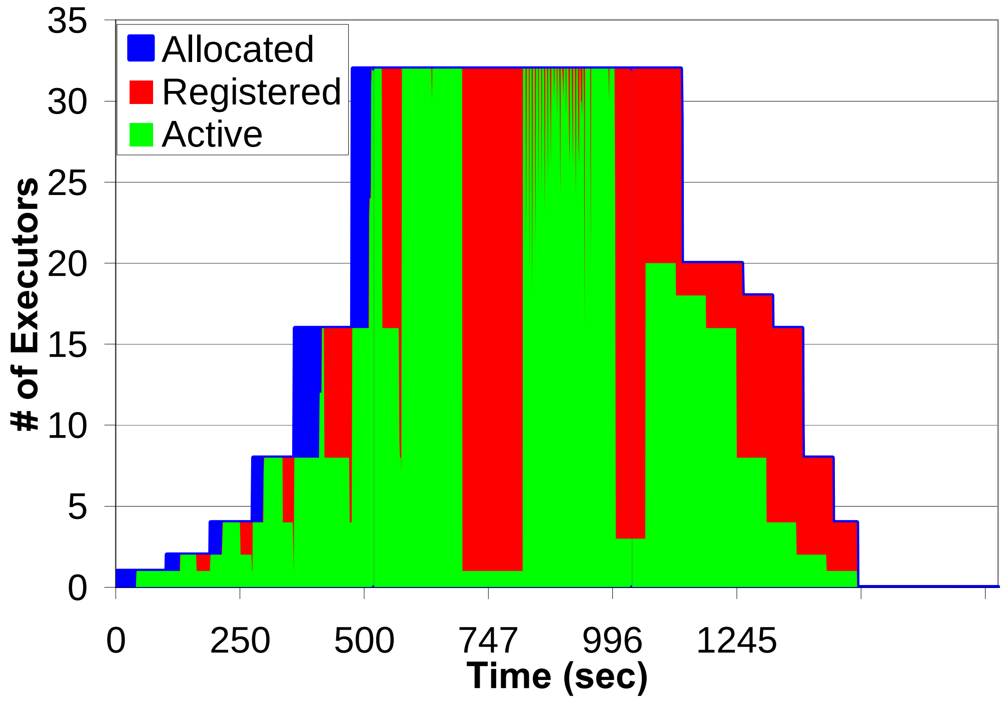


Figure 35: Dynamic Resource Provisioning: Synthetic workload for Falkon-180

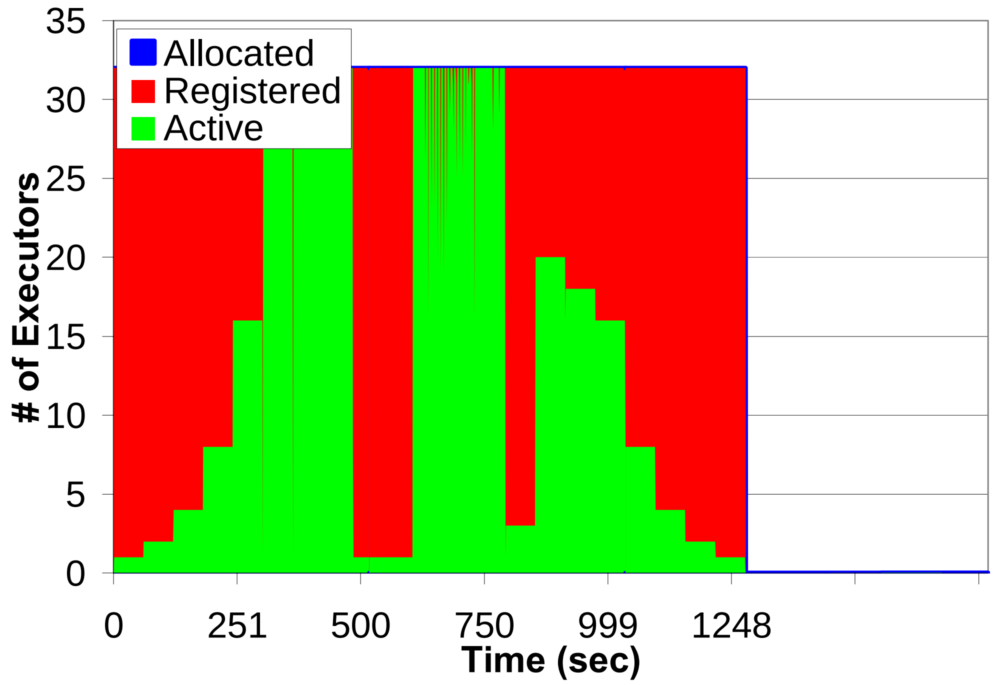


Figure 36: Dynamic Resource Provisioning: Synthetic workload for Falkon-inf

5.4 Conclusions

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

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

Dynamic resource provisioning can lead to significant savings in end-to-end application execution time, enable the use of batch-scheduled Grids for interactive use, and alleviate the high queue wait times typically found in production Grid environments. We have described a dynamic resource provisioning architecture and presented performance results we obtained on the TeraGrid. We have also integrated the dynamic resource provisioning into Falkon, a Fast and Light-weight tasK executiON framework, which allowed us to measure various performance aspects of resource provisioning with both real applications and synthetic workloads. Microbenchmarks show that DRP can allocate resources on the order of 10s of seconds across multiple Grid sites and can reduce average queue wait times by up to 95% (effectively yielding queue wait times within 3% of ideal).

Chapter 6 covers more results on dynamic resource provisioning for a variety of data-intensive workloads.

6 Towards Data Intensive Many-Task Computing with Data Diffusion

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 to support many-task computing commonly found in scientific computing (i.e. the reliance on parallel file systems with static configurations) do not scale to today's largest systems for data intensive application, as the rate of increase in the number of processors per system is outgrowing the rate of performance increase of parallel file systems. We argue that in such circumstances, data locality is critical to the successful and efficient use of large distributed systems for data-intensive applications. We propose a “data diffusion” approach to enable data-intensive many-task computing. Data diffusion acquires compute and storage resources dynamically, replicates data in response to demand, and schedules computations close to data, effectively harnessing data locality in application data access patterns. As demand increases, more resources are acquired, thus allowing faster response to subsequent requests that refer to the same data; when demand drops, resources are released. This approach can provide the benefits of dedicated hardware without the associated high costs, depending on workload and resource characteristics. To explore the feasibility of data diffusion, we offer both a theoretical and empirical analysis. We define an abstract model for data diffusion, define and implement scheduling policies with heuristics that optimize real world performance, and develop a competitive online caching eviction policy. We also offer many empirical experiments to explore the benefits of data diffusion, both under static and dynamic resource provisioning. We show performance improvements of one to two orders of magnitude across three diverse workloads when compared to the performance of parallel file systems with throughputs approaching 80Gb/s on a modest cluster of 200 processors. We also compare data diffusion with a best model for active storage, contrasting the difference between a pull-model found in data diffusion and a push-model found in active storage, on up to 5832 processors. We finally conclude with some preliminary results for collective I/O, an alternative technique to data diffusion we have used to scale data-intensive applications to the largest supercomputers with 98K processors.

The results presented in this chapter have been published in [1, 3, 14, 17, 19, 21, 23].

6.1 Overview

Within the science domain, the data that needs to be processed generally grows faster than computational resources and their speed. The scientific community is facing an imminent flood of data expected from the next generation of experiments, simulations, sensors and satellites. Scientists are now attempting calculations requiring orders of magnitude more computing and communication than was possible only a few years ago. Moreover, in many currently planned and future experiments, they are also planning to generate several orders of magnitude more data than has been collected in the entire human history [39].

Many applications in the scientific computing generally use a shared infrastructure such as TeraGrid [44] and Open Science Grid [45], where data movement relies on shared or parallel file systems. The rate of increase in the number of processors per system is outgrowing the rate of performance increase of parallel file systems, which requires rethinking existing data management techniques. For example, a cluster that was placed in service in 2002 with 316 processors has a parallel file system (i.e. GPFS [46]) rated at 1GB/s, yielding 3.2MB/s per processor of bandwidth. The second largest open science supercomputer, the IBM Blue Gene/P from Argonne National Laboratory, has 160K processors, and a parallel file system (i.e. also GPFS) rated at 8GB/s, yielding a mere 0.05MB/s per processor. That is a 65X reduction in bandwidth between a system from 2002 and one from 2008. Unfortunately, this trend is not bound to stop, as advances multi-core and many-core processors will increase the number of processor cores one to two orders of magnitude over the next decade. [18]

We believe that data locality is critical to the successful and efficient use of large distributed systems for data-intensive applications [19, 47] in the face of a growing gap between compute power and storage performance. Large scale data management needs to be a primary objective for any middleware targeting to support MTC workloads, to ensure data movement is minimized by intelligent data-aware scheduling both among distributed computing sites, and among compute nodes.

We propose an alternative *data diffusion* approach [1], in which resources required for data analysis are acquired dynamically from a local resource manager (LRM), in response to demand. Resources may be acquired either “locally” or “remotely”; their location only matters in terms of associated cost tradeoffs. Both data and applications “diffuse” to newly acquired resources for processing. Acquired resources and the data that they hold can be cached for some time, allowing more rapid responses to subsequent requests. Data diffuses over an increasing number of processors as demand increases, and then contracts as load reduces, releasing processors back to the LRM for other uses.

Data diffusion involves a combination of dynamic resource provisioning, data caching, and data-aware scheduling. The approach is reminiscent of cooperative caching [133], cooperative web-caching [134], and peer-to-peer storage systems [135]. Other data-aware scheduling approaches tend to assume static resources [106, 136], in which a system configuration dedicates nodes with roles (i.e. clients, servers) at startup, and there is no support to increase or decrease the ratio between client and servers based on load. However, in our approach we need to acquire dynamically not only storage resources but also computing resources. In addition, datasets may be terabytes in size and data access is for analysis (not retrieval). Further complicating the situation is our limited knowledge of workloads, which may involve many different applications. In principle, data diffusion can provide the benefits of dedicated hardware without the associated high costs. The performance achieved with data diffusion depends crucially on the characteristics of application workloads and the underlying infrastructure.

This chapter is organized as follows. Section 6.2 covers our proposed support for data-intensive many-task computing, specifically through our work with the Falkon [4, 2] light-weight task execution framework and its data management capabilities in data diffusion [1, 3, 23, 21]. This section discusses the data-aware scheduler and scheduling policies. Section 6.3 defines a *data diffusion abstract model*; towards developing provable results we offer 2Mark, an *O(NM)*-competitive caching eviction policy, for a constrained problem on *N* stores each holding at most *M* pages. This is the best possible such algorithm with matching upper and lower bounds (barring a constant factor). Section 6.4 offer a wide range of micro-benchmarks evaluating data diffusion, our parallel file system performance, and the data-aware scheduler performance. Section 6.5 explores the benefits of both static and dynamic resource provisioning through three synthetic workloads. The first two workloads explore dynamic resource provisioning through the Monotonically-Increasing workload and the Sin-Wave workload. We also explore the All-Pairs workload [137] which allows us to compare data diffusion with a best model for active storage [138]. Section 7.7 covers a real large-scale application from the astronomy domain, and how data diffusion improved its performance and scalability.

Section 2.5 covers related work, which have addressed data management issues to support data intensive applications. We finally conclude the chapter with Section 6.6.

6.2 Data Diffusion Architecture

We implement data diffusion [1] in the Falkon task dispatch framework [4]. We describe Falkon and data diffusion, offer justifications to why we chose a centralized scheduler approach, and finally discuss the data-aware scheduler and its various scheduling policies.

6.2.1 Falkon and Data Diffusion

To enable the rapid execution of many tasks on distributed resources, Falkon combines (1) multi-level scheduling [92] to separate resource acquisition (via requests to batch schedulers) from task dispatch, and (2) a streamlined dispatcher to achieve several orders of magnitude higher throughput (487 tasks/sec) and scalability (54K executors, 2M queued tasks) than other resource managers [4]. Recent work has achieved throughputs in excess of 3750 tasks/sec and scalability up to 160K processors [2].

Figure 37 shows the Falkon architecture, including both the data management and data-aware scheduler components. Falkon is structured as a set of (dynamically allocated) *executors* that cache and analyze data; a *dynamic resource provisioner* (DRP) that manages the creation and deletion of executors; and a *dispatcher* that dispatches each incoming task to an executor. The provisioner uses tunable allocation and de-allocation policies to provision resources adaptively. Falkon supports the queuing of incoming tasks, whose length triggers the dynamic resource provisioning to allocate resources via

GRAM4 [127] from the available set of resources, which in turn allocates the resources and bootstraps the executors on the remote machines. Individual executors manage their own caches, using local eviction policies, 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 [139] 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 37, 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.

In our experiments, 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 [48]). Thus, we avoid complicated and expensive cache coherence schemes other parallel file systems enforce. We implement four cache eviction policies:

Random, *FIFO*, *LRU*, and *LFU* [133]. Our empirical experiments all use LRU, and we will study the other policies in future work.

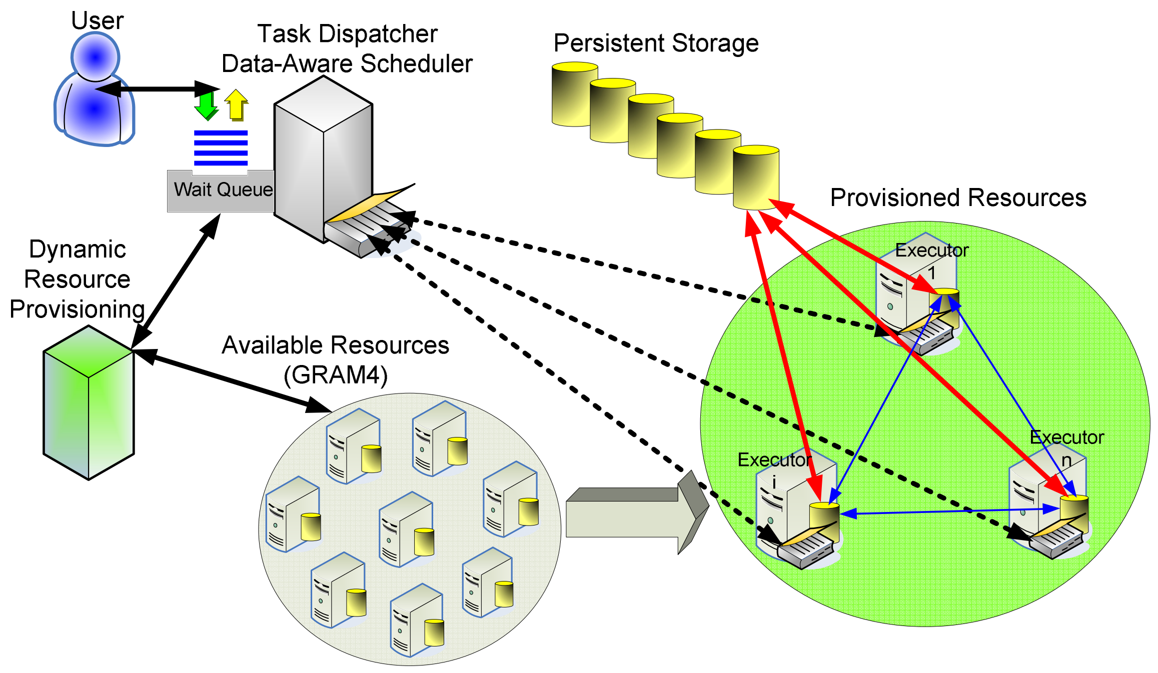


Figure 37: Architecture overview of Falkon extended with data diffusion (data management and data-aware scheduler)

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 [48]. 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. The next section (Section 6.2.2) covers a deeper analysis in the difference between a centralized index and a distributed one, and under what conditions a distributed index is preferred.

6.2.2 Centralized vs. Distributed Cache Index

Our central index and the separate per-executor indices are implemented as in-memory hash tables. The hash table implementation in Java 1.5 requires about 200 bytes per entry, allowing for index sizes of 8M entries with 1.5GB of heap, and 43M entries with 8GB of heap. Update and lookup performance on the hash table is excellent, with insert times in the 1~3 microseconds range (tested on up to 8M entries), and lookup times between 0.25 and 1 microsecond (tested on up to 8M entries) on a modern 2.3GHz Intel Xeon processor. Thus, we can achieve an upper bound throughput of 4M lookups/sec.

In practice, the scheduler may make multiple updates and lookups per scheduling decision, so the effective scheduling throughput that can be achieved is lower. Falkon's non-data-aware load-balancing scheduler can dispatch tasks at rates of 3800 tasks/sec on an 8-core system, which reflects the costs of communication. In order for the data-aware scheduler to not become the bottleneck, it needs to make decisions within 2.1 milliseconds, which translates to over 3700 updates or over 8700 lookups to the hash table. Assuming we can keep the number of queries or updates within these bounds per scheduling decision, the rate-liming step remains the communication between the client, the service, and the executors.

Nevertheless, our centralized index could become saturated in a sufficiently large enough deployment. In that case, a more distributed index might perform and scale better. Such an index could be implemented using the peer-to-peer replica location service (P-RLS) [100] or distributed hash table (DHT) [140]. Chervenak et al. [100] report that P-RLS lookup latency for an index of 1M entries increases from 0.5 ms to just over 3 ms as the number of P-RLS nodes grows from 1 to 15 nodes. To compare their data with a central index, we present in Figure 38. We see that although P-RLS latencies do not increase significantly with number of nodes (from 0.5 ms with 1 node to 15 ms with 1M nodes) [1], a considerable number of nodes are required to match that of an in-memory hash table. P-RLS would need more than 32K nodes to achieve an aggregate throughput similar to that of an in-memory hash table, which is 4.18M lookups/sec. In presenting these results we do not intend to argue that we need 4M+ lookups per second to maintain 4K scheduling decisions per second. However, these results do lead us to conclude that a centralized index can often perform better than a distributed index at small to modest scales.

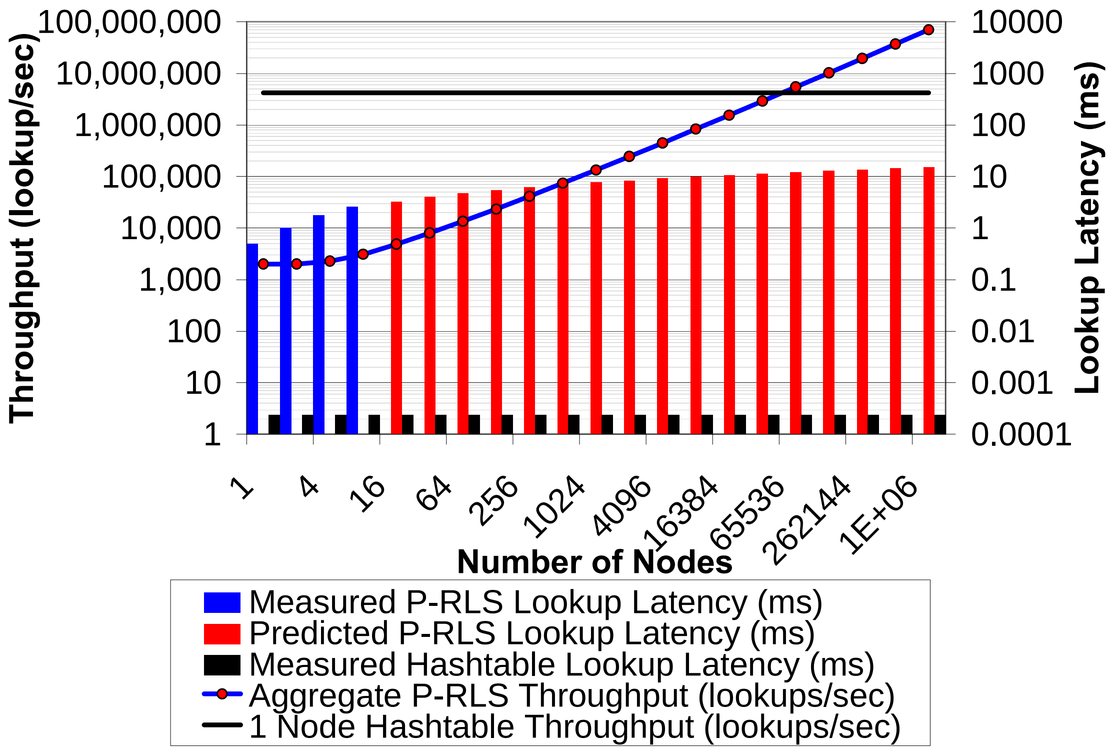


Figure 38: P-RLS vs. Hash Table performance for 1M entries

There are two disadvantages to our centralized index. The first is the requirement that the index fit in memory. Single SMP nodes can be bought with 256GB of memory, which would allow 1.3B entries in the index. Large-scale applications that are data intensive [137, 12, 66, 68] typically have terabytes to tens of terabytes of data spread over thousands to tens of millions of files, which would comfortably fit on a single node with 8GB of memory. However, this might not suffice for applications that have datasets with many small files. The second disadvantage is the single point of failure; it is worth noting that other systems, such as Hadoop [48], also have a single point of failure in the NameNode which keeps track of the global state of data. Furthermore, our centralized index load would be lower than that of Hadoop as we operate at the file level, not block level, which effectively reduces the amount of metadata that must be stored at the centralized index.

We have investigated distributing the entire Falkon service in the context of the IBM Blue Gene/P supercomputer, where we run N dispatchers in parallel to scale to 160K processors; we have tested N up to 640. However, due to limitations of the operating systems on the compute nodes, we do not yet support data diffusion on this system. Furthermore, the client submitting the workload is currently not dispatcher-aware to optimize data locality across multiple dispatchers, and currently only performs load-balancing in the event that the dispatcher is distributed over multiple nodes. There is no technical reason for not adding this feature to the client, other than not having the need for this feature so far. An alternative solution would be to add support for synchronization among the distributed dispatchers, to allow them to forward tasks amongst each other to optimize data locality. We will explore both of these alternatives in future work.

6.2.3 Data-Aware Scheduler

Data-aware scheduling is central to the success of data diffusion, as harnessing data-locality in application access patterns is critical to performance and scalability. We implement four dispatch policies.

The **first-available** (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 **max-cache-hit** (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 **max-compute-util** (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 **good-cache-compute** (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.

The scheduler is a window based one, that takes the scheduling window W size (i.e. $|W|$ is the number of tasks to consider from the wait queue when making the scheduling decision), and starts to build a per task scoring cache hit function. If at any time, a best task is found (i.e. achieves a 100% hit rate to the local cache), the scheduler removes this task from the wait queue and adds it to the list of tasks to dispatch to this executor. This is repeated until the maximum number of tasks were retrieved and prepared to be sent to the executor. If the entire scheduling window is exhausted and no best task was found, the m tasks with the highest cache hit local rates are dispatched. In the case of MCU, if no tasks were found that would yield any cache hit rates, then the top m tasks are taken from the wait queue and dispatched to the executor. For MCH, no tasks are returned, signaling that the executor is to return to the free pool of executors. For GCC, the aggregate CPU utilization at the time of scheduling decision determines which action to take. Pre-binding of tasks to nodes can negatively impact cache-hit performance if multiple tasks are assigned to the same node, and each task requires the entire cache size, effectively thrashing the cache contents at each task invocation. In practice, we find that per task working sets are small (megabytes to gigabytes) while cache sizes are bigger (tens of gigabytes to terabytes) making the worst case not a common case.

We define several variables first in order to understand the scheduling algorithm pseudo-code (see Figure 39 and Figure 40); the algorithm is separated into two sections, as the first part (Figure 39) decides which executor will be notified of available tasks, while the second part (Figure 40) decides which task to be submitted to the respective executor:

Q	wait queue
T_i	task at position i in the wait queue
E_{set}	executor sorted set; element existence indicates the executor is free, busy, or pending
I_{map}	file index hash map; the map key is the file logical name and the value is an executor sorted set of where the file is cached
E_{map}	executor hash map; the map key is the executor name, and the value is a sorted set of logical file names that are cached at the respective executor
W	scheduling window of tasks to consider from the wait queue when making the scheduling decision

The scheduler's complexity varies with the policy used. For FA, the cost is constant, as it simply takes the first available executor and dispatches the first task in the queue. MCH, MCU, and GCC are more complex with a complexity of $O(|T_i| + \min(|Q|, W))$, where T_i is the task at position i in the wait queue and Q is the wait queue. This could equate to many operations for a single scheduling decision, depending on the maximum size of the scheduling window and queue length. Since all data structures used to keep track of executors and files use in-memory hash maps and sorted sets, operations are efficient (see Section 6.2.2).

```
while Q !empty
  foreach files in $T_0$
    tempSet = $I_{map}(file_i)$
    foreach executors in tempSet
      candidates[$tempSet_j$]++
    end
  end
  sort candidates[] according to values
  foreach candidates
      if $E_{set}(candidate_i)$ = freeState then
        Mark executor $candidate_i$ as pending
      Remove $T_0$ from wait queue and mark as pending
        sendNotificatoin to $candidate_i$ to pick up $T_0$
        break
      end
      if no candidate is found in the freeState then
        send notification to the next free executor
      end
    end
  end
```

Figure 39: Pseudo Code for part #1 of algorithm which sends out the notification for work

```
while tasksInspected < W
  $fileSet_i$ = all files in $T_i$
  $cacheHit_i$ = |intersection $fileSet_i$ and $E_{map}$(executor)|
  if $cacheHit_i$ > minCacheHit || CPUutil < minCPUutil then
    remove $T_i$ from Q and add $T_i$ to list to dispatch
  end
  if list of tasks to dispatch is long enough then
    assign tasks to executor
    break
   end
end
```

Figure 40: Pseudo Code for part #2 of algorithm which decides what task to assign to each executor

6.3 Theoretical Evaluation

We define an abstract model that captures the principal elements of data diffusion in a manner that allows analysis. We first define the model and then analyze the computational time per task, caching performance, workload execution times, arrival rates, and node utilization. Finally, we present an O(NM)-competitive algorithm for the scheduler as well as a proof of its competitive ratio.

6.3.1 Abstract Model

Our abstract model includes computational resources on which tasks execute, storage resources where data needed by the tasks is stored, etc. Simplistically, we have two regimes: the working data set fits in cache, S≥W, where S is the aggregate allocated storage and W is the working data set size; and the working set does not fit in cache, S<W. We can express the time T required for a computation associated with a single data access as follows (see Equation 3), both depending on H_l (data found on local disk), H_c (remote disks), or H_s (centralized persistent storage):

$$S \geq W \quad : (R_l+C) \leq T \leq (R_c+C)$$

$$S < W \quad : (R_c+C) \leq T < (R_s+C)$$

Equation 3: Time T required to complete computation with a single data access

Where R_l, R_c, R_s are the average cost of accessing local data (l), cached data (c), or persistent storage (s), and C is the average amount of computing per data access. The relationship between cache hit performance and T can be found in Equation 4.

$$S \geq W \quad : T = (R_l+C)*HR_l + (R_c+C)*HR_c$$

$$S < W \quad : T = (R_c+C)*HR_c + (R_s+C)*HR_s$$

Equation 4: Relationship between cache hit performance and time T

Where HR_l is the cache hit local disk ratio, HR_c is the remote cache ratio, and HR_s is the cache miss ratio; $HR_{l/c/s} = H_{L/C/S}/(H_L + H_C + H_S)$. We can merge the two cases into a single one, such as the average time to complete task i is $TK_i = (R_l+C)*HR_l+(R_c+C)*HR_c+(R_s+C)*HR_s$. This can also be expressed as (see Equation 5):

$$TK_i = C + R_l*HR_l + R_c*HR_c + R_s*HR_s$$

Equation 5: Average time to complete task i

The time needed to complete an entire workload D with K tasks on N processors is, where D is a function of K, W, A, C, and L (see Equation 6):

$$T_N(D) = \sum_{i=1}^{K} TK_i$$

Equation 6: Time to complete an entire workload D

Having defined the time to complete workload D, we define speedup as Equation 7:

$$SP = T_1(D) / T_N(D)$$

Equation 7: Speedup

And efficiency can be defined as (see Equation 8):

$$EF = SP / N$$

Equation 8: Efficiency

What is the maximum task arrival rate (A) that a particular scenario can sustain? We have (see Equation 9):

$$S \geq W \quad : N*P/(R_l+C) \leq A_{max} \leq N*P/(R_c+C)$$
$$S < W \quad : N*P/(R_c+C) \leq A_{max} < N*P/(R_s+C)$$

Equation 9: Maximum task arrival rate (A) that can be sustained

Where P is the execution speed of a single node. These regimes can be collapsed into a single formula (see Equation 10):

$$A = (N*P/T)*K$$

Equation 10: Arrival rate

We can express a formula to evaluate tradeoffs between node utilization (U) and arrival rate; counting data movement time in node utilization, we have (see Equation 11):

$$U = A*T/(N*P)$$

Equation 11: Utilization

Although the presented model is quite simplistic, it manages to model quite accurately an astronomy application with a variety of workloads (the topic of Section 7.7.6)

6.3.2 *O(NM)*-Competitive Caching

Among known algorithms with provable performance for minimizing data access costs, there are none that can be applied to data diffusion, even if restricted to the caching problem it entails. For instance, LRU maximizes the local store performance, but is oblivious of the cached data in the system and persistent storage. As a step to developing a provably sound algorithm we present an online algorithm that is *O(NM)*-competitive to the offline optimum for a constrained version of the caching problem. For definitions of competitive ratio, online algorithm, and offline optimum see [142]. In brief, an online algorithm solves a problem without knowledge of the future, an offline optimal is a hypothetical algorithm that has knowledge of the future. The competitive ratio is the worst-case ratio of their performance and is a measure of the quality of the online algorithm, independent of a specific request sequence or workload characteristics.

In the constrained version of the problem there are N stores each capable of holding M objects of uniform size. Requests are made sequentially to the system, each specifying a particular object and a particular store. If the store does not have the object at that time, it must load the object to satisfy the request. If the store is full, it must evict one object to make room for the new object. If the object is present on another store in the system, it can be loaded for a cost of R_c, which we normalize to 1. If it is not present in another store, it must be loaded from persistent storage for a cost of R_s, which we normalize to $s = R_s / R_c$. Note that if $R_s < R_c$ for some reason, we can use LRU at each node instead of $\mathsf{2Mark}$ to maintain competitive performance. We assume R_l is negligible.

All stores in the system our allowed to cooperate (or be managed by a single algorithm with complete state information). This allows objects to be transferred between stores in ways not directly required to satisfy a request (e.g., to back up an object that would otherwise be evicted). Specifically, two stores may exchange a pair of objects for a cost of *1* without using an extra memory space. Further, executors may write to an object in their store. To prevent inconsistencies, the system is not allowed to contain multiple copies of one object simultaneously on different stores.

We propose an online algorithm $\mathsf{2Mark}$ (which uses the well known marking algorithm [142] at two levels) for this case of data diffusion. Let the corresponding optimum offline algorithm be OPT. For a sequence σ, let $\mathsf{2Mark}(\sigma)$ be the cost $\mathsf{2Mark}$ incurs to handle the sequence and define $\mathsf{OPT}(\sigma)$ similarly. $\mathsf{2Mark}$ may mark and unmark objects in two ways, designated *local-marking* an object and *global-marking* an object. An object may be local-marked with respect to a particular store (a bit corresponding to the object is set only at that store) or global-marked with respect to the entire system. $\mathsf{2Mark}$ interprets the request sequence as being composed of two kinds of phases, *local-phases* and *global-phases*. A local-phase for a given store is a contiguous set of requests received by the store for *M* distinct objects, starting with the first request the store receives. A global-phase is a contiguous set of requests received by the entire system for *NM* distinct objects, starting with the first request the system receives. We prove

Equation 12 which establishes that 2Mark is *O(NM)-competitive*. From the lower bound on the competitive ratio for simple paging [142], this is the best possible deterministic online algorithm for this problem, barring a constant factor.

$$\text{2Mark}(\sigma) \leq (NM + 2M/s + NM/(s+v)) \cdot \text{OPT}(\sigma) \text{ for all sequences } \sigma$$

Equation 12: The relation we seek to prove to establish that 2Mark is O(NM)-competitive

2Mark essentially uses an *M*-competitive marking algorithm to manage the objects on individual stores and the same algorithm on a larger scale to determine which objects to keep in the system as a whole. When a store faults on a request for an object that is on another store, it exchanges the object it evicts for the object requested (see Figure 41). We will establish a bound on the competitive ratio by showing that every cost incurred by 2Mark can be correlated to one incurred by OPT. These costs may be *s-faults* (in which an object is loaded from persistent storage for a cost of *s*) or they may be *1-faults* (in which an object is loaded from another cache for a cost of *1*). The number of 1-faults and s-faults incurred by 2Mark can be bounded by the number of 1-faults and s-faults incurred by OPT in sequence σ, as described in the following.

To prevent multiple copies of the same object in different caches, we assume that the request sequence is *renamed* in the following manner: when some object *p* requested at a store *X* is requested again at a different store *Y* we rename the object once it arrives at *Y* to *p'* and rename all future requests for it at *Y* to *p'*. This is done for all requests. Thus if this object is requested at *X* in the future, the object and all requests for it at *X* are renamed to *p"*. This ensures that even if some algorithm inadvertently leaves behind a copy of *p* at *X* it is not used again when *p* is requested at *X* after being requested at *Y*. Observe that the renaming does not increase the cost of any correct algorithm.

Consider the *ith* global phase. During this global phase, let OPT load objects from persistent storage u times and exchange a pair of objects between stores v times, incurring a total cost of $su+v$.

Every object loaded from persistent storage by 2Mark is globally-marked and not evicted from the system until the end of the global phase. Since the system can hold at most NM objects, the number of objects loaded by 2Mark in the *ith* global phase is at most NM. We claim OPT loads at least one object from persistent storage during this global phase. This is trivially true if this is the first global phase as all the objects loaded by 2Mark have to be loaded by OPT as well. If this is not the first global phase, OPT must satisfy each of the requests for the distinct NM objects in the previous global phase by objects from the system and thus must s-fault at least once to satisfy requests in this global phase.

Within the *ith* global phase consider the *jth* local phase at some store X. The renaming of objects ensures that any object p removed from X because of a request for p at some other store Y is never requested again at X. Thus the first time an object is requested at X in this local phase, it is locally marked and remains in X for all future requests in this local phase. Thus X can 1-fault for an object only once during this local phase. Since X can hold at most M objects, it incurs at most M 1-faults in the *jth* local phase. We claim that when $j\neq1$ OPT incurs at least one 1-fault in this local phase. The reasoning is similar to that for the *ith* global phase: since OPT satisfies each of the requests for M distinct objects in the previous local phase from cache, it must 1-fault at least once in this local phase. When $j=1$, however, it may be that the previous local phase did not contain requests for M distinct objects. There are, however, at most NM 1-faults by 2Mark in all the local phases in which j=1, for the N stores each holding M objects, in the *ith* global phase.

Since OPT has the benefit of foresight, it may be able to service a pair of 1-faults through a single exchange. In this both the stores in the exchange get objects which are useful to them, instead of just one store benefiting from the exchange. Thus since OPT has v exchanges in the *ith* global phase, it may satisfy at most $2v$ 1-faults and 2Mark correspondingly has at most $2vM + NM$ 1-faults. The second term is due 1-faults in the first local phase for each store in this global phase.

Thus the total cost in the *ith* global phase by 2Mark is at most $sNM + 2vM + NM$, while that of OPT is at least $s+v$, since $u \geq 1$ in every global phase. This completes the proof.

```
Input:   Request for object $p$ at store $X$ from sequence $\sigma$
1  if $p$ is not on $X$ then
2    if $X$ is not full then /* No eviction required */
3      if $p$ is on some store $Y$ then
4        Transfer $p$ from $Y$ to $X$
5      else
6        Load $p$ to $X$ from persistent storage
7      end
8    else /* Eviction required to make space in X */
9      if all objects on $X$ are local-marked then
10         local-unmark all  /*Begins new local phase */
11       end
12       if p is on some store $Y$ then
13         Select an arbitrary local-unmarked object q on $X$
14         Exchange $q$ and $p$ on $X$ and $Y$
       /* X now has $p$ and $Y$ has $q$ */
15         if $p$ was local-marked on $Y$ then
16           local-mark $q$ on $Y$
17         end
18       else /* p must be loaded from persistent storage */
19         if all objects in system are global-marked then
20           global-unmark and local-unmark all objects
             /*Begins new global phase  & local phases at each store */
21         end
22         if all objects on $X$ are global-marked then
23           Select an arbitrary local-unmarked object $q$ on $X$
24           Select an arbitrary store $Y$ with at least one global-unmarked object or empty space
25           Transfer $q$ to $Y$, replacing an arbitrary global-unmarked object or empty space
26         else
27           Evict an arbitrary global-unmarked object $q$ on $X$
28         end
29         Load $p$ to $X$ from persistent storage
30       end
31     end
32 end
33 global-mark and local-mark $p$
```

Figure 41: Algorithm 2Mark

6.4 Micro-Benchmarks

This section describes our performance evaluation of data diffusion using micro-benchmarks.

6.4.1 Testbed Description

Table 8 lists the platforms used in the micro-benchmark experiments. The UC_x64 node was used to run the Falkon service, while the TG_ANL_IA32 and TG_ANL_IA64 clusters [144] were used to run the executors. Both clusters are connected internally via Gigabit Ethernet, and have a shared file system (GPFS) mounted across both clusters that we use as the "persistent storage" in our experiments. The GPFS file system has 8 I/O nodes to handle the shared file system traffic. We assume a one-to-one mapping between executors and nodes in all experiments. Latency between UC_x64 and the compute clusters was between one and two milliseconds.

Table 8: Platform descriptions

Name	# of Nodes	Processors	Memory	Network
TG_ANL_IA32	98	Dual Xeon 2.4 GHz	4GB	1Gb/s
TG_ANL_IA64	64	Dual Itanium 1.3 GHz	4GB	1Gb/s
UC_x64	1	Dual Xeon 3GHz w/ HT	2GB	100Mb/s

6.4.2 File System Performance

In order to understand how well the proposed data diffusion works, we decided to model the shared file system (GPFS) performance in the ANL/UC TG cluster where we conducted all our experiments. The following graphs represent 160 different experiments, covering 19.8M files transferring 3.68TB of data and consuming 162.8 CPU hours; the majority of this time was spent measuring the GPFS performance, but a small percentage was also spent measuring the local disk performance of a single node. The dataset we used was composed of 5.5M files making up 2.4TB of data. In the hopes to eliminate as much variability or bias as possible from the results (introduced by Falkon itself), we wrote a simple program that took in some parameters, such as the input list of files, output directory, length of time to run experiment (while never repeating any files for the corresponding experiment); the program then randomized the input files and ran the workload of reading or reading+writing the corresponding files in 32KB chunks (larger buffers than 32KB didn't offer any improvement in read/write performance for our testbed). Experiments were ordered in such a manner that the same files would only be repeated after many other accesses, making the probability of those files being in any cache of the operating system or parallel file system I/O nodes small. Most graphs (unless otherwise noted) represent the GPFS read or read+write performance for 1 to 64 (1, 2, 4, 8, 16, 32, 64) concurrent nodes accessing files ranging from 1 byte to 1GB in size (1B, 1KB, 10KB, 100KB, 1MB, 10MB, 100MB, 1GB).

As a first comparison, we measured the read and read+write performance of one node as it performed the operations on both the local disk and the shared file system. Figure 42 shows that local disk is more than twice as fast when compared to the shared file system performance in all cases, a good motivator to favor local disk access whenever possible. Note that this gap would likely grow as the access patterns shift from few large I/O calls (as was the case in these experiments) to many small I/O calls (which is the case for many astronomy applications and datasets).

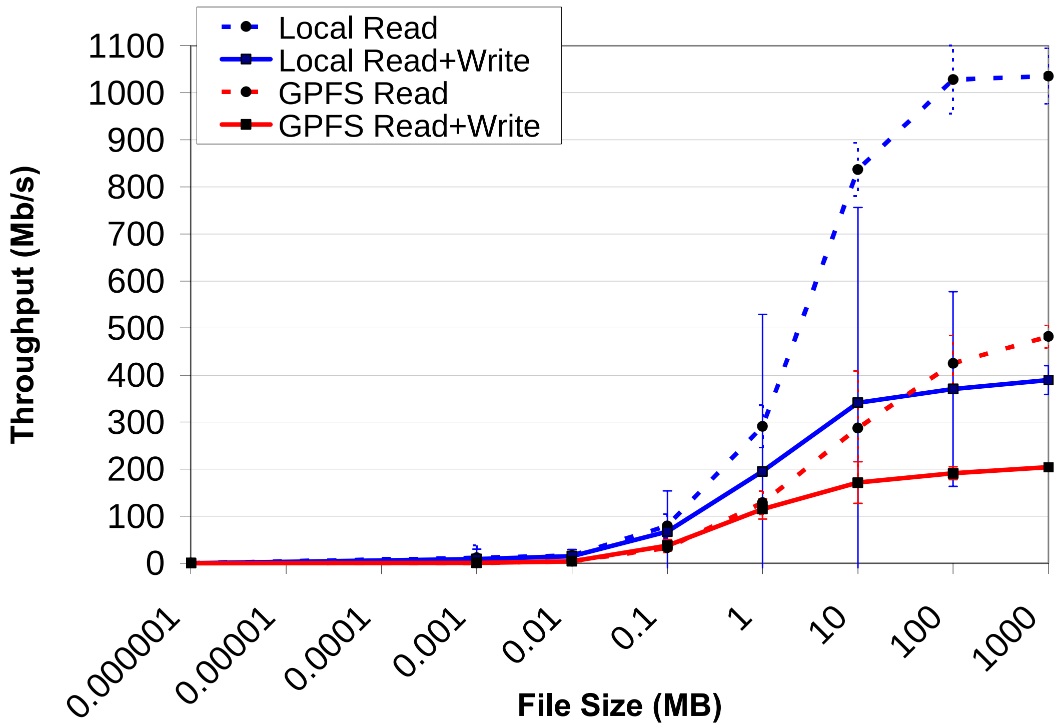


Figure 42: Comparing performance between the local disk and the shared file system GPFS from one node

Notice that the GPFS read performance (Figure 43) tops out at 3420 Mb/s for large files, and it can achieve 75% of its peak bandwidth with files as small as 1MB if there are enough nodes concurrently accessing GPFS. It is worth noting that the performance increase beyond 8 nodes is only apparent for small files; for large files, the difference is small (<6% improvement from 8 nodes to 64 nodes). This is due to the fact that there are 8 I/O servers serving GPFS, and 8 nodes are enough to saturate the 8 I/O servers given large enough files.

The read+write performance (Figure 44) is lower than that of the read performance, as it tops out at 1123Mb/s. Just as in the read experiment, there seems to be little gain from having more than 8 nodes concurrently accessing GPFS (with the exception of small files).

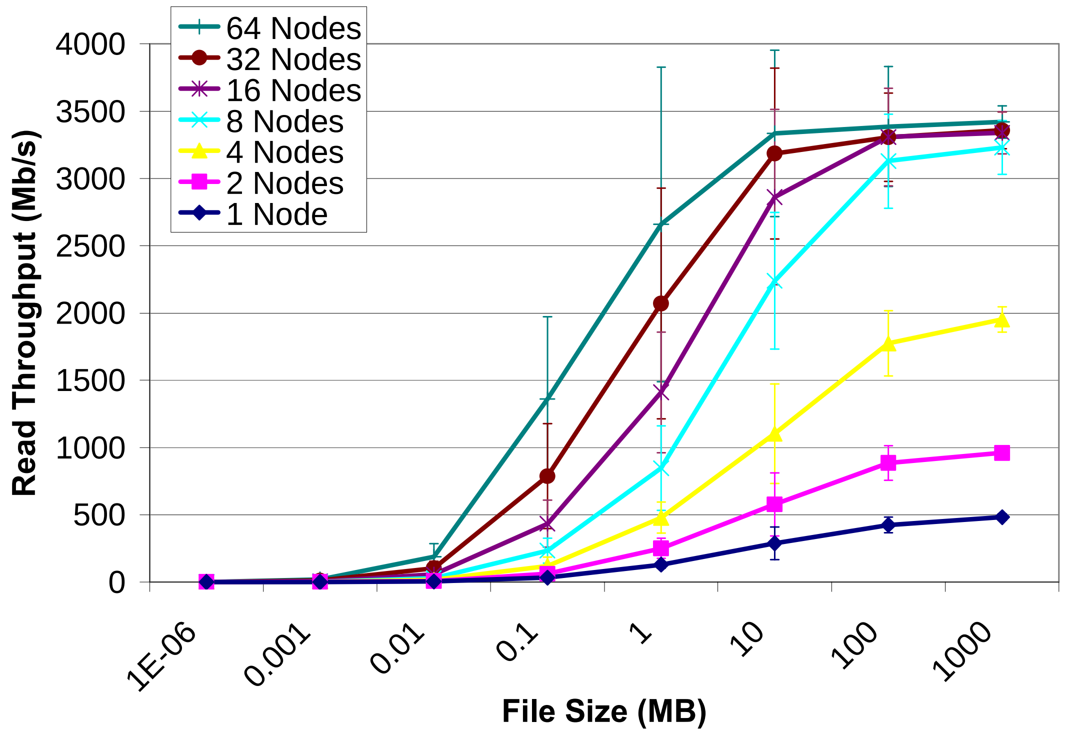


Figure 43: Read performance for GPFS expressed in Mb/s; only the x-axis is logarithmic; 1-64 nodes for GPFS; 1B – 1GB files

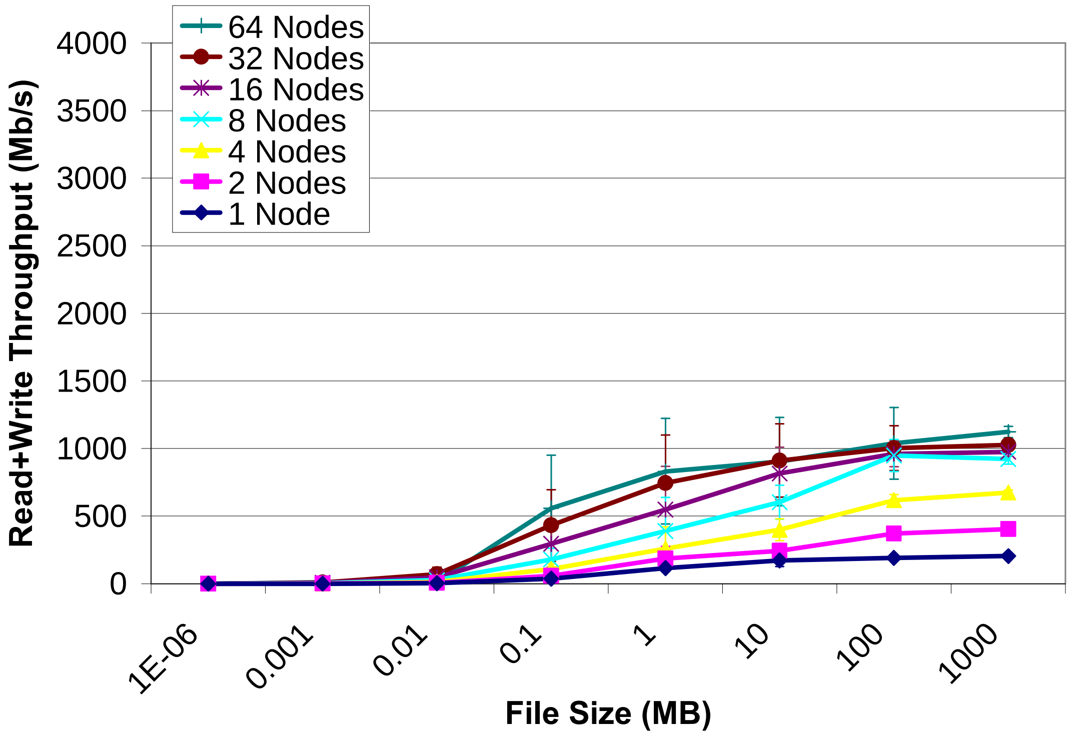


Figure 44: Read+write performance for GPFS expressed in Mb/s; only the x-axis is logarithmic; 1-64 nodes for GPFS; 1B – 1GB files

These final two graphs (Figure 45 and Figure 46) show the theoretical read+write and read throughput (measured in Mb/s) for local disk access. These results are theoretical, as they are simply a derivation of the 1 node performance (see Figure 42), extrapolated to additional nodes (2, 4, 8, 16, 32, 64) linearly (assuming that local disk accesses are completely independent of each other across different nodes). Notice the read+write throughput approaches 25GB/s (up from 1Gb/s for GPFS) and the read throughput 76Gb/s (up from 3.5Gb/s for GPFS). This upper bound potential is a great motivator for applications to favor the use of local disk over that of shared disk, especially as applications scale beyond the size of the statically configured number of I/O servers servicing the shared file systems normally found in production clusters and grids.

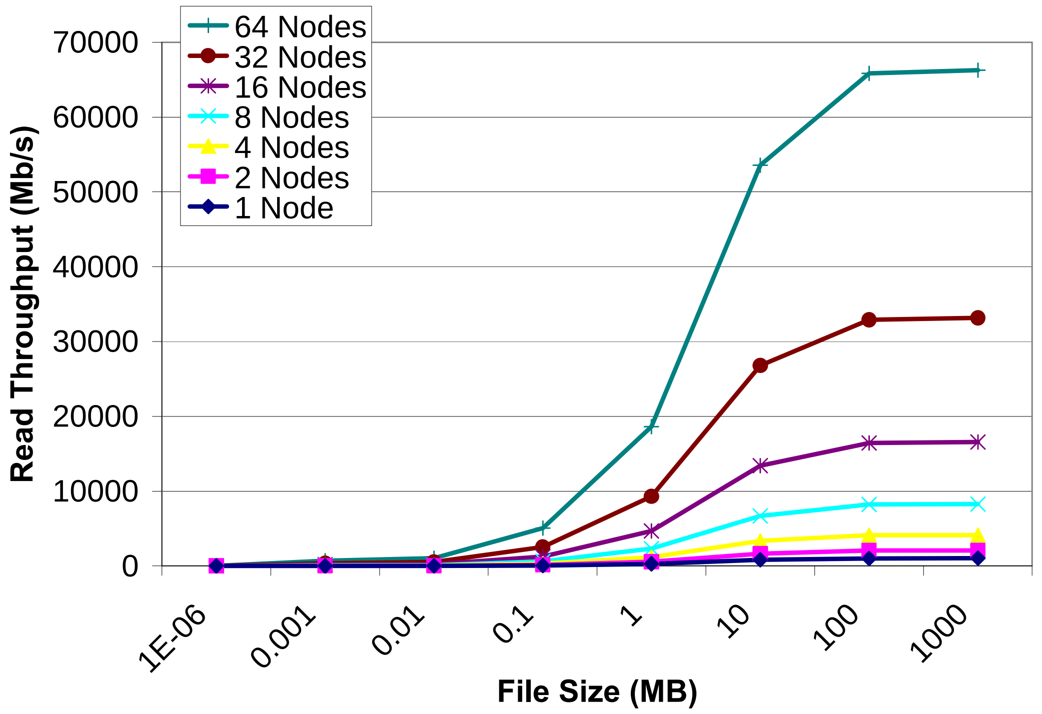


Figure 45: Theoretical read performance of local disks expressed in Mb/s; only the x-axis is logarithmic; 1-64 nodes for local disk access; 1B – 1GB files

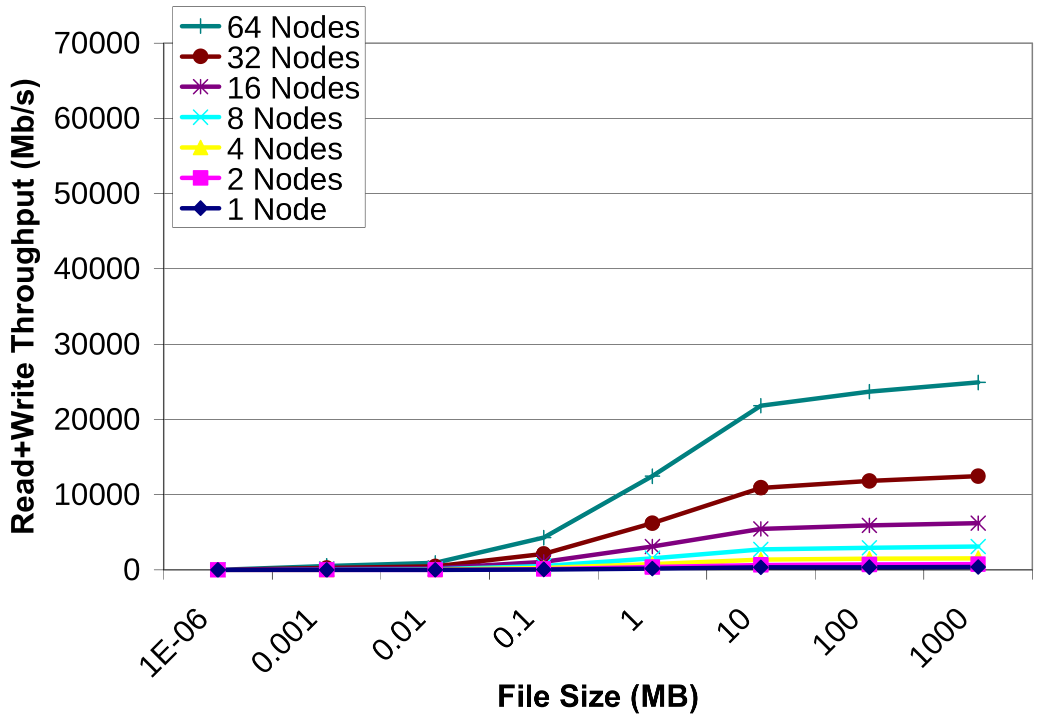


Figure 46 (local model 1-64 nodes r+w): Theoretical read+write performance of local disks expressed in Mb/s; only the x-axis is logarithmic; 1-64 nodes for local disk access; 1B – 1GB files

6.4.3 Data Diffusion Performance

We measured performance for five configurations, two variants (read and read+write), seven node counts (1, 2, 4, 8, 16, 32, 64), and eight file sizes (1B, 1KB, 10KB, 100KB, 1MB, 10MB, 100MB, 1GB), for a total of 560 experiments. For all experiments (with the exception of the 100% data locality experiments where the caches were warm), data was initially located only on persistent storage, which in our case was GPFS parallel file system. The six configurations are: Model (local disk), Model (persistent storage), FA Policy, MCU Policy (0% locality), and MCU Policy (100% locality).

Figure 47 shows read throughput for 100MB files, seven of the eight configurations, and varying numbers of nodes. Configuration (8) has the best performance: 61.7Gb/s with 64 nodes (~94% of ideal). Even the first-cache-available policy which dispatches tasks to executors without concern for data location performs better (~5.7Gb/s) than the shared file system alone (~3.1Gb/s) when there are more than 16 nodes. With eight or less nodes, data-unaware scheduling with 100% data locality performs worse than GPFS (note that GPFS also has eight I/O servers); one hypothesis is that data is not dispersed evenly among the caches, and load imbalances reduce aggregate throughput, but we need to investigate further to better understand the performance of data-unaware scheduling at small scales.

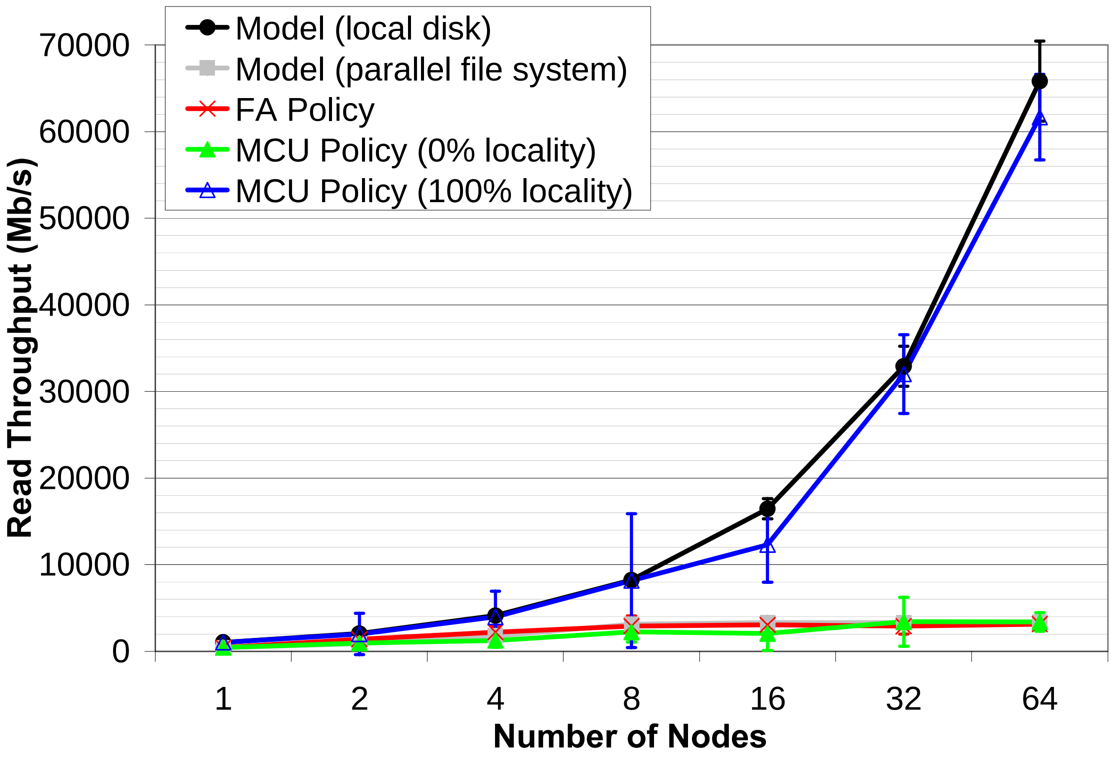


Figure 47: Read throughput (Mb/s) for large files (100MB) for seven configurations for 1 – 64 nodes

Figure 48 shows read+write performance, which is also good for the max-compute-util policy, yielding 22.7Gb/s (~96% of ideal). Without data-aware scheduling, throughput is 6.3Gb/s; when simply using persistent storage, it is a mere 1Gb/s. In Figure 47 and Figure 48, we omit configuration (4) as it had almost identical performance to configuration (3). Recall that configuration (4) introduced a wrapper script that created a temporary sandbox for the application to work in, and afterwards cleaned up by removing the sandbox. The performance of these two configurations was so similar here because of the large file sizes (100MB) used, which meant that the cost to create and remove the sand box was amortized over a large and expensive operation.

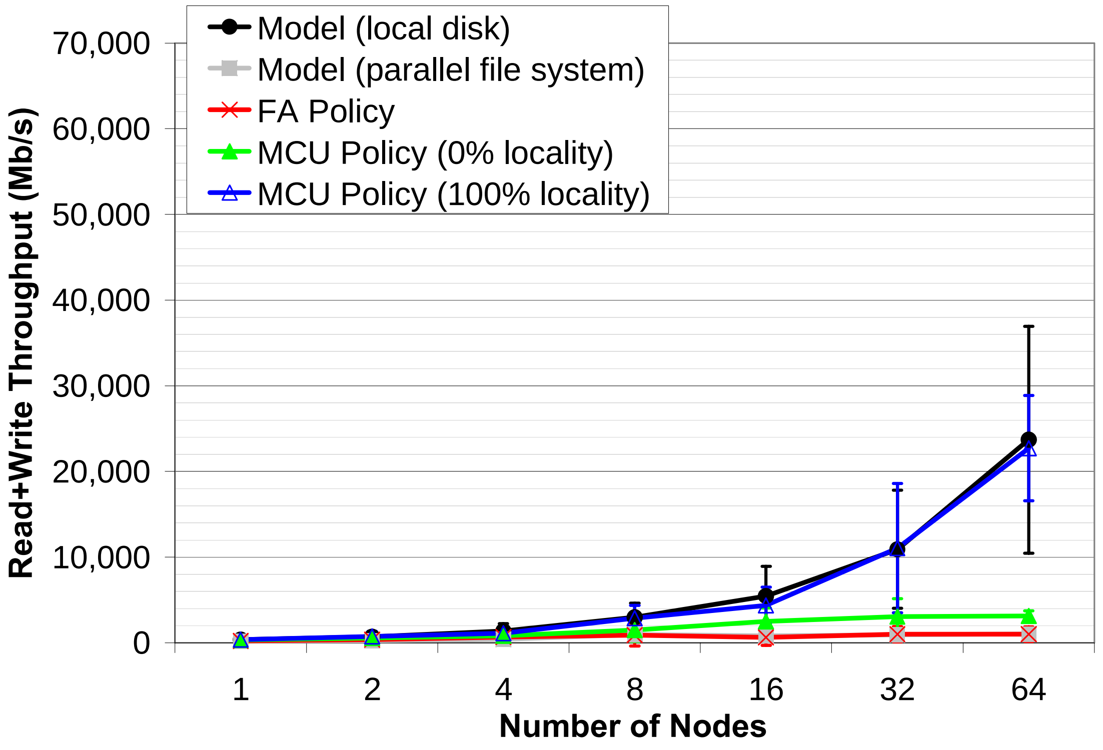


Figure 48: Read+Write throughput (Mb/s) for large files (100MB) for seven configurations and 1 – 64 nodes

In order to show some of the large overheads of parallel file systems such as GPFS, we execute the FA policy using a wrapper script similar to that used in many applications to create a sandbox execution environment. The wrapper script creates a temporary scratch directory on persistent storage, makes a symbolic link to the input file(s), executes the task, and finally removes the temporary scratch directory from persistent storage, along with any symbolic links. Figure 49 shows read and read+write performance on 64 nodes for file sizes ranging from 1B to 1GB and comparing the model performance with the FA policy with and without a wrapper. Notice that for small file sizes (1B to 10MB), the FA policy with wrapper had one order of magnitude lower throughput than those without the wrapper. We find that the best throughput that can be achieved by 64 concurrent nodes with small files is 21 tasks/sec. The limiting factor is the need, for every task, to create a directory on persistent storage, create a symbolic link, and remove the directory. Many applications that use persistent storage to read and write files from many compute processors use this method of a wrapper to cleanly separate the data between different application invocations. This offers further example of how GPFS performance can significantly impact application performance, and why data diffusion is desirable as applications scale.

Overall, the shared file system seems to offer good performance for up to eight concurrent nodes (mostly due to there being eight I/O nodes servicing GPFS), however when more than eight nodes require access to data, the data diffusion mechanisms significantly outperform the persistent storage system. The improved performance can be attributed to the linear increase in I/O bandwidth with compute nodes, and the effective data-aware scheduling performed.

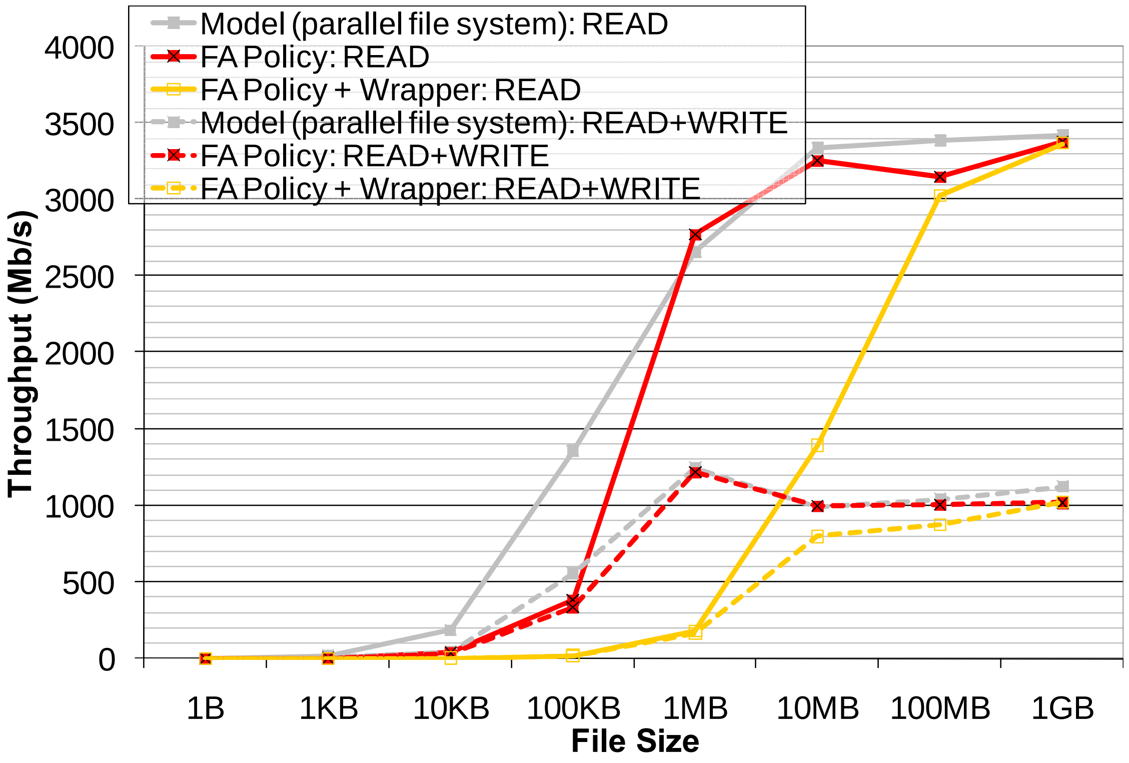


Figure 49: Read and Read+Write throughput (Mb/s) for a wide range of file sizes for three configurations on 64 nodes

6.4.4 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 50 shows the scheduler performance under different scheduling policies.

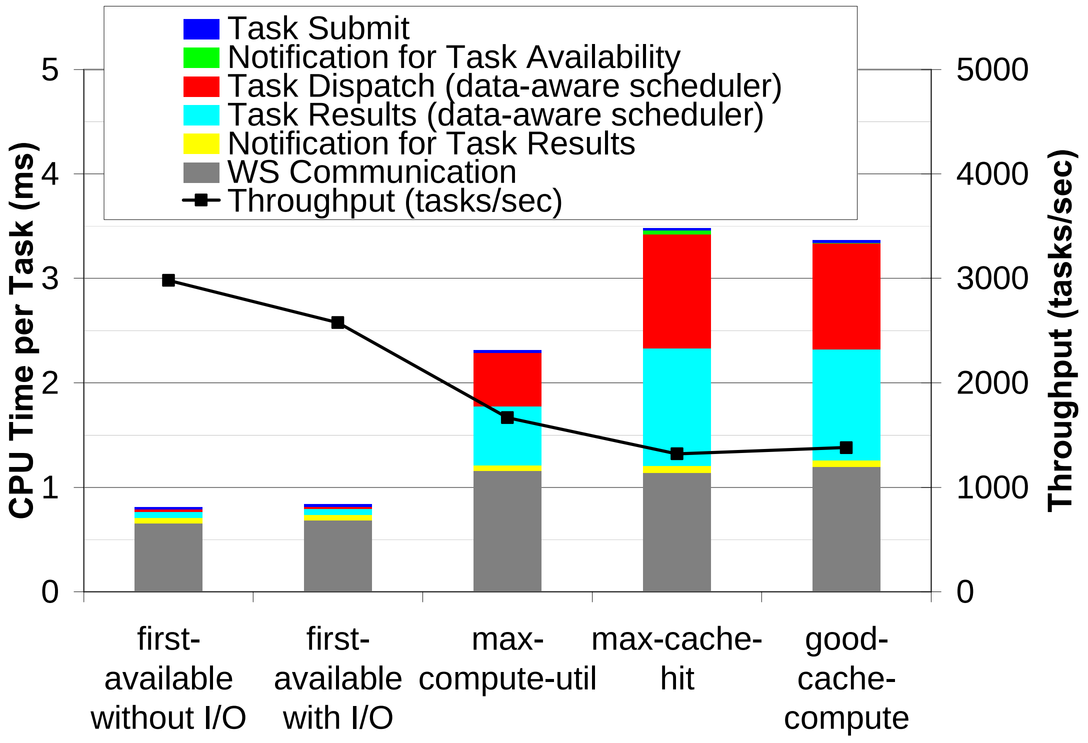


Figure 50: Data-aware scheduler performance and code profiling for the various scheduling policies

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). It is worth pointing out 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 3 data-aware policies, the scheduling costs (red and light blue areas) are more significant.

6.5 Synthetic Workloads

We measured the performance of the data-aware scheduler on various workloads, both with static (SRP) and dynamic (DRP) resource provisioning, and ran experiments on the ANL/UC TeraGrid [144] (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 [46] with <1ms latency to executors.

The three sub-sections that follow cover three diverse workloads: Monotonically-Increasing (MI: Section 6.5.1), Sine-Wave (SI: Section 6.5.2), and All-Pairs (AP: Section 6.5.3). We use workloads MI and SI to explore the dynamic resource provisioning support in data diffusion, and the various scheduling policies (e.g. FA, GCC, MCH, MCU) and cache sizes (e.g. 1GB, 1.5GB, 2GB, 4GB). We use the AP workload to compare data diffusion with active storage [137].

6.5.1 Monotonically Increasing Workload

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 increasing function is shown in Equation 13.

$$A_i = \min \left[\mathit{ceiling}(\ A_{i-1}\ *\ 1.3\ )\ ,\ 1000 \right],\ 0\ \le\ i\ <\ 24$$

Equation 13: Monotonically Increasing Workload arrival rate function

This 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 (see Section 6.2.2), 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. This workload is depicted in Figure 51.

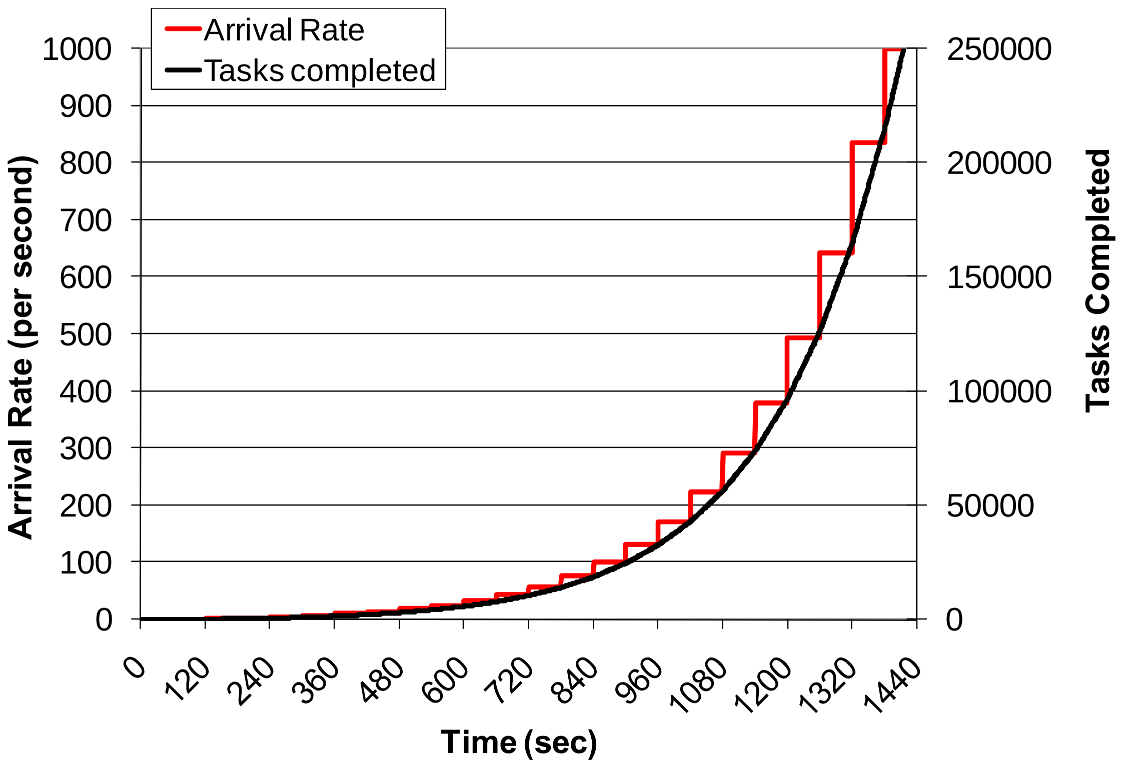


Figure 51: Monotonically Increasing workload overview

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). Several measured or computed metrics are relevant in understanding the following set of graphs:

Demand (Gb/s): throughput needed to satisfy arrival rate

Throughput (Gb/s): measured aggregate transfer rates

Wait Queue Length: number of tasks ready to run

Cache Hit Global: file access from a peer executor cache

Cache Hit Local: file access from local cache

Cache Miss: file accesses from the parallel file system

Speedup (SP): SP = T_N(FA) / T_N(GCC|MCH|MCU)

CPU Time (CPU_T): the amount of processor time used

Performance Index (PI): PI=SP/CPU_T, normalized [0…1]

Average Response Time (AR_i): time to complete task *i,* including queue time, execution time, and communication costs

Slowdown (SL): measures the factor by which the workload execution times are slower than the ideal workload execution time

6.5.1.1 Cache Size Effects on Data Diffusion

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. At the transition point when the arrival rate increased beyond 59, the wait queue length also started growing to an eventual maximum of 198K tasks. The workload execution time was 5011 seconds, yielding 28% efficiency (ideal being 1415 seconds).

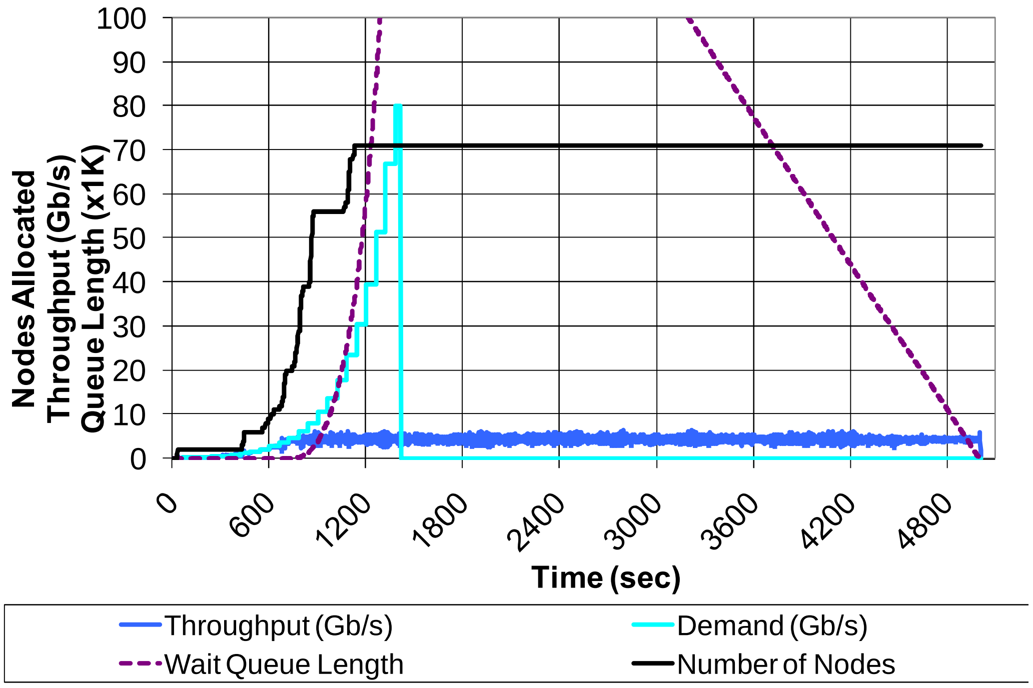


Figure 52: MI workload, 250K tasks, 10MB:10ms ratio, up to 64 nodes using DRP, FA policy

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 dataset was diffused from GPFS to local disk caches with every cache miss (the red area in the graphs); global cache hits are in yellow and local cache hits in green. 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.

Figure 53 shows the 1GB cache size experiment. 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 stabilizes at an average of 5.2Gb/s. Within 800 seconds, working set caching reaches a steady state with a throughput of 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, down from 5011 seconds for the FA policy, with 38% efficiency.

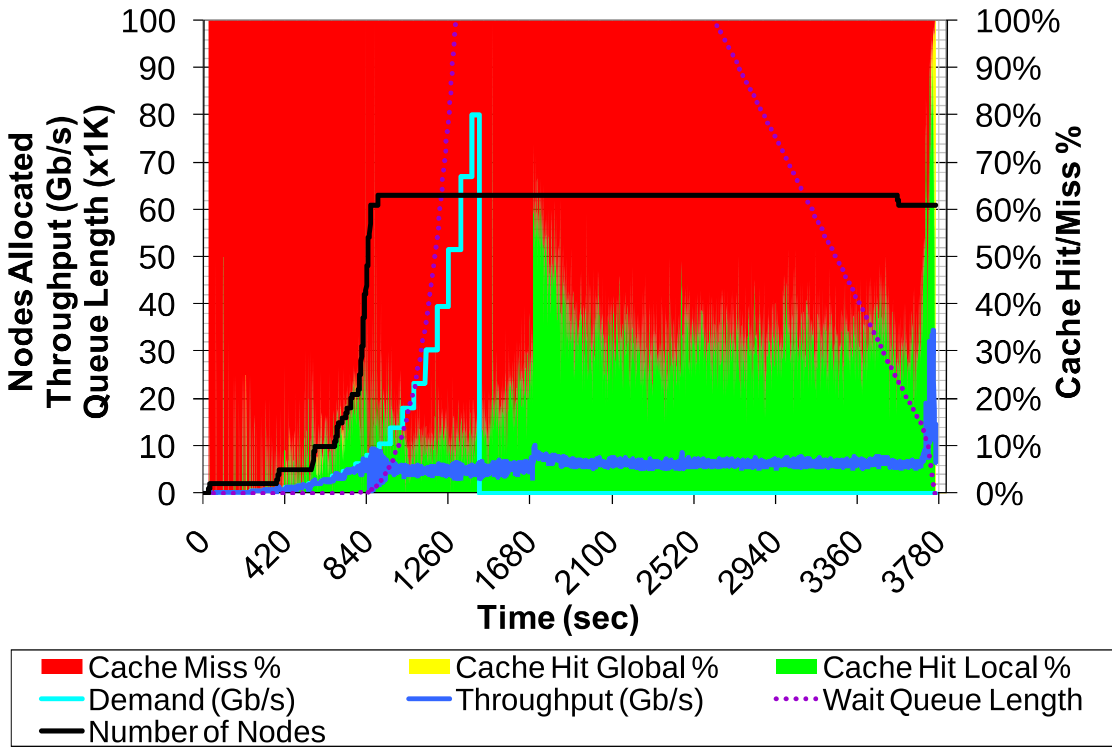


Figure 53: MI workload, 250K tasks, 10MB:10ms ratio, up to 64 nodes using DRP, GCC policy, 1GB caches/node

Figure 54 increases the per node cache size from 1Gb to 1.5GB, which increases the aggregate cache size to 96GB, almost enough to hold the entire working set of 100GB. Notice that the throughput hangs on further to the ideal throughput, up to 132 tasks/sec when the throughput increase stops and stabilizes at an average of 6.3Gb/s. Within 350 seconds of this stabilization, the cache hit performance increased significantly from 25% cache hit rates to over 90% cache hit rates; this increase in cache hit rates also results in the throughput increase up to an average of 45.6Gb/s for the remainder of the experiment. Overall, it achieved 78% cache hit rates, 1% cache hit rates to remote caches, and 21% cache miss rates. The workload execution time was reduced drastically from the 1GB per node cache size, down to 1596 seconds; this yields an 89% efficiency when compared to the ideal case. Both the 1GB and 1.5GB cache sizes achieve reasonable cache hit rates, despite the fact that the cache sizes are smaller than the working set; this is due to the fact that the data-aware scheduler looks deep (i.e. window size set to 2500) in the wait queue to find tasks that will improve the cache hit performance.

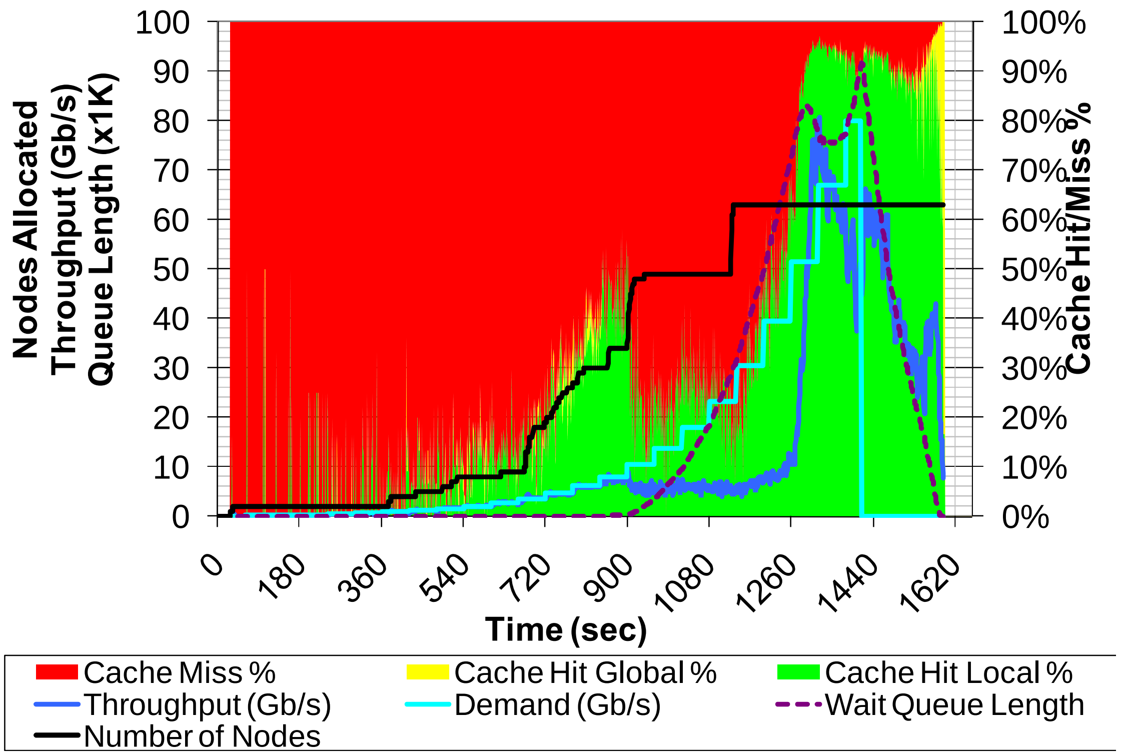


Figure 54: MI workload, 250K tasks, 10MB:10ms ratio, up to 64 nodes using DRP, GCC policy, 1.5GB caches/node

Figure 55 shows results with 2GB local caches (128GB aggregate). Aggregate throughput is close to 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%. Note that the queue length never grew beyond 7K tasks, significantly less than for the other experiments (91K to 198K tasks long). With an execution time of 1436 seconds, efficiency was 98.5%.

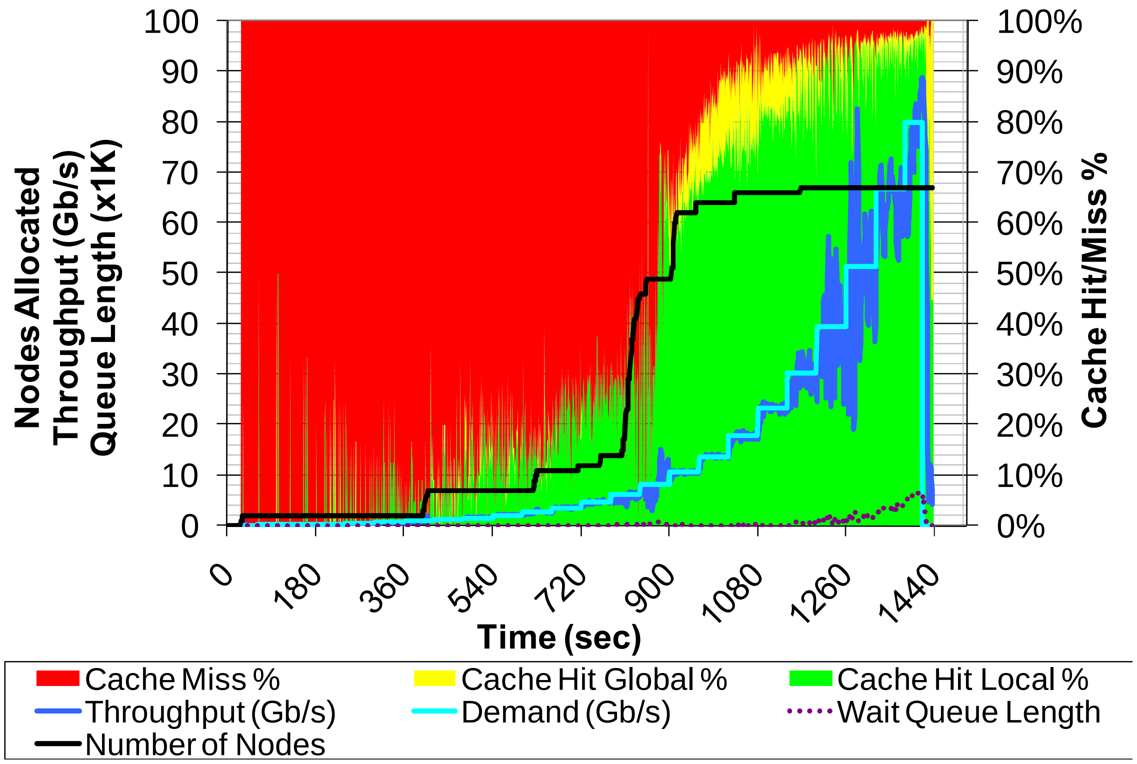


Figure 55: MI workload, 250K tasks, 10MB:10ms ratio, up to 64 nodes using DRP, GCC policy, 2GB caches/node

Investigating if it helps to increase the cache size further to 4GB per node, we conduct the experiment whose results are found in Figure 56. We see no significant improvement in performance from the experiment with 2GB caches. The execution time is reduced slightly to 1427 seconds (99.2% efficient), and the overall cache hit rates are improved to 88% cache hit rates, 6% remote cache hits, and 6% cache misses.

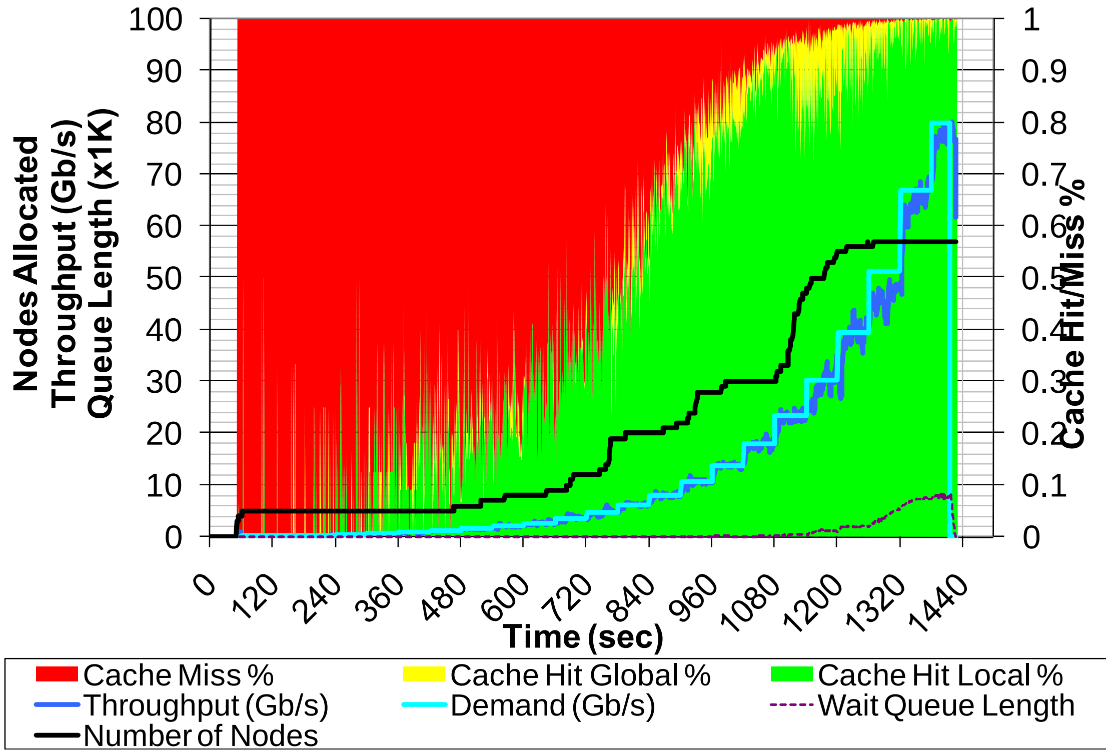


Figure 56: MI workload, 250K tasks, 10MB:10ms ratio, up to 64 nodes using DRP, GCC policy, 4GB caches/node

6.5.1.2 Comparing Scheduling Policies

To study the impact of scheduling policy on performance, we reran the workload for which we just gave GCC results using MCH and MCU with 4GB caches. Using the MCH policy (see Figure 57), we obtained 95% cache hit performance % (up from 88% with GCC), but poor processor utilization (43%, down from 95% with GCC). Note that the MCH policy always schedules tasks according to where the data is cached, even if it has to wait for some node to become available, leaving some nodes processors idle. Notice a new metric measured (dotted thin black line), the CPU utilization, which shows clear poor CPU utilization that decreases with time as the scheduler has difficulty scheduling tasks to busy nodes; the average CPU utilization for the entire experiment was 43%.

Overall workload execution time increased, to 2888 seconds (49% efficiency, down from 99% for GCC).

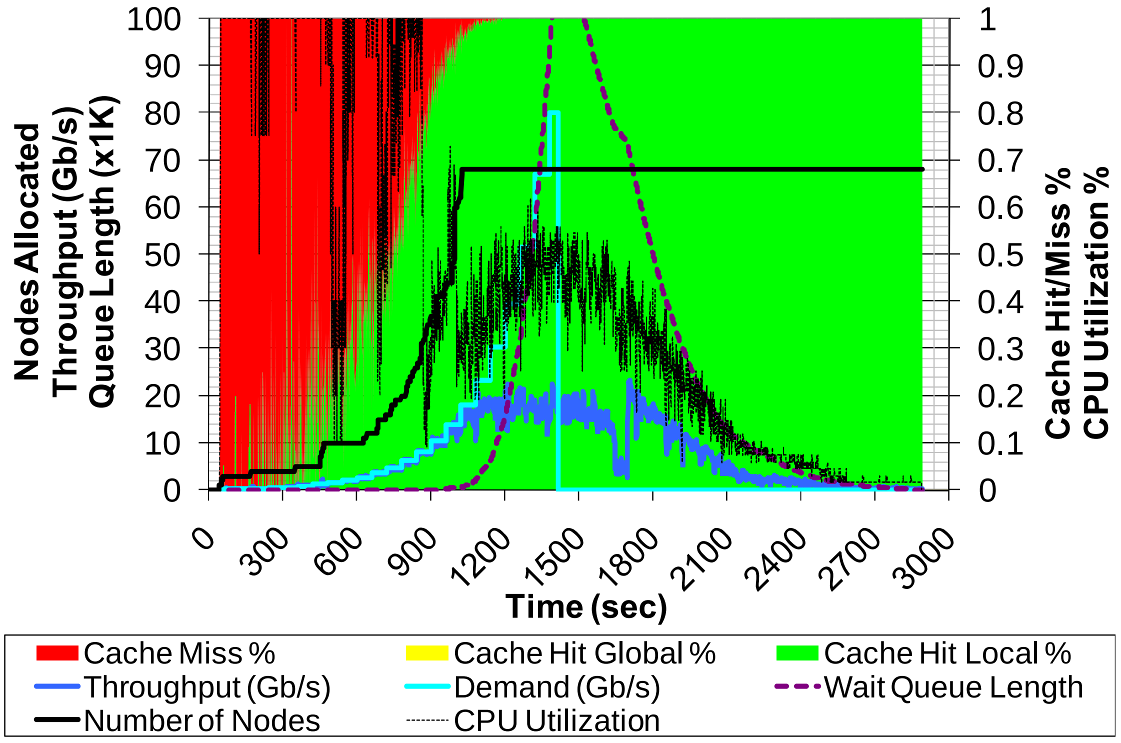


Figure 57: MI workload, 250K tasks, 10MB:10ms ratio, up to 64 nodes using DRP, MCH policy, 4GB caches/node

Figure 58 shows the MCU policy, which attempts to maximize the CPU utilization at the expense of data movement. We see the workload execution time is improved (compared to MCH) down to 2037 seconds (69% efficient), but it is still far from the GCC policy that achieved 1436 seconds. The major difference here is that the there are significantly more cache hits to remote caches as tasks got scheduled to nodes that didn't have the needed cached data due to being busy with other work. We were able to sustain high efficiency with arrival rates up to 380 tasks/sec, with an average throughput for the steady part of the experiment of 14.5 Gb/s. It is interesting to see the cache hit local performance at time 1800~2000 second range spiked from 60% to 98%, which results in a spike in throughout from 14Gb/s to 40Gb/s. Although we maintained 100% CPU utilization, due to the extra costs of moving data from remote executors, the performance was worse than the GCC policy when 4.5% of the CPUs were left idle.

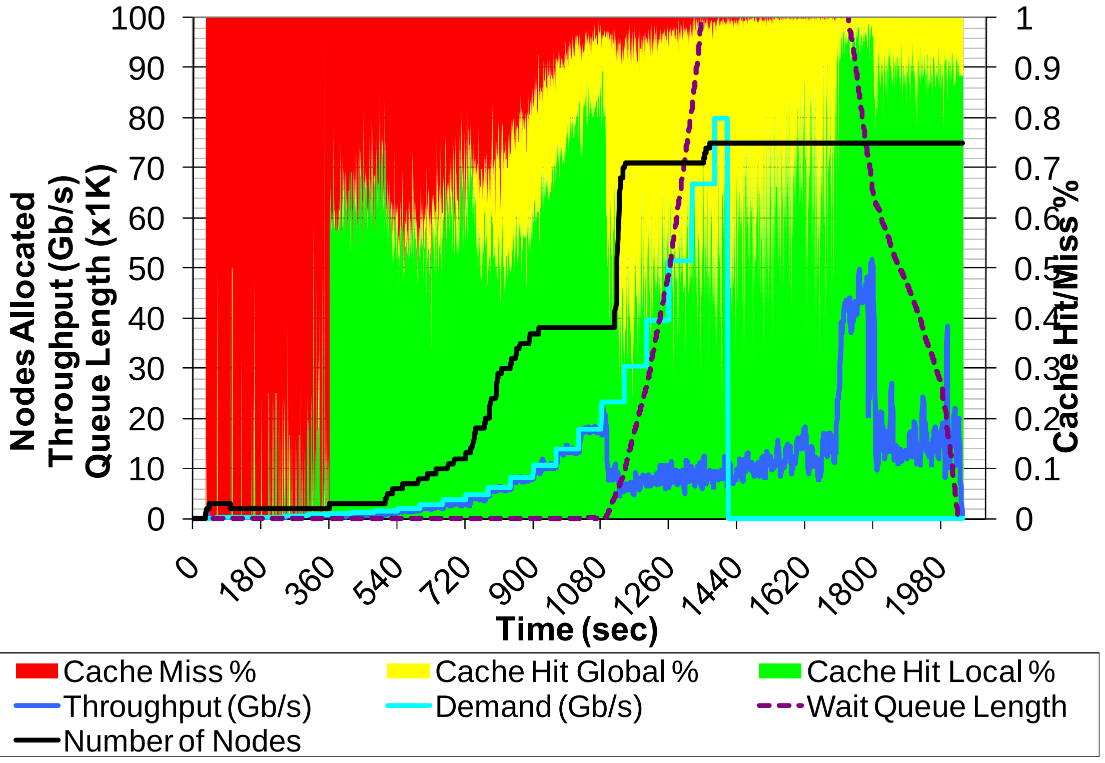


Figure 58: MI workload, 250K tasks, 10MB:10ms ratio, up to 64 nodes using DRP, MCU policy, 4GB caches/node

The next several sub-sections will summarize these experiments, and compare them side by side.

6.5.1.3 Cache Performance

Figure 59 shows cache performance over six experiments involving data diffusion, the ideal case, and the FA policy which does not cache any data. We see a clear separation in the cache miss rates (red) for the cases where the working set fit in cache (1.5GB and greater), and the case where it did not (1GB). For the 1GB case, the cache miss rate was 70%, which is to be expected considering only 70% of the working set fit in cache at most, and cache thrashing was hampering the scheduler's ability to achieve better cache miss rates. The other extreme, the 4GB cache size cases, all achieved near perfect cache miss rates of 4%~5.5%.

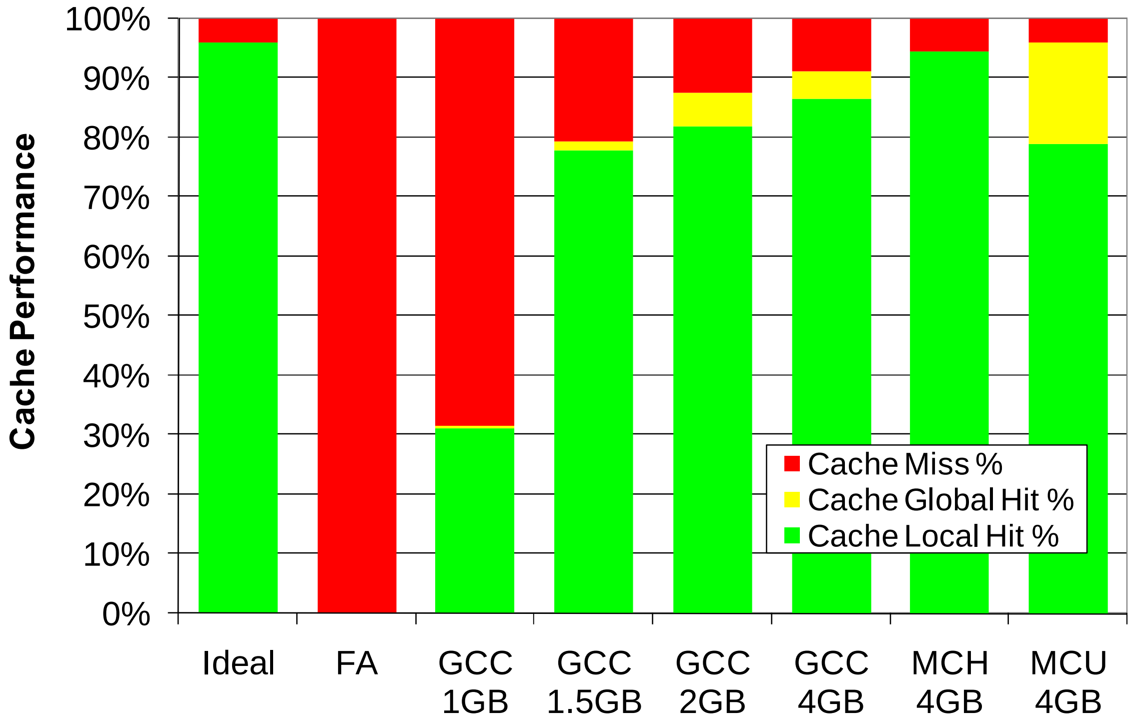


Figure 59: MI workload cache performance

6.5.1.4 Throughput

Figure 60 summarizes the aggregate I/O throughput measured in each of the seven experiments. 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.

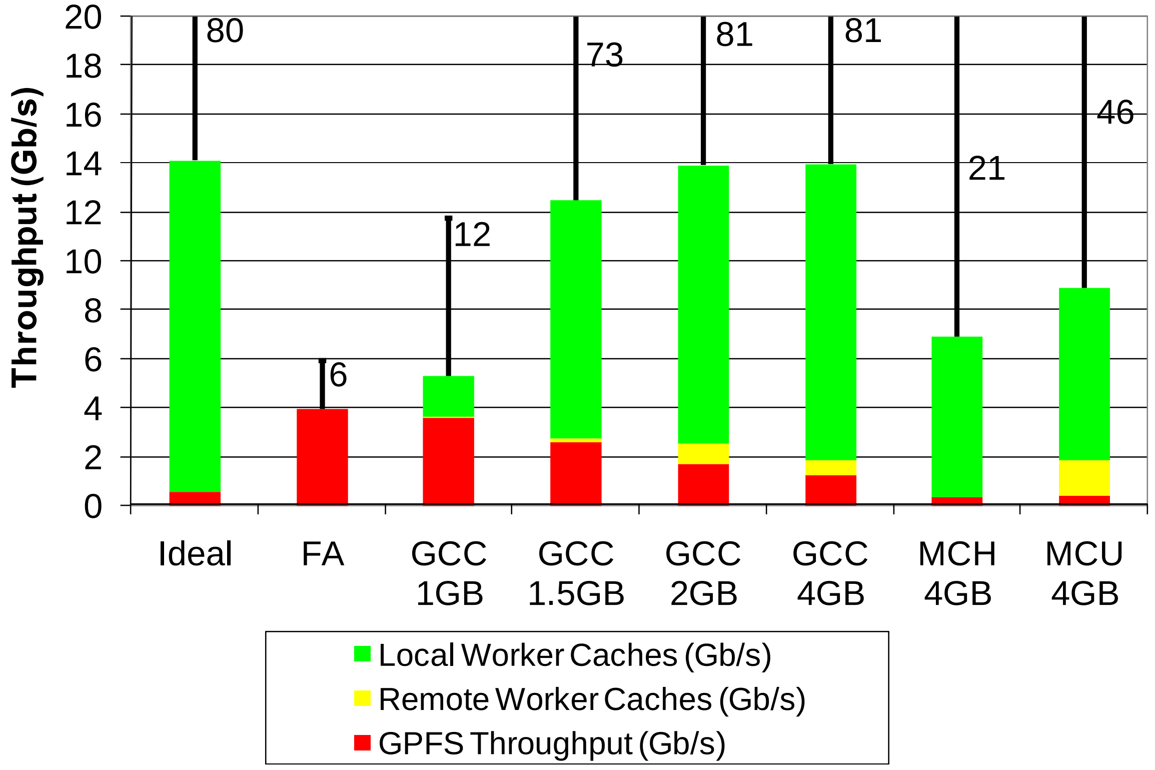


Figure 60: MI workload average and peak (99 percentile) throughput

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

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

6.5.1.5 Performance Index and Speedup

The performance index attempts to capture the speedup per CPU time achieved. Figure 61 shows PI and speedup data.

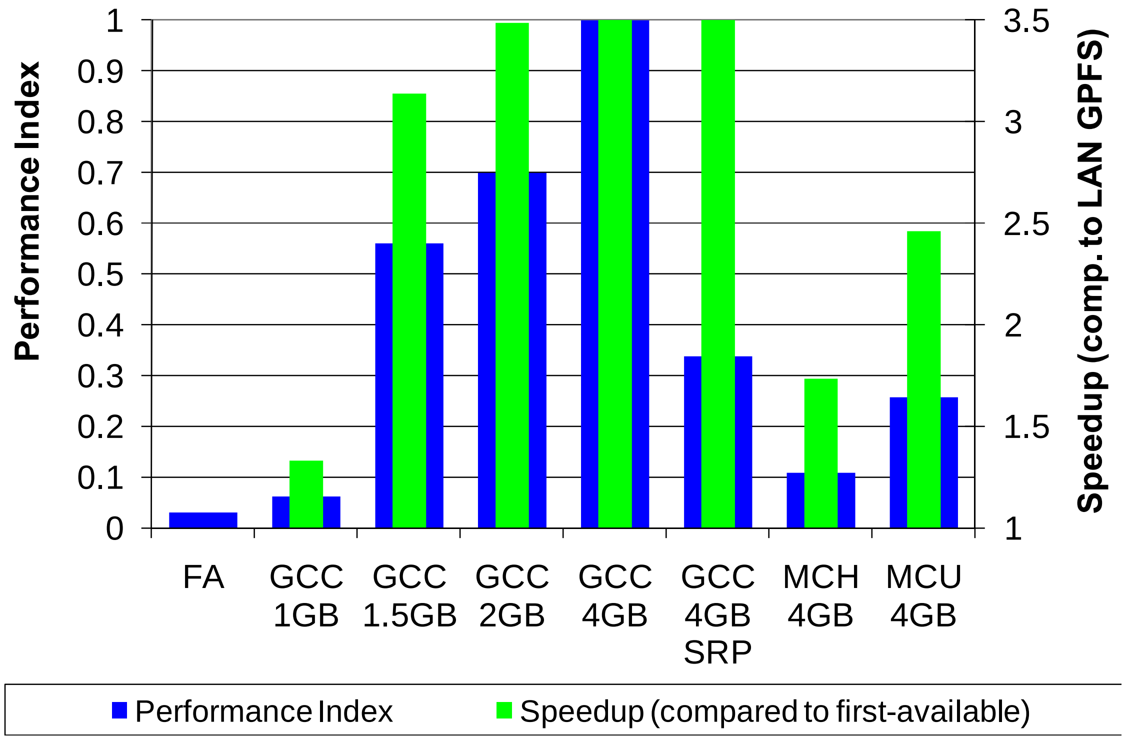


Figure 61: MI workload PI and speedup comparison

Notice that while GCC with 2GB and 4GB caches each achieve the highest speedup of 3.5X, the 4GB case achieves a higher performance index of 1 as opposed to 0.7 for the 2GB case. This is due to the fact that fewer resources were used throughout the 4GB experiment, 17 CPU hours instead of 24 CPU hours for the 2GB case. This reduction in resource usage was due to the larger caches, which in turn allowed the system to perform better with fewer resources for longer durations, and hence the wait queue did not grow as fast, which resulted in less aggressive resource allocation. Notice the performance index of the FA policy which uses GPFS solely; although the speedup gains with data diffusion compared to the FA policy are modest (1.3X to 3.5X), the performance index of data diffusion is significantly more (2X to 34X).

6.5.1.6 Slowdown

Speedup compares data diffusion to the base case, but does not tell us how well data diffusion performed in relation to the ideal case. Recall that the ideal case is computed from the arrival rate of tasks, assuming zero communication costs and infinite resources to handle tasks in parallel; in our case, the ideal workload execution time is 1415 seconds. Figure 62 shows the slowdown for our experiments as a function of arrival rates. *Slowdown (SL)* measures the factor by which the workload execution times are slower than the ideal workload execution time; the ideal workload execution time assumes infinite resources and 0 cost communication, and is computed from the arrival rate function.

These results in Figure 62 show the arrival rates that could be handled by each approach, showing the FA policy (the GPFS only case) to saturate the earliest at 59 tasks/sec denoted by the rising red line. It is evident that larger cache sizes allowed the saturation rates to be higher (essentially perfect for some cases, such as the GCC with 4GB caches). It interesting to point out the GCC policy with 1.5GB caches slowdown increase relatively early (similar to the 1GB case), but then towards the end of the experiment the slowdown is reduced from almost 5X back down to an almost ideal 1X. This sudden improvement in performance is attributed to a critical part of the working set being cached and the cache hit rates increasing significantly. Also, note the odd slowdown (as high as 2X) of the 4GB cache DRP case at arrival rates 11, 15, and 20; this slowdown matches up to the drop in throughput between time 360 and 480 seconds in Figure 58 (the detailed summary view of this experiment), which in turn occurred when an additional resource was allocated. It is important to note that resource allocation takes on the order of 30~60 seconds due to LRM's overheads, which is why it took the slowdown 120 seconds to return back to the normal (1X), as the dynamic resource provisioning compensated for the drop in performance.

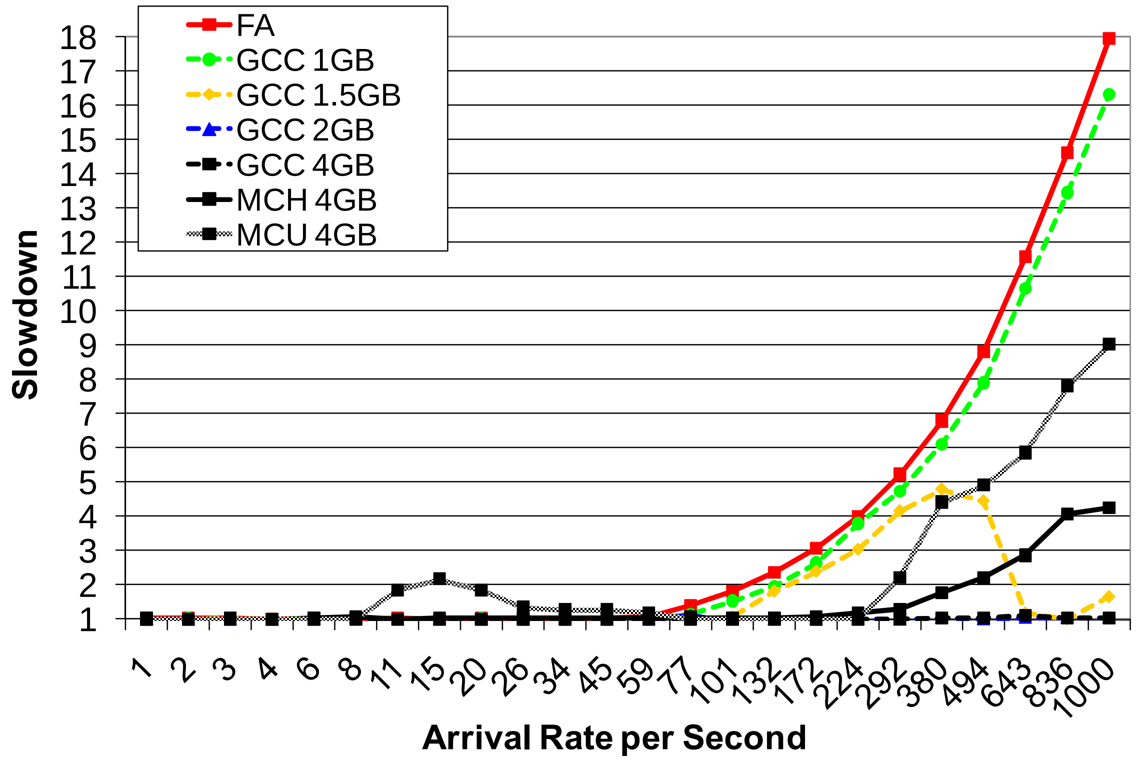


Figure 62: MI workload slowdown as we varied arrival rate

6.5.1.7 Response Time

The response time is probably one of the most important metrics interactive applications. *Average Response Time (AR_i)* is the end-to-end time from task submission to task completion notification for task *i*; $AR_i = WQ_i+TK_i+D_i$, where WQ_i is the wait queue time, TK_i is the task execution time, and D_i is the delivery time to deliver the result. Figure 63 shows response time results across all 14 experiments in log scale.

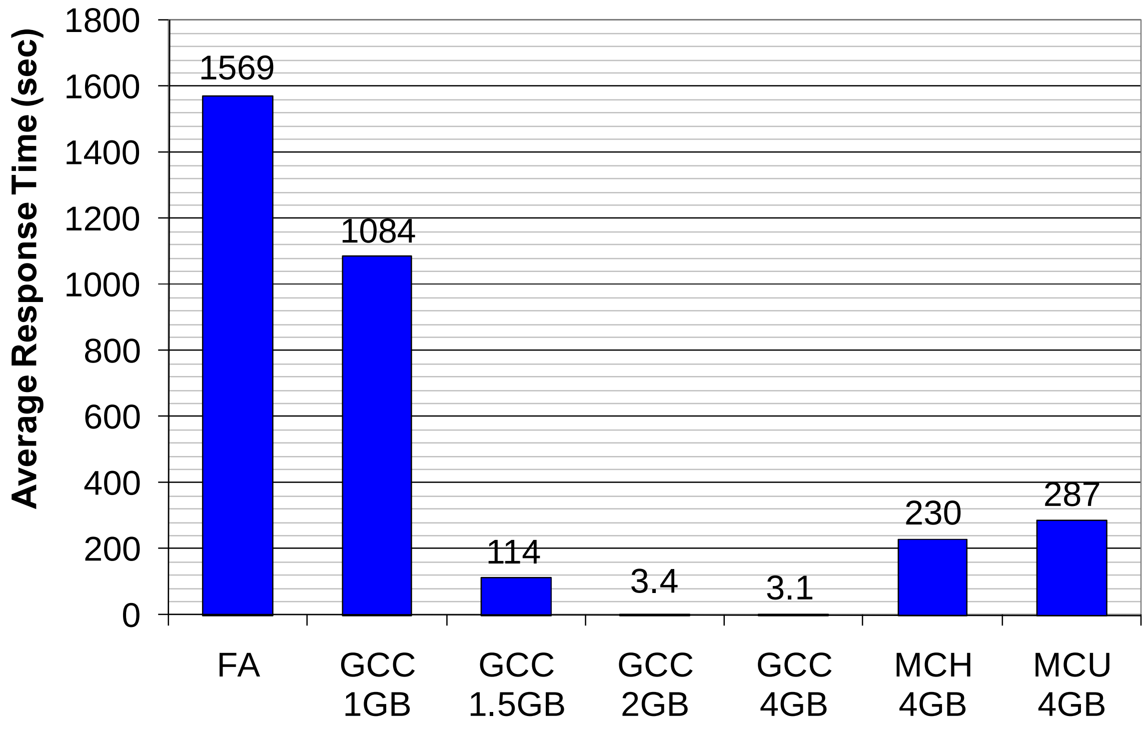


Figure 63: MI workload average response time

We see a significant different between the best data diffusion response time (3.1 seconds per task) to the worst data diffusion (1084 seconds) and the worst GPFS (1870 seconds). That is over 500X difference between the data diffusion GCC policy and the FA policy (GPFS only) 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.

6.5.2 Sine-Wave Workload

The previous sub-section explored a workload with monotonically increasing arrival rates. To explore how well data diffusion deals with decreasing arrival rates as well, we define a sine-wave (SW) workload that follows the function (where time is elapsed minutes from the beginning of the experiment):

$$A = \left\lfloor \left(\sin\left(sqrt\left(time + 0.11 \right) * 2.859678 \right) + 1 \right) * \left(time + 0.11 \right) * 5.705 \right\rfloor$$

Equation 14: Sine-Wave Workload arrival rate function

This workload aims to explore the data-aware scheduler’s ability to optimize data locality in face frequent joins and leaves of resources due to variability in demand. This function is essentially a sine wave pattern (see Figure 64, red line), in which the arrival rate increases in increasingly stronger waves, increasing up to 1000 tasks/sec arrival rates. The working set is 1TB large (100K files of 10MB each), and the I/O to compute ratio is 10MB:10ms. The workload is composed of 2M tasks (black line) where each task accesses a random file (uniform distribution), and takes 6505 seconds to complete in the ideal case.

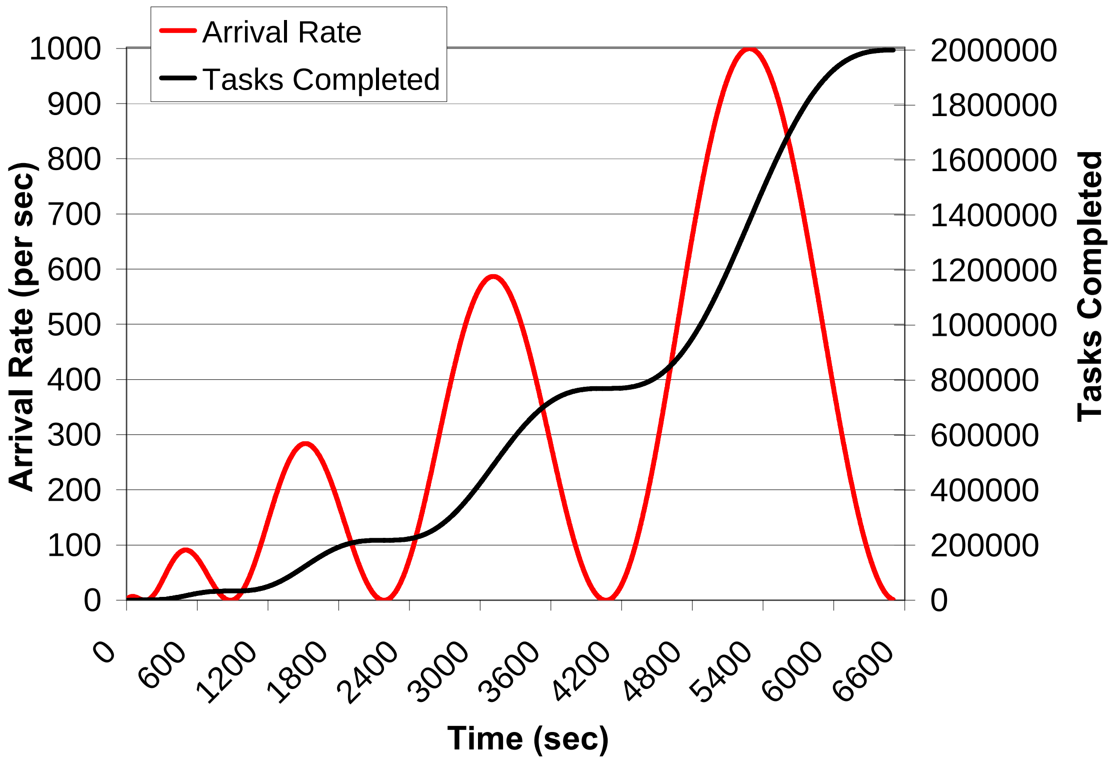


Figure 64: SW workload overview

Our first experiment consisted of running the SW workload with all computations running directly from the parallel file system and using 100 nodes with static resource provisioning. We see the measured throughput keep up with the demand up to the point when the demand exceeds the parallel file system peak performance of 8Gb/s; beyond this point, the wait queue grew to 1.4M tasks, and the workload needed 20491 seconds to complete (instead of the ideal case of 6505 seconds), yielding an efficiency of 32%. Note that although we are using the same cluster as in the MI workload (Section 6.5.1), GPFS's peak throughput is higher (8Gb/s vs. 4Gb/s) due to a major upgrade to both hardware and software in the cluster between running these experiments.

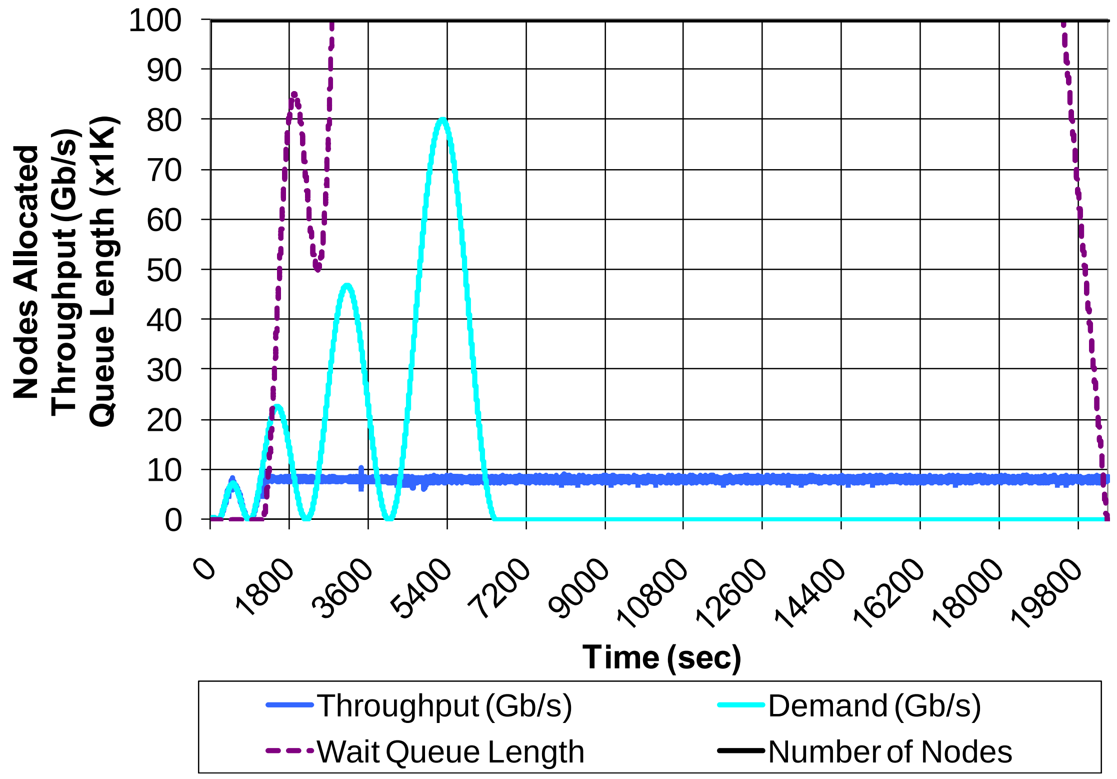


Figure 65: SW workload, 2M tasks, 10MB:10ms ratio, 100 nodes, FA policy

Enabling data diffusion with the GCC policy, setting the cache size to 50GB, the scheduling window size to 2500, and the processor utilization threshold to 90%, we get a run that took 6505 seconds to complete (see *Figure 66*), yielding an efficiency of 100%. We see the cache misses (red) decrease from 100% to 0% over the course of the experiment, while local cache hits (green) frequently making up 90%+ of the cache hits. Note that the data diffusion mechanism was able to keep up with the arrival rates throughout with the exception of the peak of the last wave, when it was only able to achieve 72Gb/s (instead of the ideal 80Gb/s), at which point the wait queue grew to its longest length of 50K tasks. The global cache hits (yellow) is stable at about 10% throughout, which is reflected from the fact that the GCC policy is oscillating between optimizing cache hit performance and processor utilization around the configured 90% processor utilization threshold.

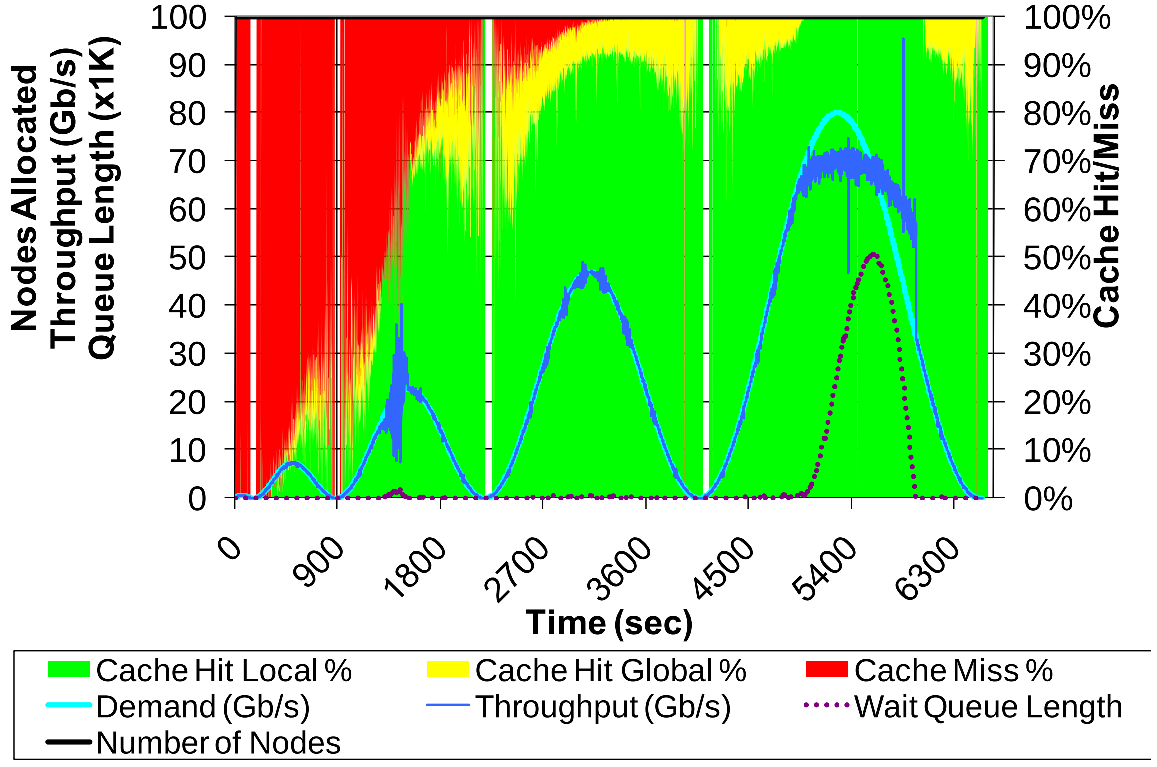


Figure 66: SW workload, 2M tasks, 10MB:10ms ratio, 100 nodes, GCC policy, 50GB caches/node

Enabling dynamic resource provisioning, Figure 67 shows the workload still manages to complete in 6697 seconds, yielding 97% efficiency. In order to minimize wasted processor time, we set each worker to release its resource after 30 seconds of idleness. Note that upon releasing a resource, its cache is reset; thus, after every wave, cache performance is again poor until caches are rebuilt. The measured throughput does not fit the demand line as well as the static resource provisioning did, but it increases steadily in each wave, and achieves the same peak throughput of 72Gb/s after enough of the working set is cached.

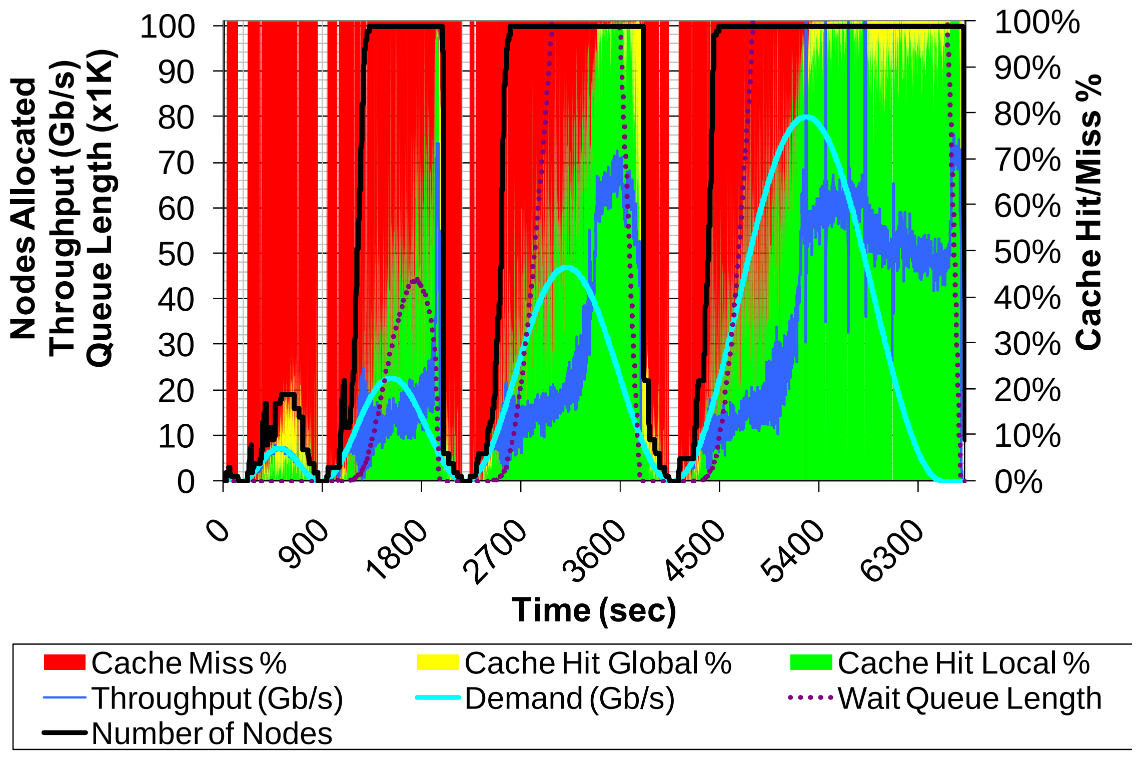


Figure 67: SW workload, 2M tasks, 10MB:10ms ratio, up to 100 nodes with DRP, GCC policy, 50GB caches/node

In summary, we see data diffusion make a significant impact. Using the dynamic provisioning where the number of processors is varied based on load does not hinder data diffusion's performance significantly (achieves 97% efficiency) and yields less processor time consumed (253 CPU hours as opposed to 361 CPU hours for SRP with data diffusion and 1138 CPU hours without data diffusion).

6.5.3 All-Pairs Workload Evaluation

In previous work, several of the co-authors addressed large scale data intensive problems with the Chirp [138] distributed filesystem. 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 [137], it typically distributes an entire application working data set to each compute node local disk prior to the application running. We call the All-Pairs use of Chirp *active storage*. 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. This section aims to compare the performance between data diffusion and a best model of active storage.

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 (see Figure 68). [137] The AP problem is easy to express in terms of two nested for loops over some parameter space (see Figure 69). This regular structure also makes it easy to optimize its data access operations. Thus, AP is a challenging benchmark for data diffusion, due to its on-demand, pull-mode data access strategy. Figure 70 shows a sample 100x100 problem space, where each black dot represents a computation computing some function F on data at index i and j; in this case, the entire compute space is composed of 10K separate computations.

```
    All-Pairs( set A, set B, function F ) returns matrix M:
    Compare all elements of set A to all elements of set B via function F, yielding matrix M,
such that M[i,j] = F(A[i],B[j])
```

Figure 68: All-Pairs definition

```
foreach $i in A
        foreach $j in B
                submit_job F $i $j
        end
end
```

Figure 69: All-Pairs Workload Script

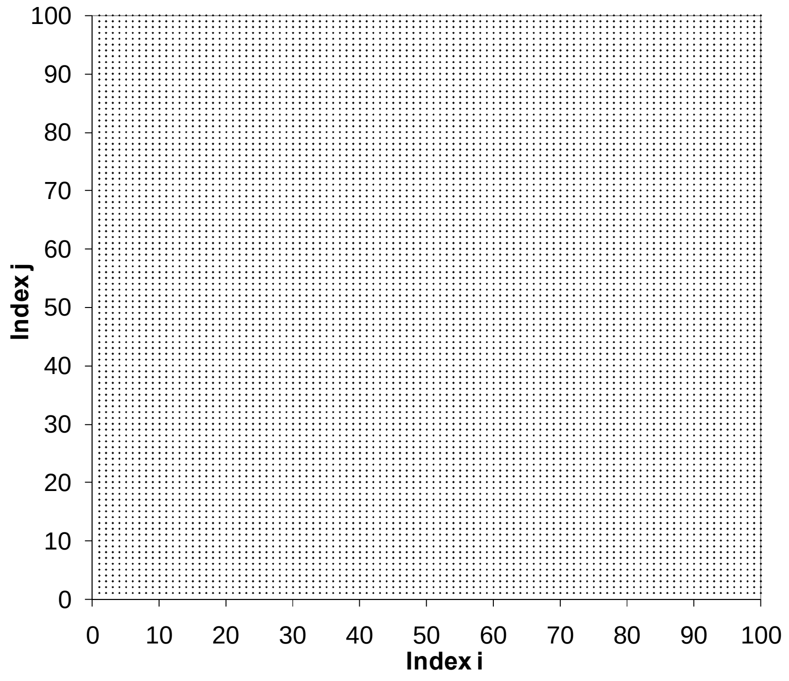


Figure 70: Sample 100x100 All-Pairs problem, where each dot represents a computation at index i,j

In previous work [137], we 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 [137], and compare measured 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 (see Figure 71), data diffusion achieves a throughput that is 80% of the best case of all data accesses occurring to local disk (see Figure 75).

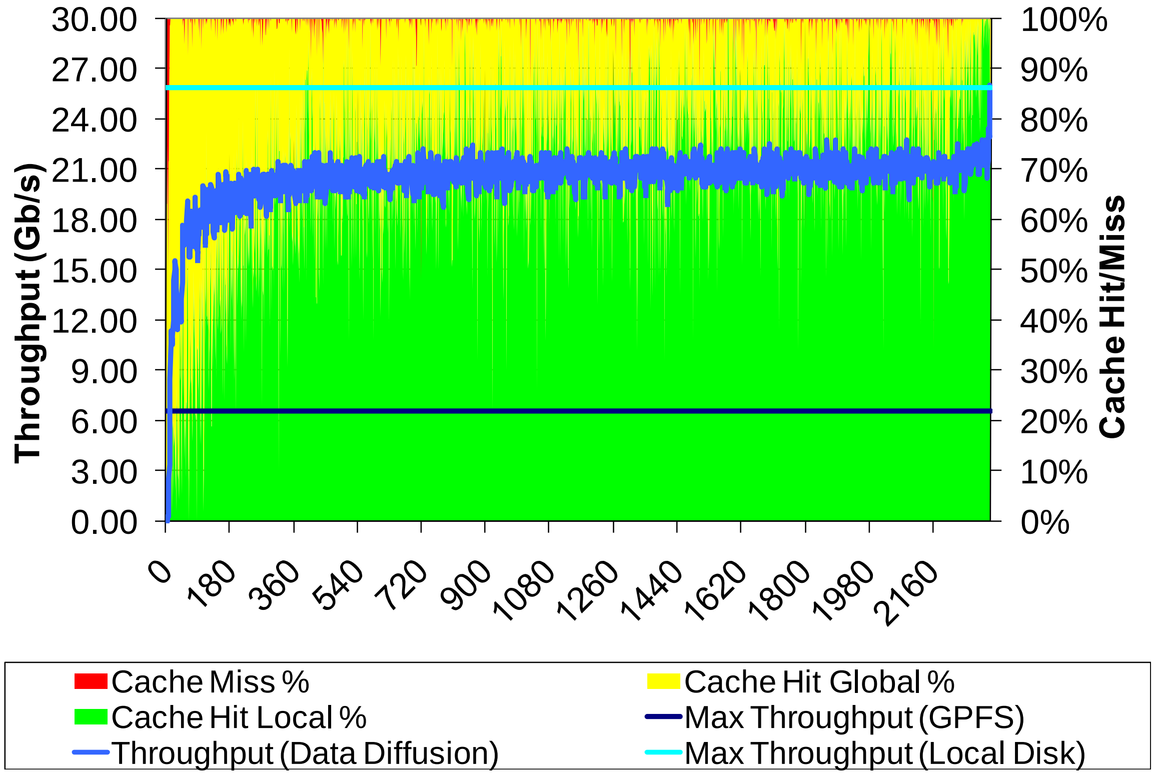


Figure 71: AP workload, 500x500=250K tasks, 24MB:1000ms, 100 nodes, GCC policy, 50GB caches/node

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 [137] 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 throughput of the purely local solution.

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 (see *Figure 72*).

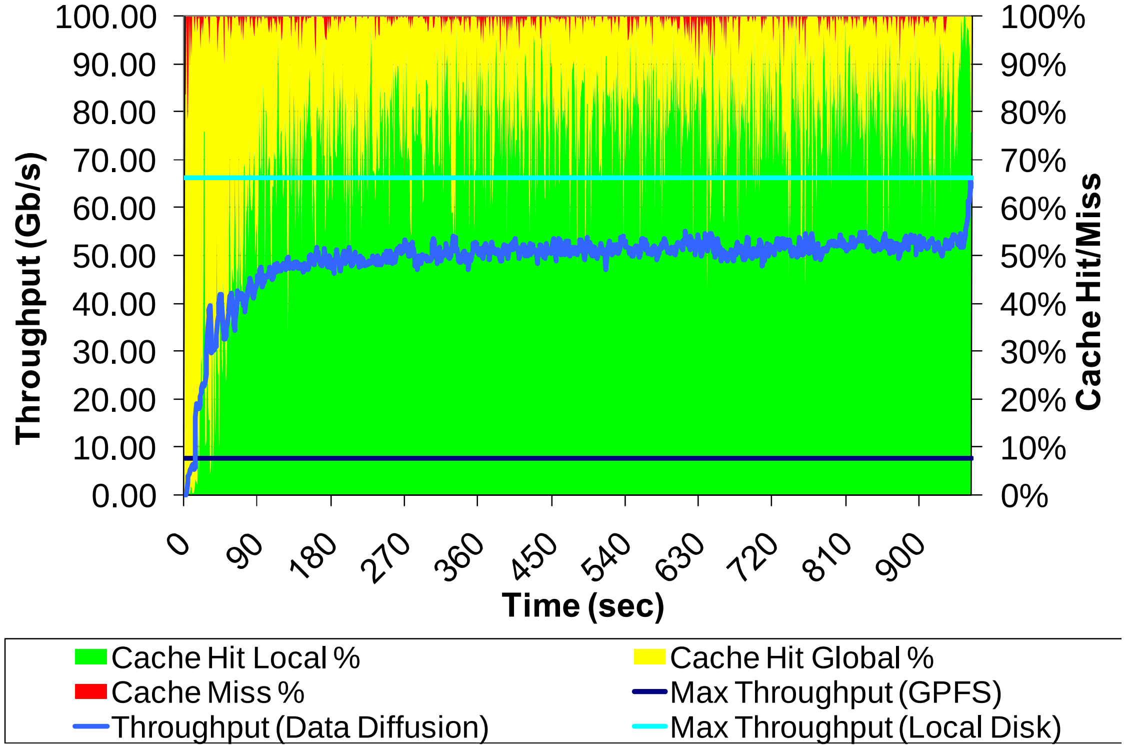


Figure 72: AP workload, 500x500=250K tasks, 24MB:100ms, 100 nodes, GCC policy, 50GB caches/node

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). With such similar bandwidth usage throughout the system, similar efficiencies were to be expected.

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) two systems, an IBM Blue Gene/P and a SiCortex. 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 and SiCortex 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 (see Table 9).

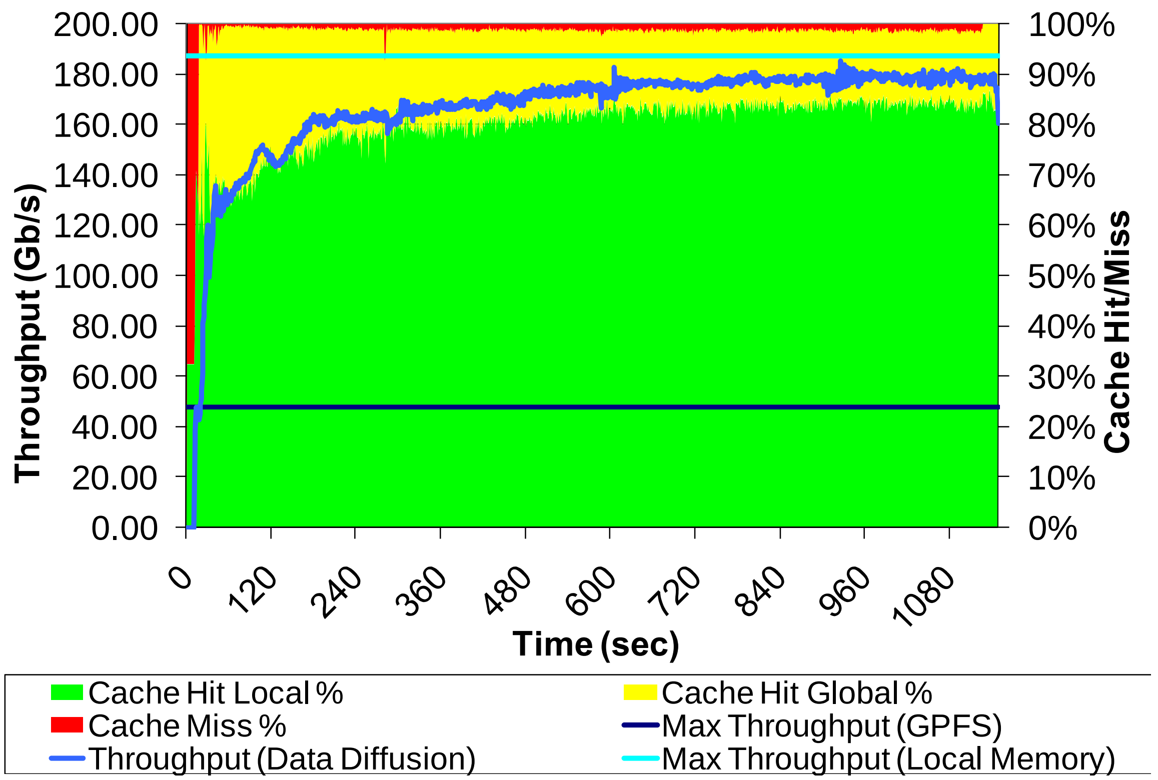


Figure 73: AP workload on emulated Blue Gene/P, 1000x1000=1M tasks, 24MB:4000ms, 1024 nodes (4096 processors), GCC policy, 2GB caches/node

The emulated SiCortex was configured with 5832 processors, 3.6GB caches, and a relatively slow parallel file system rated at 4Gb/s. The throughput on the SiCortex reached 90Gb/s, far from the upper bound of 268Gb/s. It is interesting that the overall efficiency for data diffusion on the SiCortex was 27%, the same efficiency that the best case active storage achieved. The slower parallel file system significantly reduced the efficiency of the data diffusion (as data diffusion performs the initial cache population completely from the parallel file system, and needed 906GB of parallel file system bandwidth), however it had no effect on the efficiency of the active storage as the spanning tree only required one read of the dataset from the parallel file system (a total of 24GB). With sufficiently large workloads, data diffusion would likely improve its efficiency as the expensive cost to populate its caches would get amortized over more potential cache hits.

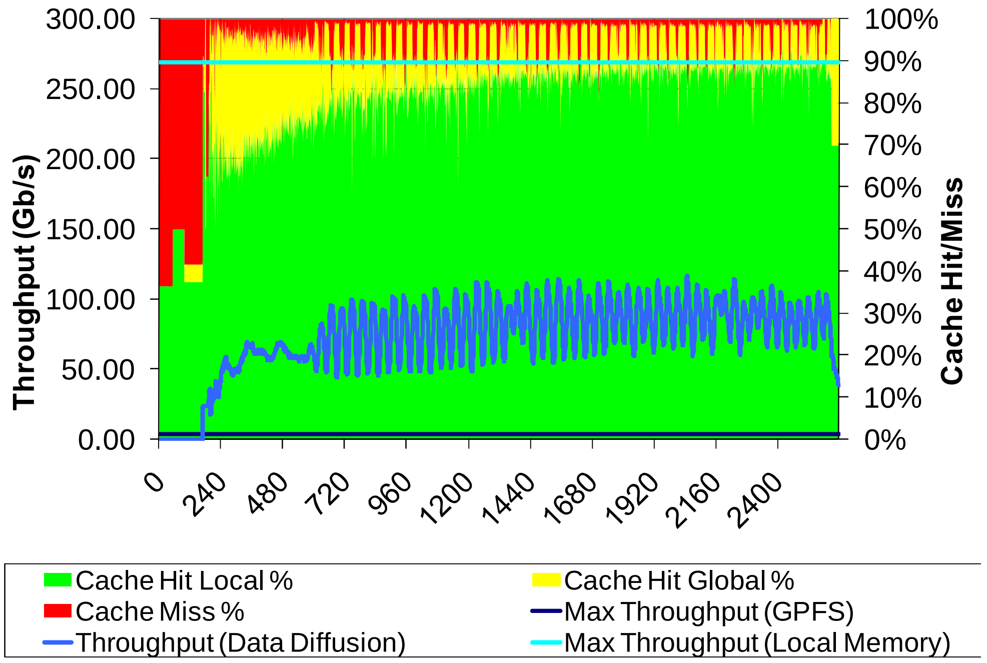


Figure 74: AP workload on emulated SiCortex, 1000x1000=1M tasks, 24MB:4000ms, 972 nodes (5832 processors), GCC policy, 3.6GB caches/node

There are some interesting oscillations in the cache hit/miss ratios as well as the achieved. We believe the oscillation occurred due to the slower parallel file system of the SiCortex, which was overwhelmed by thousands of processors concurrently accessing tens of MB each. Further compounding the problem is the fact that there were twice as many cache misses on the SiCortex than there were on the Blue Gene/P, which seems counter-intuitive as the per processor cache size was slightly larger (500MB for Blue Gene/P and 600MB for the SiCortex). We will investigate these oscillations further to find their root cause, which might very well be due to our emulation, and might not appear in real world examples.

In reality, the best case active storage would require cache sizes of at least 24GB, and the existing 2GB or 3.6GB cache sizes for the Blue Gene/P and SiCortex respectively would only be sufficient for an 83X83 problem size, so this comparison (Figure 73 and Figure 74) is not only emulated, but also hypothetical. Nevertheless, it is interesting to see the significant difference in efficiency between data diffusion and active storage at this larger scale.

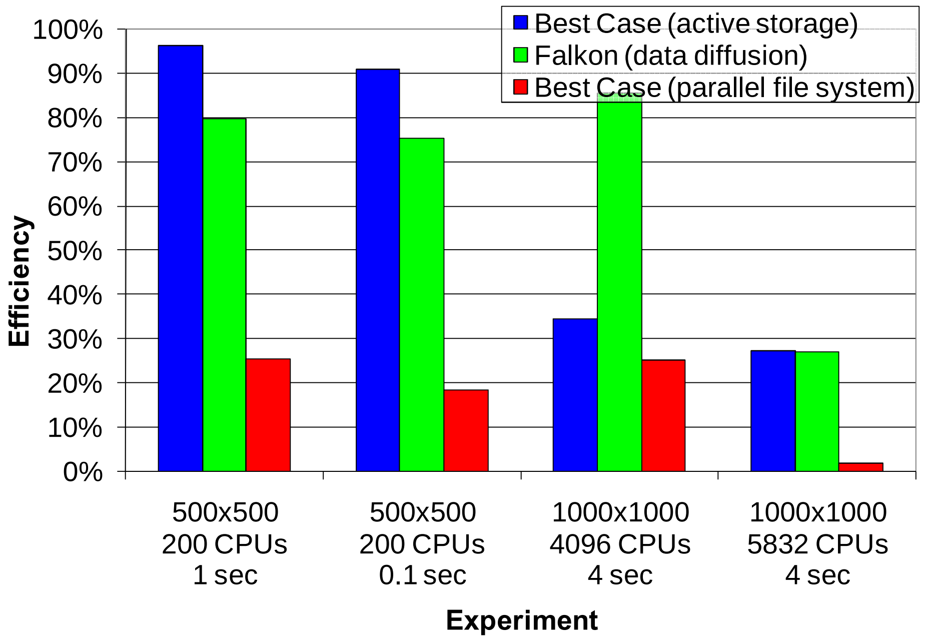


Figure 75: AP workload efficiency for 500x500 problem size on 200 processor cluster and 1000x1000 problem size on the Blue Gene/P and SiCortex emulated systems with 4096 and 5832 processors respectively

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 (as could be seen in comparing the Blue Gene/P and SiCortex results), 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.

Table 9: Data movement for the best case active storage and Falkon data diffusion

Experiment	Approach	Local Disk/Memory (GB)	Network (node-to-node) (GB)	Shared File System (GB)
500x500 200 CPUs 1 sec	Best Case (active storage)	6000	1536	12
	Falkon (data diffusion)	6000	1698	34
500x500 200 CPUs 0.1 sec	Best Case (active storage)	6000	1536	12
	Falkon (data diffusion)	6000	1528	62
1000x1000 4096 CPUs 4 sec	Best Case (active storage)	24000	12288	24
	Falkon (data diffusion)	24000	4676	384
1000x1000 5832 CPUs 4 sec	Best Case (active storage)	24000	12288	24
	Falkon (data diffusion)	24000	3867	906

6.6 Collective I/O for MTC

Collective I/O is orthogonal to the data diffusion presented in this chapter, but nevertheless has similar goals to enable data-intensive MTC. This section evaluates a prototype collective IO model for file-based MTC. The model enables efficient and easy distribution of input data files to computing nodes and gathering of output results from them. It eliminates the need for such manual tuning and makes the programming of large-scale clusters using a loosely coupled model easier. Our approach, inspired by in-memory approaches to collective operations for parallel programming, builds on fast local file systems to provide high-speed local file caches for parallel scripts, uses a broadcast approach to handle distribution of common input data, and uses efficient scatter/gather and caching techniques for input and output. We describe the design of the prototype model, its implementation on the Blue Gene/P supercomputer, and present preliminary measurements of its performance on synthetic benchmarks and on a large-scale molecular dynamics application. Note that while our data diffusion experiments were emulated on the Blue Gene/P, our collective I/O experiments are running on the real system with real applications and data.

The specific problem we address here is that as the number of nodes in large-scale clusters contending for shared resources grows large, the IO bandwidth, volume and/or file management transaction rate exceeds some aggregate capacity limit, bottlenecks arise and the system becomes unbalanced. Thus, CPU cycles are wasted because the IO subsystem cannot service the CPUs fast enough. (We are concerned here with applications with high enough IO-to-compute ratios for IO to become the primary obstacle to parallel speedup. Applications that do relatively little IO while computing for long periods typically perform well in loosely coupled settings without any change to their IO strategy.)

While petascale systems have massive shared IO subsystems, these subsystems often have vulnerabilities in handling file management transactions (e.g., creating and writing huge numbers of files at high rates) that are ill-matched with the needs of loosely coupled programs. Our work remedies this deficiency and makes petascale systems *attractive* for this important and productive paradigm for knitting existing scientific programs into powerful workflows.

Our strategy of collective IO is inspired by the collective data operations employed by tightly coupled message passing programming models. In these models, data is exchanged, both between in-memory tasks and between tasks and files, using operations such as *scatter* (often assisted by *broadcast*) and *gather*. In our model:

- Input files are broadcast from shared file systems to local file systems.
- Output files are locally batched up from applications and efficiently transferred to shared persistent storage.
- Intermediate file systems are provided within the cluster to aid in efficient input and output staging and to overcome the limitations that large-scale clusters impose on local file system capacity.

We present measurements from the Argonne ALCF Blue Gene/P supercomputer, running under ZeptoOS and Falkon. We have evaluated various features on up to 98,304 (out of 163,840) processors. Dedicated test time on the entire facility is rare, so all tests below were done with the background noise of activity from other jobs running on other processors. Nonetheless, the trends indicated are fairly clear, and we expect that they will be verifiable in future tests in a controlled, dedicated environment. We have made measurements in both areas of the proposed collective IO primitives (denoted as CIO throughout this section), such as *input* data distribution, and *output* data collection.

6.6.1 Input Data Distribution

Our first set of results investigated how effectively compute nodes can read data from the IFSs (over the torus network), examining various data volumes and various IFS/LFS ratios. We used the lightweight Chirp file system [138] and the Fuse interface to read files from IFS to LFS. Figure 76 shows higher aggregate performance with larger files, and with higher ratios, with the best IFS performance reaching 162 MB/s for 100 MB files and a 256:1 ratio. However, as the bandwidth is split between 256 clients, the per-node throughput is only 0.6 MB/s. Computing the per-node throughput for the 64:1 ratio yields 2.3 MB/s, a significant increase. Thus, we conclude that a 64:1 ratio is good when trying to maximize the bandwidth per node. Larger ratios reduce the number of IFSs that need to be managed; however, there are practical limits that prohibit these ratios from being extremely large. In the case of a 512:1 ratio and 100 MB files, our benchmarks failed due to memory exhaustion when 512 compute nodes simultaneously connected to 1 compute node to transfer the 100 MB file. This needs further analysis.

Our next set of experiments used the lightweight MosaStore file system [145] to explore how effectively we can stripe LFSs to form a larger IFS. Our preliminary results in Figure 77 show that as we increase the degree of striping we get significant increases in aggregate throughput, up from 158 MB/s to 831 MB/s.

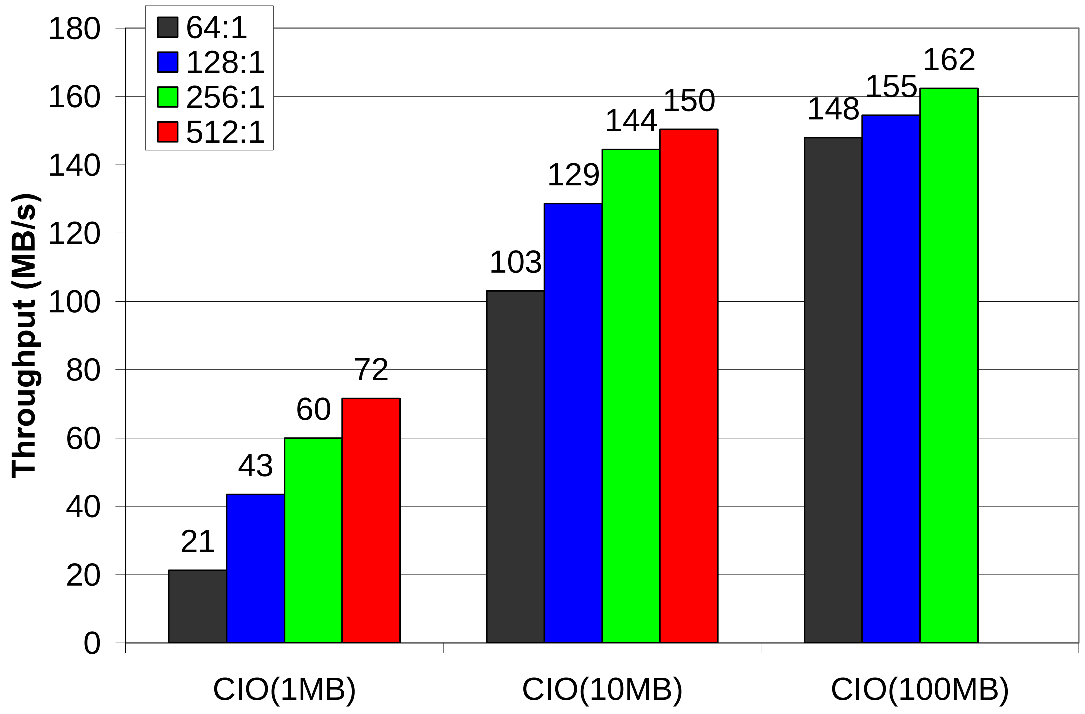


Figure 76: Read performance while varying the ratio of LFS to IFS from 64:1 to 512:1 using the Torus network.

The best performing configuration was 32 compute nodes aggregating their 2GB-per-node LFSs into a 64 GB IFS. This aggregation not only increases performance, but also allows compute nodes to keep their IO relatively local when working with large files that do not fit in a single compute node 2GB RAM-based LSF.

Our final experiment for the input data distribution section focused on how quickly we can distribute data from GFS to a set of IFSs, or potentially to LFSs. As in our previous experiment, we use Chirp (see Figure 78). Chirp has a native operation that allows a file (or set of files) to be distributed to a set of nodes over a spanning tree of copy operations. The spanning tree has the benefit of requiring fewer data transfers: log(*n*) instead of *n*, where n is the number of nodes.

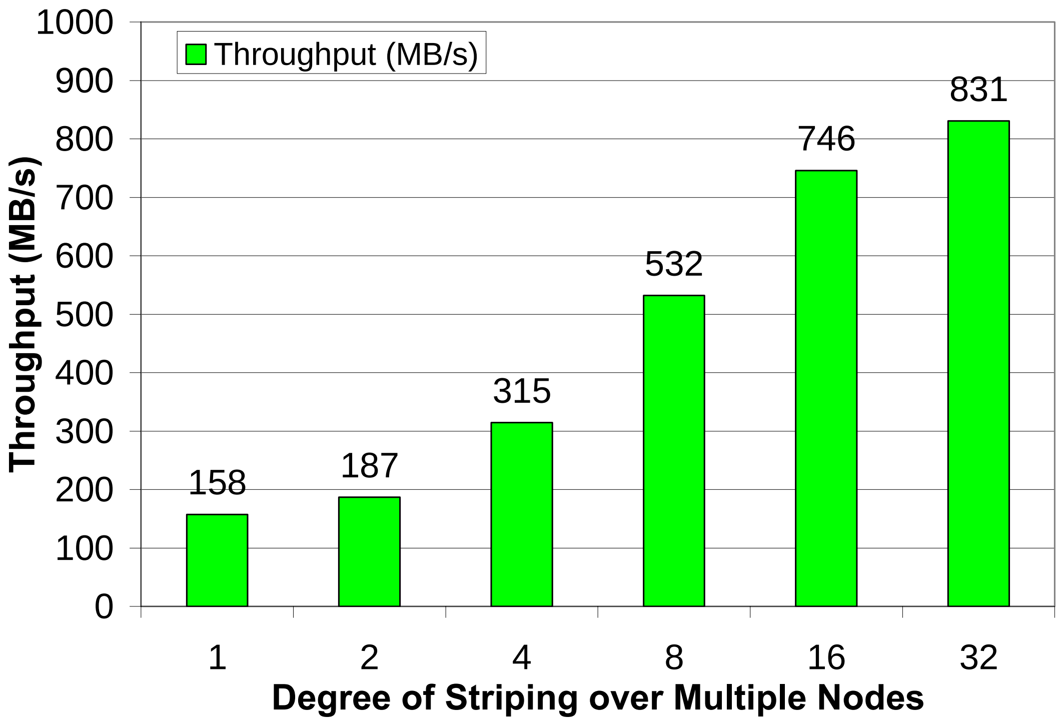


Figure 77: Read performance, varying the degree of striping of data across multiple nodes from 1 to 32 using the torus network

In the case of a naïve data distribution in which compute nodes read data directly from GFS (GPFS in our case as noted in the figure), computing the aggregate throughput is straightforward: *throughput = nodes*dataSize/workloadTime.* For the spanning tree distribution, computing the actual throughput is problematic since the number of transfers is lower than in the naïve method. To make the comparison fair, we compute throughput for the spanning tree distribution with the same formula as for the naïve data distribution, although the actual network traffic would have been significantly less. We believe this is the correct way to compare the two approaches, as it emphasizes the time to complete the workload. On up to 4K processors, GPFS achieves 2.4 GB/s at the largest scale (2.4 MB/s per node). This is the peak rated performance for the file system we tested *(/home*). However, the spanning tree approach achieves an equivalent of 12.5 GB/s on 4K processors. We plan to explore the performance of the spanning tree distribution at larger scales to find the torus network saturation point. We expect to achieve at least one order of magnitude better performance (for distributing a set of files to many compute nodes) at large scales when using the spanning tree approach as opposed to the naïve approach which reads each file from GPFS directly.

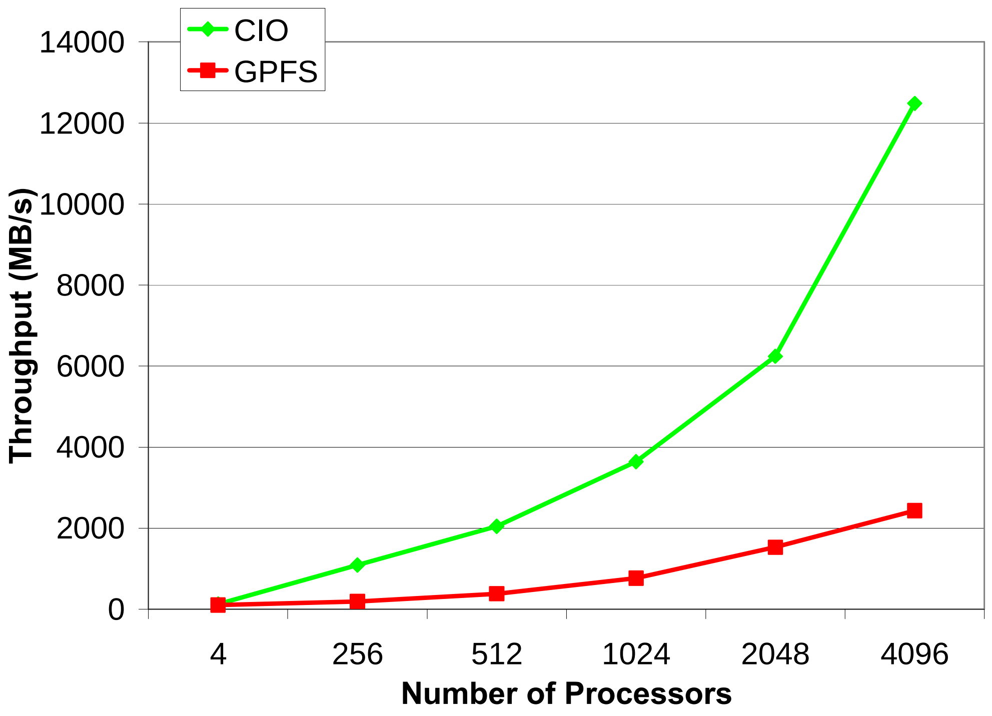


Figure 78: CIO distribution via spanning tree over Torus network vs. GPFS over Ethernet & Tree networks

6.6.2 Output Data Distribution

Our second goal for the collective IO primitives was to support the aggregation and transfer of many files from multiple LFSs or IFSs to the GFS. When writing from many compute nodes directly to GPFS (the GFS on the BG/P), care must be taken to avoid locking contention on metadata. One way to avoid this problem is to ensure that each compute node writes files to a unique directory. It is desirable to have as few clients as possible writing to GFS concurrently to limit any locking contention, and to allow the largest buffer sizes and aggregation and potentially small files into larger ones. It is also desirable to make write operations as asynchronous as possible to allow the overlap of computing and data transfer from the compute node. To achieve all these desirable features, we have implemented an output data collector (CIO, which we previously discussed) that resides on an IFS and acts as an intermediate buffer space for output generated on compute nodes. We use a ratio of 64:1 IFS to LFS, which significantly reduces the number of clients that write to GFS.

Our measurements (see Figure 79 and Figure 80) show that the CIO collector strategy yields close to the ideal efficiency when compared to compute tasks of the same length with no IO. For example, in Figure 79 we show the efficiency achieved with short tasks (4 seconds) that produce output files with sizes ranging from 1KB to 1MB. We see that CIO (the dotted lines) is able to achieve > 90% efficiency in most cases, and almost 80% in the worst case with larger files. In contrast, the same workload achieved only 10% to < 50% efficiency when using GPFS. We also observed an anomaly: a slight efficiency increase at the largest scale of 32K processors. One possible cause of this is that we reached the limit of Falkon dispatch throughput.

Figure 80 is similar to Figure 79, but uses 32 second tasks. We see a similar pattern, in which CIO achieves 90% efficiency, while GPFS achieves almost 90% efficiency with 256 processors but less than 10% on 96K processors.

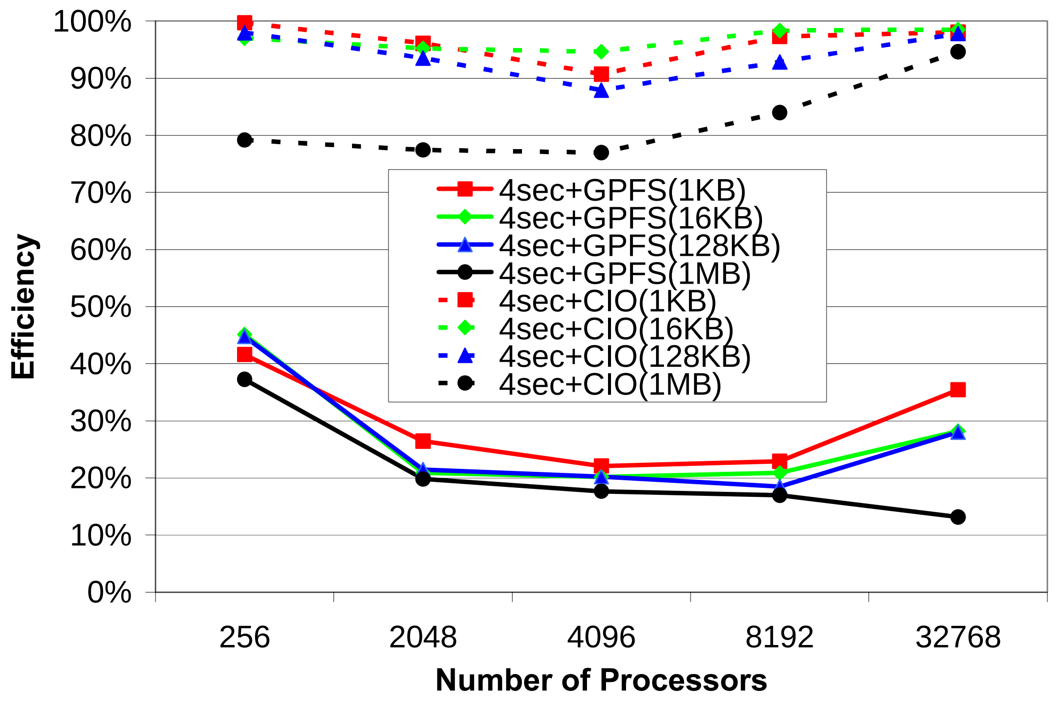


Figure 79: CIO vs. GFS efficiency for 4 second tasks, varying data size (1KB to 1MB) on 256 to 32K processors

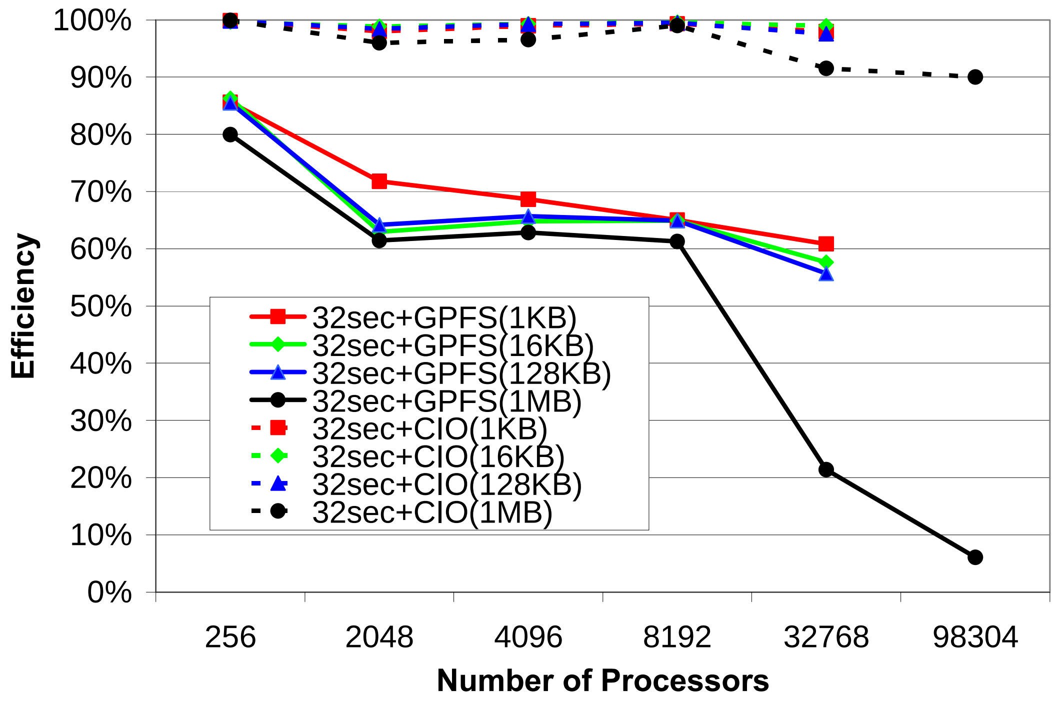


Figure 80: CIO vs GPFS efficiency for 32 second tasks, varying data size (1KB to 1MB) for 256 to 96K processors.

We also extract from these experiments the achieved aggregate throughput (shown in Figure 81). We limit this plot to the 1 MB case for readability. Notice the extremely poor GPFS write performance as the number of processors increases, peaking at only 250 MB/s. The CIO throughput is almost an order of magnitude higher, peaking at 2100 MB/s, and is within a few percent of the ideal case (tasks with the same duration, but with only local IO to RAM-based LFS, labeled 4sec+RAM and 32sec+RAM).

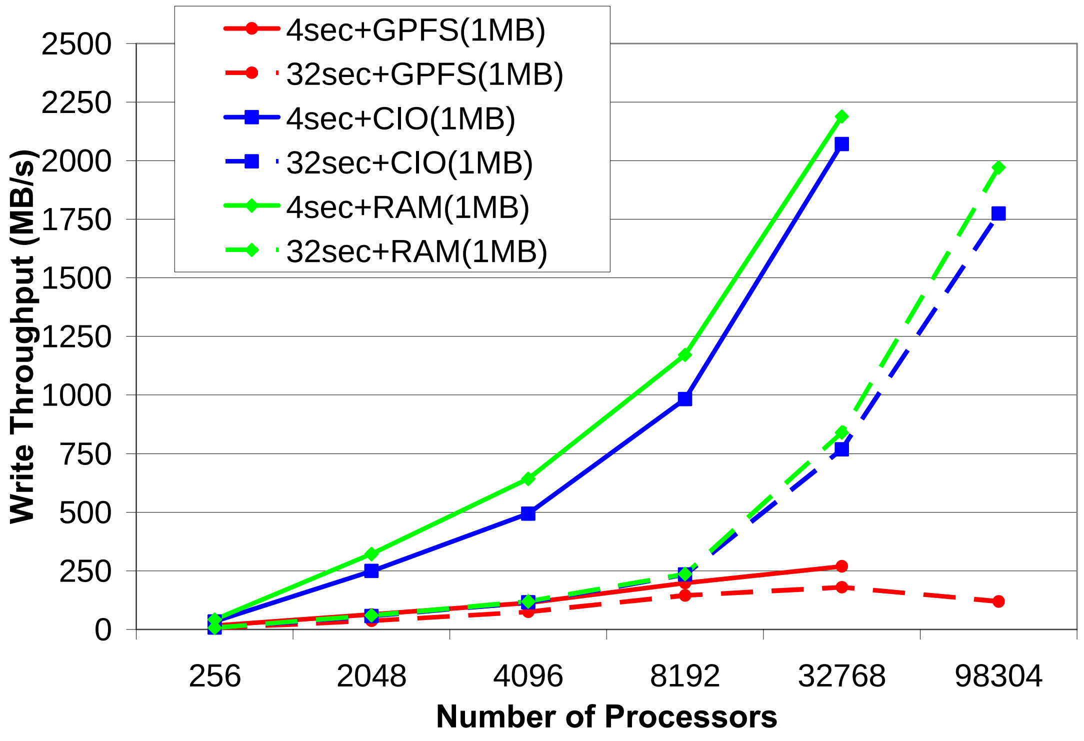


Figure 81: CIO collection write performance compared to GPFS write performance on up to 96K processors

6.7 Conclusions

We have defined a new paradigm – MTC – which aims to bridge the gap between two computing paradigms, HTC and HPC. MTC applications are typically loosely coupled that are communication-intensive but not naturally expressed using standard message passing interface commonly found in high performance computing, drawing attention to the many computations that are heterogeneous but not "happily" parallel. We believe that today's existing HPC systems are a viable platform to host MTC applications. We also believe MTC is a broader definition than HTC, allowing for finer grained tasks, independent tasks as well as ones with dependencies, and allowing tightly coupled applications and loosely coupled applications to co-exist on the same system.

Furthermore, having native support for data intensive applications is central to MTC, as there is a growing gap between storage performance of parallel file systems and the amount of processing power. As the size of scientific data sets and the resources required for analysis increase, data locality becomes crucial to the efficient use of large scale distributed systems for scientific and data-intensive applications [19]. We believe it is feasible to allocate large-scale computational resources and caching storage resources that are relatively remote from the original data location, co-scheduled together to optimize the performance of entire data analysis workloads which are composed of many loosely coupled tasks.

When building systems to perform such analyses, we face difficult tradeoffs. Do we dedicate computing and storage resources to analysis tasks, enabling rapid data access but wasting resources when analysis is not being performed? Or do we move data to compute resources, incurring potentially expensive data transfer costs? 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. This chapter proposes exactly such a stochastic optimizer. Our work is significant due to the support data intensive applications require with the growing gap between parallel file system performance and the increase in the number of processors per system. We have shown good support for MTC on a variety of resources from clusters, grids, and supercomputers through our work on Swift [13, 55, 10] and Falkon [4, 2]. Furthermore, we have addressed data-intensive MTC by offloading much of the I/O away from parallel file systems and into the network, making full utilization of caches (both on disk and in memory) and the full network bandwidth of commodity networks (e.g. gigabit Ethernet) as well as proprietary and more exotic networks (Torus, Tree, and Infiniband). [1, 17] We have used two techniques, data diffusion to harness data locality in the application data access patterns, and collective I/O to efficiently distribute data and aggregate results in an efficient manner.

We believe that there is more to HPC than tightly coupled MPI, and more to HTC than embarrassingly parallel long running jobs. Like HPC applications, and science itself, applications are becoming increasingly complex opening new doors for many opportunities to apply HPC in new ways if we broaden our perspective. We hope the definition of Many-Task Computing leads to a stronger appreciation of the fact that applications that are not tightly coupled via MPI are not necessarily embarrassingly parallel: some have just so many simple tasks that managing them is hard, some operate on or produce large amounts of data that need sophisticated data management in order to scale. There also exist applications that involve MPI ensembles, essentially many jobs where each job is composed of tightly coupled MPI tasks, and there are loosely coupled applications that have dependencies among tasks, but typically use files for inter-process communication. Efficient support for these sorts of applications on existing large scale systems, including future ones will involve substantial technical challenges and will have big impact on science.

7 Accelerating Scientific Applications on Clusters, Grids, and Supercomputers

This chapter showcases the end-product of this entire dissertation, the wide range of applications that have been run using the proposed techniques, and what improvements they have achieved in both performance and scalability. The results presented in this chapter have been published in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17].

7.1 Falkon Usage

Over the past year, Falkon [2, 4] has seen wide deployment and usage across a variety of systems, from the TeraGrid [44], the SiCortex [56] at Argonne National Laboratory (ANL), the IBM Blue Gene/P [146] supercomputer at ALCF ANL, and the Sun Constellation supercomputer from the TeraGrid [44]. Figure 82 shows plot of Falkon across these various systems from December 2007 - October 2008. 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 163K peak number of processors, with 1.4 million CPU hours consumed and 164 million tasks for an average task execution time of 31 seconds.

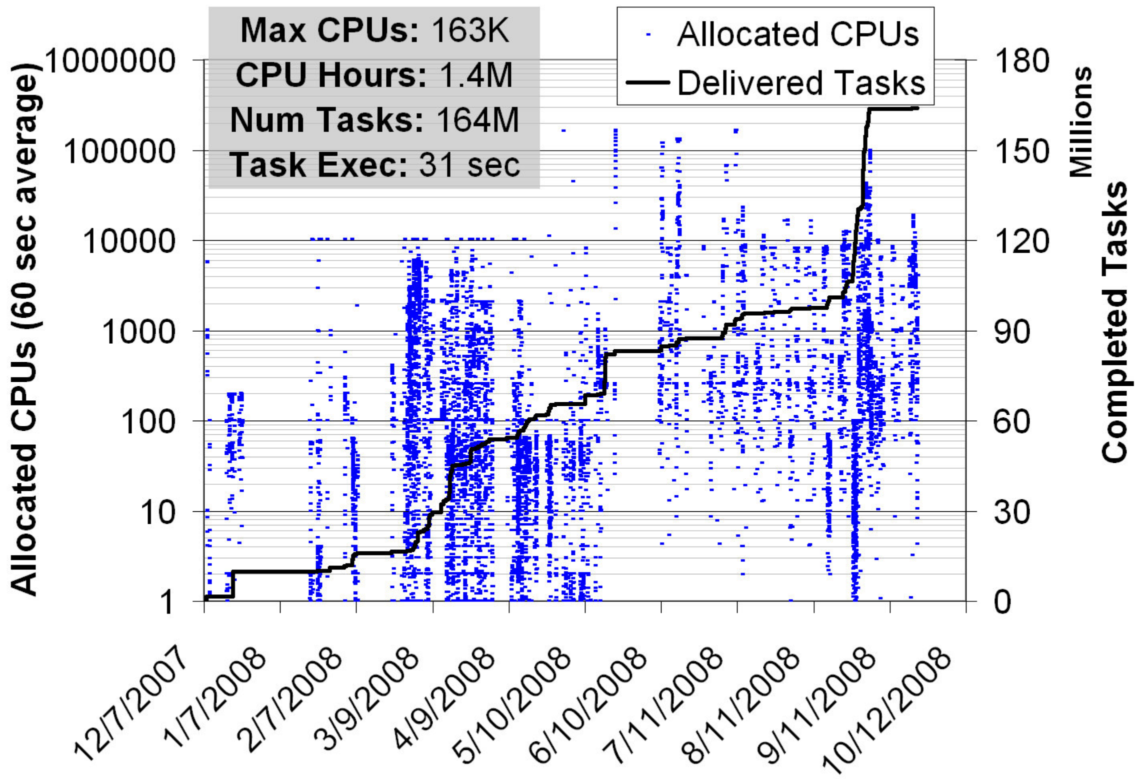


Figure 82: December 2007 - October 2008 plot of Falkon across various systems (ANL/UC TG 316 processor cluster, SiCortex 5832 processor machine, IBM Blue Gene/P 4K and 160K processor machines)

Figure 83 filters the workloads to only those executed on the IBM Blue Gene/P supercomputer (both the 4K and 160K processor machines), and extends the plot to December 2008. We see that we have substantially less overall number of tasks, down to 52 million tasks, yet the processor time consumed is still 1.4 million CPU hours. In fact, there were 200K CPU hours consumed in November and December on the Blue Gene/P machines, which allows us to deduce that the other systems (SiCortex and ANL/UC TeraGrid cluster) had consumed 200K CPU hours over the past year. On the Blue Gene machines, it appears that the average task execution time is also higher at 97 seconds per task.

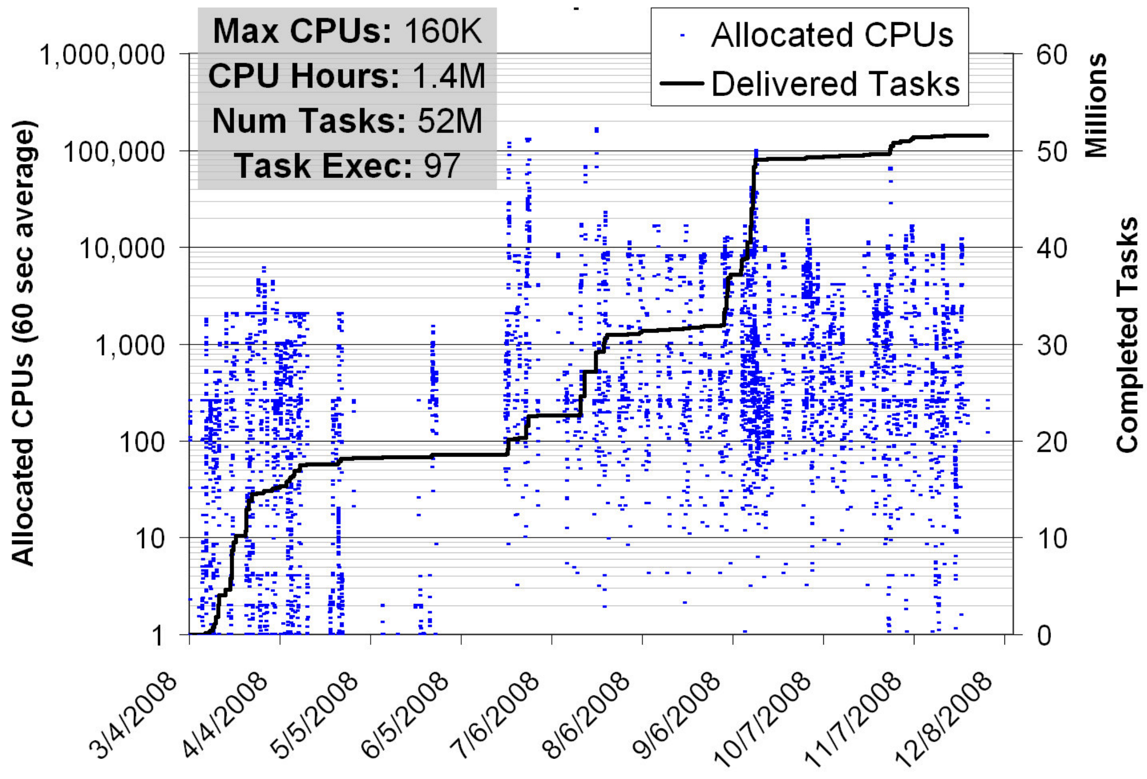


Figure 83: March 2008 – December 2008 plot of Falkon on the IBM Blue Gene/P machines with 4K and 160K processors

Many of the results presented in this chapter are represented in Figure 82 and Figure 83, although some applications were run in 2007 prior to the history log repository being instantiated.

7.2 Swift

Many of the applications presented in this chapter were executed via the Swift runtime system (see Figure 84). Swift is a scalable environment for efficient specification, scheduling, monitoring and tracking of SwiftScript programs. The system consists a SwiftScript compiler that compiles SwiftScript programs into abstract computation plans, an execute engine built on CoG Karajan and a set of libraries and tools for the scheduling and execution of the computation plans, a provenance tracking tool that records the execution process, and resource provisioning mechanism for submission to diverse computation and storage resources.

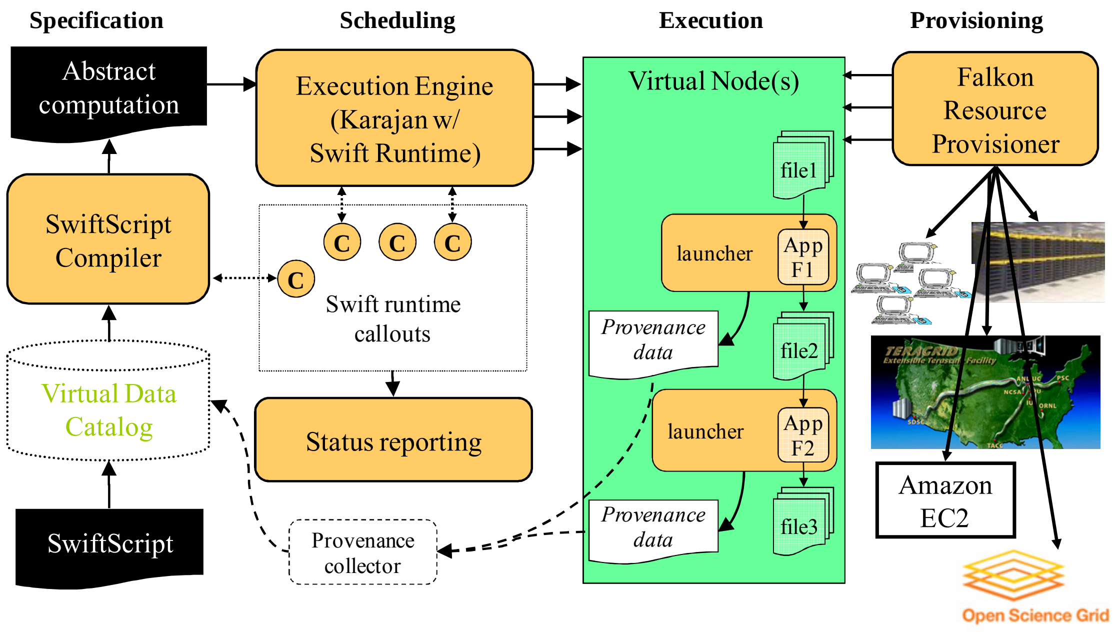


Figure 84: The Swift Runtime System

We have integrated Falkon into the Karajan [13, 55] workflow engine, which in term is used by the Swift parallel programming system [13]. Thus, Karajan and Swift applications can use Falkon without modification. We observe reductions in end-to-end run time by as much as 90% when compared to traditional approaches in which applications used batch schedulers directly.

Swift has been applied to applications in the physical sciences, biological sciences, social sciences, humanities, computer science, and science education. Table 10 characterizes some applications in terms of the typical number of tasks and stages.

Table 10: Swift applications

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

7.3 Functional Magnetic Resonance Imaging

We note that for each volume, each individual task in the fMRI 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 85 we show the execution time for different input data sizes for the fMRI workflow.

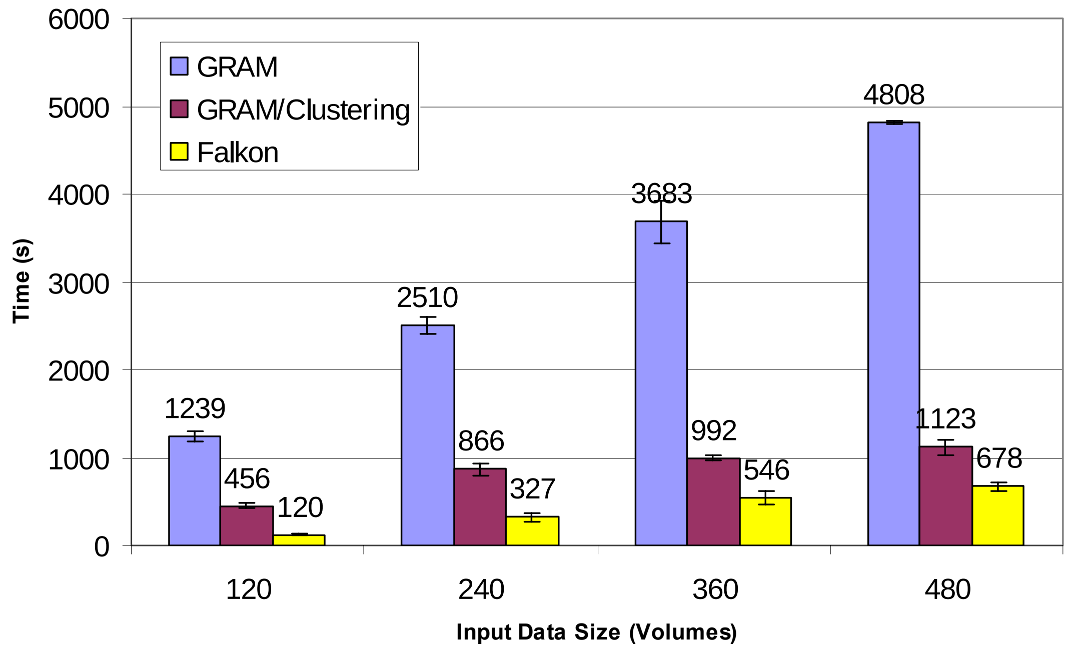


Figure 85 Execution Time for the fMRI Workflow

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, plus or minus).

7.4 MolDyn (Chemistry Domain)

We further illustrate the execution process in Falkon using a molecular dynamics (MolDyn) application. The goal of this study 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.

The MolDyn workflow consists of the following 8 stages:

- Stage 1: Annotate each of the molecules in the study with charges.

- Stage 2: For each molecule, the ligand structures are modified using Antechamber to generate the individual parameter and topology files for CHARMM Program. Antechamber automatically detects the atom type, bond type and bond order from the three-dimensional geometry of the molecule, and generates the residue topology file.
- Stage 3: The ligand structure is equilibrated using CHARMM program.
- Stage 4: Compute the solvation energy using three coupling (staging) parameters using the PERT module of the program CHARMM.
- Stage 6 computes the free energy for each of the resulting input configurations of each molecule in the study using weighted histogram analysis method (WHAM).
- Stages 7 and 8 extract the free energy value from prior stages and put the final data into a tabular form.

Stage 1 is done once for all molecules in the workflow, stages 2-8 are done for each molecule. The number of jobs in the workflow is about 1 + 84N where N is the number of molecules. The computation for each molecule takes about 235.4 minutes on one ANL_TG node processor, and 22.7 min given up to 32 ANL_TG nodes (64 processors); this resulted in a speedup of 10.4x, significantly less than 64 (the number of processors used at most), mostly due to the structure of the workflow which had several stages which were not parallelizable. Figure 86 shows the graphical representation of each task for the 1 molecule experiment. Red denotes wait queue time in Falkon and green denotes execution time per task.

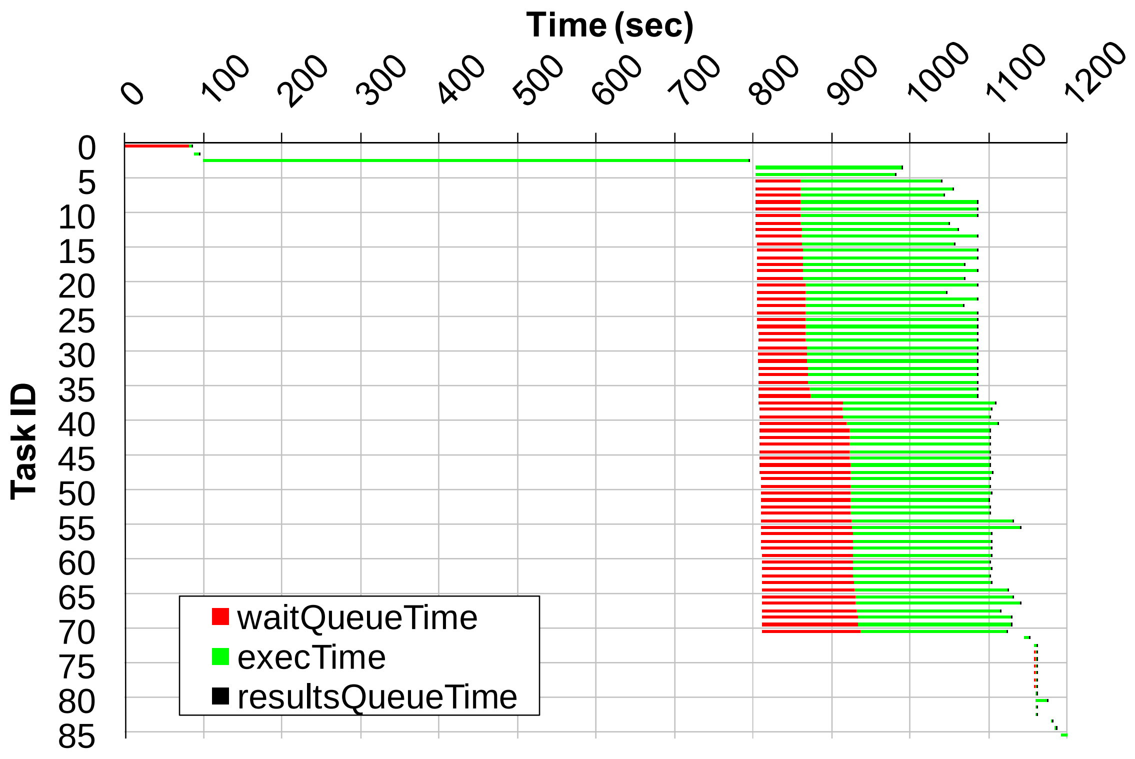


Figure 86: MolDyn 1 Molecule Workflow Task View

We were using the dynamic resource provisioning capabilities of Falkon in this experiment, and hence resources were only allocated on demand when they were needed. Notice the first job queue time is about 81 seconds, essentially the time it took from when Falkon requested one node and when it was allocated and ready to process work. Then, after the first three jobs completed in serial fashion), there was a stage with 68 independent jobs, which in turn triggered Falkon to allocated 31 more nodes (dual processors each) to be able to process all 68 independent jobs in parallel. The red lines for these 68 jobs shows the time delay that the allocation request took to traverse GRAM4 and PBS until the workers were ready to process jobs.

To better understand the structure of the MolDyn workflow, we show the graph structure of the workflow in Figure 87. Notice the large amount of parallelism that is obtainable at various stages of the workflow.

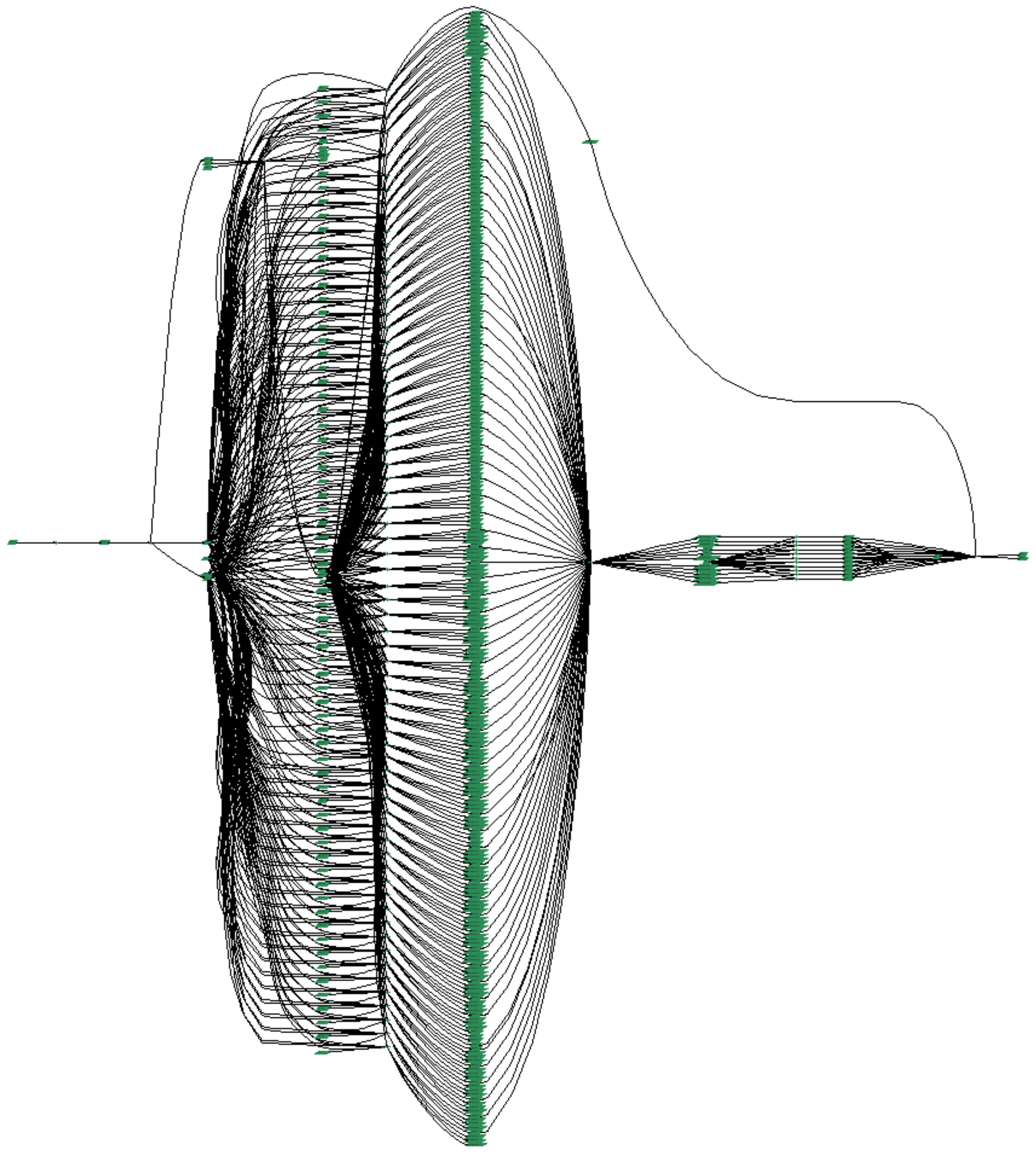


Figure 87: MolDyn Workflow Graph

Figure 86 showed the execution of 1 molecule, which is composed of 85 jobs that consume 235.4 CPU minutes. Our next 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 88 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 88 shows the same information on a per task basis, while Figure 90 shows the information on a per executor 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.

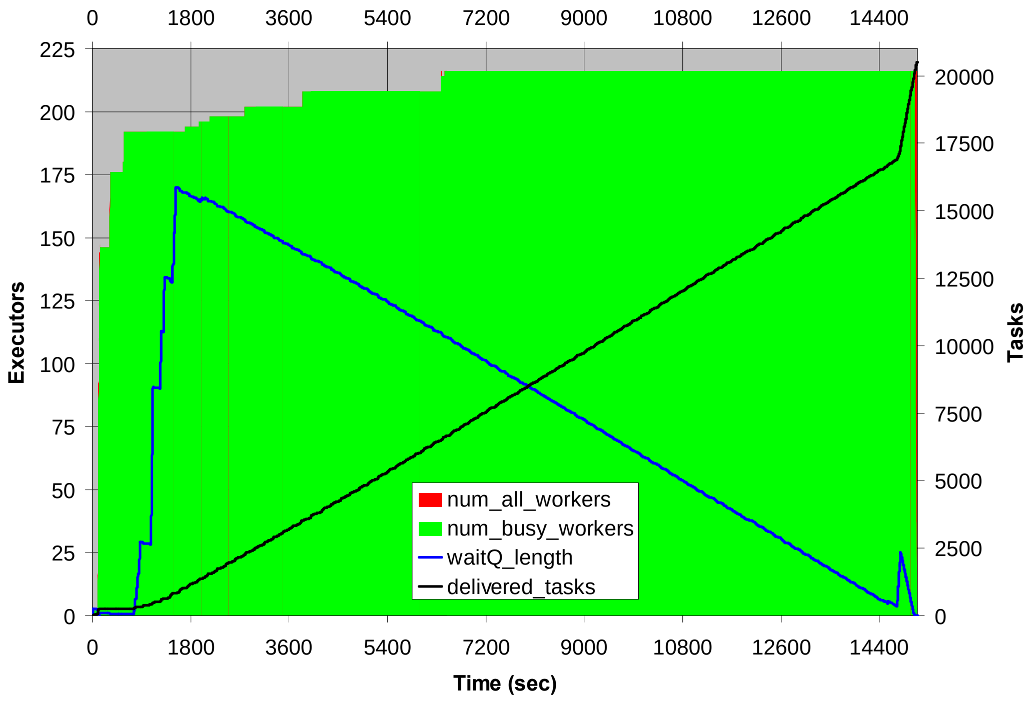


Figure 88: 244 Molecule MolDyn application; summary view showing executor's utilization in relation to the Falkon wait queue length as experiment time progressed

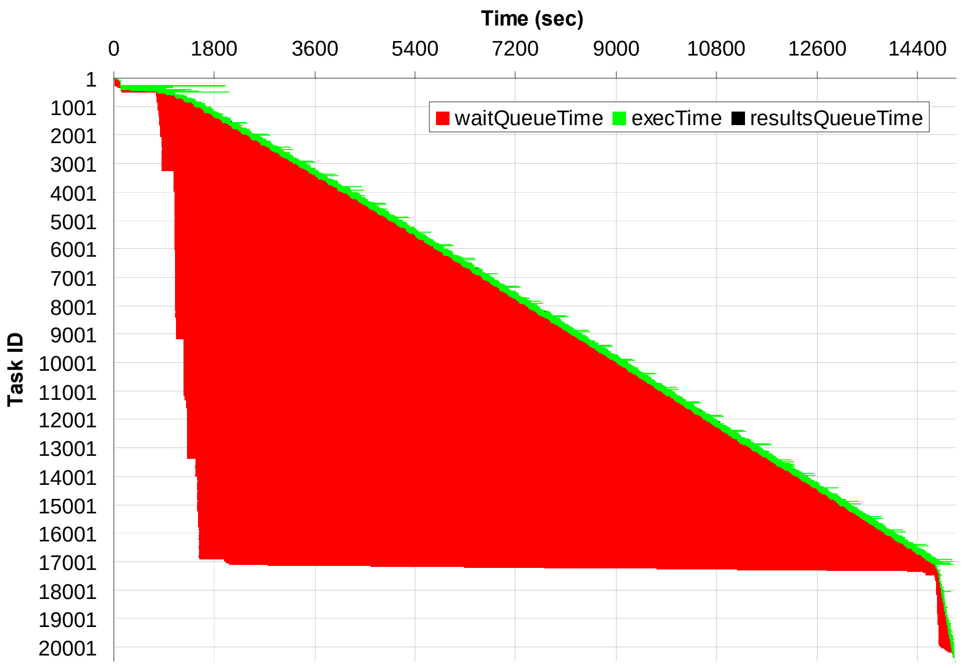


Figure 89: 244 Molecule MolDyn application; task view showing per task wait queue time and execution time as experiment time progressed

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 as we have used for Falkon, using up to 200 processors) took 25292 seconds. So we achieved a speedup of only 25.3X compared to 206.9X speedup 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; the best case scenario is that the workflow could have kept 40 machines busy, but in reality the number of concurrent jobs ranged between 30~40 jobs. 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 having a system like Falkon that allows the specific configuration of new queues that behave appropriately on a per application basis, which is quite impractical to do (mostly due to policy) in real-world deployed Grids.

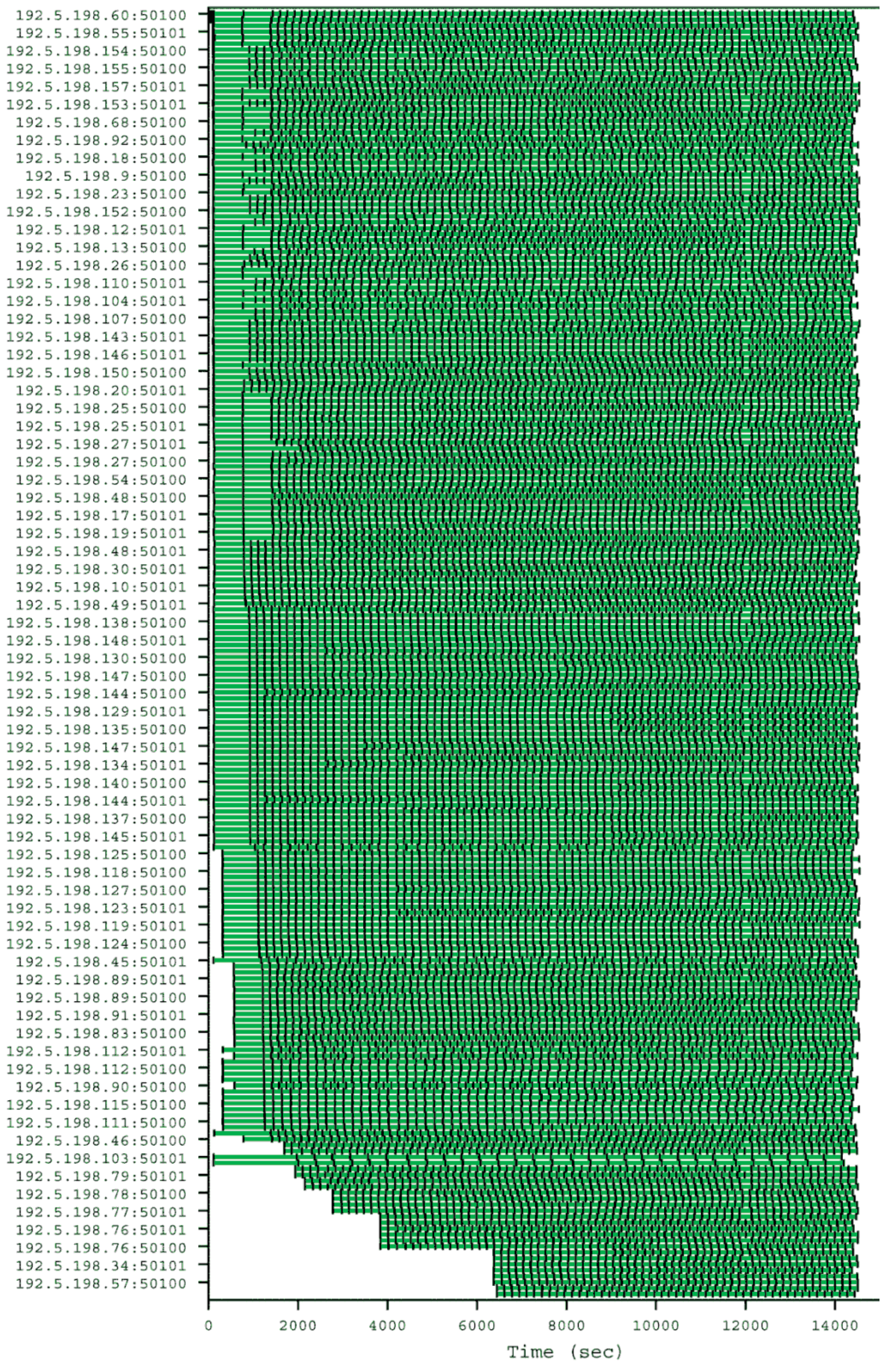


Figure 90 244 Molecule MolDyn application: Executor's view showing each task that it executed (green) delineated by the black vertical bars showing each task boundary

The evaluation of the Swift runtime system shows that it is fast and scalable in executing large scale scientific computations. Swift leverages lightweight threading techniques and can schedule hundreds of thousands of parallel computations efficiently; its combination of type checking, retry mechanism and restart log supports reliable workflow execution. The abstract provider interface and various implementations allow workflows to be executed either on a single desktop, on a cluster managed by a batch scheduler, or on multi-site distributed resources. The integration with the lightweight Falkon execution service significantly improves system throughput for large number of small jobs. In particular, we show that Swift plus Falkon can achieve comparable performance to MPI execution for the Montage workflow, and also can reduce execution time by as much as 90% for the fMRI pipeline when compared with GRAM and PBS submission. We also showed a large scale MolDyn workflow with 20K tasks that was able to achieve 99.8% efficiency on 216 processors over a period of 15K seconds; Falkon was able to achieve an application speedup of 206.9X when GRAM/PBS was only able to achieve a 25.3X speedup on the same Grid site with similar usage conditions.

7.5 Molecular Dynamics: DOCK

Our first project deals with virtual screening on the Blue Gene/P of core metabolic targets against KEGG [69] compounds and drugs, and uses the DOCK6 [61] application. 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. In this application run, nine proteins that perform key enzymatic functions in the core metabolism of bacteria and humans were selected for screening against a database of 15,351 natural compounds and existing drugs in KEGG's Ligand Database.

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.

7.5.1 Molecular Dynamics: DOCK5

Prior to running the real workload with nearly 1M molecules, which exhibits wide variability in its job durations, we investigated the scalability of the application under larger than normal I/O to compute ratios and by reducing the number of variables. From the ligand search space, we selected one that needed 17.3 seconds to complete. We then ran a workload with this specific molecule (replicated to many files) on a varying number of processors from 6 to 5760 on the SiCortex. The ratio of I/O to compute was about 35 times higher in this synthetic workload than the real workload whose average task execution time was 660 seconds. Figure 91 shows the results of the synthetic workload on the SiCortex system.

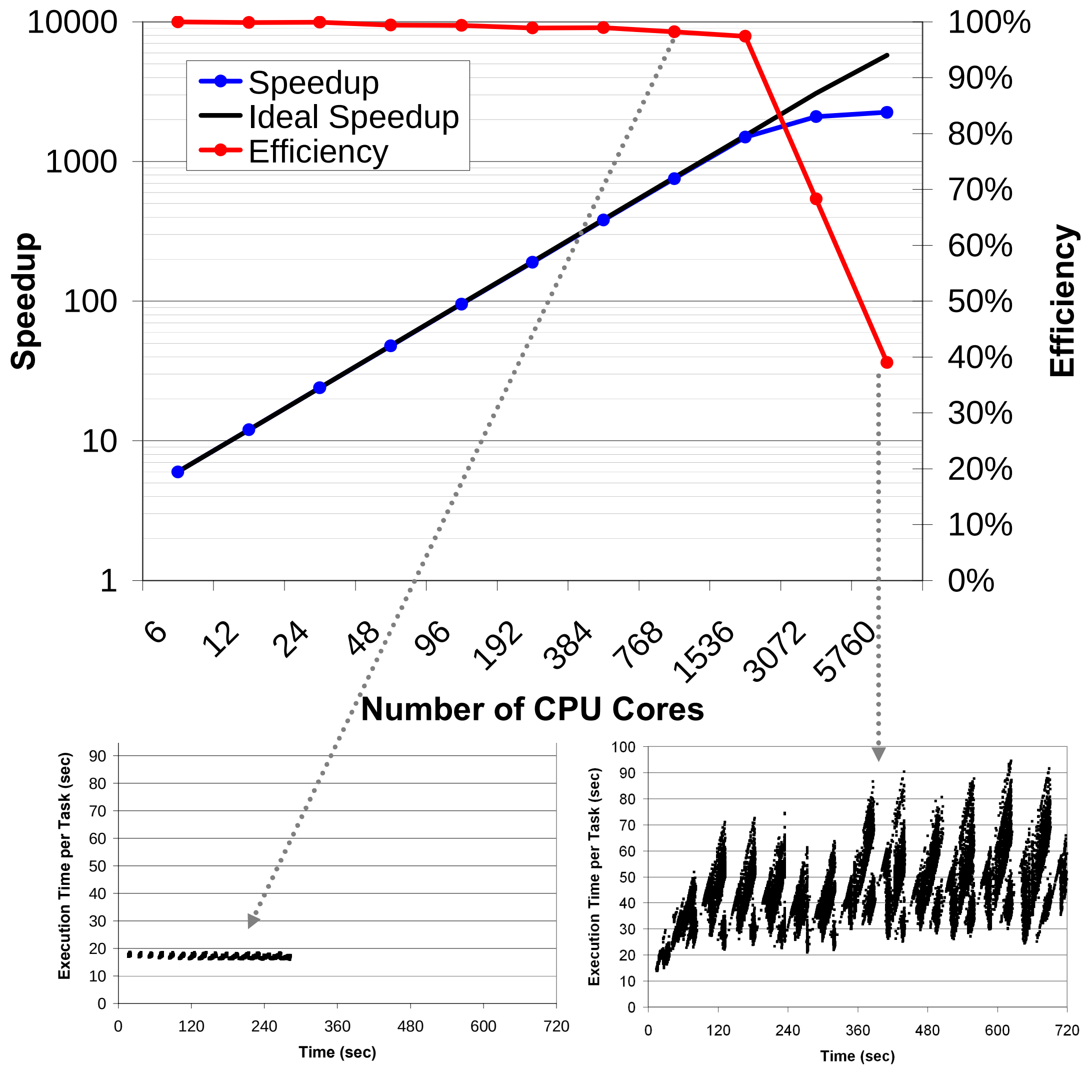


Figure 91: Synthetic workload with deterministic job execution times (17.3 seconds) while varying the number of processors from 6 to 5760 on the SiCortex

Up to 1536 processors, the application had excellent scalability with 98% efficiency, but due to shared file system contention in reading the input data and writing the output data, the efficiency dropped to below 70% for 3072 processors and below 40% for 5760 processors. We concluded that shared file system contention caused the loss in efficiency, due to the average execution time per job and the standard deviation as we increased the number of processors. Notice in the lower left corner of Figure 91 how stable the execution times are when running on 768 processors, 17.3 seconds average and 0.336 seconds standard deviation. However, the lower right corner shows the performance on 5760 processors to be an average of 42.9 seconds, and a standard deviation of 12.6 seconds. Note that we ran another synthetic workload that had no I/O (sleep 18) at the full 5760 processor machine scale, which showed an average of 18.1 second execution time (0.1 second standard deviation), which rules out the dispatch/execute mechanism. The likely contention was due to the application's I/O patterns to the shared file system.

The real workload of the DOCK application involves a wide range of job execution times, ranging from 5.8 seconds to 4178 seconds, with a standard deviation of 478.8 seconds. This workload (Figure 92 and Figure 93) has a 35X smaller I/O to compute ratio than the synthetic workload presented in Figure 91. Expecting that the application would scale to 5760 processors, we ran a 92K job workload on 5760 processors. In 3.5 hours, we consumed 1.94 CPU years, and had 0 failures throughout the execution of the workload. We also ran the same workload on 102 processors to compute speedup and efficiency, which gave the 5760 processor experiment a speedup of 5650X (ideal being 5760) and an efficiency of 98.2%. Each horizontal green line represents a job computation, and each black tick mark represents the beginning and end of the computation. Note that a large part of the efficiency was lost towards the end of the experiment as the wide range of job execution times yielded the slow ramp-down of the experiment and leaving a growing number of processors idle.

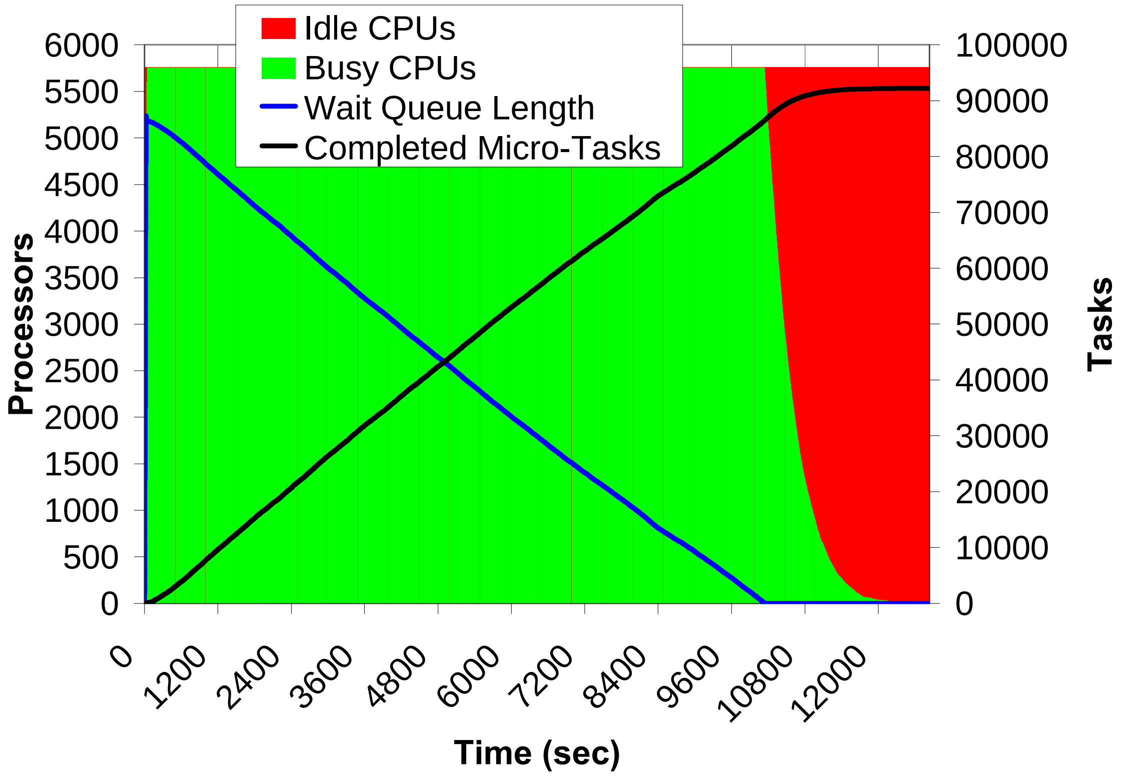


Figure 92: DOCK application (summary view) on the SiCortex; 92K jobs using 5760 processor cores

Despite the loosely coupled nature of this application, our preliminary results show that the DOCK application performs and scales well on thousands of processors. The excellent scalability (98% efficiency when comparing the 5760 processor run with the same workload executed on 102 processors) 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, with the majority of the computations being in the 100s of seconds, with an average of 660 seconds.

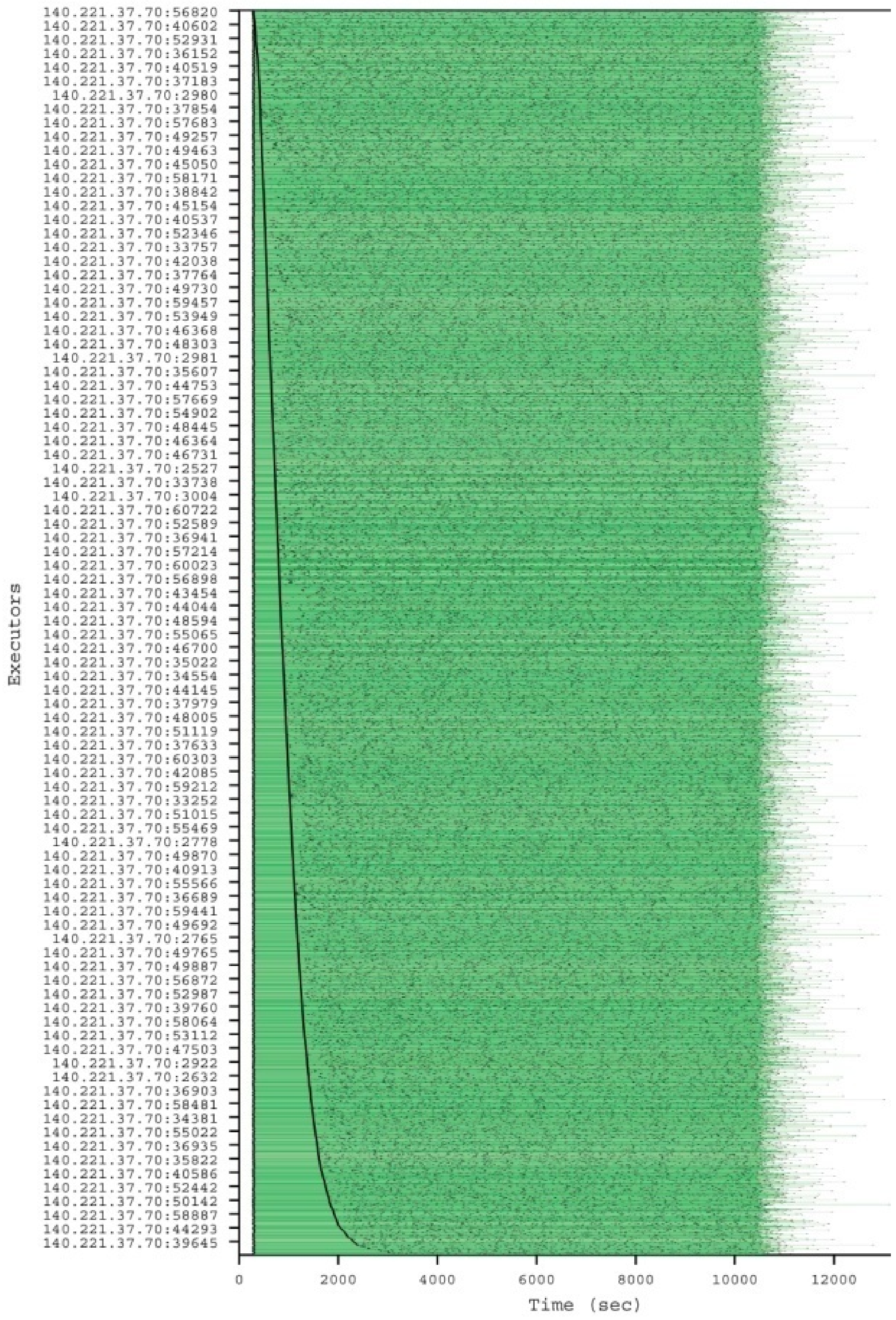


Figure 93: DOCK application (per processor view) on the SiCortex; 92K jobs using 5760 processor cores

Running the entire workload consisting of 934,803 molecules on 116K CPU cores took 2.01 hours (see Figure 94). The per-task execution time was quite varied (even more so than the DOCK6 runs from Figure 95), 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 again.

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 half the scale of 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 beginning and end of computations), with the majority of the computations being in the 100s of seconds, with an average of 713 seconds.

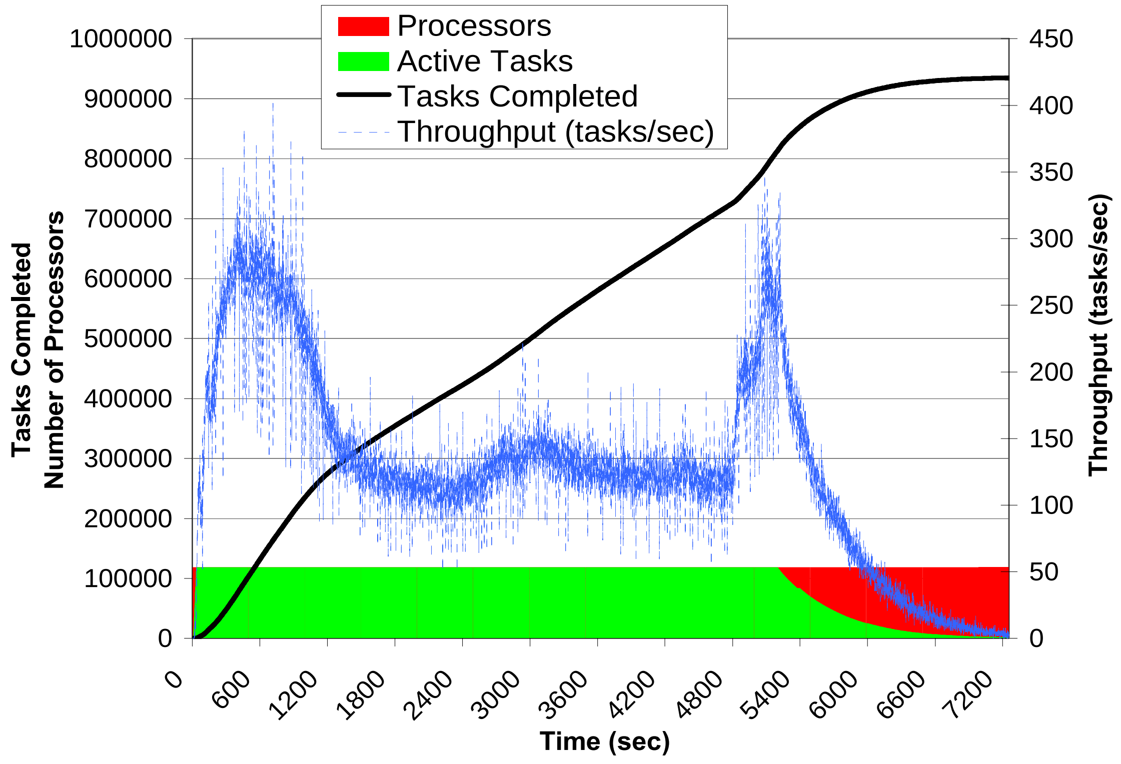


Figure 94: 934,803 DOCK5 runs on 118,784 CPU cores on Blue Gene/P

7.5.2 DOCK6 Performance Evaluation

We also ran the computation of the binding affinity between each compound in the database and each protein was performed with 138,159 runs of DOCK6 (a newer version of DOCK5 we ran in the previous sub-sections) on the Blue Gene/P. Using 32 racks on the Blue Gene/P (128K cores at 0.85 GHz), these runs took 2807 seconds (see Figure 95), totaling 3.5 CPU years.

The sustained utilization (while there were enough tasks to be done, roughly 600 seconds) was 95%, with the overall utilization being 30%. The large underutilization was caused by the heterogeneous task execution time (23/783/2802 +/- 300 seconds, for min/aver/max +/- stdev respectively). Expecting a significant underutilization, we had overlapped another application to start running as soon as the sustained period ended at around 600 seconds. The other application had enough work to be done that it actually used all of the idle CPUs from Figure 95 (the red area) with 97% utilization.

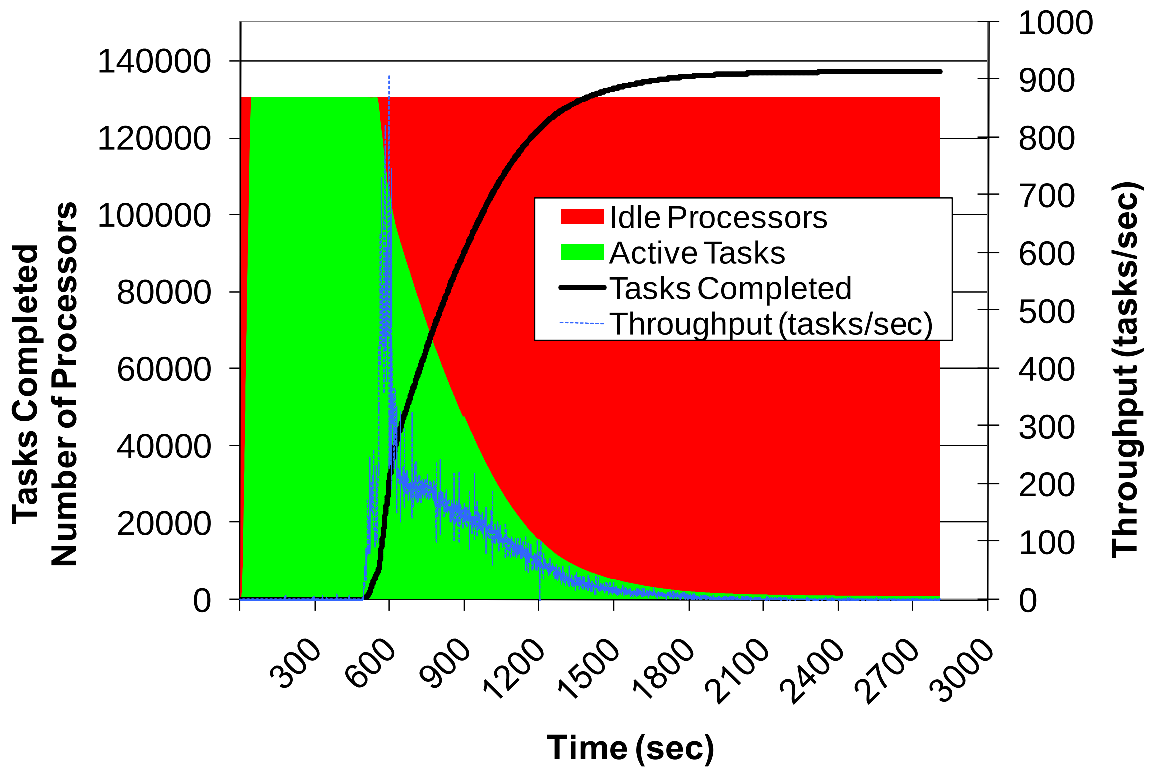


Figure 95: 138,159 DOCK6 runs on 131,072 CPU cores on Blue Gene/P

7.5.3 DOCK Application using CIO

We have shown significant performance and scalability improvements for synthetic data-intensive workloads using the proposed collective I/O in Section 6.6. To determine how these improvements translate into real application performance, we evaluated the utility of collective IO on a molecular dynamics workflow which screens candidate drug compounds against metabolic protein targets using the DOCK6 application [61]. In this application run, a database of 15,351 compounds was screened against nine proteins that perform key enzymatic functions in the metabolism of bacteria and humans.

The molecular dynamics docking workflow has 3 stages: 1) read input, compute the docking, and write output; 2) summarize, sort, and select results; and 3) archive results. In out tests, the DOCK6 invocations averaged 10KB of output every 550 seconds.

In the simple case where we use GFS, the input data of stage 1 is read from GFS to LFS, the application reads from LFS and writes its output to LFS, and finally the output is synchronously copied back to GFS. Stage 1 is parallelized to process each DOCK invocation on a separate processor core. Both stage 2 and stage 3 were originally a single process application that would run on a login node and access input data directly from GFS. In the case of using CIO, the stages are a bit different: stage 1 writes the output data from LFS to IFS asynchronously; stage 2 is parallelized across all processors and works on IFS; stage 3 copies the data from IFS to GFS. Figure 96 shows the breakdown of the 3 stages, and where time was being spent, for a total of 1412 seconds for CIO and 2140 seconds for GPFS. The first stage is negligibly faster with CIO (1.06X), and the third stage is 1.5X faster, but the second stage is 11.7X faster with 694 seconds being reduced down to 59 seconds. Stage 2 summarizes, sorts and filters the results, which CIO can handle much better in a distributed fashion (as opposed to the centralized GFS solution) with data accesses localized to IFS instead of GFS.

In order to see the effects of CIO at larger scale, we also ran the DOCK6 stage 1 with 135K tasks on 96K processors. The net result was a 1.12X speedup using CIO (1772 seconds) as compared to GPFS (1981 seconds) – a negligible speedup, as we expected for this compute-bound workload.

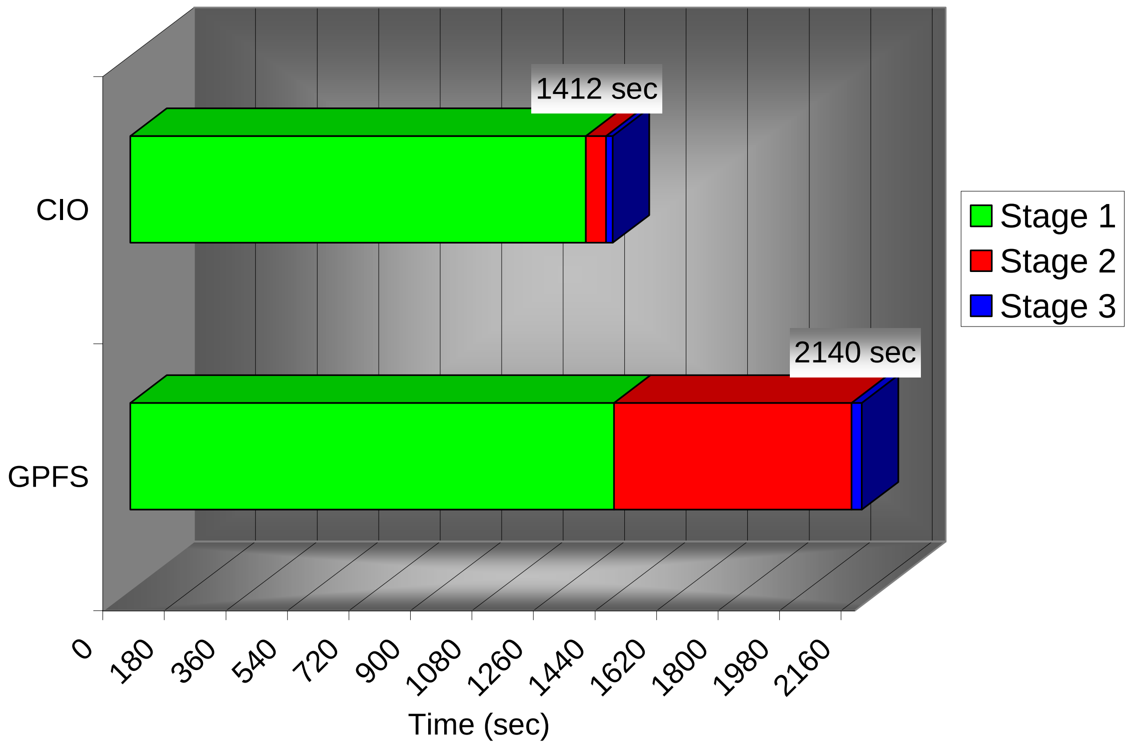


Figure 96: DOCK6 application summary with 15K tasks on 8K processor comparing CIO with GPFS

7.5.4 DOCK6 Scientific Results

The natural compound for 6 of the 8 targets scored reasonably well in terms of interaction energy and ranking (2/8 in top 2%, 2/8 in the top 10%, 2/8 in top 16%), especially considering these are natural compounds which rely on higher concentration levels for enzyme interaction compared to optimized inhibitors which rely on higher binding affinities. The 2D representation of best scoring compound, D03361, against protein NAD Kinase, is displayed in Figure 97.

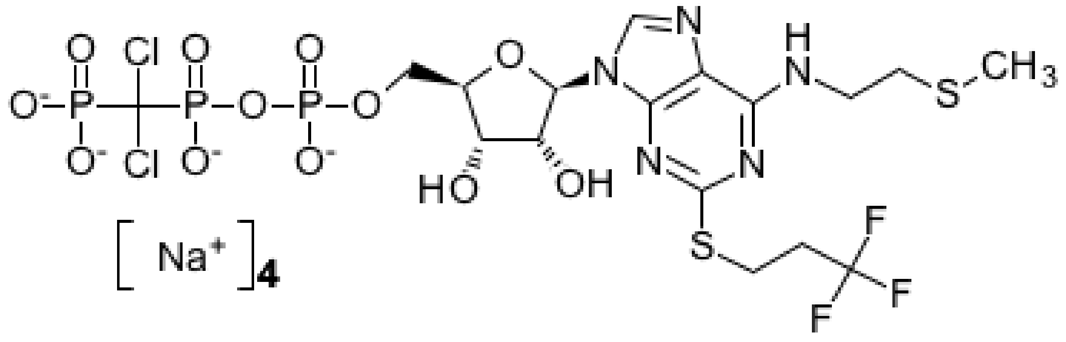


Figure 97: 2D representation of the best scoring compound, D03361, against protein NAD Kinase. Graphic is from KEGG

Reviewing the 3D structures of the compound-protein complexes generated by DOCK6 provided insight into the hydrogen bonding and chemical functional group placement within the pocket required for tight binding (see Figure 98).

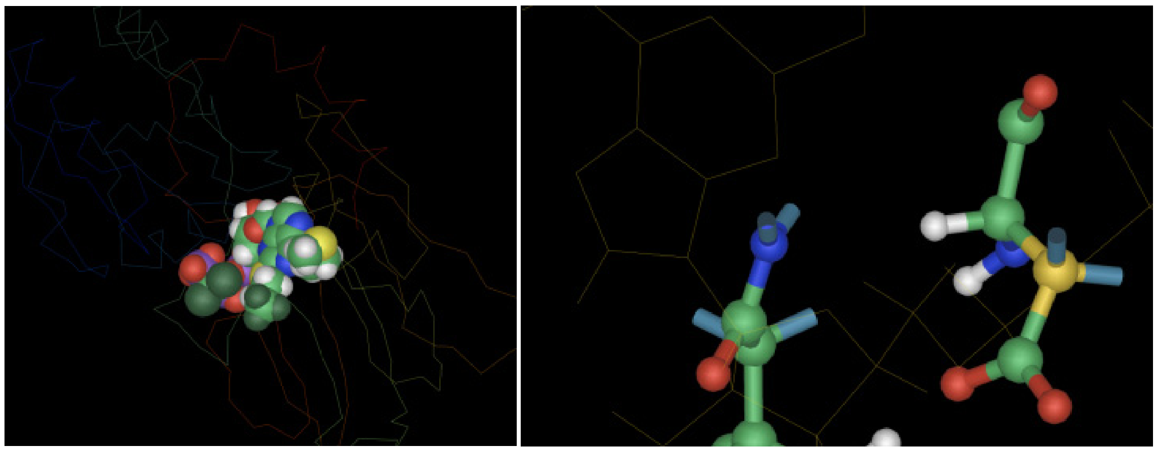


Figure 98: Left: Spherical representation of D03361, Cangrelor tetrasodium, docked in pocket of NAD kinase (showing backbone of protein as wireframe). Right: Close-up of NAD Kinase (backbone removed) with a side-chain carbon atom of residue ASP209 in yellow binding to an oxygen of D03361 (gold wire frame), and residue ASN115 interacting with core rings of D03361

Though more studies are required, the results were consistent with the idea that the pocket of a protein that binds its natural compound, that is not similar to other compounds, is more differentiated and specialized; whereas a protein pocket with a natural compound that is similar to many other compounds is more generic and lacks the shape and residue constraints for strong binding affinity. If this is the case, a protein with a more differentiated pocket (higher possible binding affinities) may be a more attractive target and should be pursued with a higher priority since it would lower concentrations of the drug required for inhibition. It is interesting to note for 7/8 of the proteins existing drugs were the top hit. The drugs included an antiplatelet agent, anti-hypertensive agent, a treatment for chronic dry eye, a vitamin precursor, and an opthalmic agent. Since these are safe for use in humans, performing follow-up wet lab assays for inhibition could lead rapidly to a novel application for an existing drug.

7.5.5 DOCK Application Conclusions

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 system that has at least tens of thousands of processors. Our loosely coupled approach holds great promise for making this problem tractable and manageable on today's largest supercomputers.

7.6 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 [62]. 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 multivariable parameter studies of the behavior of all aspects of petroleum refining covered by MARS.

As a simple test of utilizing the BG/P for refinery modeling, 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 144 model runs into a single task, yielding a workload with 1KB of input and 1KB of output data, and an average execution time of 65.4 seconds.

We executed a workload with 7 million model runs (49K tasks) on 2048 processors on the BG/P (Figure 99 and Figure 100). The experiment consumed 894 CPU hours and took 1601 seconds to complete. At the scale of 2048 processors, the per micro-task execution times were quite deterministic with an average of 0.454 seconds and a standard deviation of 0.026 seconds; this can also be seen from Figure 100 where we see all processors start and stop executing tasks at about the same time, the banding effects in the graph) . As a comparison, a 4 processor experiment of the same workload had an average of 0.449 seconds with a standard deviation of 0.003 seconds. The efficiency of the 2048 processor run in comparison to the 4 processor run was 97.3% with a speedup of 1993 (compared to the ideal speedup of 2048).

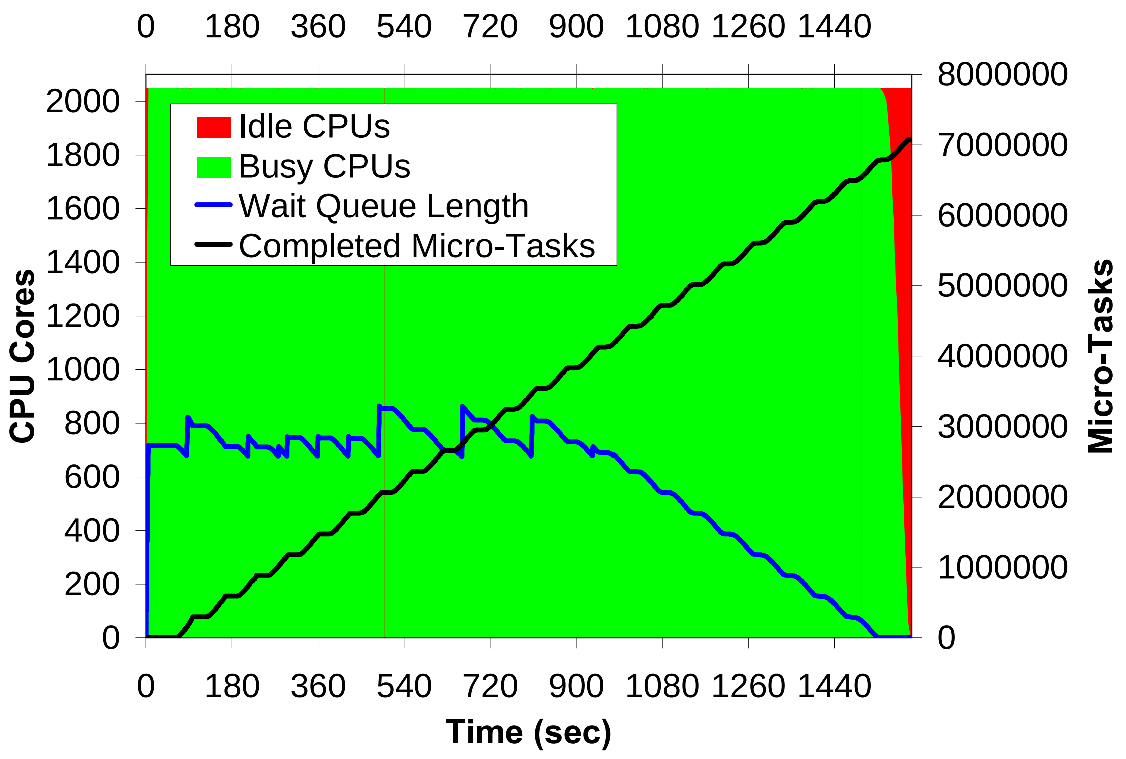


Figure 99: MARS application (summary view) on the BG/P; 7M micro-tasks (49K tasks) using 2048 processor cores

The results presented in these figures are from a static workload processed directly with Falkon. Swift on the other hand can be used to make the workload more dynamic, reliable, and provide a natural flow from the results of this application to the input of the following stage in a more complex workflow. Swift incurs its own overheads in addition to what Falkon experiences when running the MARS application. These overheads include 1) managing the data (staging data in and out, copying data from its original location to a workflow-specific location, and back from the workflow directory to the result archival location) , 2) creating per-task working directories (via mkdir on the shared file system), and 3) creation and tracking of status logs files for each task.

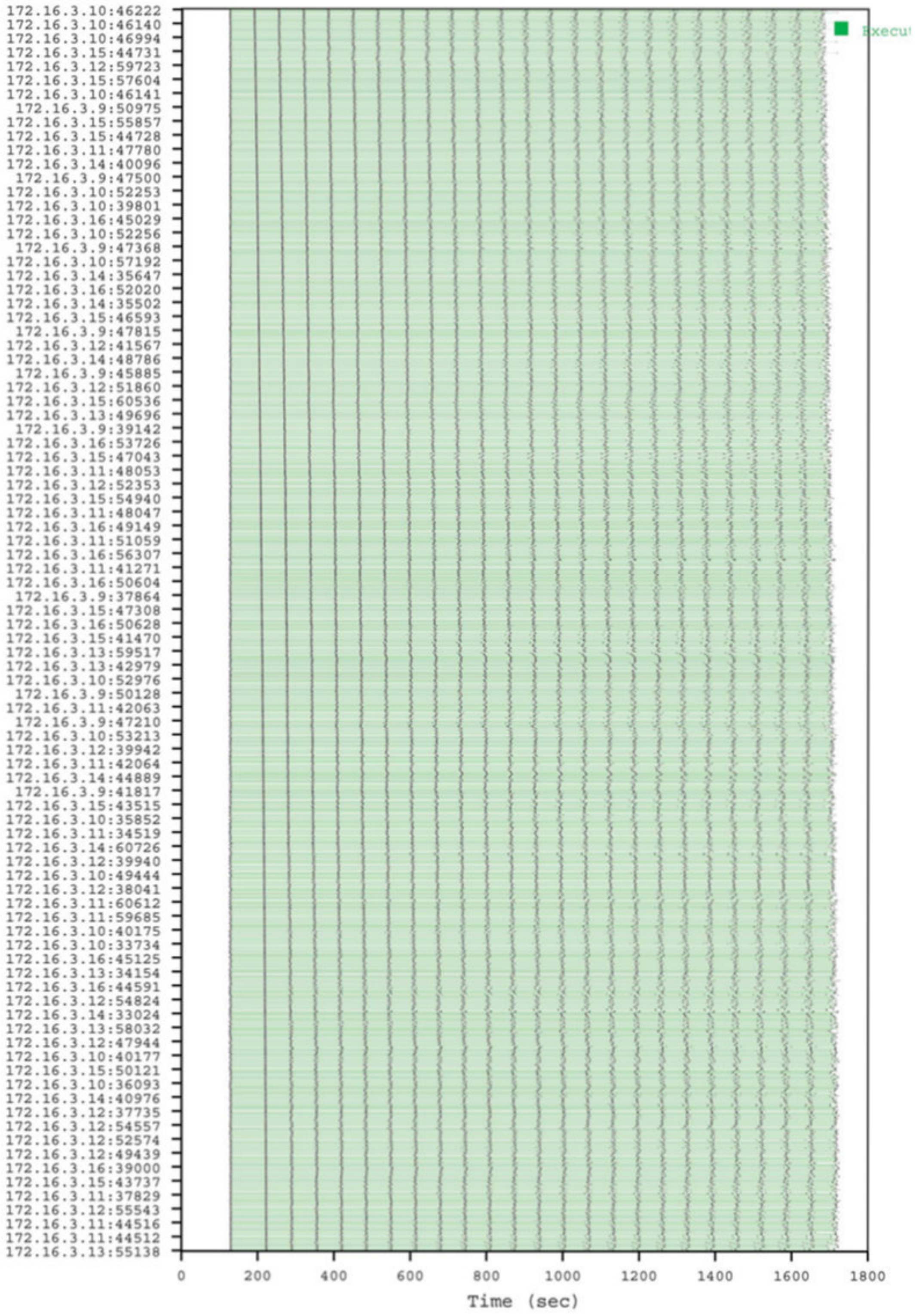


Figure 100: MARS application (per processor view) on the BG/P; 7M micro-tasks (49K tasks) using 2048 processor cores

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 101). 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 102 is a visualization of the scientific results from Figure 99. The scientists are particularly interested in the peaks and boroughs presented in the graph, as those denote potentially interesting areas to explore further.

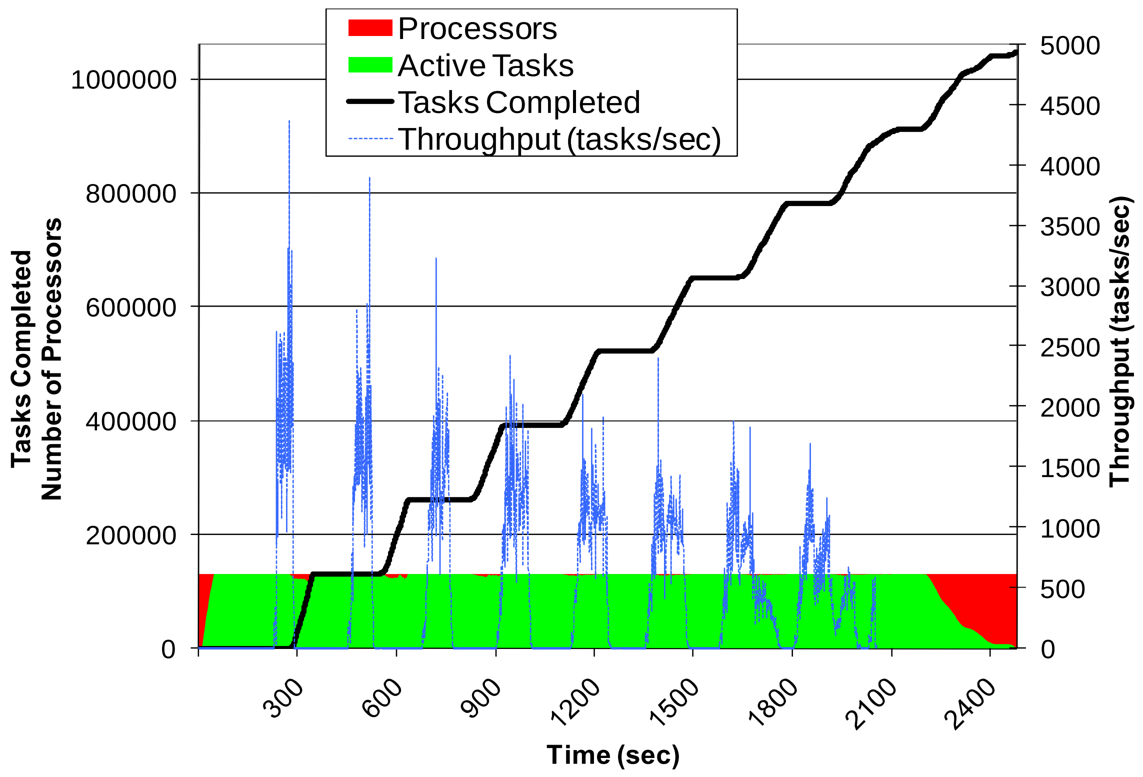


Figure 101: MARS application (summary view) on the Blue Gene/P; 1M tasks using 128K processor cores

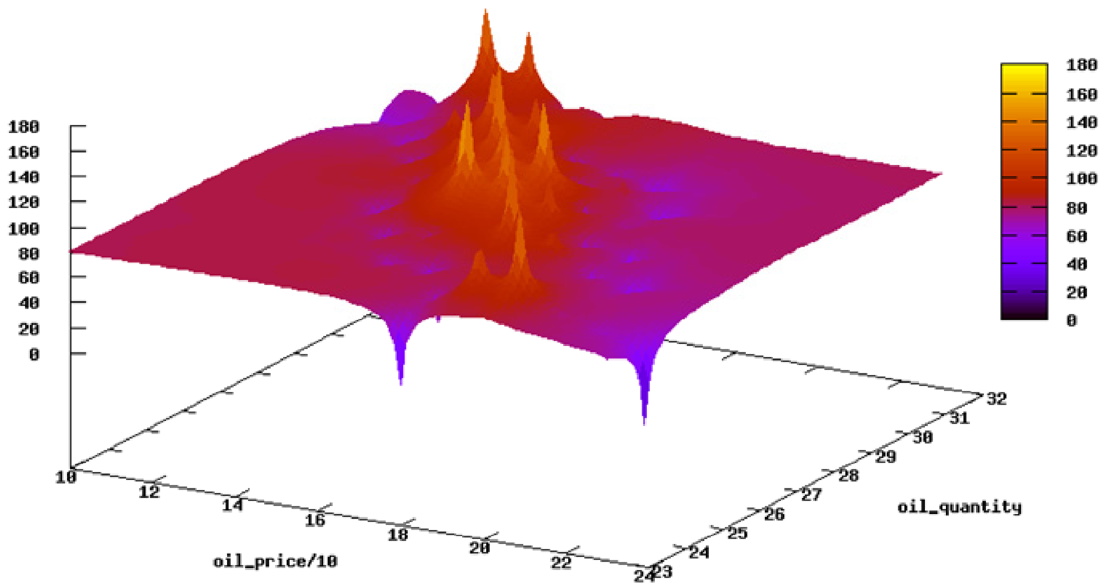


Figure 102: MARS application scientific results visualization

7.7 Large-scale Astronomy Application Performance Evaluation

Section 6.4 and Section 6.5 covered micro-benchmarks and synthetic workloads to show how well data diffusion works, and how it compares to parallel file systems such as GPFS and active storage. This section takes a specific example of a data intensive application, from the astronomy domain, and shows the benefits of data diffusion in both performance and scalability of the application.

The creation of large digital sky surveys presents the astronomy community with tremendous scientific opportunities. However, these astronomy datasets are generally terabytes in size and contain hundreds of millions of objects separated into millions of files—factors that make many analyses impractical to perform on small computers. To address this problem, 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. Building on the Falkon framework, we have built an AstroPortal prototype and implemented the "stacking" analysis that sums multiple regions of the sky, a function that can help both identify variable sources and detect faint objects. We have deployed AstroPortal on the TeraGrid distributed infrastructure and applied the stacking function to the Sloan Digital Sky Survey (SDSS), DR4, which comprises about 300 million objects dispersed over 1.3 million files, a total of 3 terabytes of compressed data, with promising results. AstroPortal gives the astronomy community a new tool to advance their research and to open new doors to opportunities never before possible on such a large scale.

The astronomy community is acquiring an abundance of digital imaging data, via sky surveys such as SDSS [40], GSC-II [147], 2MASS [148], and POSS-II [149]. However, these datasets are generally large (multiple terabytes) and contain many objects (100 million +) separated into many files (1 million +). Thus, while it is by now common for astronomers to use Web Services interfaces to retrieve individual objects, analyses that require access to significant fractions of a sky survey have proved difficult to implement efficiently. There are five reasons why such analyses are challenging: (1) *large dataset size*; (2) *large number of users* (1000s); (3) *large number of resources* needed for adequate performance (potentially 1000s of processors and 100s of TB of disk); (4) *dispersed geographic distribution of the users and resources*; and (5) *resource heterogeneity*.

7.7.1 Definition of "Stacking"

The first analysis that we have implemented in our AstroPortal prototype is "stacking," 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. AstroPortal gives the astronomy community a new tool to advance their research and opens doors to new opportunities.

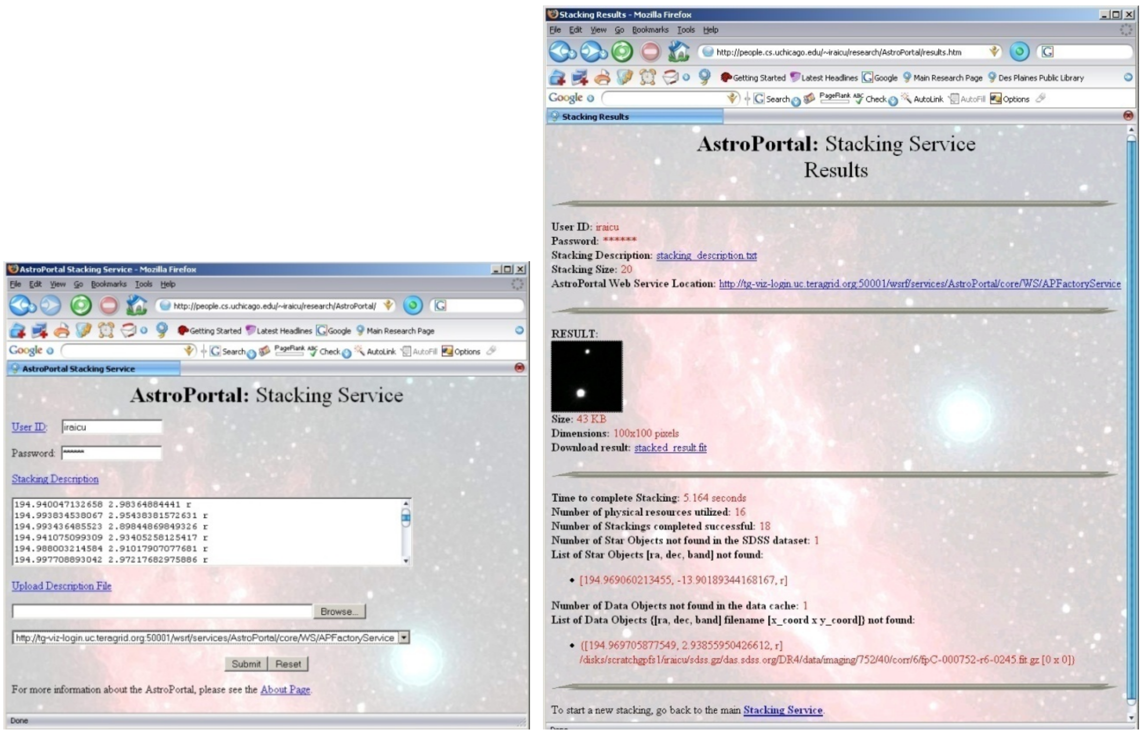
7.7.2 AstroPortal

AstroPortal provides both a Web Services and a Web portal interface. Figure 103 is a screenshot of the AstroPortal Web Portal, which allows a user to request a "stacking" operation on an arbitrary set of objects from the SDSS DR4 dataset. The AstroPortal Web Portal is implemented using Java Servlets and Java Server Pages technologies; we used Tomcat 4.1.31 as the container for our web portal.

User input comprises user ID and password, a stacking description, and the AstroPortal Service location. The user ID and password are currently created out-of-band; in the future, we will investigate alternatives to making this a relatively automated process [150]. The stacking description is a list of objects identified by the tuple {ra dec band}. The AstroPortal Web Service location is currently statically defined in the web portal interface, but in the future we envision a more dynamic discovery mechanism.

Following submission (see Figure 103, left), the user gets a status screen showing the progress of the stacking, including percentage completed and an estimated completion time. Once the stacking is complete, the results are returned along with additional information about performance and any errors encountered. Figure 103 (right) shows an example result from the stacking of 20 objects. The results include summary, results, and statistics and errors. The results displays a JPEG equivalent of the result for quick interpretation, along with the size of the result (in KB), the physical dimensions of the result (in pixels x pixels), and a link to the result in FIT format [151].

The final section specifies the completion time, number of computers used, number of objects found, the number (and address) of star objects not found in the SDSS dataset, and the number (and address) of data objects not found in the data cache. Some star objects might not be found in SDSS since the SDSS dataset does not cover the entire sky; other objects might not be found in the data cache due to inconsistencies (e.g., read permission denied, corrupt data, data cache inaccessible) between the original data archive and the live data cache actually used in the stacking.


AstroPortal Stacking Service - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://people.cs.uchicago.edu/~iraicu/research/AstroPortal/
Getting Started Latest Headlines Google Main Research Page
Search PageRank Check AutoLink AutoFill Options
AstroPortal Stacking Service
Browse...
Submit Reset
Done
Stacking Results - Mozilla Firefox
http://people.cs.uchicago.edu/~iraicu/research/AstroPortal/results.htm
Getting Started Latest Headlines Google Main Research Page Des Plaines Public Library
Stacking Results

AstroPortal: Stacking Service

User ID: iraicu

Password: ******

Stacking Description

```
194.940047132658 2.98364884441 r
194.993834538067 2.95438381572631 r
194.993436485523 2.89844869849326 r
194.941075099309 2.93405258125417 r
194.988003214584 2.91017907077681 r
194.997708893042 2.97217682975886 r
```

Upload Description File

http://tg-viz-login.uc.teragrid.org:50001/wsrf/services/AstroPortal/core/WS/APFactoryService

For more information about the AstroPortal, please see the About Page.

AstroPortal: Stacking Service Results

User ID: iraicu
Password: ******
Stacking Description: stacking_description.txt
Stacking Size: 20
AstroPortal Web Service Location: http://tg-viz-login.uc.teragrid.org:50001/wsrf/services/AstroPortal/core/WS/APFactoryService

RESULT:

Size: 43 KB
Dimensions: 100x100 pixels
Download result: stacked_result.fit

Time to complete Stacking: 5.164 seconds
Number of physical resources utilized: 16
Number of Stackings completed successful: 18
Number of Star Objects not found in the SDSS dataset: 1
List of Star Objects [ra, dec, band] not found:

- [194.969060213455, -13.90189344168167, r]

Number of Data Objects not found in the data cache: 1
List of Data Objects {[ra, dec, band] filename [x_coord x y_coord]} not found:

- {[194.969705877549, 2.93855950426612, r] /disks/scratchgpfs1/iraicu/sdss.gz/das.sdss.org/DR4/data/imaging/752/40/corr/6/fpC-000752-r6-0245.fit.gz [0 x 0]}

To start a new stacking, go back to the main Stacking Service.

Figure 103: Left: AstroPortal Web Portal Stacking Service; Right: Stacking Service Result

7.7.3 Workload Characterization

Astronomical surveys produce terabytes of data, and contain millions of objects. For example, the SDSS DR5 dataset (which we base our experiments on) has 320M objects in 9TB of images [40]. To construct realistic workloads, we identified the interesting objects (for a quasar search) from SDSS DR5; we used the CAS SkyServer [152] to issue the SQL command from Figure 104. This query retrieved 168,529 objects, which after removal of duplicates left 154,345 objects per band (there are 5 bands, u, g, r, I, and z) stored in 111,700 files per band.

```
select SpecRa, SpecDec
from QsoConcordanceAll
where bestMode=1
  and SpecSciencePrimary=1
  and SpecRa<>0
```

Figure 104: SQL command to identify interesting objects for a quasar search from the SDSS DR5 dataset

The entire working set 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 (see Table 11).

Table 11: Workload characteristics

Locality	Number of Objects	Number of Files
1	111700	111700
1.38	154345	111699
2	97999	49000
3	88857	29620
4	76575	19145
5	60590	12120
10	46480	4650
20	40460	2025
30	23695	790

These workloads had 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 30 objects on average of).

7.7.4 Stacking Code Profiling

We first profile the stacking code to see where time is spent. We partition time into four categories, as follows.

open: open Fits file for reading

radec2xy: convert coordinates from RA DEC to X Y

readHDU: reads header and image data

getTile: perform extraction of ROI from memory

curl: convert the 1-D pixel data (as read from the image file) into a 2-dimensional pixel array

convertArray: convert the ROI from having SHORT value to having DOUBLE values

calibration: apply calibration on ROI using the SKY and CAL variables

interpolation: do the appropriate pixel shifting to ensure the center of the object is a whole pixel

doStacking: perform the stacking of ROI that are stored in memory

writeStacking: write the stacked image to a file

To simplify experiments, we perform tests with a simple standalone program on 1000 objects of 100x100 pixels, and repeat each measurement 10 times, each time on different objects residing in different files. In Figure 105, the Y-axis is time per task per code block measured in milliseconds (ms).

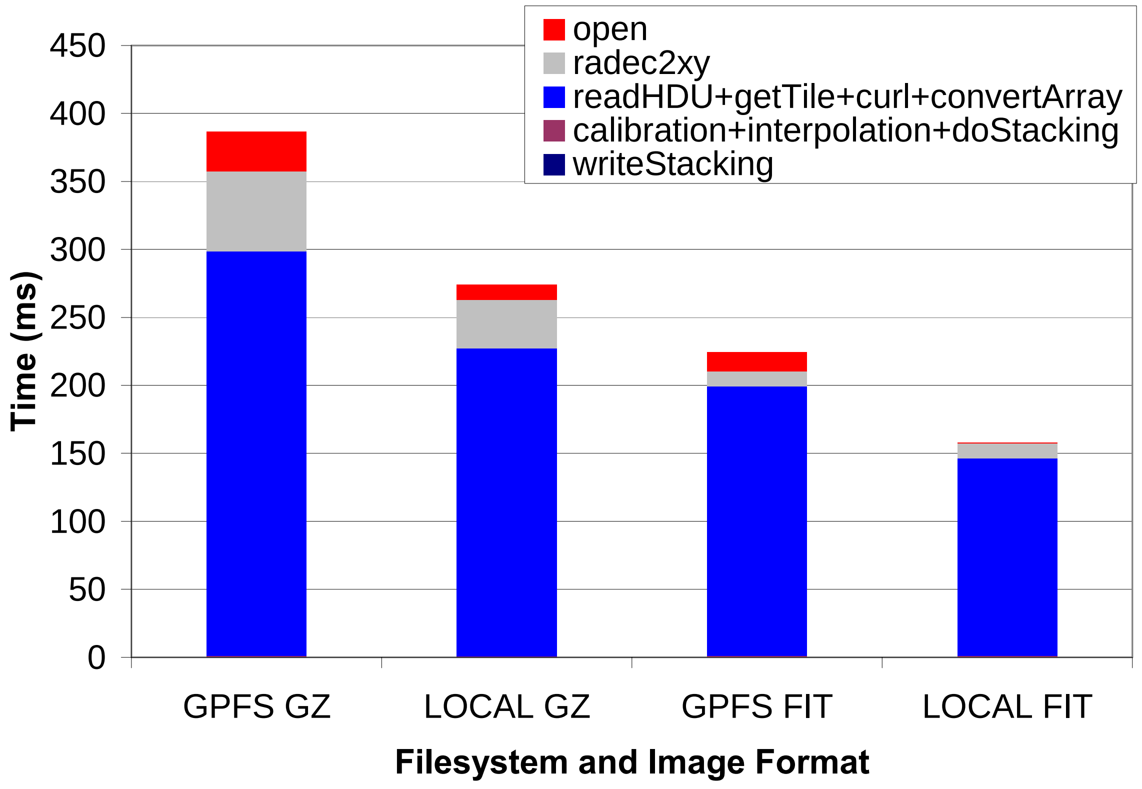


Figure 105: Stacking code performance profiling for 1 CPU

Having the image data in compressed format affects the time to stack an image significantly, increasing the time needed by a factor of two. Similarly, accessing the image data from local disk instead of the shared file system speeds up processing 1.5 times. In all cases, the dominant operations are file metadata and I/O operations. For example, calibration, interpolation, and doStacking take less than 1 ms in all cases. Radec2xy consumes another 10~20% of total time, but the rest is spent opening the file and reading the image data to memory. In compressed format (GZ), there is only 2MB of data to read, while in uncompressed format (FIT) there are 6MB to read. However, uncompressing images is CPU intensive, and in the case of a single CPU, it is slower than if the image was uncompressed. In the case of many CPUs, the compressed format is faster mostly due to limitations imposed by the shared file system. Overall, Figure 105 shows the stacking analysis to be I/O bound and data intensive.

7.7.5 Performance Evaluation

All tests performed in this section were done using the testbed described in Table 8, using from 1 to 64 nodes, and the workloads (described in Table 8) that had locality ranging from 1 to 30. 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). In the data diffusion experiments, we use the MCU policy and cache 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.

Figure 106 shows the performance difference between data diffusion and GPFS when data locality is small (1.38). We normalize the results here by showing the time per stacking operation per processor used; with perfect scalability, the time per stack should remain constant as we increase the number of processors. We see that data diffusion and GPFS perform quite similarly when locality is low, with data diffusion slightly faster; data diffusion has a growing advantage as the number of processors increases. This similarity in performance is not surprising because most of the data must still be read from GPFS to populate the local disk caches. Note that in with small number of processors, it is more efficient to access uncompressed data; however, as the number of processors increases, compressed data becomes preferable. A close inspection of the I/O throughput achieved reveals that GPFS becomes saturated at around 16 CPUs with 3.4Gb/s read rates. In the compressed format (which reduces the amount of data that needs to be transferred from GPFS by a factor of three), GPFS only becomes saturated at 128 CPUs. We also find that when working in the compressed format, it is faster (as much 32% less per stack time) to first cache the compressed files, uncompress the files, and work on the files in uncompressed format, as opposed to working directly on the uncompressed files from GPFS.

While the previous results from Figure 106 shows an almost worst case scenario where the data locality is small (1.38), the next set of results (Figure 107) shows a best case scenario in which the locality is high (30). Here we see an almost ideal speedup (i.e., a flat line) with data diffusion in both compressed and uncompressed formats, while the GPFS results remain similar to those presented in Figure 106.

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 (see Figure 108). The ideal cache hit ratio is computed by 1 – 1/locality; for example, with locality 3 (meaning that each file is access 3 times, one cache miss, and 2 cache hits), the ideal cache hit ratio is 1 – 1/3 = 2/3.

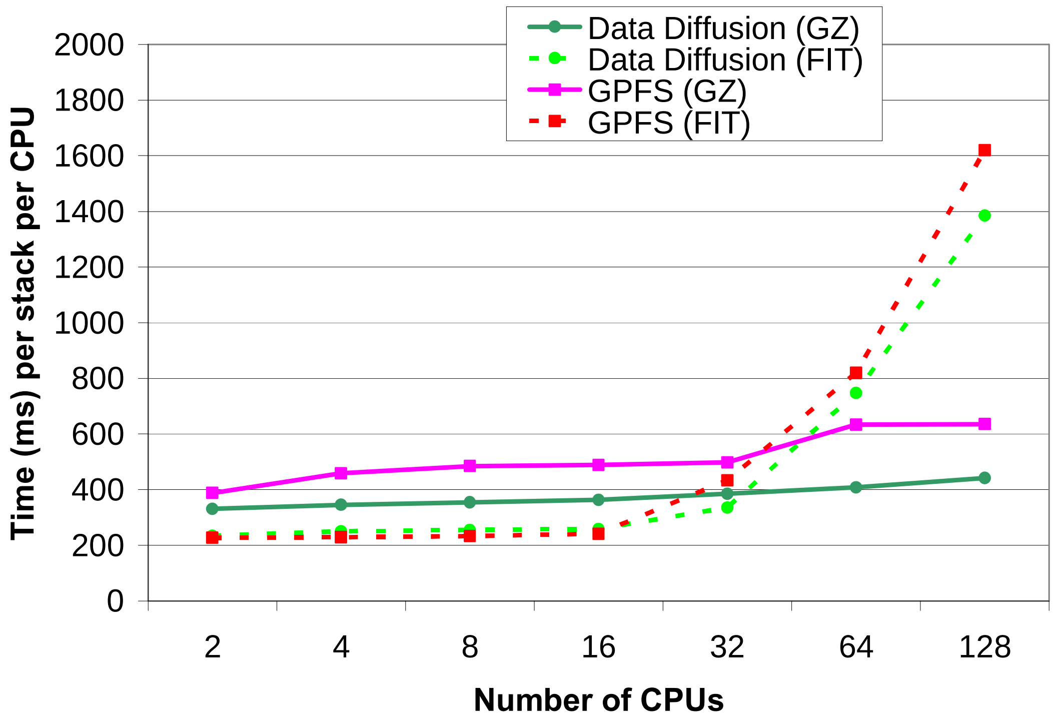


Figure 106: Performance of the stacking application for a workload data locality of 1.38 using data diffusion and GPFS while varying the CPUs from 2 to 128

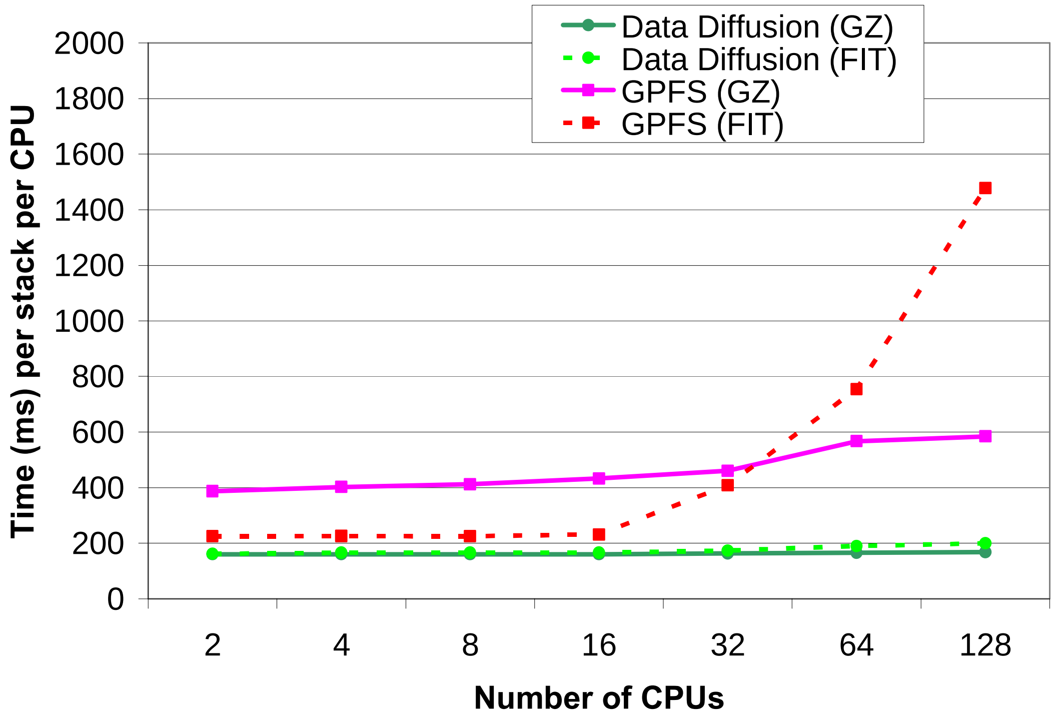


Figure 107: Performance of the stacking application for a workload data locality of 30 using data diffusion and GPFS while varying the CPUs from 2 to 128

The following experiment (Figure 109) 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.

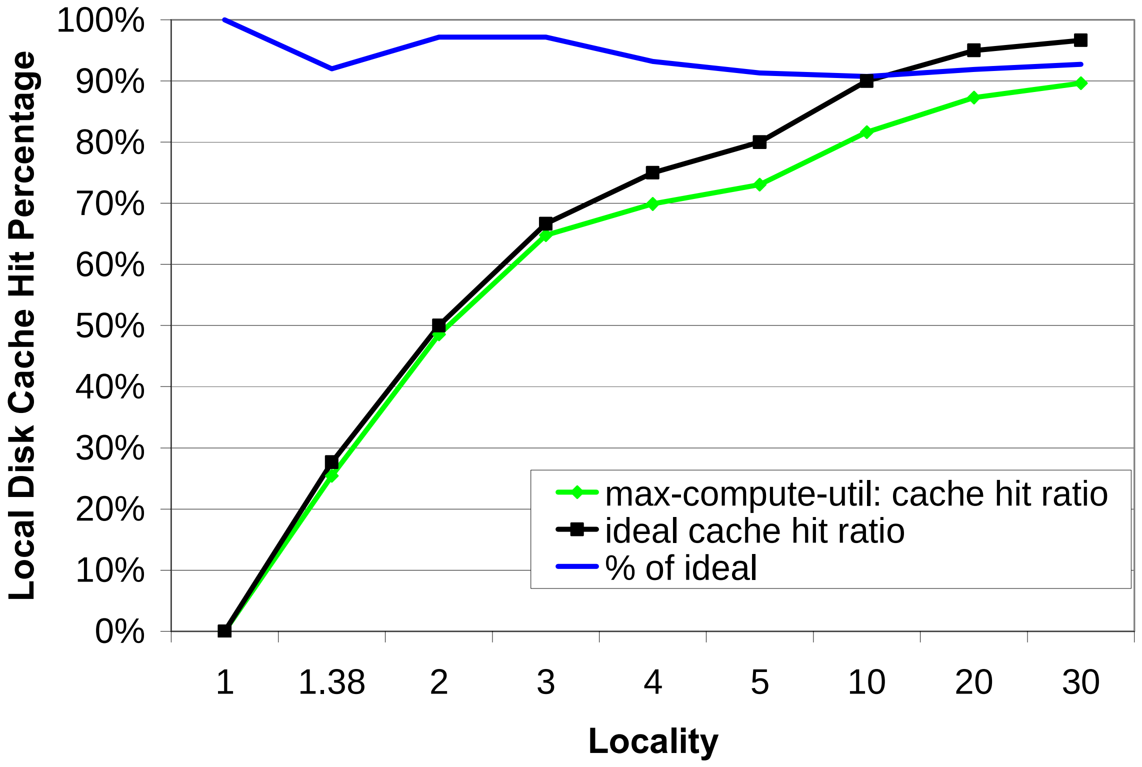


Figure 108: Cache hit performance of the data-aware scheduler for the stacking application using 128 CPUs for workloads ranging from 1 to 30 data locality using data diffusion

Figure 110 shows aggregate I/O throughput and data movement for the experiments of Figure 109. The two dotted lines show I/O throughput when performing stacking directly against GPFS: we achieve 4Gb/s with a data locality of 30. The data diffusion I/O throughput is separated into three distinct parts: 1) local, 2) cache-to-cache, and 3)

GPFS, as a stacking may read directly from local disk if data is cached on the executor node, from a remote cache if data is on other nodes, and from GPFS as some data may not have been cached at all.

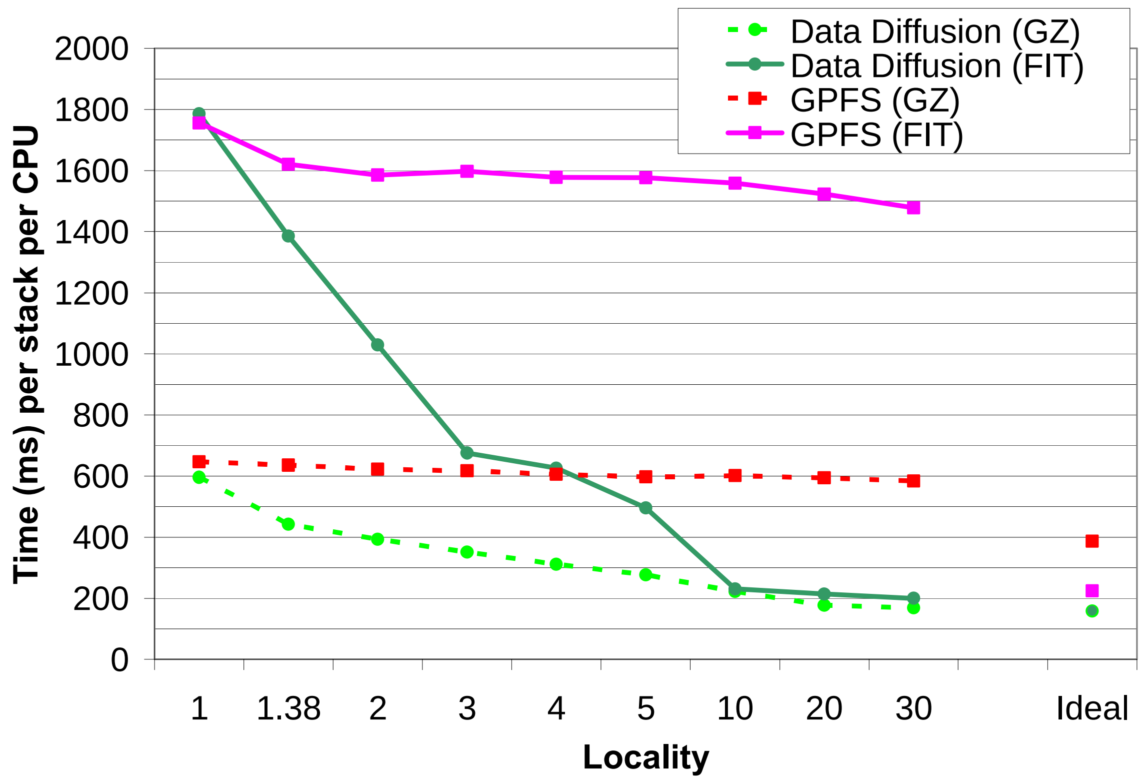


Figure 109: Performance of the stacking application using 128 CPUs for workloads with data locality ranging from 1 to 30, using data diffusion and GPFS

GPFS throughput is highest with low locality and lowest with high locality; the intuition is that with low locality, the majority of the data must be read from GPFS, but with high locality, the data can be mostly read locally. Note that cache-to-cache throughput increases with locality, but never grows significantly; we attribute this result to the good performance of the data-aware scheduler, always gets within 90% of the ideal cache hit ratio (for the workloads presented in this sub-section). 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. Finally, Figure 111 investigates the amount of data movement that occurs per stacking as we vary data locality.

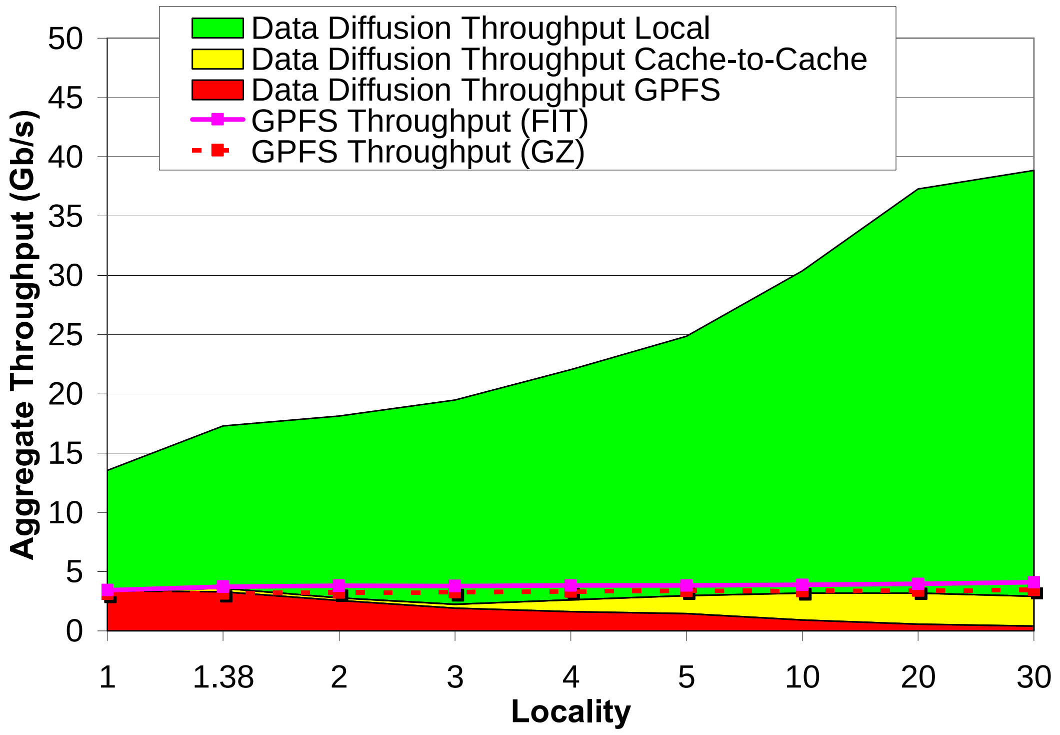


Figure 110: I/O throughput of image stacking using 128 CPUs, for workloads with data locality ranging from 1 to 30, and using both data diffusion and GPFS

In summary, data diffusion (using compressed data) transfers a total of 8MB (2MB from GPFS and 6MB from local disk) for a data locality of 1; if data diffusion is not used, we need 2MB if in compressed format, or 6MB in uncompressed format, but this data must come from GPFS. As data locality increases, data movement from GPFS does not change (given a large number of processors and the small probability of data being re-used without data-aware scheduling). However, with data diffusion, the amount of data movement decreases substantially from GPFS (from 2MB with a locality of 1 to 0.066MB with a locality of 30), while cache-to-cache increases from 0 to 0.421MB per stacking respectively. These results show the decreased load on shared infrastructure (i.e., GPFS), which ultimately allows data diffusion to scale better.

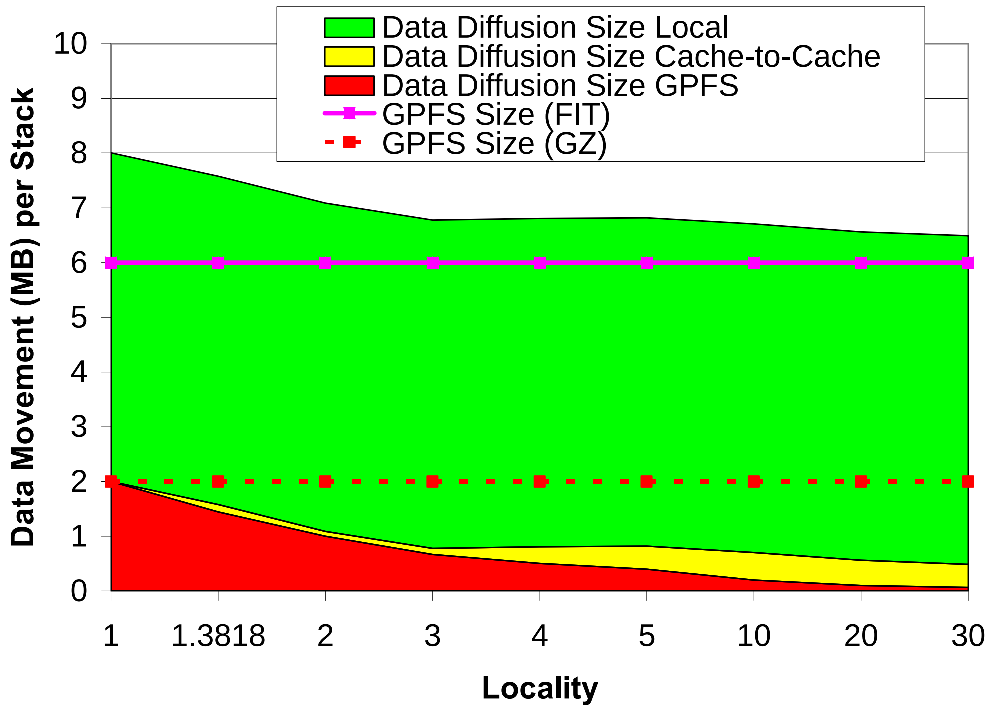


Figure 111: Data movement for the stacking application using 128 CPUs, for workloads with data locality ranging from 1 to 30, using data diffusion and GPFS

7.7.6 Abstract Model Validation

We perform a preliminary validation of our abstract model (presented in Section 6.3.1) with results from a real large-scale astronomy application [12]. We compared the model expected time $T_N(D)$ to complete workload D (with varying data locality and access patterns) on the measured completion time for N equal from 2 to 128 processors incrementing in powers of 2. For 92 experiments [1], we found an average (and median) model error 5%, with a standard deviation of 5% and 29% worst case.

Figure 112 and Figure 113 shows the details of the model error under the various experiments, presented in Section 7.7.5. These experiments were from the stacking service and had a working set of 558,500 files (1.1TB compressed and 3.35TB uncompressed). From this working set, various workloads were defined that had 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 30 objects). Experiments marked with FIT represents ones performed on uncompressed image data, GZ represents experiments on compressed image data, GPFS represents experiments ran accessing data directly on the parallel file system, and data diffusion are experiments using the data-aware scheduler. Figure 112 shows the model error for experiments that varied the number of CPUs from 2 to 128 with locality of 1, 1.38, and 30. Note that each model error point represents a workload that spanned 111K, 154K, and 23K tasks for data locality 1, 1.38, and 30 respectively.

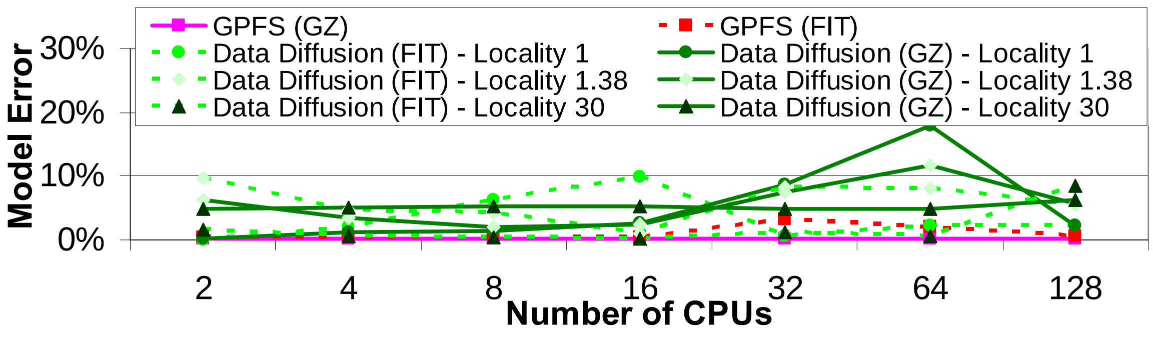


Figure 112: Model error for varying number of processors

Figure 113 shows the model error with a fixed the number of processors (128), and varied the data locality from 1 to 30. The results show a larger model error with an average of 8% and a standard deviation of 5%. We attribute the model errors to contention in the parallel file system and network resources that are only captured simplistically in the current model, and due to not having dedicated access to the 316-CPU cluster.

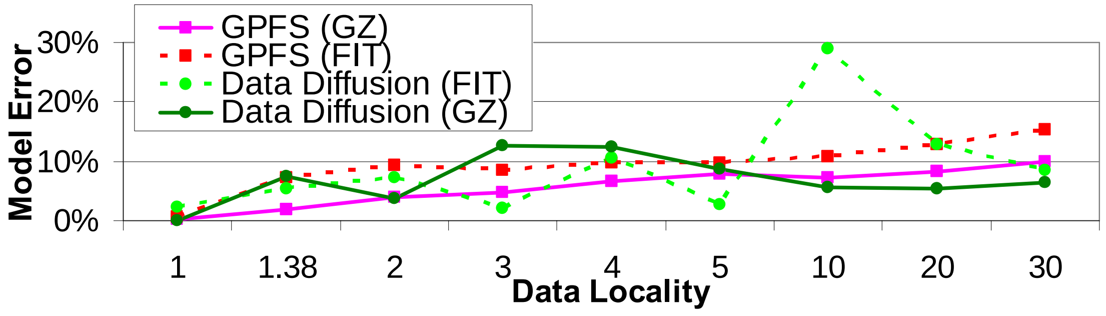


Figure 113: Model error for varying data-locality

Overall, the model seems to be a good fit for this particular astronomy application at modest scales of up to 128 processors. We did not investigate the model's accuracy under varying arrival rates, nor did we investigate the model under other applications. We plan to further the model analysis in future work, by implementing the model in a simple simulator to allow more dynamic scenarios, such as the ones found in Section 6.5.1 and Section 6.5.2.

7.8 Montage (Astronomy Domain)

The Montage workflow demonstrated similar job execution time pattern as there were many small jobs involved. We show in Figure 114 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. [153] 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.

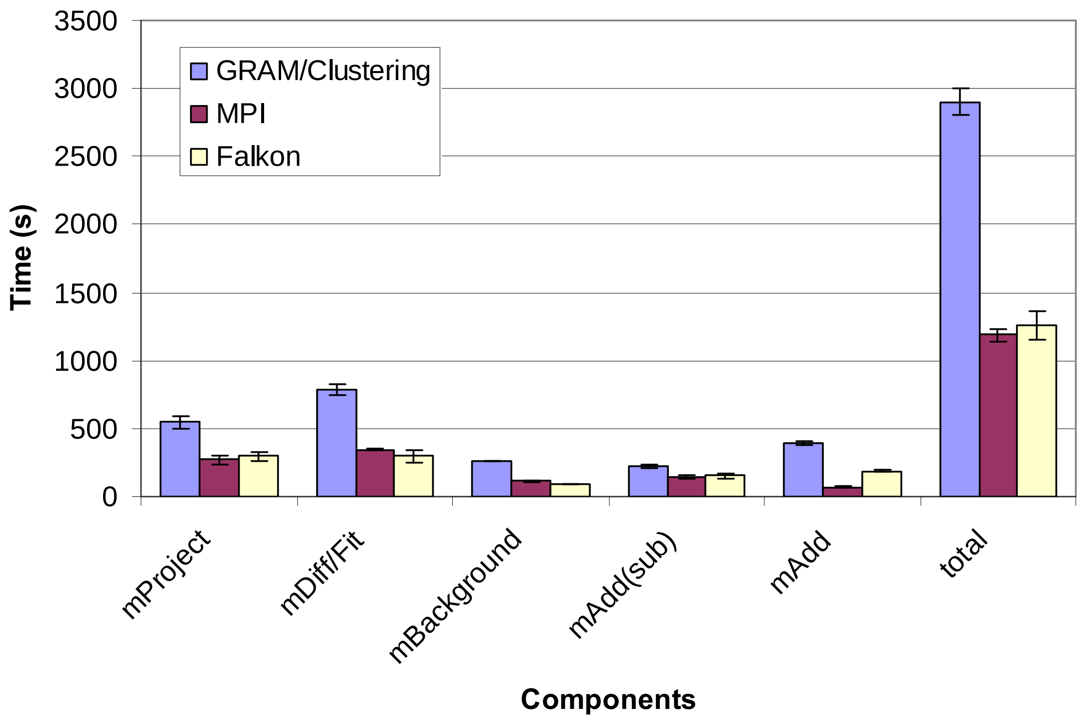


Figure 114: Execution Time for the Montage Workflow

7.9 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 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, such as distributed indexing, sorting, and clustering.

Swift is a parallel programming tool for rapid and reliable specification, execution, and management of large-scale science and engineering workflows. Swift consists of a concise scripting language called SwiftScript for specifications of complex parallel computations based on dataset typing and iterations, and dynamic dataset mappings. The runtime system relies on CoG Karajan workflow engine for scheduling and load balancing, and integrates the Falkon light-weight task execution service for optimized task throughput, resource provisioning, and to leverage data-locality found in application access patterns.

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. 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 (see Figure 115). However, for WordCount, Swift execution times were on average 75% lower (see Figure 116).

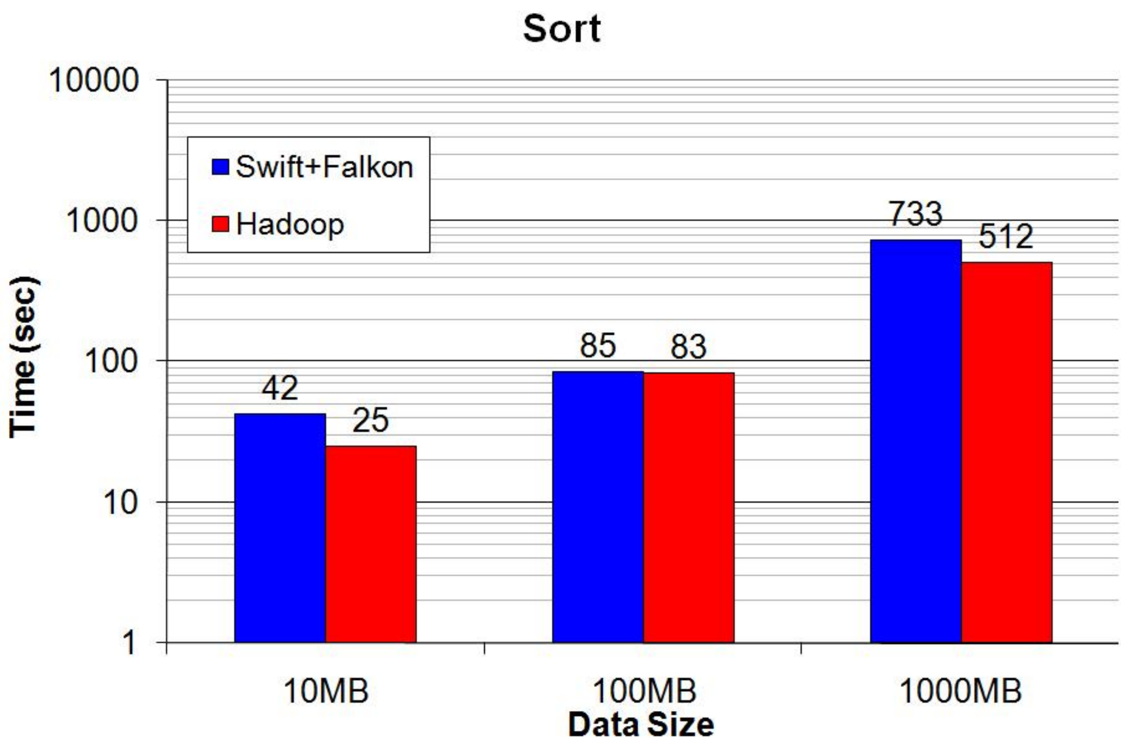


Figure 115: Swift vs. Hadoop, sort

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. We plan to do additional experiments on the TeraGrid, with larger testbeds and data sets, as well as additional benchmarks.

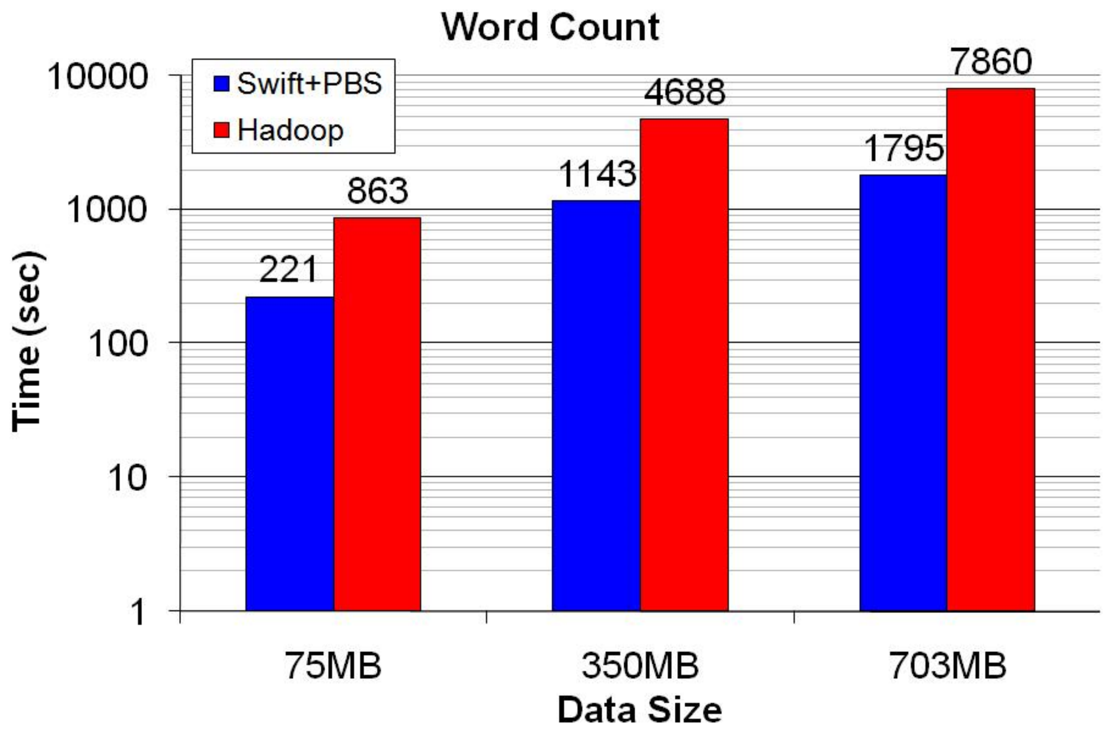


Figure 116: Swift vs. Hadoop, word count

We conclude with three questions. 1) Can MapReduce applications run on workflow systems? We believe yes, and with even better performance in some cases. 2) Is the MapReduce model an option for scientific applications? 3) What parallel programming model will be best suited for scientific applications in the coming decade? We hope future work will help answer question (2) and (3).

8 Contributions, Conclusions, and Future Research Directions

My dissertation work has focused on resource management in distributed systems. My work has enabled a new paradigm called Many-Task Computing (MTC) to operate at previously believed impossible scales with high efficiency. My interdisciplinary research has showcased how novel resource management techniques allow specific scientific applications to be solved at a larger scale, faster, and more efficient. This chapter re-iterates this dissertation's contributions, covers conclusions we have drawn, and discuss future research directions.

8.1 Dissertation Contributions

This dissertation is a fusion of many publications [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23].

We have defined a new paradigm – MTC – which aims to bridge the gap between two computing paradigms, HTC and HPC. MTC applications are typically loosely coupled that are communication-intensive but not naturally expressed using standard message passing interface commonly found in high performance computing, drawing attention to the many computations that are heterogeneous but not “happily” parallel. We believe that today’s existing HPC systems are a viable platform to host MTC applications. We also believe MTC is a broader definition than HTC, allowing for finer grained tasks, independent tasks as well as ones with dependencies, and allowing tightly coupled applications and loosely coupled applications to co-exist on the same system.

Furthermore, having native support for data intensive applications is central to MTC, as there is a growing gap between storage performance of parallel file systems and the amount of processing power. As the size of scientific data sets and the resources required for analysis increase, data locality becomes crucial to the efficient use of large scale distributed systems for scientific and data-intensive applications [19]. We believe it is feasible to allocate large-scale computational resources and caching storage resources that are relatively remote from the original data location, co-scheduled together to optimize the performance of entire data analysis workloads which are composed of many loosely coupled tasks.

We identified challenges in running these novel workloads on large-scale systems, which can hamper both efficiency and utilization. These challenges include local resource manager scalability and granularity, efficient utilization of the raw hardware, shared file system contention and scalability, reliability at scale, application scalability, and understanding the limitations of HPC systems in order to identify promising and scientifically valuable MTC applications.

We have investigated many aspects of MTC, by exploring the upper limits on the number of tasks, number of processors, and the I/O per processor particular systems could maintain. In exploring these limits, I investigated three techniques: (1) multi-level scheduling to enable dynamic resource provisioning [4, 6]; (2) streamlined task dispatching [2, 4]; and (3) data diffusion [1, 3, 14, 21].

The “data diffusion” approach acquires compute and storage resources dynamically, replicates data in response to demand, and schedules computations close to data. As demand increases, more resources are acquired, thus allowing faster response to subsequent requests that refer to the same data; when demand drops, resources are released. This approach can provide the benefits of dedicated hardware without the associated high costs, depending on workload and resource characteristics. To explore the feasibility of data diffusion, we have completed both a theoretical and empirical analysis. We defined an abstract model for data diffusion and provided a preliminary validation using a real-world large-scale astronomy application. We defined scheduling policies with heuristics to optimize real world performance, and developed a competitive online caching eviction policy. We also offered many empirical experiments to explore the benefits of dynamically expanding and contracting resources based on load, to improve system responsiveness while keeping wasted resources small.

The concepts of dynamic resource provisioning, data diffusion, and streamlined dispatching have been realized in Falkon, a Fast and Light-weight tasK executiON framework. Falkon has shown orders of magnitude improvements in performance and scalability across many diverse workloads (heterogeneous tasks from 100ms to hours long, compute intensive, data intensive, varying arrival rates) and applications (e.g. astronomy, medicine, chemistry, molecular dynamics, economic modeling, and data analytics) at scales of up to billions of tasks on up to hundreds of thousands of processors across clusters, Grids (e.g. TeraGrid), and supercomputers (e.g. IBM Blue Gene/P). 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. Falkon is an open source Globus Incubator Project [154], and can be freely downloaded from online [155].

To extend the discussion with the broader community on many-task computing, Ian Foster, Yong Zhao, and I held a workshop at IEEE/ACM Supercomputing 2008 titled "IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08)" [24]. Another related event was a bird-of-feather (BOF) session titled "Megajobs08: How to Run a Million Jobs" [25], co-organized with Marlon Pierce, Ruth Pordes, John McGee, and Dick Repasky. The half-day workshop was a success; it attracted over 100 participants who attended the presentations of the six accepted papers [17, 156, 157, 158, 159, 160] and a keynote talk by the renowned Dr. Alan Gara (IBM Blue Gene Chief Architect). The BOF was also a success, having seven short presentations and attracting over 60 participants. We plan to have follow-up workshops and special issue publications on many-task computing in the near future.

8.2 Conclusions

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

Furthermore, the rate of increase in the number of processors per system is outgrowing the rate of performance increase of parallel file systems, which requires rethinking existing data management techniques. For example, there is a 65X reduction in per core bandwidth between a system from 2002 and one from 2008. Unfortunately, this trend is not bound to stop, as advances multi-core and many-core processors will increase the number of processor cores one to two orders of magnitude over the next decade.

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.

8.3 Future Research Directions

My future work centers on resource management in large scale distributed systems, covering: 1) *Many-Task Computing*, 2) *Data Intensive Computing,* 3) *Cloud Computing & Grid Computing*, and 4) *Many-Core Computing*.

I have defined a new paradigm Many-Tasks Computing (MTC) [20] which aims to bridge the gap between high throughput computing (HTC) and high performance computing (HPC). MTC is reminiscent to HTC, but it differs in the emphasis of using many computing resources over short periods of time to accomplish many computational tasks (i.e. including both dependent and independent tasks), where the primary metrics are measured in seconds, as opposed to operations per month. MTC denotes high-performance computations comprising multiple distinct activities, coupled via file system operations or message passing. There are many challenges to enable support for MTC across clusters, Grids, and supercomputers, including scalable resource management and storage solutions, as well as having well defined standards on how applications are to interact with the new or improved middleware. I have already started a dialogue between Alan Gara (IBM Blue Gene chief architect), Ian Foster (father of the Grid), and Pete Beckman (director of ANL ALCF BG/P) about how our current research work on MTC can be applied to the next generation BG/Q in the 2010-2012 time frame.

The support for Data Intensive Computing is critical to advancing modern science as storage systems have experienced an increasing gap between its capacity and its bandwidth by more than 10-fold over the last decade. There is an emerging need for advanced techniques to manipulate, visualize and interpret large datasets. Many domains share these data management challenges, strengthening the potential road impact from a generic solution. My work in identifying the importance of data locality and results obtained in data management, collective I/O primitives, and data-aware scheduling are fundamental to future storage systems and tomorrow’s exascale systems. I plan to pursue improvements to parallel file systems (e.g. GPFS, Lustre, PVFS) to support future data intensive applications by exposing data locality information and allowing schedulers to capitalize on it with data-aware scheduling.

The Cloud Computing concept surfaced in 2007, although it is not a completely new concept; it has intricate connection to the thirteen-year established Grid Computing paradigm, and other relevant technologies such as utility computing, cluster computing, and distributed systems in general. In building the future Cloud Computing infrastructure, I believe it needs to support on-demand provisioning of "virtual systems" providing the precise capabilities needed by an end-user. There is a growing demand to define protocols that allow users and service providers to discover and hand off demands to other providers, to monitor and manage their reservations, and arrange payment. Tools need to be defined and implemented for managing both the underlying resources and the resulting distributed computations. I plan to pursue these ideas within both Clouds and Grids.

With the advent of Many-Core Computing, some are predicting that desktop machines will reach 1000s of cores within a decade. This increase in parallelism will bring many challenges to end-users, with new programming models and new thinking methods that until recently were reserved for the select few that were involved in large scale science on distributed systems. Having new tools (e.g. parallel programming languages) to harness these massively parallel machines by non-experts will be critical to allow maximal impact of the many-core computing era.

My research philosophy pivots on the importance of *computational science* to the many branches of science, and how interdisciplinary research can bridge the gap among them. Computational science has already begun to change how science is done, enabling scientific breakthroughs through new kinds of experiments that would have been impossible only a decade ago. Today's science is generating datasets that are increasing exponentially in both complexity and volume, making their analysis, archival, and sharing one of the grand challenges of the 21st century.

I believe that the intersection of computer science and the sciences has the potential to have a profound impact on science, how it is practiced, and the rate at which major advancements are achieved. Computational science represents the foundation of a new revolution in science that is just beginning, and will re-energize virtually all disciplines over the next decades. Through the spread of computational science, it will enable new kinds of science and a new era of science-based innovation that could dwarf the last decades of technology-based innovation.

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. Raicu, Y. Zhao, I. Foster. “IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08) 2008”, co-located with IEEE/ACM Supercomputing 2008, http://dsl.cs.uchicago.edu/MTAGS08/, 2008

[25] M. Pierce, I. Raicu, R. Pordes, J. McGee, D. Repasky. “How to Run One Million Jobs (Megajobs08)” Bird-of-Feather Session at IEEE/ACM Supercomputing 2008, http://gridfarm007.ucs.indiana.edu/megajobBOF/index.php/Main_Page, 2008

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

[27] IBM BlueGene/P (BG/P), http://www.research.ibm.com/bluegene/, 2008

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

[29] W. Gropp, E. Lusk, N. Doss, A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing, 22:789-828, 1996

[30] J. Dean, S. Ghemawat. “MapReduce: Simplified data processing on large clusters.” USENIX OSDI04, 2004

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

[32] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly. “Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks,” European Conference on Computer Systems (EuroSys), 2007

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

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

[35] M. Flynn. “Some Computer Organizations and Their Effectiveness”, IEEE Trans. Comput. C-21, pp. 948, 1972

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

[37] J. Appavoo, V. Uhlig, A. Waterland. “Project Kittyhawk: Building a Global-Scale Computer,” ACM Sigops Operating System Review, 2008

[38] Top500, June 2008, http://www.top500.org/lists/2008/06, 2008

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

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

[41] CERN’s Large Hadron Collider, http://lhc.web.cern.ch/lhc, 2008

[42] GenBank, http://www.psc.edu/general/software/packages/genbank, 2008

[43] European Molecular Biology Laboratory, http://www.embl.org, 2008

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

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

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

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

[48] 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

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

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

[51] J. Cope, M. Oberg, H.M. Tufo, T. Voran, M. Woitaszek. "High Throughput Grid Computing with an IBM Blue Gene/L," Cluster 2007

[52] 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

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

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

[55] "Swift Workflow System": www.ci.uchicago.edu/swift, 2008

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

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

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

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

[60] T. Stef-Praun, B. Clifford, I. Foster, U. Hasson, M. Hategan, S. Small, M. Wilde and Y. Zhao. "Accelerating Medical Research using the Swift Workflow System", Health Grid , 2007

[61] 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

[62] 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

[63] T. Stef-Praun, G. Madeira, I. Foster, R. Townsend. “Accelerating solution of a moral hazard problem with Swift”, e-Social Science, 2007

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

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

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

[67] US National Virtual Observatory (NVO), http://www.us-vo.org/index.cfm, 2008

[68] ASC / Alliances Center for Astrophysical Thermonuclear Flashes, http://www.flash.uchicago.edu/website/home/, 2008

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

[70] PL Protein Library, http://protlib.uchicago.edu/, 2008

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

[72] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, D.J. Lipman. "Basic local alignment search tool". J Mol Biol 215 (3): 403–410, 1990

[73] Computational Neuroscience Applications Research Infrastructure, http://www.ci.uchicago.edu/wiki/bin/view/CNARI/WebHome, 2008

[74] D. Bernholdt, S. Bharathi, D. Brown, K. Chanchio, M. Chen, A. Chervenak, L. Cinquini, B. Drach, I. Foster, P. Fox, J. Garcia, C. Kesselman, R. Markel, D. Middleton, V. Nefedova, L. Pouchard, A. Shoshani, A. Sim, G. Strand, and D. Williams, “The Earth System Grid: Supporting the Next Generation of Climate Modeling Research”, Proceedings of the IEEE, 93 (3), p 485-495, 2005

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

[76] D. Bader, R. Pennington. "Cluster Computing: Applications". Georgia Tech College of Computing, June 1996

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

[78] F. Gagliardi, The EGEE European Grid Infrastructure Project, LNCS, Volume 3402/2005, p. 194-203, 2005

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

[80] D. W. Erwin and D. F. Snelling, “UNICORE: A Grid Computing Environment”, Euro-Par 2001, LNCS Volume 2150/2001: p. 825-834, 2001

[81] P. Helland, Microsoft, “The Irresistible Forces Meet the Movable Objects”, November 9th, 2007

[82] “ZeptoOS: The Small Linux for Big Computers”, http://www-unix.mcs.anl.gov/zeptoos/, 2008

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

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

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

[86] F.J.L. Reid, “Task farming on Blue Gene”, EEPC, Edinburgh University, 2006

[87] H. Casanova, M. Kim, J.S. Plank, J.J. Dongarra, “Adaptive Scheduling for Task Farming with Grid Middleware”, International Euro-Par Conference, 1999

[88] M. Danelutto, “Adaptive Task Farm Implementation Strategies”, Euromicro Conference on Parallel, Distributed and Network-based Processing, pp. 416-423, IEEE press, ISBN 0-7695-2083-9, 2004, A Coruna (E), 11-13 February 2004

[89] H. González-Vélez, “An Adaptive Skeletal Task Farm for Grids”, International Euro-Par 2005 Parallel Processing, pp. 401-410, 2005

[90] E. Heymann, M.A. Senar, E. Luque, and M. Livny, “Adaptive Scheduling for Master-Worker Applications on the Computational Grid”, IEEE/ACM International Workshop on Grid Computing (GRID00), 2000

[91] D. Petrou, G.A. Gibson, G.R. Ganger, “Scheduling Speculative Tasks in a Compute Farm”, IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC05), 2005

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

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

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

[95] 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

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

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

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

[99] L. Ramakrishnan, L. Grit, A. Iamnitchi, D. Irwin, A. Yumerefendi, J. Chase. “Toward a Doctrine of Containment: Grid Hosting with Adaptive Resource Control,” IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC06), 2006

[100] A.L. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, R. Schwartzkopf, “The Replica Location Service”, International Symposium on High Performance Distributed Computing Conference (HPDC-13), June 2004

[101] A. Chervenak, R. Schuler. “The Data Replication Service”, Technical Report, USC ISI, 2006.

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

[103] T. Kosar. “A New Paradigm in Data Intensive Computing: Stork and the Data-Aware Schedulers”, IEEE CLADE 2006

[104] O. Tatebe, N. Soda, Y. Morita, S. Matsuoka, S. Sekiguchi, “Gfarm v2: A Grid file system that supports high-performance distributed and parallel data computing”, Computing in High Energy and Nuclear Physics (CHEP04), 2004.

[105] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, S. Sekiguchi, “Grid Datafarm Architecture for Petascale Data Intensive Computing”, IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002), pp.102-110, 2002

[106] W. Xiaohui, W.W. Li, O. Tatebe, X. Gaochao, H. Liang, J. Jiubin. “Implementing data aware scheduling in Gfarm using LSF scheduler plugin mechanism”, International Conference on Grid Computing and Applications (GCA'05), pp.3-10, 2005

[107] X. Wei, W.W. Li, O. Tatebe, G. Xu, L. Hu, and J. Ju. “Integrating Local Job Scheduler – LSF with Gfarm”, Parallel and Distributed Processing and Applications, Springer Berlin, Vol. 3758/2005, pp 196-204, 2005

[108] D. Culler, et al. “Parallel computing on the berkeley now”, Symposium on Parallel Processing, 1997

[109] R. Arpaci-Dusseau. “Run-time adaptation in river”, ACM Transactions on Computer Systems, 21(1):36–86, 2003

[110] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. “Bigtable: A Distributed Storage System for Structured Data”, Symposium on Operating System Design and Implementation (OSDI'06), 2006

[111] S. Ghemawat, H. Gobioff, S.T. Leung. “The Google file system,” 19th ACM SOSP, 2003

[112] R.L Grossman, Y. Gu. “Data Mining Using High Performance Clouds: Experimental Studies Using Sector and Sphere”, Proceedings of The 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2008), 2008

[113] Y. Gu, R.L. Grossman, A. Szalay, A. Thakar. “Distributing the sloan digital sky survey using udt and sector”. In Proceedings of e-Science 2006, 2006

[114] D.L. Adams, K. Harrison, C.L. Tan. “DIAL: Distributed Interactive Analysis of Large Datasets”, Conference for Computing in High Energy and Nuclear Physics (CHEP 06), 2006

[115] A. Chervenak, at al. “High-Performance Remote Access to Climate Simulation Data: A Challenge Problem for Data Grid Technologies”, Parallel Computing, Special issue: High performance computing with geographical data, Volume 29 , Issue 10 , pp 1335 – 1356, 2003

[116] M. Beynon, T.M. Kurc, U.V. Catalyurek, C. Chang, A. Sussman, J.H. Saltz. “Distributed Processing of Very Large Datasets with DataCutter”, Parallel Computing, Vol. 27, No. 11, pp. 1457-1478, 2001

[117] D.T. Liu, M.J. Franklin. “The Design of GridDB: A Data-Centric Overlay for the Scientific Grid”, VLDB04, pp. 600-611, 2004

[118] D. Olson and J. Perl, “Grid Service Requirements for Interactive Analysis”, PPDG CS11 Report, September 2002

[119] S. Langella, S. Hastings, S. Oster, T. Kurc, U. Catalyurek, J. Saltz. “A Distributed Data Management Middleware for Data-Driven Application Systems” IEEE International Conference on Cluster Computing, 2004.

[120] M. Branco, “DonQuijote - Data Management for the ATLAS Automatic Production System”, Computing in High Energy and Nuclear Physics (CHEP04), 2004.

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

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

[123] I. Foster, C. Kesselman, S. Tuecke, “The Anatomy of the Grid”, International Journal of Supercomputer Applications, 15 (3). 200-222, 2001

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

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

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

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

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

[129] T. Tannenbaum. “Condor RoadMap”, Condor Week 2007

[130] J.E. Moreira, et al. “Blue Gene/L programming and operating environment”, IBM Journal of Research and Development, Volume 49, Number 2/3, 2005

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

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

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

[134] R. Lancellotti, M. Colajanni, B. Ciciani, “A Scalable Architecture for Cooperative Web Caching”, Workshop in Web Engineering, Networking 2002, 2002

[135] R. Hasan, Z. Anwar, W. Yurcik, L. Brumbaugh, R. Campbell. “A Survey of Peer-to-Peer Storage Techniques for Distributed File Systems”, International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II - Volume 02, 2005

[136] P. Fuhrmann. “dCache, the commodity cache,” Twelfth NASA Goddard and Twenty First IEEE Conference on Mass Storage Systems and Technologies 2004

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

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

[139] 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

[140] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet application", ACM SIGCOMM, 2001

[141] K. Pruhs, J, Sgall, E. Torng. "Online scheduling", in Handbook of Scheduling: Algorithms, Models, and Performance Analysis, 2004

[142] E. Torng, "A unified analysis of paging and caching", Algorithmica 20, 175–200, 1998

[143] S. Irani. "Randomized Weighted Caching with Two Page Weights", Algorithmica, 32:4, 624-640, 2002

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

[145] S. Al-Kiswany, M. Ripeanu, S. Vazhkudai, "A Checkpoint Storage System for Desktop Grid Computing", Networked Systems Lab, U. of British Columbia, Tech Report NetSysLab-TR-2007-04, 2007

[146] 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

[147] GSC-II: Guide Star Catalog II, http://www-gsss.stsci.edu/gsc/GSChome.htm, 2008

[148] 2MASS: Two Micron All Sky Survey, http://irsa.ipac.caltech.edu/Missions/2mass.html, 2008

[149] POSS-II: Palomar Observatory Sky Survey, http://taltos.pha.jhu.edu/~rrg/science/dposs/dposs.html, 2008

[150] I. Foster, V. Nefedova, L. Liming, R. Ananthakrishnan, R. Madduri, L. Pearlman, O. Mulmo, M. Ahsant. "Streamlining Grid Operations: Definition and Deployment of a Portal-based User Registration Service." Workshop on Grid Applications: from Early Adopters to Mainstream Users 2005

[151] R.J. Hanisch, et al. "Definition of the Flexible Image Transport System (FITS)", Astronomy and Astrophysics, v.376, p.359-380, September 2001

[152] CAS SkyServer, http://cas.sdss.org/dr6/en/tools/search/sql.asp, 2007

[153] 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

[154] Globus Incubation Management Project, http://dev.globus.org/wiki/Incubator, 2008

[155] Falkon Globus Incubator Project, http://dev.globus.org/wiki/Incubator/Falkon, 2008

[156] Y. Gu, R. Grossman. "Exploring Data Parallelism and Locality in Wide Area Networks", IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08) 2008

[157] E.V. Hensbergen, Ron Minnich. "System Support for Many Task Computing", IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08) 2008

[158] L. Hui, Y. Huashan, L. Xiaoming. "A lightweight execution framework for massive independent tasks", IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08) 2008

[159] A.T. Thor, G.V. Zaruba, D. Levine, K. De, T.J. Wenaus. "ViGs: A Grid Simulation and Monitoring Tool for Grid Workflows", IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08) 2008

[160] E. Afgan, P. Bangalore. "Embarrassingly Parallel Jobs Are Not Embarrassingly Easy to Schedule on the Grid", IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08) 2008