Towards Data Intensive Many-Task Computing

Ioan Raicu[1], Ian Foster[1,2,3], Yong Zhao[4], Alex Szalay[5]
Philip Little[6], Christopher M. Moretti[6], Amitabh Chaudhary[6], Douglas Thain[6]

iraicu@cs.uchicago.edu, foster@mcs.anl.gov, yozha@microsoft.com, szalay@jhu.edu
plittle1@nd.edu, cmoretti@nd.edu, achaudha@nd.edu, dthain@nd.edu

[1]Department of Computer Science, University of Chicago, USA
[2]Computation Institute, University of Chicago, USA
[3]Mathematics and Computer Science Division, Argonne National Laboratory, USA
[4]Microsoft Corporation, USA
[5]Department of Physics and Astronomy, The Johns Hopkins University, USA
[6]Department of Computer Science & Engineering, University of Notre Dame, USA

ABSTRACT

Many-task computing aims to bridge the gap between two computing paradigms, high throughput computing and high performance computing. Many task computing denotes high-performance computations comprising multiple distinct activities, coupled via file system operations. The aggregate number of tasks, quantity of computing, and volumes of data may be extremely large. Traditional techniques 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. 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 conclude the chapter with performance results from a large scale astronomy application demonstrating that our approach improves both its performance and scalability.

1. INTRODUCTION

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

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

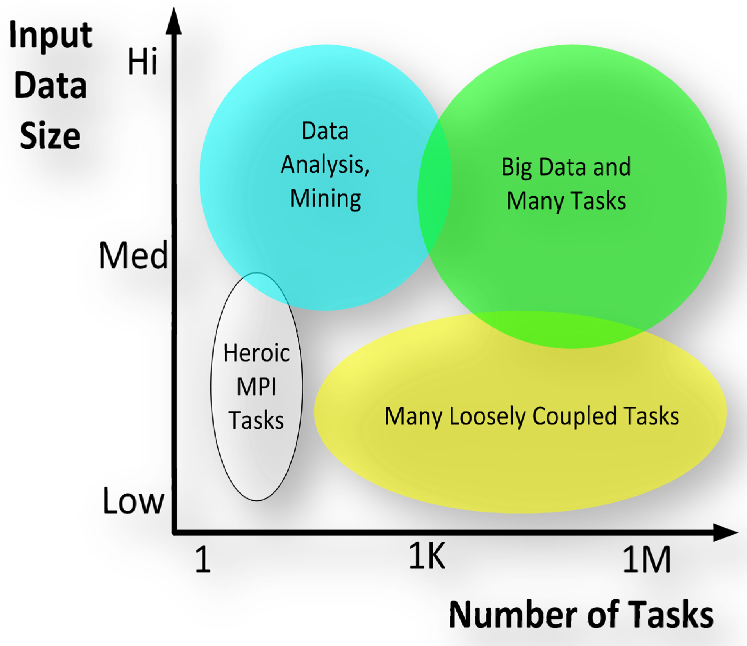


Figure 1: Problem types with respect to data size and number of tasks

High performance computing can be considered to be part of the first category (denoted by the white area). High throughput computing [13] can be considered to be a subset of the third category (denoted by the yellow area). Many-Task Computing [9] can be considered as part of categories three and four (denoted by the yellow and green areas). This chapter focuses on techniques to enable the support of data-intensive many-task computing (denoted by the green area), and the challenges that arise as datasets and computing systems are getting larger and larger.

Clusters and Grids [14, 15] have been the preferred platform for loosely coupled applications that have been traditionally part of the high throughput computing class of applications, which are managed and executed through workflow systems or parallel programming systems. Various properties of a new emerging applications, such as large number of tasks (i.e. millions or more), relatively short per task execution times (i.e. seconds to minutes long), and data intensive tasks (i.e. tens of MB of I/O per CPU second of compute) have 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 [13]. 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. Tasks may be small or large, uniprocessor or multiprocessor, compute-intensive or data-intensive. The set of tasks may be static or dynamic, homogeneous or heterogeneous, loosely coupled or tightly coupled. The aggregate number of tasks, quantity of computing, and volumes of data may be extremely large. The new term MTC draws attention to the many computations that are heterogeneous but not "happily" parallel.

Within the science domain, the data that needs to be processed generally grows faster than computational resources and their speed. The scientific community is facing an imminent flood of data expected from the next generation of experiments, simulations, sensors and satellites. Scientists are now attempting calculations requiring orders of magnitude more computing and communication than was possible only a few years ago. Moreover, in many currently planned and future experiments, they are also planning to generate several orders of magnitude more data than has been collected in the entire human history [16].

For instance, in the astronomy domain the Sloan Digital Sky Survey [17] 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 [18] 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 [19] and European Molecular Biology Laboratory (EMBL) [20] has been following an exponential trend, with a doubling time estimated to be 9-12 months. A large class of applications in Many-Task Computing will be applications that analyze large quantities of data, which in turn would require that data and computations be distributed over many hundreds and thousands of nodes in order to achieve rapid turnaround times.

Many applications in the scientific computing generally use a shared infrastructure such as TeraGrid [21] and Open Science Grid [22], 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 [23]) 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. [4]

We believe that data locality is critical to the successful and efficient use of large distributed systems for data-intensive applications [24, 25] 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 [10, 26], 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 [27], cooperative web-caching [28], and peer-to-peer storage systems [29]. Other data-aware scheduling approaches tend to assume static resources [30, 31], 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 a culmination of a collection of papers [7, 9, 10, 24, 26, 32, 33, 34, 35] dating back to 2006, and includes a deeper analysis of previous results as well as some new results. Section 2 covers our proposed support for data-intensive many-task computing, specifically through our work with the Falkon [7, 33] light-weight task execution framework and its data management capabilities in data diffusion [10, 26, 32, 34]. This section discusses the data-aware scheduler and scheduling policies. Section 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 4 offer a wide range of micro-benchmarks evaluating data diffusion, our parallel file system performance, and the data-aware scheduler performance. Section 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 [36] which allows us to compare data diffusion with a best model for active storage [37]. Section 6 covers a real large-scale application from the astronomy domain, and how data diffusion improved its performance and scalability. Section 7 covers related work, which have addressed data management issues to support data intensive applications. We finally conclude the chapter with Section 8.

2. DATA DIFFUSION ARCHITECTURE

We implement data diffusion [10] in the Falkon task dispatch framework [7]. 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.

2.1 Falkon and Data Diffusion

To enable the rapid execution of many tasks on distributed resources, Falkon combines (1) multi-level scheduling [38] 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 [7]. Recent work has achieved throughputs in excess of 3750 tasks/sec and scalability up to 160K processors [33].

Figure 2 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 [39] 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 [40] at each executor, which allows other executors to read data from its cache. This scheduling information are only hints, as remote cache state can change frequently and is not guaranteed to be 100% in sync with the global index. In the event that a data item is not found at any of the known cached locations, it attempts to retrieve the item from persistent storage; if this also fails, the respective task fails. In Figure 2, the black dotted lines represent the scheduler sending the task to the compute nodes, along with the necessary information about where to find input data. The red thick solid lines represent the ability for each executor to get data from remote persistent storage. The blue thin solid lines represent the ability for each storage resource to obtain cached data from another peer executor.

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 [41]). 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* [27]. Our empirical experiments all use LRU, and we will study the other policies in future work.

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 [41]. 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 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.

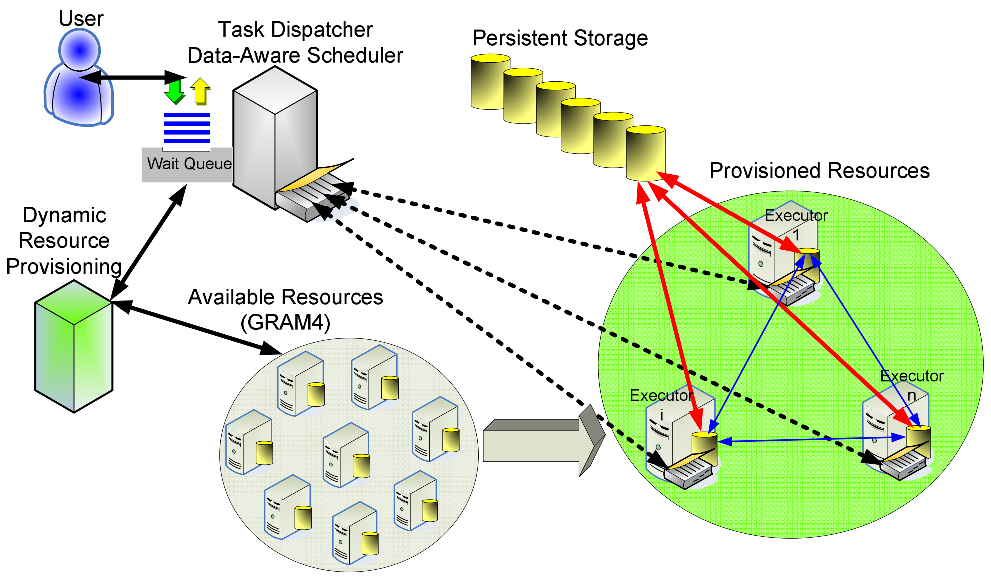


Figure 2: Architecture overview of Falkon extended with data diffusion (data management and data-aware scheduler)

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) [42] or distributed hash table (DHT) [43]. Chervenak et al. [42] 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 3. 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) [10], 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.

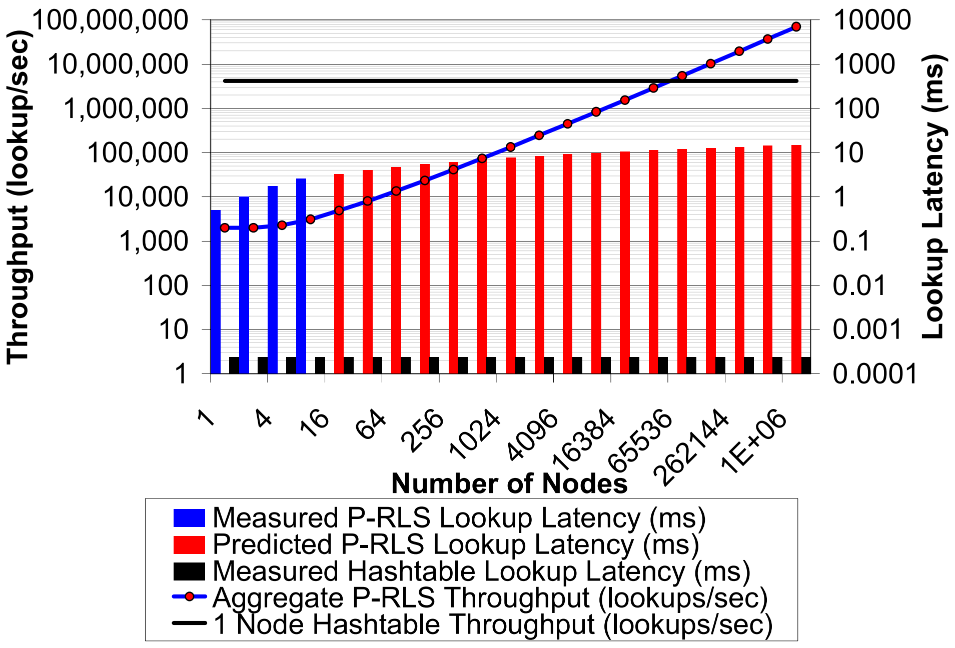


Figure 3: P-RLS vs. Hash Table performance for 1M entries

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 [36, 44, 45, 46] 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 [41], 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.

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 4 and Figure 5); the algorithm is separated into two sections, as the first part (Figure 4) decides which executor will be notified of available tasks, while the second part (Figure 5) 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

```
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 4: Pseudo Code for part #1 of algorithm which sends out the notification for work

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 2.2).

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

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.

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 1), 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 1: 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 2.

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

Equation 2: 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 (see Equation 3):

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

Equation 3: Average time to complete task i

Which can also be expressed as (see Equation 4):

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

Equation 4: 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 5):

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

Equation 5: Time to complete an entire workload D

Having defined the time to complete workload D, we define speedup as Equation 6:

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

Equation 6: Speedup

And efficiency can be defined as (see Equation 7):

$$EF = SP / N$$

Equation 7: Efficiency

What is the maximum task arrival rate (*A*) that a particular scenario can sustain? We have (see Equation 8):

$$\begin{array}{ll} S \geq W & : N*P/(R_l+C) \leq A_{max} \leq N*P/(R_c+C) \\ S < W & : N*P/(R_c+C) \leq A_{max} < N*P/(R_s+C) \end{array}$$

Equation 8: 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 9):

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

Equation 9: 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 10):

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

Equation 10: 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 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 [47]. 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 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 2Mark (which uses the well known marking algorithm [47] at two levels) for this case of data diffusion. Let the corresponding optimum offline algorithm be OPT. For a sequence σ, let 2Mark(σ) be the cost 2Mark incurs to handle the sequence and define OPT(σ) similarly. 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. 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 11 which establishes that 2Mark is *O(NM)-competitive*. From the lower bound on the competitive ratio for simple paging [47], 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 11: 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 6). 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 $j \neq 1$ 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 6: Algorithm 2Mark

4. MICRO-BENCHMARKS

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

4.1 Testbed Description

Table 1 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 [48] 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 1: 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

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 7 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.

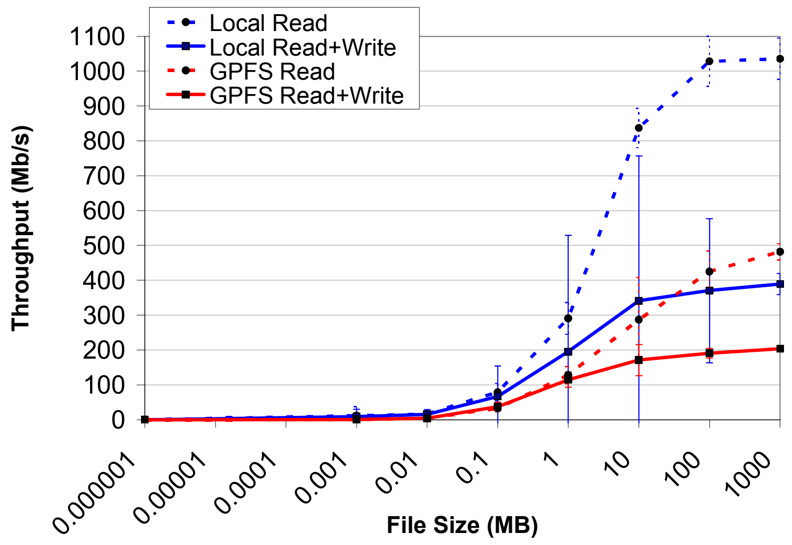


Figure 7: Comparing performance between the local disk and the shared file system GPFS from one node

Notice that the GPFS read performance (Figure 8) 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.

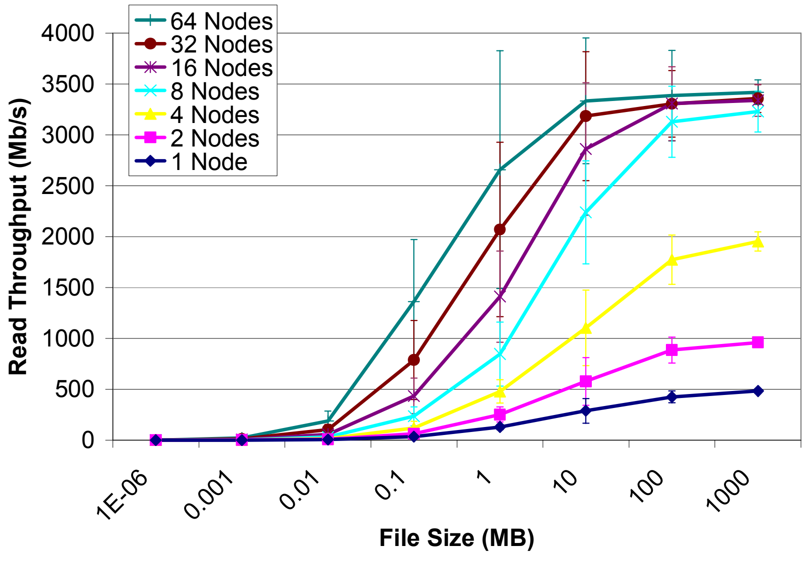


Figure 8: Read performance for GPFS expressed in Mb/s; only the x-axis is logarithmic; 1-64 nodes for GPFS; 1B – 1GB files

The read+write performance (Figure 9) 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).

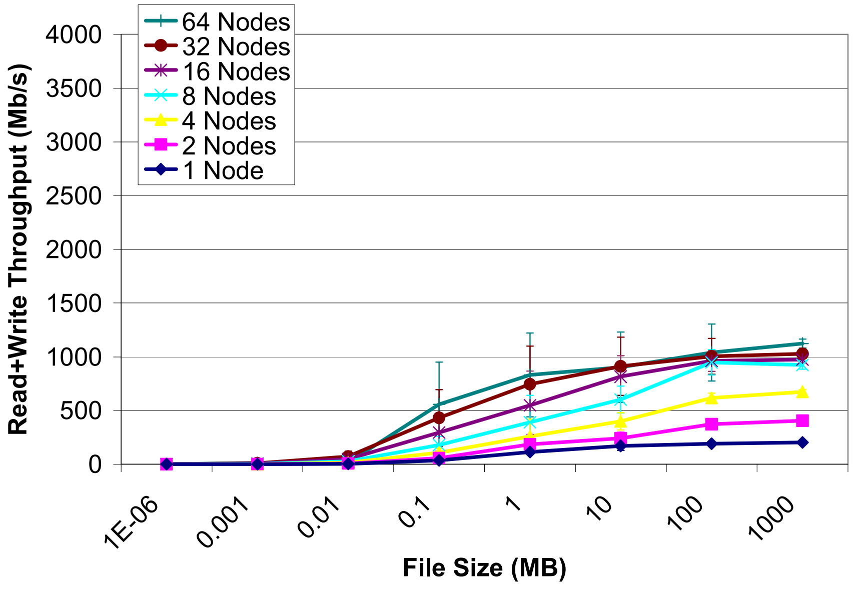


Figure 9: Read+write performance for GPFS expressed in Mb/s; only the x-axis is logarithmic; 1-64 nodes for GPFS; 1B – 1GB files

These final two graphs (Figure 10 and Figure 11) show the theoretical read+write and read throughput (measured in Mb/s) for local disk access.

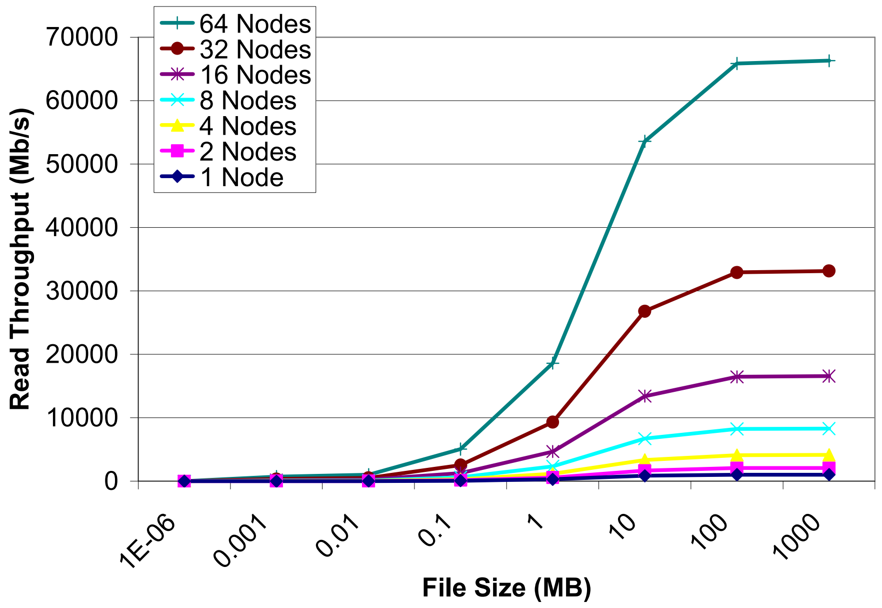


Figure 10: Theoretical read performance of local disks expressed in Mb/s; only the x-axis is logarithmic; 1-64 nodes for local disk access; 1B – 1GB files

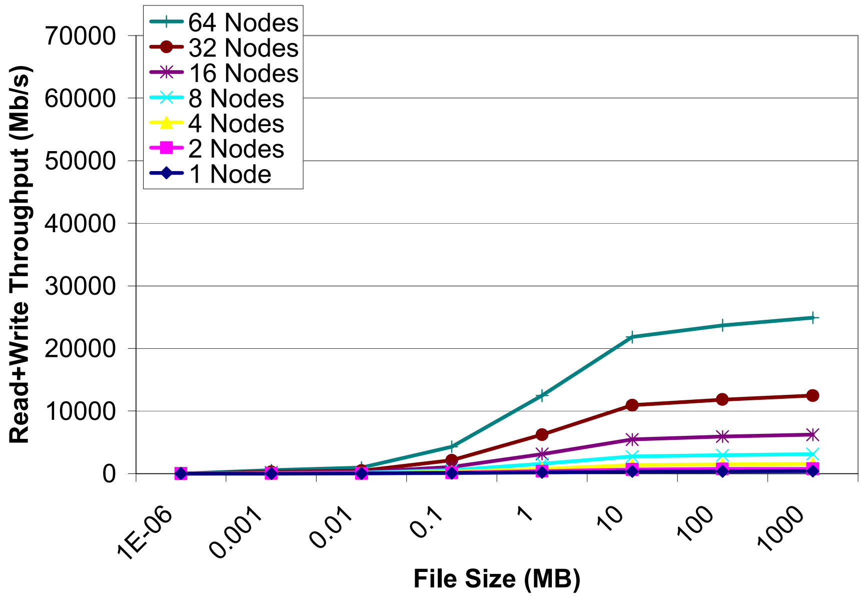


Figure 11 (local model 1-64 nodes r+w): Theoretical read+write performance of local disks expressed in Mb/s; only the x-axis is logarithmic; 1-64 nodes for local disk access; 1B – 1GB files

These results are theoretical, as they are simply a derivation of the 1 node performance (see Figure 7), 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.

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 12 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.

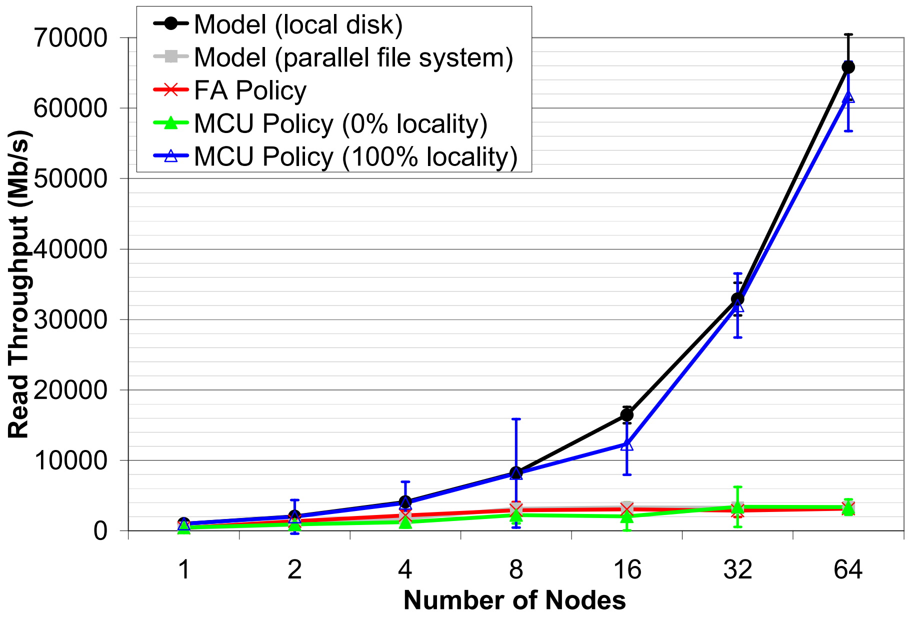


Figure 12: Read throughput (Mb/s) for large files (100MB) for seven configurations for 1 – 64 nodes

Figure 13 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 12 and Figure 13, 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.

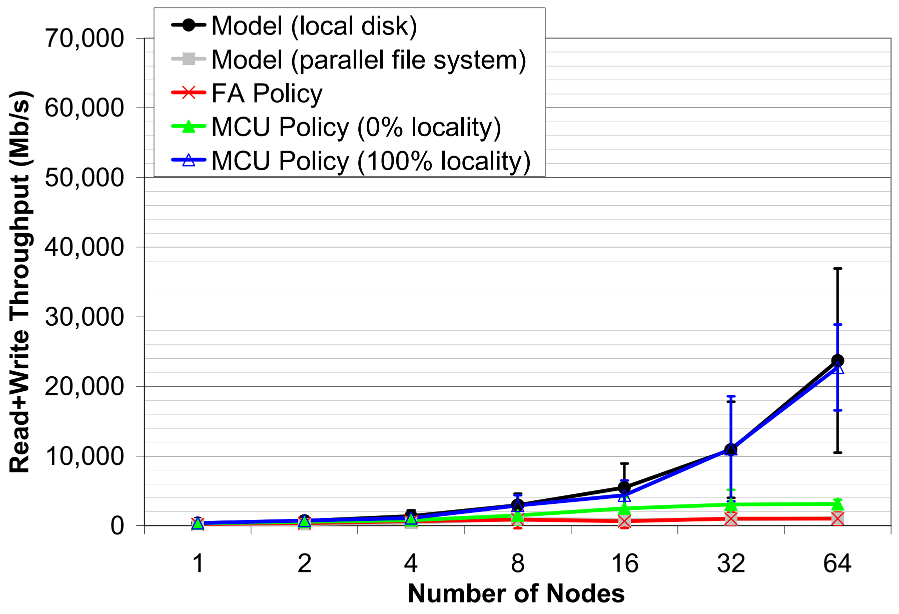


Figure 13: Read+Write throughput (Mb/s) for large files (100MB) for seven configurations and 1 – 64 nodes

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 14 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.

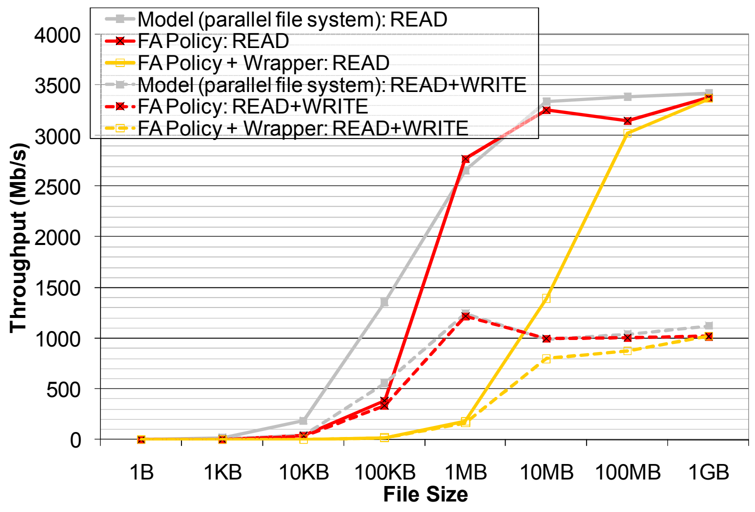


Figure 14: Read and Read+Write throughput (Mb/s) for a wide range of file sizes for three configurations on 64 nodes

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 15 shows the scheduler performance under different scheduling policies.

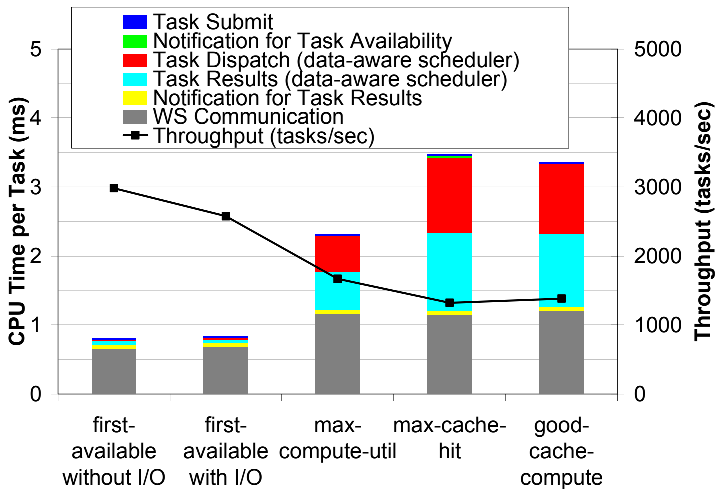


Figure 15: Data-aware scheduler performance and code profiling for various scheduling policies

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.

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 [48] (up to 100 nodes, 200 processors). The Falkon service ran on an 8-core Xeon@2.33GHz, 2GB RAM, Java 1.5, 100Mb/s network, and 2 ms latency to the executors. The persistent storage was GPFS [23] with <1ms latency to executors.

The three sub-sections that follow cover three diverse workloads: Monotonically-Increasing (MI: Section 5.1), Sine-Wave (SI: Section 5.2), and All-Pairs (AP: Section 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 [36].

All our workloads were generated at random, using uniform distribution. Although this might not be representative of all applications, we believe workloads with uniform distribution will stress the data-aware scheduler far more than if the distribution followed zipf where few files were extremely popular and many files were unpopular. In general, zipf distribution workloads require a smaller aggregate cache in relation to the workload working set. From data locality perspective, uniform distribution workloads offer the worst case scenario. It should be noted that zipf distributions would have caused hot-spots to occur for popular files, which does not naturally occur in uniform distribution workloads. However, due to our dynamic resource provisioning, when many new compute and storage resources are joining frequently, the initial local storage population of data can certainly cause the same kind of hot-spots by putting heavier stress on existing nodes. Data diffusion handles these hot-spots naturally, as temporarily popular data (due to new nodes joining and having an empty cache) get replicated at the new storage locations, and subsequent accesses to this data can be served locally. Therefore, in our system, these hot-spots are only temporary, while the popular data diffuses to other storage nodes, and subsequent accesses to this data is then localized effectively eliminating the hot-spot (even in the face of continuing access patterns that favor several popular files).

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 12.

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

Equation 12: 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 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 16.

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

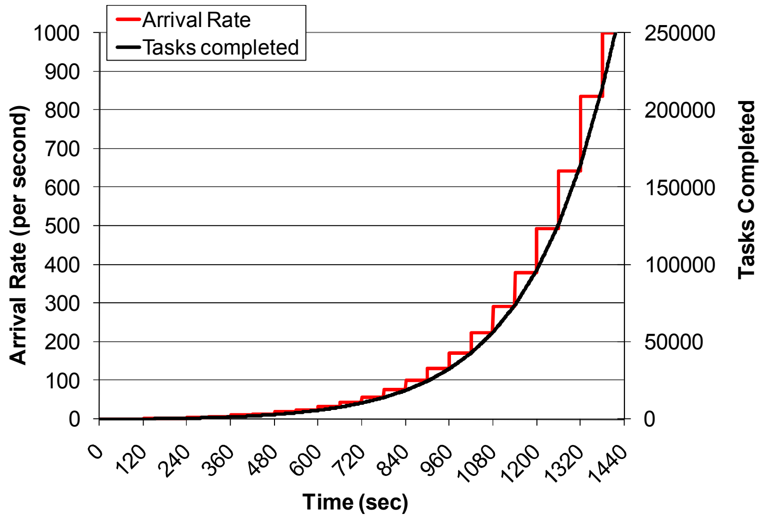


Figure 16: Monotonically Increasing workload overview

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).

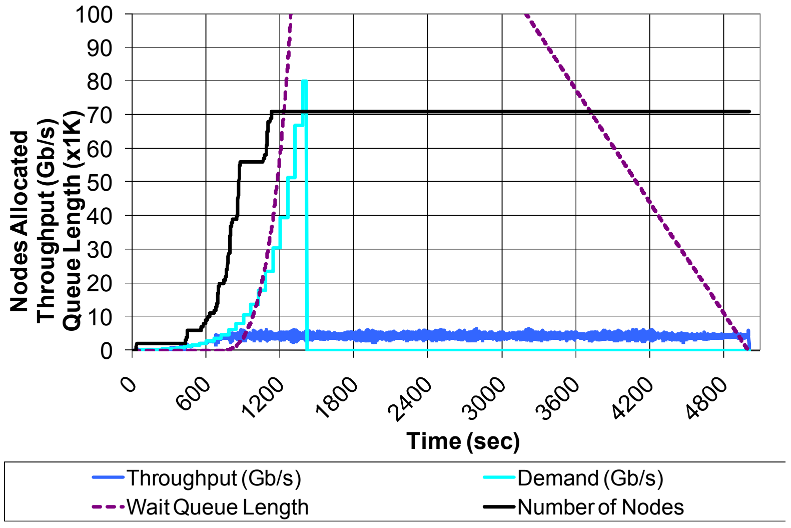


Figure 17: MI workload, 250K tasks, 10MB:10ms ratio, up to 64 nodes using DRP, FA policy

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 18 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.

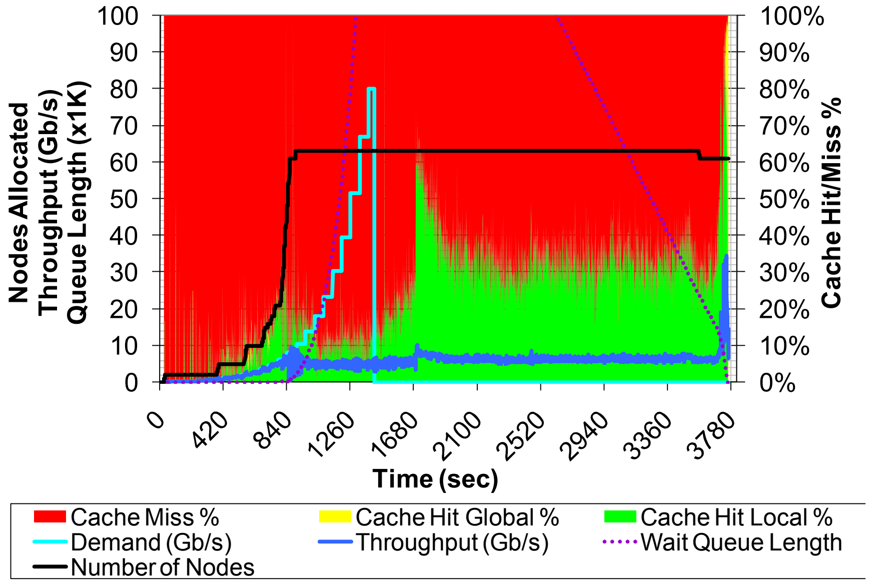


Figure 18: MI workload, 250K tasks, 10MB:10ms ratio, up to 64 nodes using DRP, GCC policy, 1GB caches/node

Figure 19 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. Overall, 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.

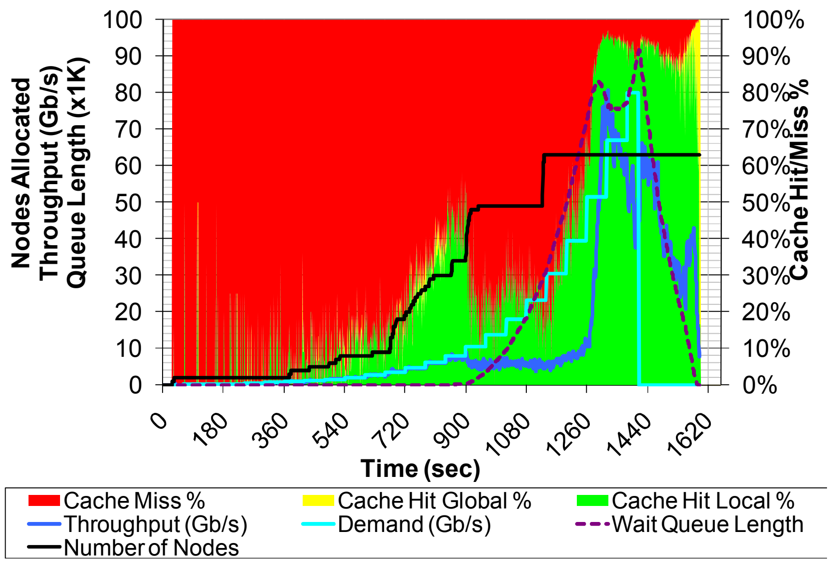


Figure 19: MI workload, 250K tasks, 10MB:10ms ratio, up to 64 nodes using DRP, GCC policy, 1.5GB caches/node

Figure 20 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%.

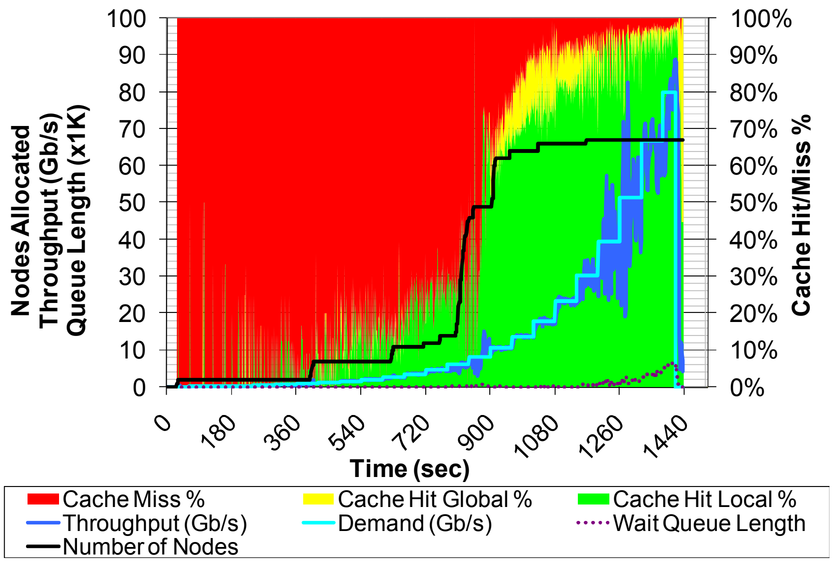


Figure 20: MI workload, 250K tasks, 10MB:10ms ratio, up to 64 nodes using DRP, GCC policy, 2GB caches/node

Investigating if it helps to increase the cache size further to 4GB per node, we conduct the experiment whose results are found in Figure 21. 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.

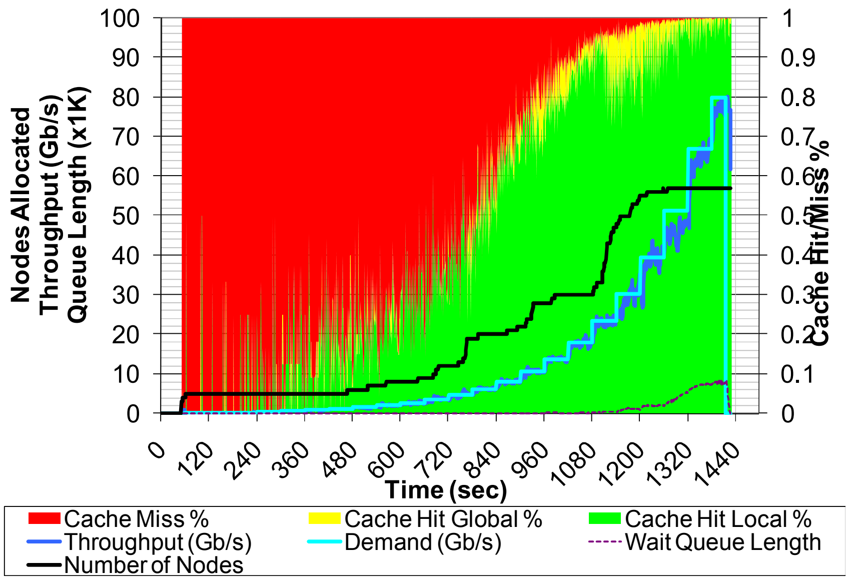


Figure 21: MI workload, 250K tasks, 10MB:10ms ratio, up to 64 nodes using DRP, GCC policy, 4GB caches/node

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 22), 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).

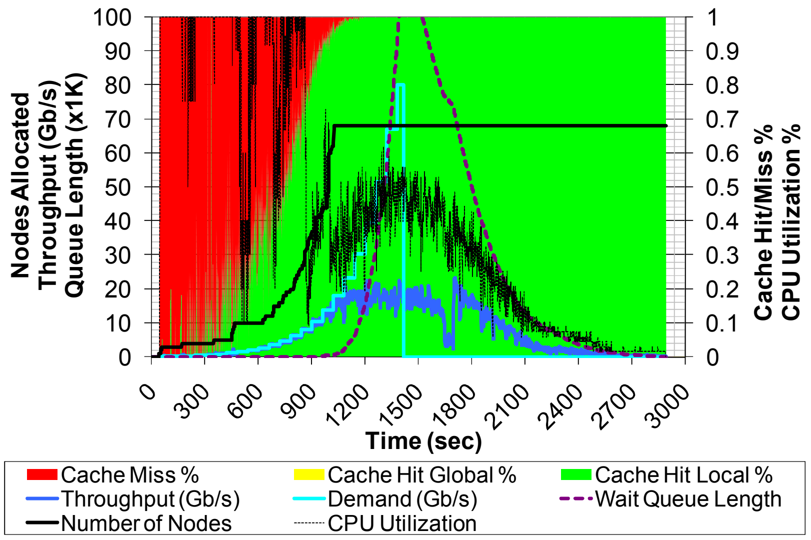


Figure 22: MI workload, 250K tasks, 10MB:10ms ratio, up to 64 nodes using DRP, MCH policy, 4GB caches/node

Figure 23 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.

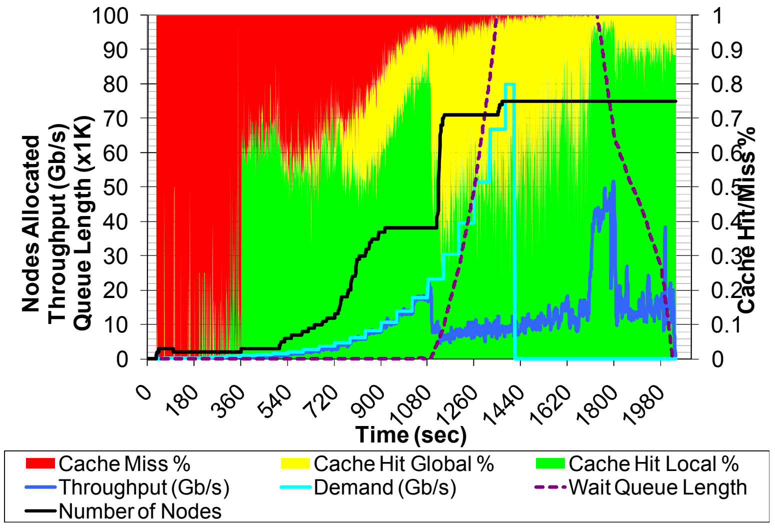


Figure 23: MI workload, 250K tasks, 10MB:10ms ratio, up to 64 nodes using DRP, MCU policy, 4GB caches/node

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.

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

5.1.3 Cache Performance

Figure 24 shows cache performance over six experiments involving data diffusion, the ideal case, and the FA policy which does not cache any data.

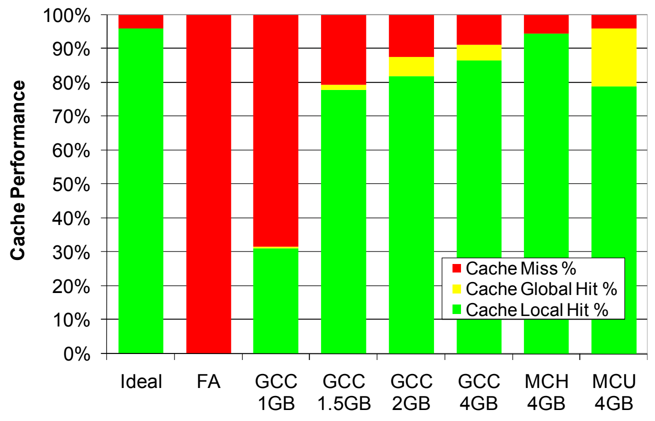


Figure 24: MI workload cache performance

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%.

5.1.4 Throughput

Figure 25 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.

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

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

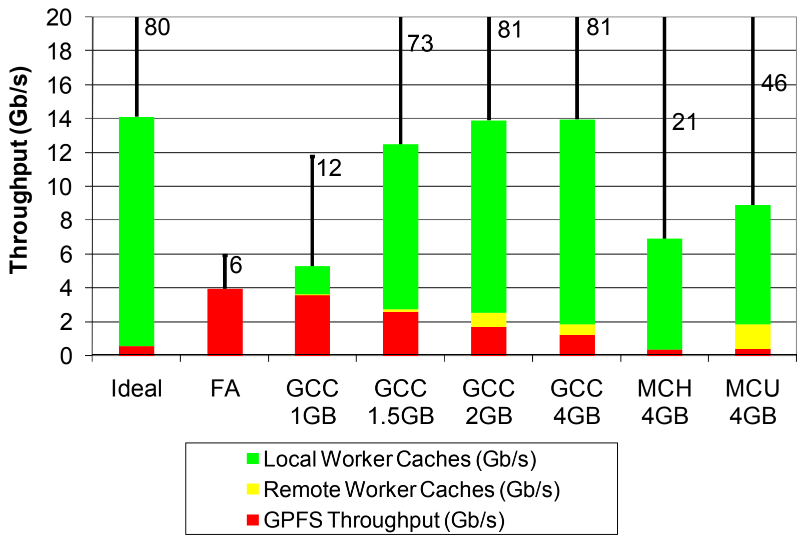


Figure 25: MI average and peak (99 percentile) throughput

5.1.5 Performance Index and Speedup

The performance index attempts to capture the speedup per CPU time achieved. Figure 26 shows PI and speedup data. 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).

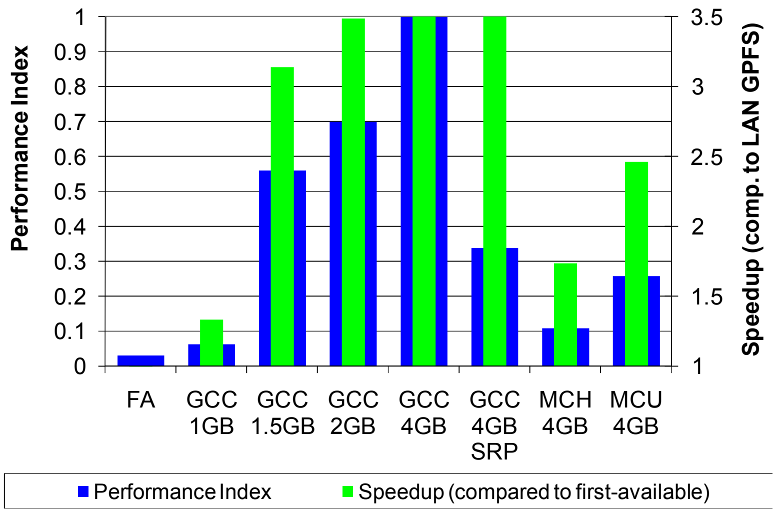


Figure 26: MI workload PI and speedup comparison

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 27 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 27 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 23 (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.

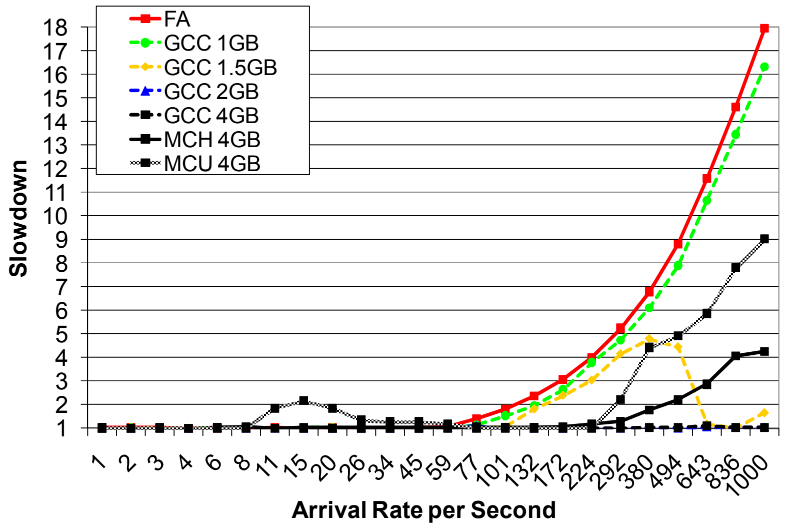


Figure 27: MI workload slowdown as we varied arrival rate

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 28 shows response time results across all 14 experiments in log scale.

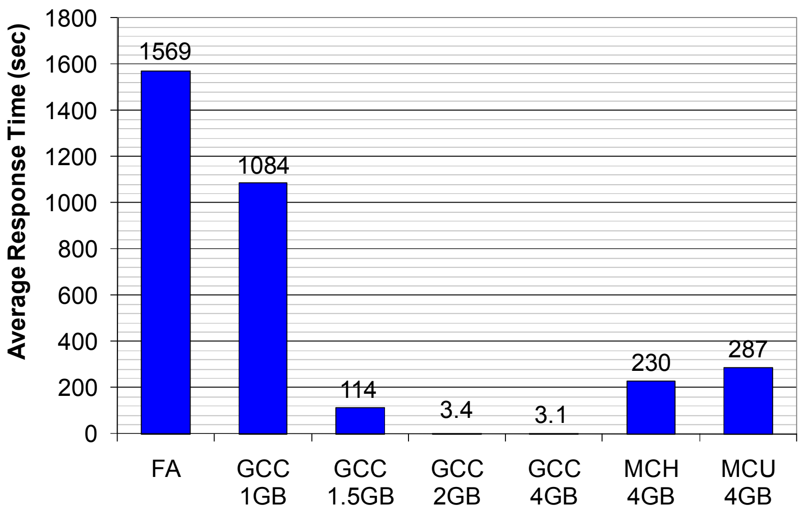


Figure 28: MI workload average response time

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.

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 (see Figure 29) that follows the function shown in Equation 13 (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 13: Sine-Wave Workload arrival rate function

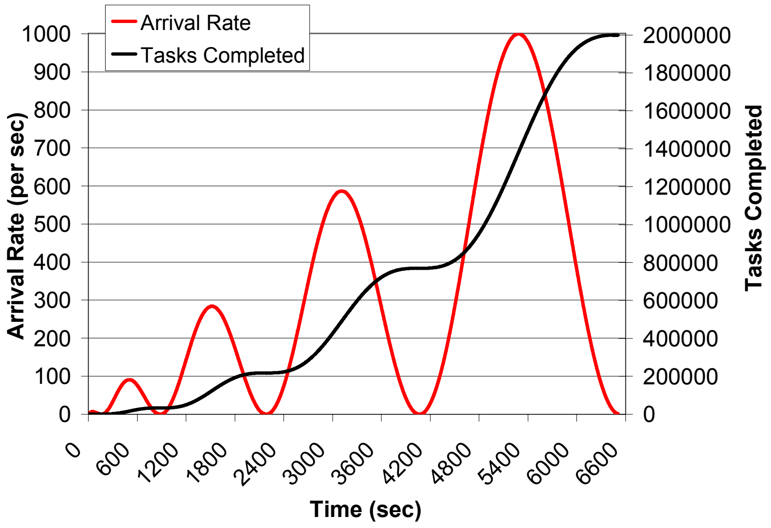


Figure 29: SW workload overview

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 (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.

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 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.

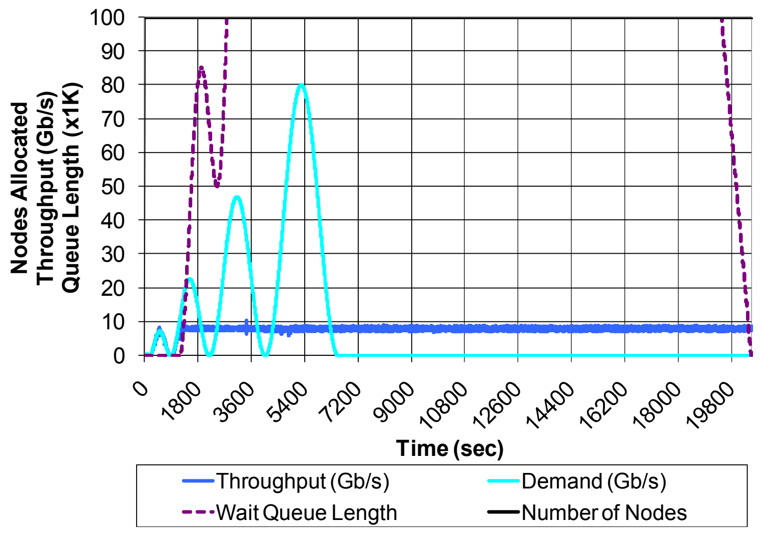


Figure 30: SW workload, 2M tasks, 10MB:10ms ratio, 100 nodes, FA policy

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 31), 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.

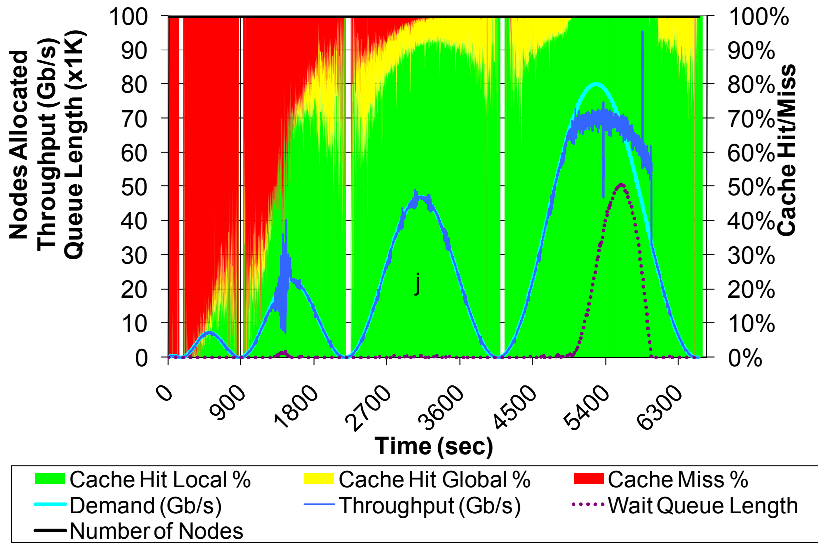


Figure 31: SW workload, 2M tasks, 10MB:10ms ratio, 100 nodes, GCC policy, 50GB caches/node

Enabling dynamic resource provisioning, Figure 32 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.

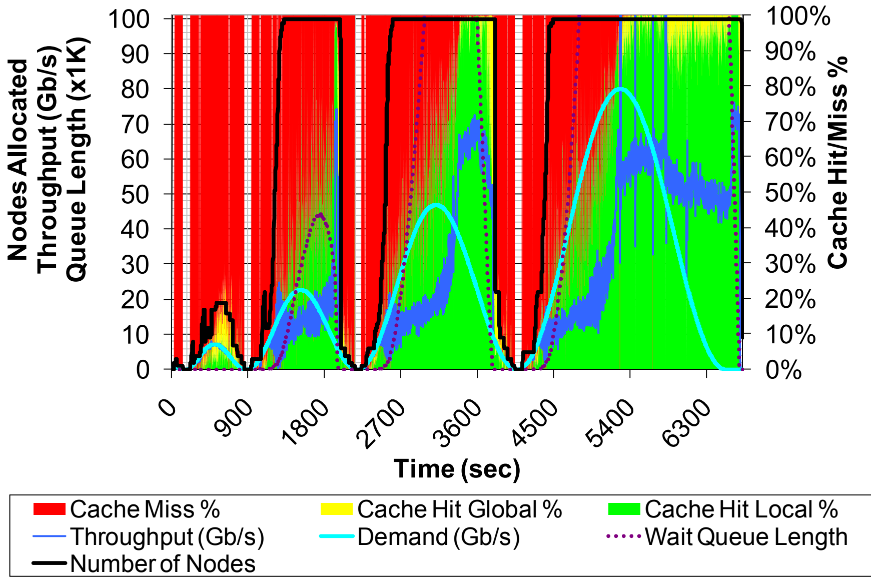


Figure 32: SW workload, 2M tasks, 10MB:10ms ratio, up to 100 nodes with DRP, GCC policy, 50GB caches/node

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).

5.3 All-Pairs Workload Evaluation

In previous work, several of the co-authors addressed large scale data intensive problems with the Chirp [37] 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 [36], 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 33). [36] The AP problem is easy to express in terms of two nested for loops over some parameter space (see Figure 34). 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 35 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 33: All-Pairs definition

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

Figure 34: All-Pairs Workload Script

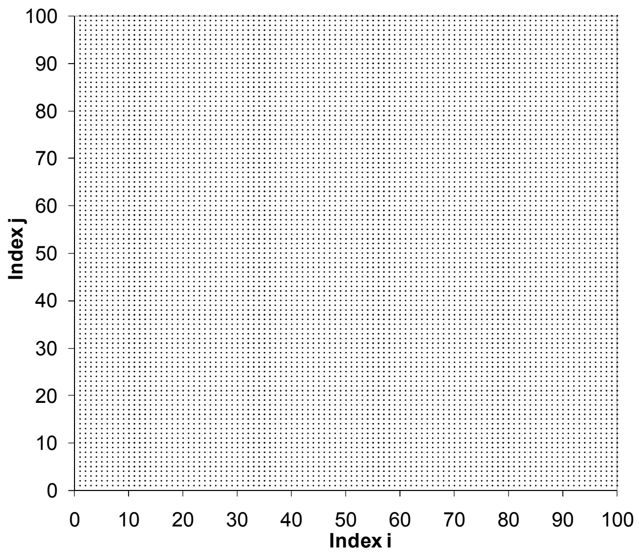


Figure 35: Sample 100x100 AP problem, where each dot represents a computation at index i,j

In previous work [36], 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 [36], 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 36), data diffusion achieves a throughput that is 80% of the best case of all data accesses occurring to local disk (see Figure 40). 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 [36] 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.

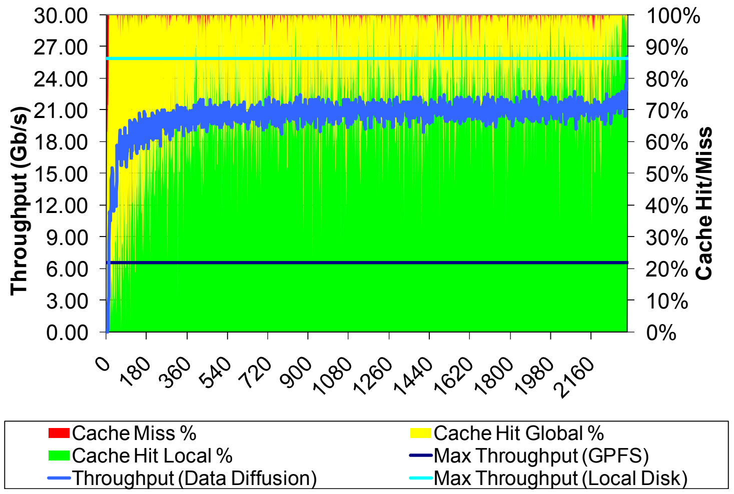


Figure 36: AP workload, 500x500=250K tasks, 24MB:1000ms, 100 nodes, GCC policy, 50GB caches/node

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 37).

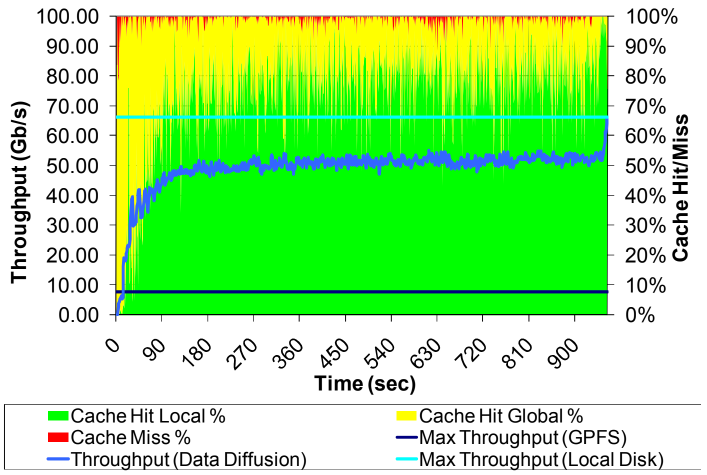


Figure 37: AP workload, 500x500=250K tasks, 24MB:100ms, 100 nodes, GCC policy, 50GB caches/node

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 [2] and a SiCortex SC5832 [49]. 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 2).

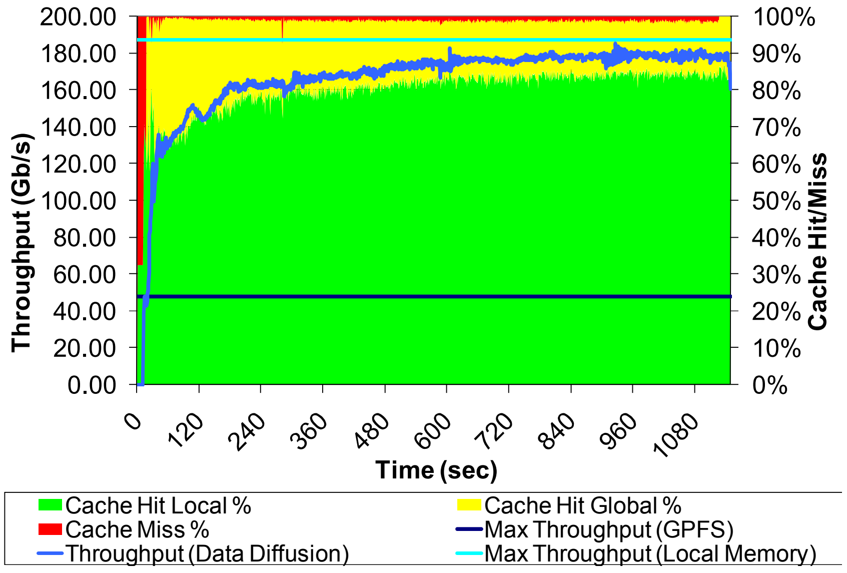


Figure 38: AP workload on emulated Blue Gene/P, 1000x1000=1M tasks, 24MB:4000ms, 1024 nodes (4096 processors), GCC policy, 2GB caches/node

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.

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.

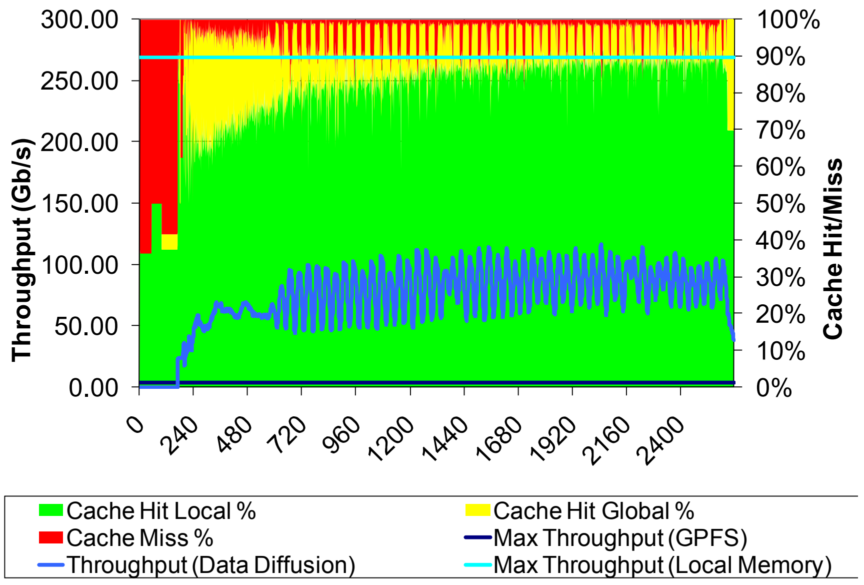


Figure 39: AP workload on emulated SiCortex, 1000x1000=1M tasks, 24MB:4000ms, 972 nodes (5832 processors), GCC policy, 3.6GB caches/node

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 38 and Figure 39) is not only emulated, but also hypothetical. Nevertheless, it is interesting to see the significant difference in efficiency between data diffusion and active storage at this larger scale.

Our comparison between data diffusion and active storage fundamentally boils down to a comparison of pushing data versus pulling data. The active storage implementation pushes all the needed data for a workload to all nodes via a spanning tree. With data diffusion, nodes pull only the files immediately needed for a task, creating an incremental spanning forest (analogous to a spanning tree, but one that supports cycles) at runtime that has links to both the parent node and to any other arbitrary node or persistent storage. We measured data diffusion to perform comparably to active storage on our 200 processor cluster, but differences exist between the two approaches. Data diffusion is more dependent on having a well balanced persistent storage for the amount of computing power (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.

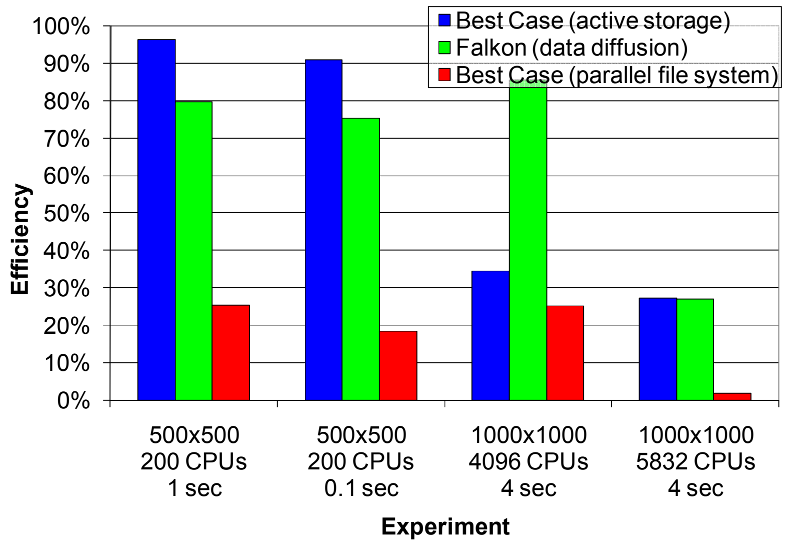


Figure 40: AP workload efficiency for 500x500 problem size on 200 processor cluster and 1000x1000 problem size on the Blue Gene/P and SiCortex emulated systems with 4096 and 5832 processors respectively

Table 2: Data movement comparing 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. LARGE-SCALE ASTRONOMY APPLICATION PERFORMANCE EVALUATION

Section 4 and Section 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 [17], GSC-II [50], 2MASS [51], and POSS-II [52]. 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*.

6.1 Definition of "Stacking"

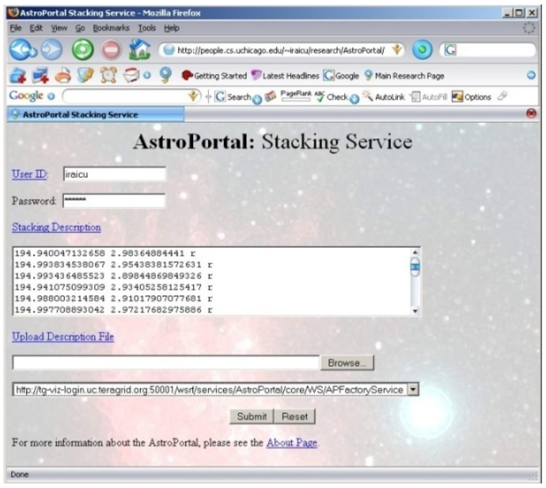
The first analysis that we have implemented in our AstroPortal prototype is "stacking," image cutouts from different parts of the sky. This function can help to statistically detect objects too faint otherwise. Astronomical image collections usually cover an area of sky several times (in different wavebands, 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.

6.2 AstroPortal

AstroPortal provides both a Web Services and a Web portal interface. Figure 41 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. 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 41, 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 41 (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 [53].


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

AstroPortal: Stacking Service

User ID: iraicu

Password: ******

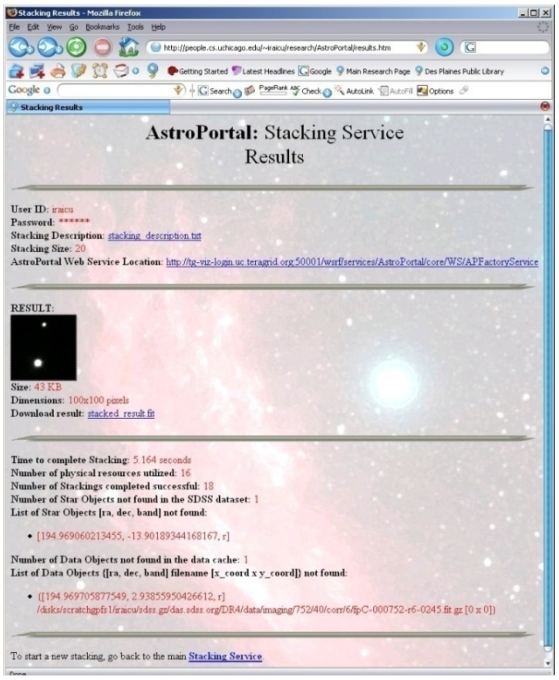
Stacking Description

```
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.


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

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 41: Left: AstroPortal Web Portal Stacking Service; Right: Stacking Service Result

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.

6.3 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 [17]. To construct realistic workloads, we identified the interesting objects (for a quasar search) from SDSS DR5; we used the CAS SkyServer [54] to issue the SQL command from Figure 42. 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 42: 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 3) 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 on average of).

Table 3: 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

6.4 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 43, the Y-axis is time per task per code block measured in milliseconds (ms).

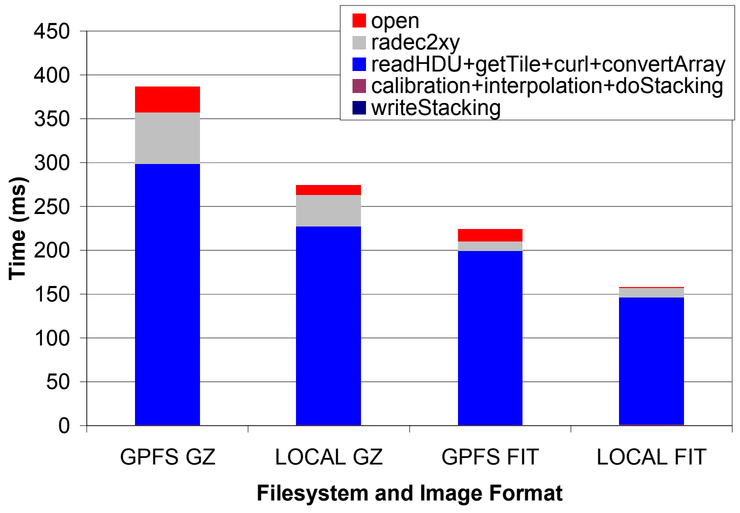


Figure 43: Stacking code performance profiling for 1 CPU

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 43 shows the stacking analysis to be I/O bound and data intensive.

6.5 Performance Evaluation

All tests performed in this section were done using the testbed described in Table 1, using from 1 to 64 nodes, and the workloads (described in Table 3) 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 44 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 (as described in Section 6.4 and Figure 43) per processor used; with perfect scalability, the time per stack should remain constant as we increase the number of processors.

We see in Figure 44 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.

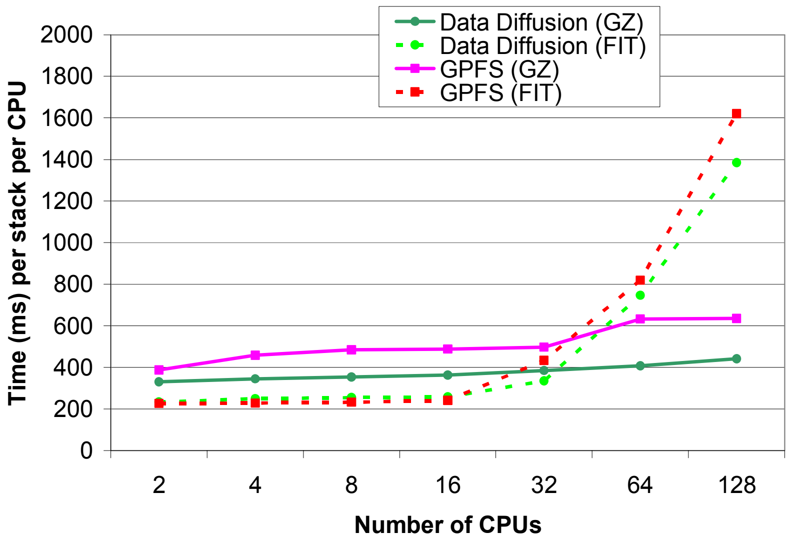


Figure 44: Performance of the stacking application for a workload data locality of 1.38 using data diffusion and GPFS while varying the CPUs from 2 to 128

While the previous results from Figure 44 shows an almost worst case scenario where the data locality is small (1.38), the next set of results (Figure 45) 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 44.

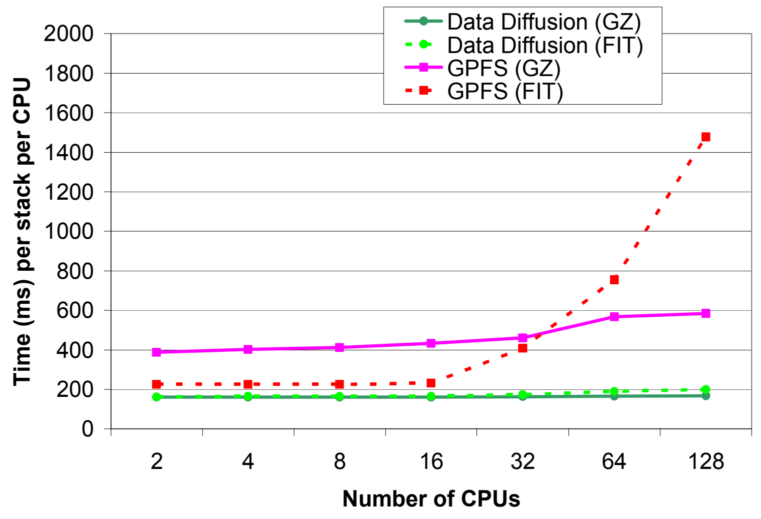


Figure 45: Performance of the stacking application for a workload data locality of 30 using data diffusion and GPFS while varying the CPUs from 2 to 128

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 46). 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.

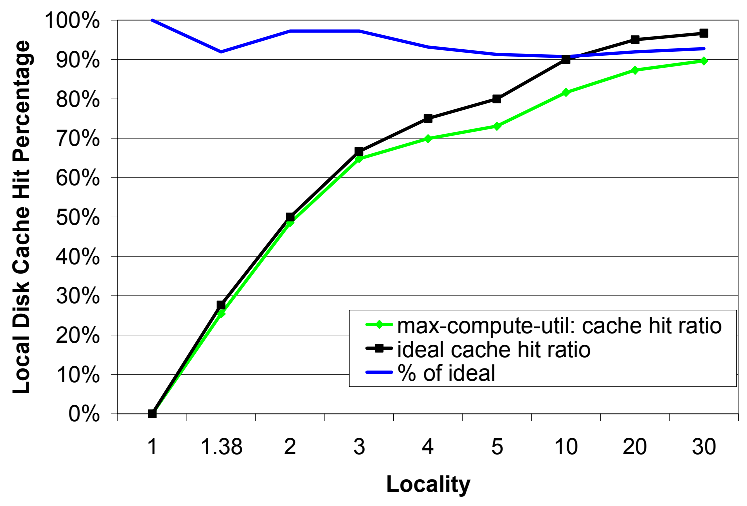


Figure 46: Cache hit performance of the data-aware scheduler for the stacking application using 128 CPUs for workloads ranging from 1 to 30 data locality using data diffusion

The following experiment (Figure 47) 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.

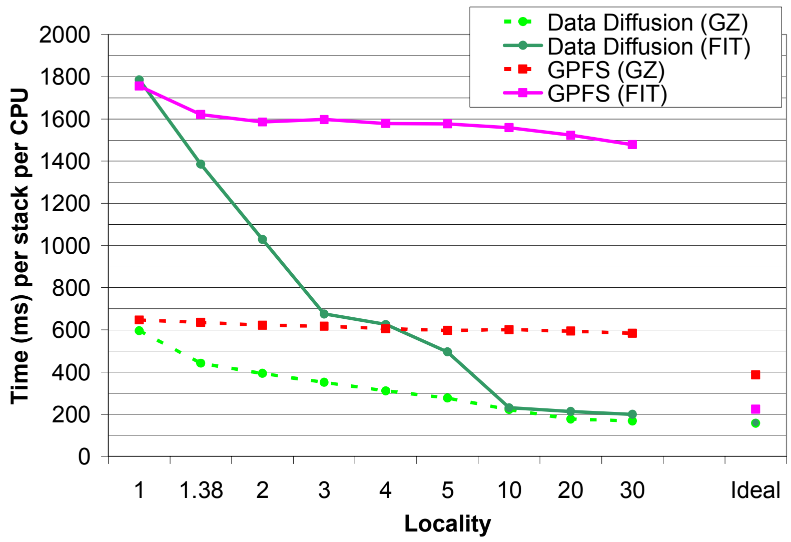


Figure 47: Performance of the stacking application using 128 CPUs for workloads with data locality ranging from 1 to 30, using data diffusion and GPFS

Figure 48 shows aggregate I/O throughput and data movement for the experiments of Figure 47. 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.

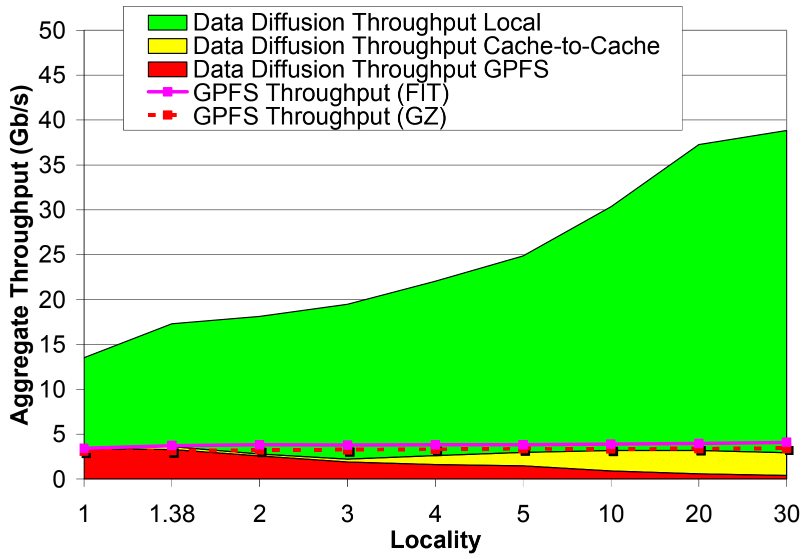


Figure 48: I/O throughput of image stacking using 128 CPUs, for workloads with data locality ranging from 1 to 30, and using both data diffusion and GPFS

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 49 investigates the amount of data movement that occurs per stacking as we vary data locality.

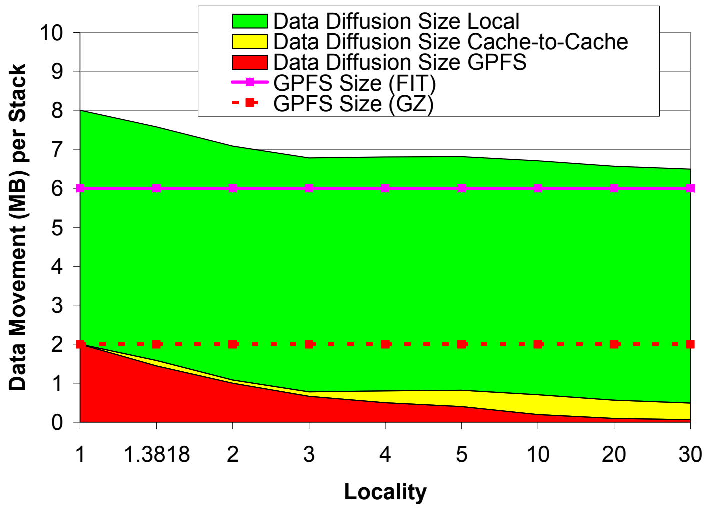


Figure 49: Data movement for the stacking application using 128 CPUs, for workloads with data locality ranging from 1 to 30, using data diffusion and GPFS

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.

6.6 Abstract Model Validation

We perform a preliminary validation of our abstract model (presented in Section 3.1) with results from a real large-scale astronomy application [44]. 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 [10], we found an average (and median) model error 5%, with a standard deviation of 5% and 29% worst case.

Figure 50 and Figure 51 shows the details of the model error under the various experiments, presented earlier in this section. 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 50 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.

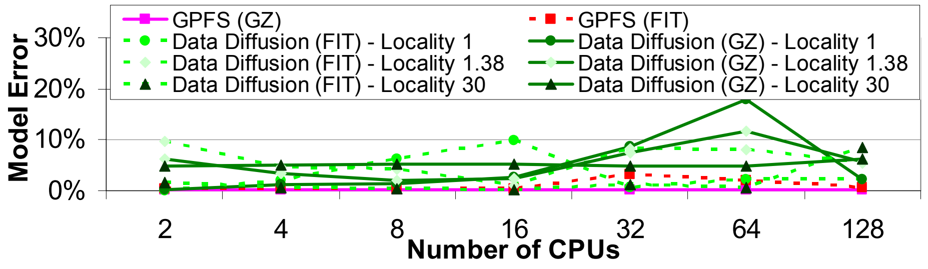


Figure 50: Model error for varying number of processors

Figure 51 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.

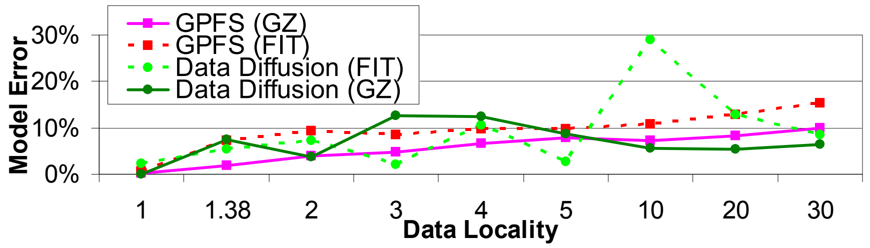


Figure 51: Model error for varying data-locality

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 5.1 and Section 5.2.

7. RELATED WORK

There has been much work in the general space of data management in distributed systems over the last decade. This section is placed at the end, as some of the comparisons to other systems get into subtle details, which are better appreciated after being familiar with data diffusion.

The Stork [55] 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 co-schedule CPU and storage resources, there is no attempt to retain nodes and harness data locality in data access patterns between tasks.

The GFarm team implemented a data-aware scheduler in Gfarm using an LSF scheduler plugin [30, 56, 57, 58]. 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 [59] 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 [60] 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 [6, 61] 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 [62], Google File System (GFS) [63], MapReduce [5], and Hadoop [41] 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 [6, 61] 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 [64] and Sector [65]. 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 [66, 67]. Chervenak et al. developed the Earth System Grid-I prototype to analyze climate simulation data using data Grid technologies [68]. The Mobius project developed a sub-project DataCutter for distributed processing of large datasets [69]. A database oriented view for data analysis is taken in the design of GridDB, a data-centric overlay for the scientific Grid [70]. Finally, Olson et al. discusses Grid service requirements for interactive analysis of large datasets [71].

With respect to provable performance results, several online competitive algorithms are known for a variety of problems in scheduling (see [72] for a survey) and for some problems in caching (see [47] 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 [73]; it allows for different page costs but assumes a single cache.

Finally, there has been some interest from the database community to do multi-query optimization. One such effort is the active semantic caching [75], which performs the identification and transparent reuse of data and computation in the presence of multiple queries. They explore a similar space of applications for which Falkon and data diffusion is most suitable for, namely scientific applications; furthermore, they draw similar conclusions that we do, such as that data locality is critical to obtaining good performance. However, their work centers on databases, which inherently limits the types of operations and applicability of their proposed system. We argue that the Falkon system in conjunction with the Swift parallel programming system is a much more general, powerful, and accepted set of tools for the broader scientific community.

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.

8. 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 [24]. 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 [6, 61, 74] and Falkon [7, 33]. 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). [10, 35]

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.

9. ACKNOWLEDGEMENTS

This work was supported in part by the NASA Ames Research Center GSRP Grant Number NNA06CB89H and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. This research was also supported in part by the National Science Foundation through TeraGrid resources provided by UC/ANL. We thank the Computation Institute at University of Chicago for hosting part of the experiments. We also thank the numerous people that have contributed to ideas, papers, implementations that have made this chapter possible; in no particular order, Jim Gray, Julian Bunn, Catalin Dumitrescu, Mike Wilde, Zhao Zhang, Ben Clifford, and Mihael Hategan.

10. REFERENCES

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

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

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

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

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

[6] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, M. Wilde. "Swift: Fast, Reliable, Loosely Coupled Parallel Computation", IEEE Workshop on Scientific Workflows 2007

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

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

[9] I. Raicu, I. Foster, Y. Zhao. "Many Task Computing: Bridging the Gap Between High Throughput Computing and High Performance Computing", IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08) 2008

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[29] 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

[30] 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

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

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

[33] 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

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

[35] 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

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

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

[38] 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

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

[40] 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

[41] 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

[42] 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

[43] 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

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

[45] 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

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

[47] E. Torng, “A unified analysis of paging and caching”, Algorithmica 20, 175–200, 1998

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

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

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

[51] 2MASS: Two Micron All Sky Survey, http://irsa.ipac.caltech.edu/Missions/2mass.html, 2008

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

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

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

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

[56] 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.

[57] 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

[58] 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

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

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

[61] 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

[62] 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

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

[64] 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

[65] 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

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

[67] 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

[68] A. Chervenak, E. Deelman, C. Kesselman, B. Allcock, I. Foster, V. Nefedova, J. Lee, A. Sim, A. Shoshani, B. Drach, D. Williams, D. Middleton. “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

[69] 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

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

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

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

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

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

[75] H. Andrade, T. Kurc, A. Sussman, J. Saltz. "Active Semantic Caching to Optimize Multidimensional Data Analysis in Parallel and Distributed Environments", Parallel Computing Journal, Vol. 33, Nos. 7-8, August 2007