Reconciling Scratch Space Consumption, Exposure, and Volatility to Achieve Timely Staging of Job Input Data

Henry M. Monti, Ali R. Butt
Dept. of Computer Science
Virginia Tech.
Blacksburg, Virginia, USA
Email: {hmonti, butta}@cs.vt.edu

Sudharshan S. Vazhkudai
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
vazhkudaiss@ornl.gov

***Abstract*—Innovative scientific applications and emerging dense data sources are creating a data deluge for high-end computing systems. Processing such large input data typically involves copying (or *staging*) onto the supercomputer's specialized high-speed storage, *scratch* space, for sustained high I/O throughput. The current practice of conservatively staging data as early as possible makes the data vulnerable to storage failures, which may entail re-staging and consequently reduced job throughput. To address this, we present a timely staging framework that uses a combination of job start-up time predictions, user-specified intermediate nodes, and decentralized data delivery to coincide input data staging with job start-up. By delaying staging to when it is necessary, the exposure to failures and its effects can be reduced.**

Evaluation using both PlanetLab and simulations based on three years of Jaguar (No. 1 in Top500) job logs show as much as 85.9% reduction in staging times compared to direct transfers, 75.2% reduction in wait time on scratch, and 2.4% reduction in usage/hour.

***Keywords*-High performance data management, data-staging, HPC center serviceability, end-user data delivery**

I. Introduction

The advent of extremely powerful computing systems, e.g., Petaflop supercomputers, and the data they can process, e.g., from emerging sources such as space observatories and large-scale particle colliders, are pushing the envelope on dataset sizes. For instance, the Large Hadron Collider (LHC) at CERN [1] or the Spallation Neutron Source (SNS) at Oak Ridge National Laboratory (ORNL) [2] will generate petabytes of data. These large datasets are processed by a geographically dispersed user base, often times, on high-end computing systems. Therefore, result output data from High-Performance Computing (HPC) simulations are not the only source that is driving dataset sizes. Input data sizes are also growing many fold [1], [2], [3], [4].

To match the I/O capabilities with the computational power in a supercomputing center, the required input data for a given job is almost always copied to the fast local storage – the scratch parallel file system – at the center before the job is started. This process is commonly referred to as *staging*. Modern applications usually encompass complex analyses, which can involve staging gigabytes to terabytes of input data, using point-to-point transfer tools (e.g., scp, hsi [23]), from observations or experiments. Many times, the applications also involve comparing the above analysis data against large-scale simulation results to see how theoretical models fit real experimental results. Thus, input data can originate from multiple data sources ranging from end-user sites, remote archives (e.g., HPSS [5]), Internet repositories (e.g., NCBI [6], SDSS [3]), collaborating sites and other centers that run pieces of the job workflow (e.g., Figure 1).

Once submitted, the job waits in a *batch queue* at the HPC center until it is selected for running, while the input data "waits" on the scratch space. HPC centers are heavily crowded and it is not uncommon for a job to spend hours—or even days on end—in the queue. In the best case when the data is staged at job submission, the time a job takes to complete, i.e., *(wall_time + wait_time)*, is also the time the input data spends in the scratch space. In the worst case, which is more common, the data wait time is longer as users conservatively (manually) stage in the data much earlier than job submission, let alone job startup.

Scratch space is an expensive commodity, and provisioning and maintaining it usually consumes a notable fraction of the HPC center's operations budget. The cost of scratch is often to the tone of millions of dollars for state-of-the-art supercomputers such as Jaguar [7] (which comprises of 14,000 disks, 192 object storage servers, 1300 object storage targets and 48 controller pairs). More importantly, the scratch space is meant for facilitating currently running or soon to run jobs. From a center standpoint, sub-optimal use of the scratch resource could impact the center's serviceability, i.e., the ability to serve more incoming jobs. That is why, even with a huge scratch capacity, Jaguar administrators constantly trim usage through purge policies and send periodic reminders every week to users to delete their data from scratch. From a user standpoint, the input data is exposed to potential unavailability due to storage system failure [8], [9], [10] while it is waiting for the job to be scheduled. Consequently, when the job is selected for running, crucial pieces of input data may be unavailable, requiring a rescheduling (delay on the order of hours to days). *What is needed is a framework*

that enables timely staging of large input datasets for jobs.

A. Design Challenges

We now present the challenges and issues involved in designing a timely staging service for HPC centers. In order to stage the data to be coincident with job startup, we need intelligent estimates of the following. First, we need to know when the user's job will commence. This has been explored extensively [11], [12], and HPC schedulers (e.g., PBS Pro [13], Moab [14]) can also provide a batch queue wait time estimate based on current and historical (jobs with a similar profile) data. However, a simple and direct use of batch queue predictions in staging is not appropriate due to sudden changes in schedules. For example, an unexpected failure can cause a 10,000 node job to suddenly exit, resulting in many jobs being promoted to "ready to run" state all too quickly. This prospect needs to be factored into the staging mechanism.

Second, we need an estimate of how long the data staging would take from the input locations to the HPC center. We need continuous bandwidth measurements so they can be factored in to revise the route dynamically and adapt to changing network conditions. The upshot is that both the queue wait time estimates and network bandwidth estimates are volatile and "soft". Consequently, our staging solution needs to be resilient to adapt to these transient conditions.

B. Contributions

In this paper, we present a timely staging framework that attempts to have the data available at the scratch storage, from multiple input sources, just before the job is about to run, thereby mitigating the aforementioned issues. The basic idea is to reduce the staging time by proactively bringing the data to intermediate storage locations on the path from the end-user site to the HPC center, then transferring the data to the scratch space as late as possible without delaying the job's scheduled start time.

The novelty of our work is the integration of decentralized data transfer systems into HPC management to improve overall scratch utilization, and not another decentralized transfer mechanism. Thus, the framework uses an innovative combination of high-efficiency data dissemination (BitTorrent [15]) and network monitoring (Network Weather Service (NWS) [16]) to exploit orthogonal, residual bandwidth and to dynamically adapt to network volatility, respectively. Further, the framework constantly adjusts to changes in the predicted job start time, e.g., due to job cancellations or improved estimates. Such dynamic adaptation achieves just-in-time data staging to meet the job's commencement schedule. We stress that despite prior work in developing decentralized data transfer schemes [17], [18], [19], [20], [21], such schemes have not been adopted by HPC centers. Centers, without an exception, still use simple point-to-point protocols, e.g., `scp`, `sftp`, GridFTP [22], hsi [23], etc., due to lack of seamless integration with critical HPC services that the users care about, e.g., job startup prediction, PBS [13], and scratch space. Moreover, no prior work reconciles scratch space consumption with volatility (both network and storage) and timely staging, which is the overarching unique goal of our work.

We have evaluated our solutions using both real-world experiments on PlanetLab [24] as well as extensive simulations using three years worth of job logs from the ORNL Jaguar supercomputer (No. 1 in Top500) [7]. Our approach optimizes precious scratch space usage and minimizes the exposure of input data at center storage. Such an approach is a fundamentally novel way of staging data into HPC centers. Extant data staging techniques are point-to-point, not fault-tolerant and do not factor in scratch space optimization or job startup schedules.

Evaluation of our timely staging framework shows as much as 85.9% reduction in staging times compared to direct transfers, and reduced exposure to scratch failures: 75.2% reduction in wait time on scratch, and 2.4% reduction in usage/hour.

II. Design

In this section, we first present the goals of our timely-staging framework, then we discuss the framework components in detail.

A. Objectives

In designing a timely-staging service for HPC centers, there is a need to reconcile several factors. We highlight these in the following discussion.

Timely delivery of input data: Our primary goal is to deliver application input data to center local storage from multiple sources on time, in the face of both transient network conditions and changing batch queue job wait times. Not properly accounting for such dynamism can have adverse effects on the staging framework: data delivery is delayed and, consequently, job turnaround time is increased.

Minimize transfer times: Ability to minimize transfer times by choosing optimal routes and constantly re-evaluating them is critical for reacting to changes. For instance, optimal routing can mitigate the effect of a sudden tightening in the delivery deadline that can occur due to an unexpected cancellation of a large job.

Reduce duration of scratch space consumption: From a center standpoint, it is desirable to stage the data of a waiting job as late as possible so that the scratch space is available for all of the currently running jobs' I/O (e.g., checkpointing and output). Consequently, if the waiting jobs' duration of scratch usage is reduced, it would help the HPC center better service the currently running jobs.

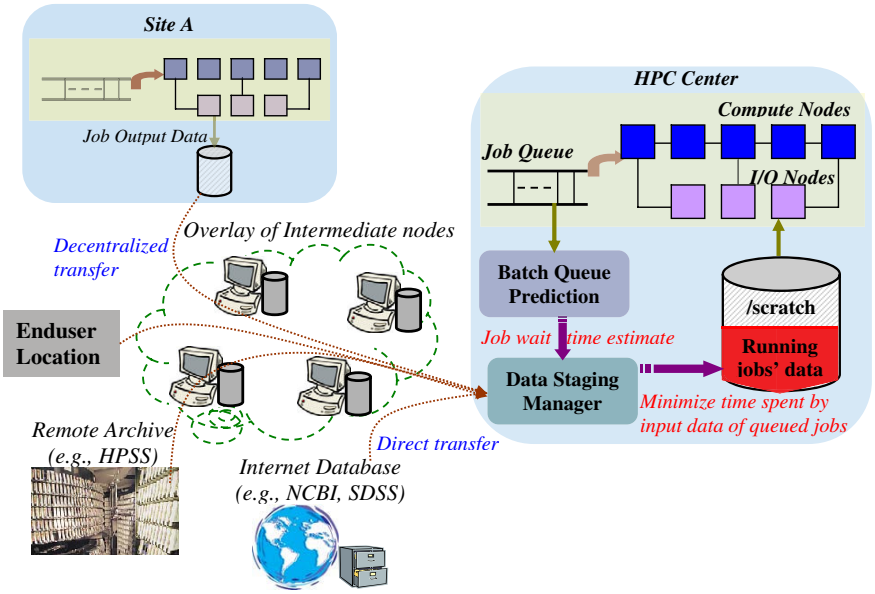


Figure 1. Overview of the timely staging framework, and interactions between the components.

Reduce exposure window: Another downside of staging the data early is its exposure to potential storage system failure. We refer to the time elapsed between when data is staged in until the associated job starts running as exposure window, E_w. To protect against storage failures, it is desirable to minimize E_w, preferably as close to 0 as possible.

The reason for minimizing E_w is stressed by the observation that in a supercomputer with tens of thousands of disks, failure is a norm and not an exception. Scratch is prone to failures due to the sheer number of disks, I/O nodes and controllers used in realizing the system. To put things in perspective, supercomputers such as Jaguar, ASCI Q, ASCI White, and PSC Lemieux all cite storage as a primary reason for system downtime with MTBF of 37.5 hrs, 6.5 hrs, 40 hrs, and 6.5 hrs, respectively [25].

Avoid starvation: Finally, from a center serviceability perspective, it is absolutely essential that the job scheduler not be rendered idle because the input data of a waiting job has not been completely staged-in.

B. Architecture

In the following, we detail our framework components, and how they are integrated to realize timely staging. Figure 1 shows the overall architecture overview, and illustrates interactions between components.

1) Intermediate Nodes: Our framework uses intermediate nodes (N_is) that can provide temporary storage for data on the path from the source to the HPC center. The intuition behind using N_is is that nodes closer to the center than the user site can support faster data transfers for staging and reduce staging times. This provides for delaying the staging to much later than when using a direct transfer, which also reduces E_w.

These nodes can be the user's own collaborating sites, from where other input data can also be staged, ensuring that the data is transferred through a dependable substrate. Using these nodes, the HPC center can also asynchronously retrieve data from other sources, decoupled from the user site. Intermediate nodes provide multiple data flow paths from the user site to the center, which lead to better bandwidth utilization, faster staging speeds, as well as fault-tolerance in the face of failures.

Motivation for collaboration: In today's HPC environment, supercomputing jobs are almost always collaborative in nature. In fact, a quick survey of jobs awarded compute time on the ORNL NLCF, through the DOE's INCITE [26] program, suggests that these jobs involve multiple users from multiple institutions. This collaborative property is even more true in the TeraGrid [27], where jobs are usually from a *virtual organization*, which is a set of geographically dispersed users from different sites, coming together to solve a problem of mutual interest for a certain duration. In such cases, it is clear that many users, from different sites will be interested in seeing the job run to completion, with as little delay as possible. This emerging property of collaborative science can be exploited to perform a collaborative stage-in of job input data. We therefore argue that there exists a *natural incentive* to provide resources and to participate in the timely staging process.

A set of intermediate nodes is typically employed for staging data to a single HPC center at a time. However, the design certainly allows usage scenarios that may include employing the same intermediate nodes to support staging

for multiple HPC centers. We argue against this case, as intermediate nodes are temporarily setup to support a particular collaborative project, and are chosen not by center policies, but by the users' collaborations. Therefore, such overlap across centers is not likely.

Landmark Nodes: The reliance of our design on intermediate nodes exposes the data delivery system to possible failures due to lack of sufficient N_i's. For instance, the end-user site may not have access to any (or sufficient enough) intermediate nodes on the path to the HPC center. This could be either due to the lack of many participating sites in the job or due to the volatility of the intermediate nodes. To avoid such a scenario, we propose to utilize a number of geographically distributed *Landmark nodes* that are always available and can serve as intermediate nodes. The Landmark nodes can be other HPC centers, or nodes along national links such as, Internet2 [28], REDDNET [29], Lambda Rail [30] or the TeraGrid [27] to which many end-users may be connected. The location and number of the Landmarks is determined through out-of-band agreements with the HPC center. For instance, HPC centers can setup such an infrastructure to benefit the whole range of users that it caters to. Consider the following scenario where a collaboration near SDSC (a TeraGrid site) runs a job on the Kraken machine at the University of Tennessee (also a TeraGrid site). An elegant way to dispatch the large input data to the computation would be to exploit the connectivity between the two landmark sites (SDSC and UT) and use the intermediate storage overlay between the end-user and SDSC. A challenging research question to answer is the concerted use of Landmarks and intermediate storage to achieve an efficient data delivery schedule. For instance, a direct GridFTP transfer is well suited for delivering data between two well-endowed sites, whereas a decentralized delivery is better equipped to exploit the intermediate storage. In the following sections, we highlight the use of a combination of direct and decentralized delivery schemes to achieve timely end-user data delivery.

Impact on infrastructure costs: We reiterate that our design does not require the explicit setup and management of landmark and intermediate nodes. Instead, it leverages and "piggybacks" on *existing* infrastructure. Several national testbeds, e.g., TeraGrid [27], REDDNET [29], etc., are already in production and can act as such nodes, without incurring any additional costs such as electricity, manpower, and management costs. Moreover, intermediate nodes use resources that are already part of the "collaborative" job. We also do not require extra provisioning of network bandwidth, rather employ the residual bandwidth that would have otherwise gone unused. Thus, our design also achieves better utilization of resources and possibly a higher system-wide efficiency.

2) Queue Prediction as Staging Deadline: In our design, the HPC center is expected to support a batch queue prediction service (e.g., NWS batch queue prediction [31]), which the users can query before submitting their jobs to get an estimate of queue wait times. Scheduling based on queue wait times is already popular in TeraGrid [27] supercomputer centers. In fact, modern resource managers (e.g., Moab [14]) are beginning to provide services that would enable users to query and obtain start times of queued jobs. The prediction service can usually provide both wait time estimates as well as the probability of a job starting by a user-specified deadline [31]. In cases where direct wait time predictions are unavailable, the user can pose a query to the service, with a deadline, and determine the likelihood of the job starting by the deadline. A 90% or higher probability can be treated as an affirmation of the user-specified deadline and can be used as the job startup time and, consequently, the staging deadline.

However, the job can potentially start earlier than this predicted deadline due to inaccuracies in the prediction or due to failure of other running jobs. Similarly, a lower probability may mean that the job may not commence by the user-specified deadline, but is only an estimate. To accommodate this, we can let the user tweak the estimate by up to a fixed factor, f, moving the deadline earlier. This can be done for one of two reasons: (i) the user may wish to use the prediction with "guarded optimism" to account for jobs starting earlier than estimated; or (ii) may wish to finish staging the data as early as possible by using an artificial tighter deadline, thereby shifting the burden of protecting the data during the prolonged wait time to the center. While (i) is acceptable, (ii) works against the basis of our timely staging; allowing f to be large can unduly affect other jobs, which have genuine tight deadlines. Thus, limiting the adjustment to only a factor is necessary to ensure global fairness in the staging of all jobs. Consequently, the estimate is reported to the *staging manager* so it can ensure that the user-submitted deadlines are within the factor.

3) Timely Staging Algorithm: Once a deadline for completing the input data staging is determined, the user submits a job script to the staging manager at the center with a description of the job and other details necessary for timely staging. The script includes attributes such as the user-adjusted job startup deadline, the set of intermediate nodes, $< \ N_i, P_i >$, where P_i denotes the usage properties of the intermediate nodes N_i, for the decentralized staging process, and the sizes and locations of the input datasets, D_j. The staging manager also takes as input the current snapshot, BW_i, of the observed NWS bandwidth between the HPC center and N_i as well as between the N_i's themselves. The manager reconciles the predicted job start deadline with the user-adjusted one to determine if it can allow the user's tight deadline. This reconciled deadline is denoted by $T_{JobStartup}$. Based on these parameters, the manager decides upon a data staging schedule, X_j, for each D_j, which delivers the dataset in time, $T_j =$

$Min(DirectTransfer, DecentralizedTransfer)$. To estimate these times, the manager uses the measured available bandwidth to the user site as well as the intermediate nodes. To create a distributed schedule, the intermediate nodes are sorted based on available bandwidth and then the number of nodes to which data is sent is increased until overall transfer times that are better than a direct transfer (if possible) can be achieved. This choice is dictated largely by the available bandwidth and storage at the intermediate nodes. When the intermediate nodes can provide a faster transfer, a decentralized transfer is scheduled. Each dataset could come from a variety of sources, including those wherein our decentralized transfer software cannot be installed. In such cases, the manager relies on *just-in-time probes* to the data source to judge if a direct transfer to the HPC center is most appropriate. Alternatively, such input data could also be transferred through the intermediate nodes by having the edge-level nodes pull the data from the source, enabling decentralized staging.

The multi-input stage-in should obviously also complete before job startup and should satisfy the property, $Max(T_j) \leq T_{JobStartup}$. Minimizing transfer times by choosing optimal routes helps achieve this goal. At the same time, each of the input stage-ins, X_j, is also started as late as possible to reduce the duration of scratch space consumption and, consequently, the exposure window, E_w of the datasets. The exposure window for each input dataset is: $E_{wj} = T_{JobStartup} - T_j$. Then, total exposure of all input data is, $E_w = Sum(E_{wj})$. The closer E_w is to 0, the better. Thus, the ideal start time for each input dataset is the one that achieves, $T_{JobStartup} - T_j = 0$. In practice, however, a small difference is desirable to safeguard against unexpected delay. This approach factors in both timely delivery as well as scratch space usage optimization.

4) Feedback and Re-evaluating Staging Decisions: Even after a particular course of action, e.g., decentralized transfer, is chosen, the manager periodically re-evaluates the data staging based on an updated $< N_i, P_i, BW'_i >$, where BW'_i is the latest snapshot of NWS bandwidth measurements. If the re-evaluated time to staging, T'_j, satisfies the property, $T'_j > T_{JobStartup}$, then, alternate routes are taken (if available) to stage the data before job startup, enabling us to meet the staging deadline.

In addition to constantly re-evaluating the network routes based on latest bandwidth measurements, the staging manager also has to account for batch queue status changes as discussed earlier. We address this by having the manager periodically obtain new estimates $T'_{JobStartup}$ from the batch queue service. If the staging schedules reflect that $T_j > T'_{JobStartup}$, then alternate routes are evaluated to ensure timely delivery. This also has the desired side effect of preventing the job scheduler from starvation due to inability to schedule jobs as a result of unfinished stage-ins.

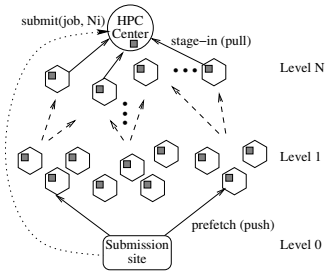


Figure 2. The data flow path from the client site to the HPC center. Each intermediate node (hexagon) runs NWS (gray square) for bandwidth monitoring.

5) Staging and Compute Dependency: Upon receiving a job script, the manager splits it into the compute job and the staging task. The compute job is submitted to the batch queue to ensure that it is in line to start to run by the user deadline. The staging task is placed on a data job queue to start data delivery as necessary. The manager also sets up a dependency such that the compute job does not begin until the staging has finished. To this end, we use and extend our earlier works [32], [33] on instrumenting the job submission system, and the *stagesub* tool used in the ORNL Jaguar machine.

C. Supporting Timely Staging

Once the data staging is initiated, the client chooses a number of nodes from the set of N_i's (fan-out) ordered by available bandwidth. The cardinality of the fan-out is chosen to stage-in all the necessary data before the predicted job start time. These chosen N_i's serve as the Level-1 intermediate nodes. Note that the selected fan-out is not static, and can vary depending on the actual transfer speeds and the impending deadline. The manager monitors the changing bandwidths periodically (using NWS) to determine if a chosen fan-out needs to be increased. Next, the input data is split into chunks and parallel transfer of the chunks to Level-1 nodes is initiated. The transfer may also involve further levels of intermediate nodes (up to Level-N). The choice of the number of levels of intermediate nodes is left to the users, and does not have a direct bearing on the center to Level-N node performance that is critical for our design. The levels simply enable users to provide multiple data-flow paths to the center, and we foresee the levels to be not more than two in typical scenarios. Additionally, depending on the availability of intermediate nodes, the client can also stage the data to Level-N nodes much earlier than the deadline.

As the job startup deadline approaches, the close proximity of the Level-N nodes to the center allows them to quickly move the input data to the center's scratch space. Also, this design allows the Level-N nodes to stage the data at peak (pre-specified) bandwidth at the most appropriate time without worrying about the availability (and connection speed) of the submission site (Figure 2).

Intermediate nodes provide multiple data-flow paths as well as several alternative options for data delivery. For instance, data may be replicated across different N_i's during the transfer from one level to the other. This will allow the center to pull data from a number of locations, thus providing fault tolerance against node failure, as well as better utilization of the available in-bandwidth at the center.

D. Discussion

Recent studies have shown the high rate of storage system failures [8], [9], [10] and the complexity of ensuring reliability in large-scale installations [34], [35], [36] such as the HPC scratch space. Improving reliability in such fixed installations entail going through a rigorous and time-consuming acquisition process mired with delays. In contrast, the collective use of less-reliable individual intermediate nodes can provide a solution that can be arbitrarily grown to accommodate any desired level of reliability. Thus, we argue that although individual intermediate nodes may be more prone to errors compared to individual disks in an HPC center, as a system our approach is able to provide better reliability due to its flexibility. Plus, this reliability comes for free as we use resources volunteered by collaborators, which would otherwise not be used [37].

Alternative design considerations: There are several possible alternative solutions for the HPC staging problem, namely, adding more scratch space, streaming data directly and not using the scratch space, and moving computations closer to data. In the following, we discuss why we did not adopt these options in our design.

First, we reiterate that simply adding more scratch is not practical (Section I), as scratch is a precious commodity and provisioning more scratch means taking dollars away from buying FLOPS, and more FLOPS are how most HPC acquisition proposals are won.

Second, streaming data online and bypassing scratch to support HPC applications is not viable and sustainable (based on Top500 supercomputers). Be it large input/output or checkpoint data, scratch is desirable (and mandated by the HPC center) for its high-speed parallel I/O bandwidth (e.g., Jaguar [7] scratch offers I/O rates of 256 GB/s). A 100,000-core job cannot afford to idle its cores (wasting compute time), waiting for the input data to be streamed in from remote locations. Furthermore, streaming mechanisms cannot match the I/O rates required to keep such large systems busy. In fact, as pointed out earlier, HPC centers spend millions of dollars provisioning and optimizing scratch exactly to avoid this scenario.

Third, moving computation closer to data is a compelling idea, but there are numerous HPC applications, e.g., DOE supercomputer and NSF TeraGrid applications, which cannot be sustained on users' local clusters where data may be available. Our design takes all these factors into consideration for realizing a practical solution to the staging problem.

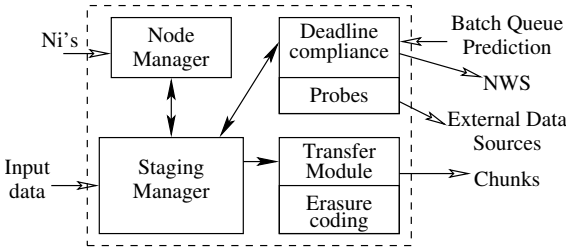


Figure 3. Implementation architecture for timely staging.

```
#PBS -N myjob
#PBS -l nodes=128, walltime=12:00
mpirun -np 128 ~/MyComputation
#Stagein file://SubmissionSite:/home/user/input1
file:///home/scratch/user/input1
#Stagein wget://WebRepo:/input2
file:///home/scratch/user/input2
#InterNode node1.Site1:49665:50GB
#InterNode nodeN.SiteN:49665:30GB
#JobStartDeadline 11/14/2008:12:00
```

Figure 4. An instrumented PBS script for timely staging.

III. Implementation

We have implemented the timely staging manager using about 3500 lines of C code, with the p2p overlay created using FreePastry [38]. Figure 3 shows the overall architecture as well as the interactions between the manager components.

Integration with Job Submission: To facilitate easy adoption of our scheme by the community, we have integrated it with the widely-used PBS [39] job submission system. Specifically, we instrument the job submission scripts to let users specify intermediate nodes and deadlines. An example instrumented PBS script is shown in Figure 4, where the user specifies intermediate nodes and deadlines as well as details such as available storage capacities. The nodes listed in the script are just a suggestion, and the actual runtime queries these nodes directly for availability as needed.

The annotated script is submitted to the staging manager on the center, which filters out the staging-specific directives and forwards the remaining script to the standard batch queue, but with a dependency on the staging task.

Integration with BitTorrent and NWS: We exploit BitTorrent's [15] scatter-gather protocol for transferring data by extending the protocol to use NWS bandwidth measurements. The NWS measurements are integrated with bittorrent to dynamically select locations from where to retrieve a particular dataset, and adapt to changing network behavior by adjusting fan-out to enable staging of data in time. This is in contrast to the standard protocol, which will continue to use a location(s) even if the performance degrades. This allows efficient use of the orthogonal bandwidth, and provides opportunities to improve overall transfer times. The *Staging Manager* creates a "torrent" file for the subset of data to be transmitted to a set of chosen intermediate

nodes. Upon receiving the torrent file, the nodes use the metadata information in the file along with a BitTorrent tracker to "download" the data subset to their local storage. The process is repeated at all the intermediate node levels. When the job is about to run at the center, the Manager can use appropriate torrent files to pull the input data from the intermediate nodes to the center, thus completing the staging process.

Center-wide Global Staging Considerations: Since we anticipate all jobs, along with their staging needs will be submitted through the staging manager, we have instrumented into the manager certain global optimizations that can be performed across all jobs. (1) All jobs that desire a staging to the Level-N, i.e., one hop away from the center, can be started immediately. Since these stage-in operations do not use any center resources — neither occupying scratch space nor consuming bandwidth — the data can be brought closer to the center and pulled in much faster when needed. (2) A job whose startup deadline tightens during the course of a previously initiated stage-in will be given higher priority if it is determined that the staging may not complete in time. For instance, this could mean providing more flows to maximize the last leg of the transfer, using more of the center's incoming bandwidth.

Ensuring Data Reliability: To ensure that data is reliably staged on the center, we employ replication of data by sending out chunks to more than a single location. This is a tunable parameter in our implementation and users can specify the minimum number of replicas that should be created for a given dataset. If necessary, more space-efficient erasure codes can be used. The erasure code that we have used in our implementation is Reed-Solomon [40] in 4:5 coding configuration, i.e., four input chunks are coded to produce five output chunks, with a redundancy of 25%. The chunk-size is also a tunable parameter which can be set based on the size of the datasets being transferred.

Multi-Input Staging: Our implementation is capable of retrieving data from more than a single source, directly as well as incorporating it into the decentralized transfer. The data sources are provided as links in the job-submission script. If the external data source runs an instance of our software, the staging manager can simply use the NWS information to decide between direct or decentralized staging. However, if the external source does not support NWS, the staging manager uses small scale tests, e.g., partial download from a web repository, to determine expected transfer times and make staging decisions. In this case, the goal of the staging manager is to ensure staging of all input data from all sources before the predicted job startup time.

IV. Evaluation

In this section, we present an evaluation of our timely data staging using both our implementation on the PlanetLab testbed [24], and a simulator driven by three-year job-statistics logs from the Jaguar [7] supercomputer. We also compare our results to the popular direct transfer technique that is the default approach for staging input data in many HPC centers.

Table I
AVERAGE OBSERVED BANDWIDTH BETWEEN PLANETLAB NODES DURING EXPERIMENTATION. ALL NUMBERS ARE IN MB/S.

	Center	Client	Level-1	Level-2
Center	-	3.82	-	10.9
Client	3.07	-	5.22	-
Level-1	-	3.86	-	4.22
Level-2	9.47	-	5.66	-

Table II
TRANSFER TIMES (IN SECONDS) USING A DIRECT TRANSFER (`scp`) AND OUR DECENTRALIZED STAGING.

Step	File size		
	1 GB	2 GB	5 GB
Direct	864	2034	5188
Client offload	703	1264	4082
Center pull	155	337	731

A. Implementation Results

First, we use the PlanetLab [24] testbed to study the effectiveness of our decentralized staging in a true distributed environment. We chose 20 PlanetLab nodes arranged in a tree-structure: one as the client site and root of the tree, one as the HPC center, 10 Level-1 nodes, and 8 Level-2 nodes. Table I shows the average bandwidths observed between the nodes during the course of our experiments. Our results represent averages over a set of three runs.

1) Decentralized Staging vs. Direct Transfer: In this experiment, we compare decentralized staging to a point-to-point direct transfer using `scp`. For this purpose, we used a range of file sizes from 1 GB to 5 GB, limited by PlanetLab policies, and measured the time to transfer data under the two schemes. Table II shows the times for direct data transfer from client to HPC center (Direct), from client to Level-1 nodes (Client offload), and from Level-2 to the center (Center Pull). Compared to a direct transfer, the decentralized staging can reduce the last-hop transfer times by 82.1% to 85.9% for 1 GB and 5 GB data sizes, respectively.

This implies that the decentralized staging can delay copying of data to scratch space by a factor of 6.2 on average across the studied file sizes, and still get the data to the center in time for the job to start. Thus, it reduces the time the scratch space has to hold the data, consequently, reducing the exposure window (E_w), and improving center serviceability.

2) Effect of Using NWS Measurements: Next, we compare our NWS-based monitored transfer approach with a standard BitTorrent-based data transfer. In this case, we use

Table III
THE TIME TO TRANSFER A 2 GB FILE USING STANDARD BITTORRENT. THE EQUIVALENT PHASES FOR OUR SCHEME ARE SHOWN IN BRACKETS.

Phase	Time(s)
Send to intermediate nodes (Client offload)	1428
Download at HPC center (Center pull)	362

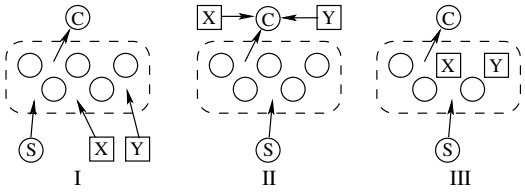


Figure 5. Configurations used in Multi-Input test.

NWS bandwidth measurements to greedily provision Level-2 nodes to increase the fan-in, i.e., the number of nodes simultaneously transferring data to the center, to utilize the maximum center in-bound bandwidth. Table III shows the times taken to deliver a 2.0 GB file using standard BitTorrent protocol. Compare these to the transfer times using our timely staging shown earlier in Table II: both Client offload and Center pull in our approach out-perform by 11.5% and 6.8%, respectively, the corresponding steps in regular BitTorrent transfer. These results show that active bandwidth monitoring provides a good tool for improving staging times.

3) Employing Decentralized Staging: In the above experiments, the bandwidth available between the Level-2 nodes and the center, which dictates Center pull times, is greater than that between the client and the center, which dictates direct transfer time. Thus, the center always decided to perform decentralized staging. In the next experiment, we modified the setup to use a faster node as the client site, and repeated the experiment for staging a 2 GB file. First, we do the transfer without considering direct transfer and always using decentralized staging. Second, we repeat the experiment with the ability to choose between direct and decentralized staging depending on the ability to meet a transfer deadline (job startup). We observed that for the first case, the time to stage and transfer the data to the center was 2867 seconds. In contrast, for the second case the direct transfer completed in 968 seconds, an improvement of 66.2%. This stresses the need for the staging mechanisms to dynamically adjust to the variations in the system behavior, and to not be hard-wired to simply always do a staged transfer or a direct transfer.

4) Multi-Input Staging: Next, we study the ability of our decentralized staging to accommodate input data from multiple sources. We consider three configurations, shown in Figure 5, with two sources (X and Y) of data in addition to the client site (S). In *I*, the data from all sources is staged in a decentralized manner. This captures retrieving data from

Table IV
TRANSFER TIMES (IN SECONDS) FOR MULTI-INPUT DATA UNDER DIRECT AND DECENTRALIZED STAGING.

Step	Configuration		
	I	*II*	*III*
Direct	652	872	844
Client Offload (S)	318	672	740
X offload	646	92	N/A
Y Offload	574	142	N/A
Center Pull	312	158	340
Staging time	312	158	340

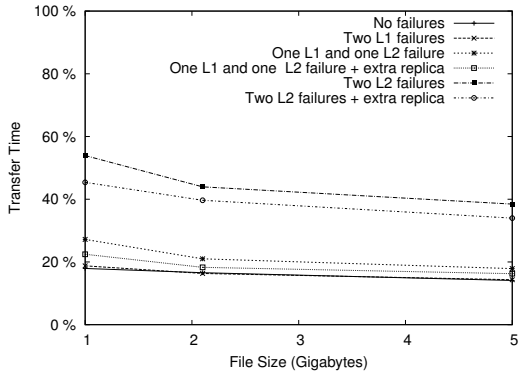
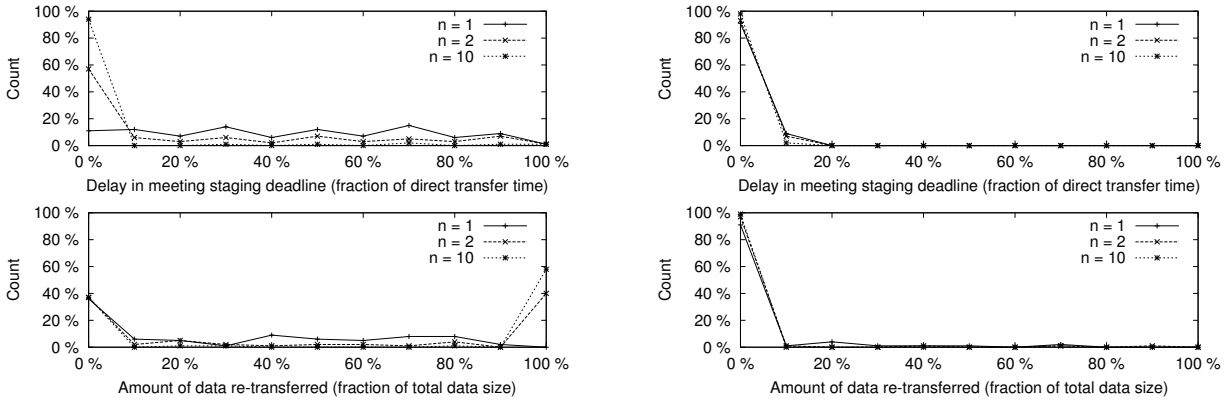


Figure 6. Transfer time as different combinations of Level 1 (L1) and Level 2 (L2) nodes are failed. The results are normalized with respect to a direct transfer.

slower external sources. In *II*, we consider fast external sources, e.g., online data repositories [6] so the center can directly retrieve from them. Finally, in *III*, the intermediate nodes may already have the data, such as collaborating sites in TeraGrid jobs [27]. For each case, we compare a direct transfer from the sources to that of our staging. Table IV shows the results. It is observed that decentralized staging is able to handle multiple sources, and outperform direct transfers by 52.1%, 81.8% and 59.7% for *I*, *II*, and *III*, respectively. Note that in real scenarios, the staging manager will switch between the various configurations depending on the transfer rates and staging deadlines.

5) Behavior Under Failures: Improved transfer times are key to delaying staging, and thus reducing scratch space usage times. Therefore, in the following set of experiments, we study how failures will affect the transfer times under our framework.

First, we examine intermediate node failures. We focus on our decentralized staging, as a failure under direct will result in data transfer to be incomplete by job startup time, consequently leading to obvious job rescheduling. Figure 6 shows transfer time achieved by our approach under various failure scenarios, normalized to direct transfer time. We failed two intermediate nodes under three different scenarios: two Level-1 nodes fail, a Level-1 and a Level-2 node fail, and two Level-2 nodes fail. In this test, the



(a) Direct transfer.

(b) Timely staging.

Figure 7. The distribution of staging delay and re-transmission overhead for 25 transfers with one scratch space failure.

number of replicas at each level is set to 3. The system tolerates two Level-1 failures, i.e., 20% of Level-1 nodes, with negligible affect. A failure at Level-2 increases the transfer time somewhat (by a factor of 1.3), but two Level-2 failures are significantly more disruptive (time increase by a factor of 2.7). However, this is an extreme case with 25% of the Level-2 nodes failing. On the plus side, the transfer time, even with these failures, is less than half (41.2% on average) that of the direct transfer. Furthermore, our flexible design can easily accommodate extra replicas to improve fault tolerance, as observed by the reduction of transfer times for each of the Level-2 failure cases when one extra replica is used. This experiment shows that dynamic rerouting of our approach can adapt to the changing network conditions and ensure meeting the staging deadline with minimal delays if any. Moreover, the use of a flexible routing path between the client site and HPC center allows for offsetting delays due to intermediate node failures.

Next, we examine how failure in scratch space affect the ability of a transfer scheme to meet a given job deadline. Here, we capture the early-transferring approach of users by starting the direct transfers as early as $T_{JobStartup} - n * T_j$, with $1 \leq n \leq 10$. Next, we randomly introduce a single failure on the scratch space between the time of starting the transfer and $T_{JobStartup}$, and determine the delay in meeting the job deadline, as well as the extra amount of data that has to be transferred. For timely staging, we assume perfect prediction, so it starts staging-in data as late as possible for a given file size. The experiment is repeated 25 times using files of sizes from 1 GB to 5 GB, for each studied n. Figure 7 shows the distribution of delay in meeting a deadline and the amount of data re-transferred, respectively. In the distributions, a higher count for a smaller x-axis value is desirable as that implies less delay and higher chances of meeting a deadline, and less data re-transfers. Our timely staging shows excellent properties with 98% of the transfers completing with no delay. In contrast, only a direct transfer that starts as early as with $n = 10$ is able to come close with 94% transfers without delay. With $n = 2$, only 31% of direct transfers complete in time. The flip side is that by staging early, the data remains exposed to the failures on the scratch and possible re-transfers. It is observed that while over 91% of the transfers in our approach had no retransmissions due to exposure to failures, that is only true for 36% of the cases with direct transfers.

Note that since we introduce a single failure, the maximum overhead is 100%. In real scenarios, multiple failures can further exacerbate the problem, as the re-transfer may now take much longer than the earlier transfer or failures in the system may prevent immediate response to a failure. This implies that delaying staging is preferable. Thus, our timely staging is able to withstand failures much close to the job deadline, and the delay if any is small. Such delay can be easily compensated by assuming a slightly tighter deadline than actual, as discussed in Section II.

Table V
STATISTICS ABOUT THE JOB LOGS USED IN THE SIMULATION STUDY.

Duration	22764 Hrs
Number of jobs	80234
Job execution time	30 s to 120892 s, average 5835 s
Input data size	2.28 MB to 3714 GB, average 32.1 GB

B. Simulation Results

In this section, we study the performance of timely staging using job-statistics logs collected over a period of three-years on the Jaguar [7] supercomputer. Table V shows some relevant characteristics of the logs.

To analyze the logs, we have developed a simulator that captures the design of our setup. The simulator models job queuing, scheduling, batch-queue prediction, job execution times, and provides data about scratch space usage and delay

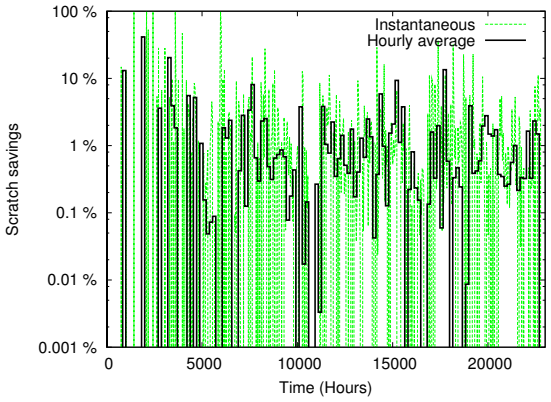


Figure 8. Scratch savings under timely staging compared to direct. Purge period is seven days.

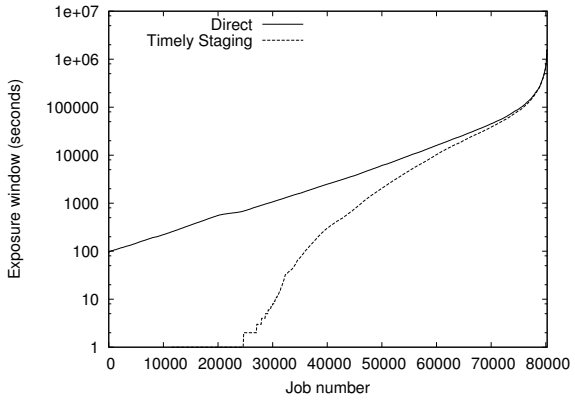


Figure 9. Size of exposure window for each job in the log.

in meeting deadlines. It also models distributed intermediate nodes, their bandwidth variations and decentralized data staging. It uses the connectivity values from PlanetLab nodes and plays the periodic snapshot NWS bandwidth measurements to emulate volatility. In the following, we use this simulator to gain insights into timely staging.

1) Impact on Scratch Space Usage: In this experiment, we quantify the impact of timely staging on scratch space usage. We play the logs in our simulator and determine the amount of scratch used both under direct and timely staging. For this test, we assume that the scratch is empty at the beginning, and use perfect batch queue prediction. Moreover, the center is setup for weekly purges of the scratch space and the maximum center in-bound bandwidth is limited to 10 Gb/s. Only input data is considered, and a data item is only purged if its associated job has completed. Figure 8 shows the instantaneous savings in scratch space usage by timely staging compared to direct, measured every 10 minutes. The instantaneous savings (associated with a job input data) become zero as the job startup time approaches, as timely staging has to bring in the necessary data. A more representative aspect is the average savings over a period of time, as it captures not only the savings but the duration for which the savings were possible. Therefore, we also show the average savings calculated per hour. Finally, we calculated the average savings per hour across the entire log, and found that staging uses 2.43% less scratch per unit of time (e.g. 24.9 GB/Hr on average per Terabyte of storage) compared to direct. Thus, timely staging is a promising way for conserving precious scratch resource.

2) Effect on Exposure Window: In this experiment, we repeat the previous experiment, but now study the exposure window (E_w), i.e., duration for which the data has to wait on the scratch before the associated job is run. Figure 9 shows the observed E_w under direct and timely staging, for each job in our log, arranged in ascending order. For 30.7% of the jobs, timely staging was effectively able to reduce E_w to zero, and for the remaining jobs it reduced E_w by 64.2%, i.e., 75.2% reduction on average across all jobs. Moreover, E_w was reduced by at least a factor of 10 for 48.3% of the jobs. However, it is seen that some jobs ($\approx$ 1.3%) with large E_ws saw only negligible ($<$ 1%) affect from timely staging. The reason for this is that: (i) many jobs require large input data, so the long duration of transfer increases the effective E_w; and (ii) many jobs in our logs arrived in bursts, and timely staging is forced to start transfers early to ensure all necessary data is available and avoid staging errors. Overall, the significantly reduced E_w for most jobs under timely staging shows that it can provide better resiliency against storage system failures and costly re-staging.

3) Effect of Job Startup Time Prediction: In this experiment, we randomly introduce up to 20% variance in the batch queue prediction and the actual job start-up time. Then, we simulate the time by which timely staging will miss the actual job start-up, i.e. staging error. Figure 10 shows the distribution of staging error for different prediction accuracies. The results show the dependence of timely staging on the accuracy of batch queue prediction: as the error in accuracy increases from 0% to 20%, the number of jobs with no staging error reduces from 95% to 75%, i.e., by 21%. However, even with increased prediction error, the number of jobs with significant delays is much less than half (30.6% of the jobs suffer a staging error of more than 1000 seconds). Note that in this test, we assumed that the prediction error remains constant, however, in real scenarios, the accuracy is improved as the start-up time draws near, implying that timely staging will have much improved performance than studied in this case. Finally, the results show that the approach can withstand some prediction errors, and with improved predictions becoming available, can provide better staging alternatives.

V. Related Work

Users either perform out-of-band manual staging, or include the staging commands in the job scripts. Manual

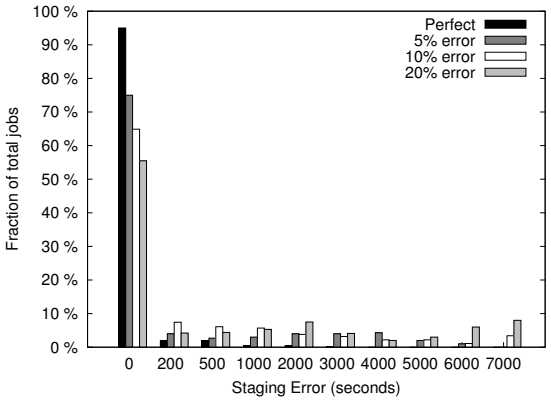


Figure 10. The distribution of staging error under different batch queue prediction accuracy.

staging lacks coordination with job start-up times. Scripted staging wastes compute allocation as allocated cores are waiting while the data is being staged. In our own earlier work, we addressed this to some extent by decoupling data movement from computation and scheduling it separately using a zero-charge data queue [32]. Our work in this paper complements this effort and can be used therein.

PBS Pro [13] supports stage-in requests with and without jobs. In the former, the job is run once the staging finishes. However, if there are other jobs waiting to be run, there is unnecessary scratch space usage and exposure of data. In the latter, prolonged exposure of data is unavoidable until the compute job is submitted. Moab [14] attempts to coordinate staging with job startup. However, these solutions do not adapt the data staging to changes in job startup times. There is no way to expedite the transfer as they only support point-to-point transfer protocols. Consequently, these solutions cannot address network volatility either.

Stork [20] a scheduler for data placement activities in a grid environment, along with Condor [41] and DAGMan [42], is used to schedule data and computation together in the face of vagaries. However, these systems are positioned as a part of the application workflow rather than a set of HPC center integrated services, where our work resides.

BatchAware Distributed File System (BAD-FS [43]) constructs a file system for large, I/O intensive batch jobs on remote clusters. BAD-FS addresses the coordination of input data and computation by exposing distributed file system decisions to an external workload-aware scheduler. We attempt to inherently improve the job workflow without creating a new file system.

Kangaroo [44] uses intermediate buffers in grid transfers, with the goal to provide reliability against transient resource availability. However, it simply provides a staged transfer mechanism and does not address network vagaries. IBP [45] uses a set of strategically placed resources to move data. Our approach also exploits the presence of pre-installed storage nodes. However, it combines both staged and decentralized transfers to deliver data under a deadline.

Systems such as Bullet [17], [18], Shark [19], CoDeeN [46], and CoBlitz [21] have explored the use of multicast and p2p-techniques for transferring large data between multiple Internet nodes. Their focus is on downloading user or multimedia data. Staging requires factoring in center-user agreements and dynamic resource availability, which are not considered in these systems. Downloading large files from several mirror sites has been validated by its wide-spread use in BitTorrent [15], and many other protocols have been proposed [47], [48], [49]. These works are complementary, and we build on their principles, especially BitTorrent.

VI. Conclusion

In this paper, we have presented the design and implementation of a timely staging framework to coincide input data delivery with job startup. Our framework leverages periodic job wait time estimates from a batch queue prediction service, user-specified intermediate nodes, and periodic network bandwidth measurements to deliver input data on time. We use this in conjunction with BitTorrent that we instrumented to use dynamic network monitoring information to adapt to transient network conditions and to tap available residual network bandwidth. Thus, our solution is able to reconcile several key factors such as reduce the duration of scratch space consumption and exposure window, adapt to volatility and deliver the data on time.

Acknowledgments

This research is sponsored by the Laboratory Directed Research and Development Program (LDRD), and the National Center for Computational Sciences (NCCS) of Oak Ridge National Laboratory (ORNL), managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725, and by the U.S. National Science Foundation CAREER Award CCF-0746832.

References

[1] LHC– the large hadron collider. http://lhc.web.cern.ch/lhc/, 2009

[2] Spallation Neutron Source. http://www.sns.gov/, 2008.

[3] Sloan digital sky survey. http://www.sdss.org, 2005.

[4] Laser Interferometer Gravitational-Wave Observatory. http://www.ligo.caltech.edu/, 2008.

[5] R.A. Coyne and R.W. Watson. The parallel i/o architecture of the high-performance storage system (hpss). In *Proc. IEEE MSS Symposium*, 1995.

[6] National center for biotechnology information. http://www.ncbi.nlm.nih.gov/, 2009.

[7] National Center for Computational Sciences. http://www.nccs.gov/, 2008.

[8] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you? In *Proc. USENIX FAST*, 2007.

[9] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *Proc. USENIX FAST*, 2007.

[10] S. Shah and J.G. Elerath. Reliability analysis of disk drive failure mechanisms. In *Proc. RAMS*, 2005.

[11] W. Smith, V.E. Taylor, and I.T. Foster. Using run-time predictions to estimate queue wait times and improve scheduler performance. In *Proc. JSSPP*, 1997.

[12] A. Downey. Using queue time predictions for processor allocation. In *Proc. JSSPP*, 1997.

[13] Pbs pro technical overview: Scheduling and file staging. https://secure.altair.com/sched_staging.html, 2008.

[14] Cluster resources. http://www.clusterresources.com/, 2008.

[15] Bram Cohen. BitTorrent Protocol Specification. http://www.bittorrent.org/protocol.html, 2007.

[16] Rich Wolski, Neil Spring, and Jim Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computing Systems*, 15(5):757–768, 1999.

[17] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. ACM SOSP*, 2003.

[18] Dejan Kostic, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin M. Vahdat. Using random subsets to build scalable network services. In *Proc. 4th USENIX USITS*, 2003.

[19] S. Annapureddy, M. J. Freedman, and D. Mazires. Shark: Scaling file servers via cooperative caching. In *Proc. 2nd USENIX NSDI*, 2005.

[20] T. Kosar and M. Livny. Stork: Making data placement a first class citizen in the grid. In *Proc. ICDCS*, 2004.

[21] KyoungSoo Park and Vivek S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *Proc. 3rd USENIX NSDI*, 2006.

[22] W. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link. The Globus Striped GridFTP framework and server. In *Proc. of Supercomputing*, 2005.

[23] M. Gleicher. HSI: Hierarchical storage interface for HPSS. http://www.hpss-collaboration.org/hpss/HSI/, 2009.

[24] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. ACM HotNets*, 2002.

[25] C. Hsu and W. Feng. A power-aware run-time system for high-performance computing. In *Proc. SC*, 2005.

[26] Department of Energy, Office of Science. Innovative and Novel Computational Impact on Theory and Experiment (INCITE), Jan 2008. http://www.er.doe.gov/ascr/incite/.

[27] TeraGrid. http://www.teragrid.org/, 2009.

[28] Internet2. http://www.internet2.edu/, 2008.

[29] Reddnet. http://www.reddnet.org/, 2009.

[30] National Lambda Rail. http://www.nlr.net/, 2008.

[31] Batch Queue Prediction. http://nws.cs.ucsb.edu/ewiki/nws.php?id=Batch+Queue+Prediction, 2008.

[32] Z. Zhang, C. Wang, S. S. Vazhkudai, X. Ma, G. Pike, J. Cobb, and F. Mueller. Optimizing center performance through coordinated data staging, scheduling and recovery. In *Proc. SC*, 2007.

[33] H. Monti, A.R. Butt, and S.S. Vazhkudai. /scratch as a cache: Rethinking hpc center scratch storage. In *Proc. ICS*, 2009.

[34] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proc. SIGMETRICS*, 2007.

[35] Ilias Iliadis, Robert Haas, Xiao-Yu Hu, and Evangelos Eleftheriou. Disk scrubbing versus intra-disk redundancy for high-reliability raid storage systems. In *Proc. SIGMETRICS*, 2008.

[36] Alma Riska and Erik Riedel. Idle read after write: Iraw. In *Proc. USENIX ATC*, 2008.

[37] Ali R. Butt, Troy A. Johnson, Yili Zheng, and Y. Charlie Hu. Kosha: A peer-to-peer enhancement for the network file system. *Journal of Grid Computing: Special issue on Global and Peer-to-Peer Computing*, 4(3):323–341, 2006.

[38] Druschel et. al. Freepastry. http://freepastry.rice.edu/, 2004.

[39] Albeaus Bayucan, Robert L. Henderson, Casimir Lesiak, Bhroam Mann, Tom Proett, and Dave Tweten. Portable Batch System: External reference specification. November 1999. http://www-unix.mcs.anl.gov/openpbs/docs/v2_2_ers.pdf.

[40] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, 1997.

[41] M. J M. J. Litzkow, M Livny, and M. W Mutka. Condor - A hunter of idle workstations. In *Proc. ICDCS*, 1988.

[42] Directed acyclic graph manager. http://www.cs.wisc.edu/condor/dagman/, 2007.

[43] J. Bent, D. Thain, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Explicit control in a batch aware distributed file system. In *Proc. NSDI*, 2004.

[44] D. Thain, S. Son J. Basney, and M. Livny. The kangaroo approach to data movement on the grid. In *Proc. HPDC*, 2001.

[45] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the network. In *Proc. Network Storage Symposium*, 1999.

[46] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Proc. USENIX ATC*, 2004.

[47] P. Rodriguez, A. Kirpal, and E. W. Biersack. Parallel-access for mirror sites in the internet. In *Proc. IEEE Infocom*, 2000.

[48] James S. Plank, Scott Atchley, Ying Ding, and Micah Beck. Algorithms for high performance, wide-area distributed file downloads. *Parallel Processing Letters*, 13(2):207–224, 2003.

[49] Rebecca L. Collins and James S. Plank. Downloading replicated, wide-area files – a framework and empirical evaluation. In *Proc. Symposium on Network Computing*, 2004.