DISC: A System for Distributed Data Intensive Scientific Computing

George Kola, Tevfik Kosar, Jaime Frey, Miron Livny
Computer Sciences Department
University of Wisconsin-Madison
kola,kosart,jfrey,miron@cs.wisc.edu

Robert Brunner
Department of Astronomy and NCSA
University of Illinois at Urbana-Champaign
rb@astro.uiuc.edu

Michael Remijan
NCSA
University of Illinois at Urbana-Champaign
remijan@ncsa.uiuc.edu

Abstract

The increasing computation and data requirements of scientific applications have necessitated the use of distributed resources owned by collaborating parties. While existing distributed systems work well for computation that requires limited data movement, they fail in unexpected ways when the computation accesses, creates, and moves large amounts of data over wide-area networks. In this work, we analyzed the problems with existing systems and used the result of this analysis to design our own system. Realizing that it takes a long while for a new system to stabilize, we tried our best to reuse existing components. We added new components only when we could not get by with adding features to existing ones. We used our system to successfully process three terabytes of DPOSS image data in under a week by using idle CPUs in desktops and commodity clusters in the UW-Madison Computer Science Department and Starlight.

1 Introduction

The scientific community has been collaborating to solve hard problems, such as finding the Higgs Boson [5] and mapping the human genome. The solutions involve a large amount of computation beyond the scope of a single organization. This has necessitated the use of distributed resources owned by collaborating parties. While existing distributed systems work well for computation that requires limited data movement, they fail in unexpected ways when the computation accesses, creates, and moves large amounts of data.

Typical scientific applications consist of multiple stages of processing performed in a pipelined manner. The data transfer between stages is mostly done via pipes and files. Thain et al. [25] mention the process in detail.

Scientists want to run a large number of instances of the pipeline, each of them operating on independent data. For instance, NCSA astronomers search for bright galaxies by running 2611 instances of the Sextractor [1] pipeline on the DPOSS [8] dataset with each pipeline accessing a different 1.1 GB image. This process can be more complex, with feedback from one set of pipelines affecting the next set. The BMRB [2] BLAST [22] pipeline uses the result of a set of protein sequence matches to refine the next set of searches.

In this work, we analyzed the problems with existing systems and used it to design our system. Realizing that it takes a long time for a new system to stabilize, we tried our best to reuse existing components and added new components only when we could not get by with adding features to existing ones. We used our system to successfully process three terabytes of DPOSS image data in under a week by using idle CPUs in desktops and commodity clusters in the UW-Madison Computer Science Department and Starlight, a network access point located in Chicago.

2 Limitations of Current Systems

Some of the limitations of the current systems in handling data intensive distributed computation are as follows.

Data movement part of computation. Existing systems closely couple data movement and computation. This results in re-computation whenever the output data transfer fails. This is especially bad for data intensive applications, where the transfers have a higher likelihood of failure because they move large amounts of data. Further, if there are intermittent network failures, we may be unable to complete the transfer unless we can resume the previous failed transfer.

Data movement not scheduled. There is no scheduling of data movement. This has resulted in storage server thrashing/crashing when a large number of compute nodes read/write data from/to it respectively. When a large number of compute nodes write data to a storage server, it acts as a distributed denial of service attack.

Inability to distinguish between transient and permanent failures. Current systems do not differentiate between transient and permanent failures and apply the same strategy to both, resulting in the system being unable to handle each class of failure appropriately. For instance, some systems waste resources trying to recover from a permanent failure, which results in a considerable delay before the user notices the problem. At other times, they give up too soon on transient failures, resulting in application level failure that needs user intervention to fix. For instance, many data transfers fail because of temporary network outage and temporary non-availability of a storage server and users would prefer the system to automatically recover from such transient failures.

Need artificial dependencies to prevent overcommitting of resources. Existing systems allow users to specify dependencies between jobs. However, they do not allow them to restrict the number of concurrent jobs of a certain class. Many applications need to restrict the number of concurrent jobs in a certain processing stage, as the jobs can open only so many database connections without overwhelming the server, or because of space limitations in the shared fileserver. To accomplish this, users have to introduce artificial dependencies between jobs/pipelines. In this case, a failure of one pipeline may prevent a set of other independent pipelines from executing because of the artificial dependency, resulting in sub-optimal throughput.

Inability to set priority between pipelines. Users want some results before other results. Current systems do not support priority between pipelines. Some support job-level priority but they do not provide a higher-level interface to set priority between pipelines. They require manual mapping from pipeline-level priority to job-level priority. Further, users may want to change the priorities between pipelines dynamically. For instance, the user may have made a mistake in a certain pipeline and when he notices the failure, he may fix it and then want the fixed pipeline to complete before some of the earlier submitted pipelines. Users cannot easily accomplish this in existing systems.

Inability to execute alternative pipeline. There is newer and/or alternative software that is faster but may fail on certain inputs. Users want to use the faster software when it works and switch to the slower, more reliable one when the faster one fails. Current systems do not support this.

Inability to dynamically balance load across multi-domain resources. The current systems do not dynamically load balance across multi-domain resources. Part of the problem is that the pipeline may be different depending on the domain. If a system is to dynamically load balance across domains, it must be able to choose the appropriate pipeline depending on the domain. Current systems cannot do that and users who want to load balance across domains have to do it statically.

Inability to dynamically adapt. Many tunable parameters depend on the current state of the network, server, and other components involved in the pipeline. Ideally, the system should be able to figure this out and adapt the application. A low-level example is that the TCP buffer size should be set equal to the bandwidth delay product to utilize the full bandwidth. A higher-level example is that to maximize throughput of a storage server, the number of concurrent data transfers should be controlled taking into account server, end host, and network characteristics. Current systems do not perform automated tuning.

3 Related Work

GridDB [19] is a grid middleware based on a data-centric model for representing workflows and their data. GridDB provides users with a relational interface through a three-tiered programming model combining procedural programs (tier 1) and their data through a functional composition language (tier 2). The relational interface (tier 3) provides an SQL-like query and data manipulation language and data definition capability. GridDB allows prioritization of parts of a computation through a tabular interface. GridDB is at a higher-level than our system. It presents a data-centric view to the user and uses the Condor [18]/Condor-G [11] batch scheduling system underneath. Since our system is a transparent layer above Condor/Condor-G, GridDB can easily use our system for data intensive applications and benefit from improved throughput, fault-tolerance, and failure handling.

Chimera [10] is a virtual data system for representing, querying, and automating data derivation. It provides a catalog that can be used by application environments to describe a set of application programs ("transformations"), and then track all the data files produced by executing those applications ("derivations"). Chimera contains a mechanism to locate the "recipe" to produce a given logical file, in the form of an abstract program execution graph. The Pegasus planner [7] maps Chimera's abstract workflow into a concrete workflow DAG that the DAGMan [6] meta-scheduler executes. DAGMan is a popular workflow scheduler and our system addresses the deficiencies of DAGMan and provides new capabilities that considerably improve fault-tolerance, increase throughput, and greatly enhance user experience. Thus, Pegasus and Chimera would benefit from our system.

4 Framework

We set out to address all the problems with existing systems mentioned in the previous section.

First, we define some terms used in our work. 'Pipeline Identifier' is a string that uniquely identifies the data to be processed by one pipeline within a dataset. 'Pipeline Generator' is a program that takes a pipeline identifier and generates the workflow to process that data. The workflow is represented as a directed acyclic graph(DAG) of processing stages in the same manner as DAGMan. For the NCSA image processing, the pipeline identifier is a string that specifies the source directory of the dataset, the filename of the 1.1 GB image file to be processed, the destination directory for the result, and the file transfer protocol to use.

Figure 1 shows an overview of our system. The user first supplies a set of pipeline generators. Individual execution domains can have their own pipeline generators. The user

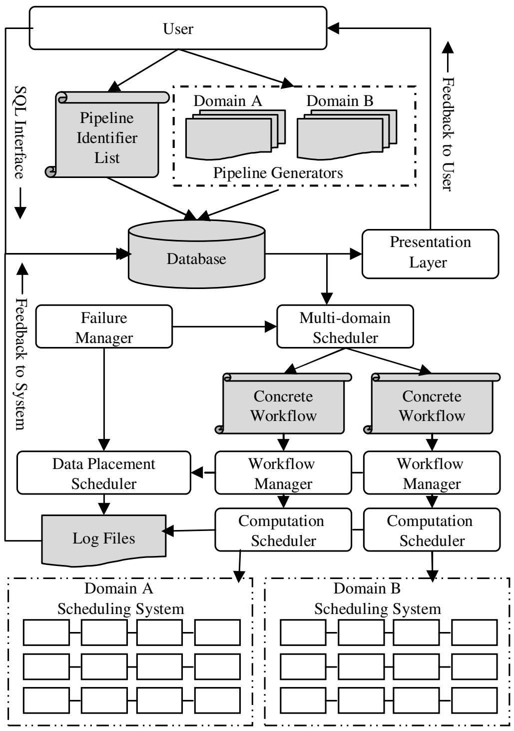

Figure 1: Shows the components of our system

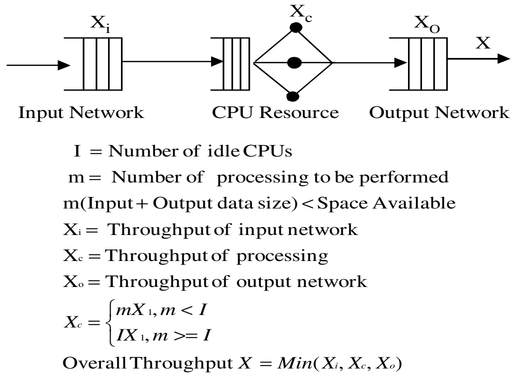

Figure 2: Shows the analytical model of processing in a domain

can also define a default set of generators that are used for domains that do not have their own generators. For each domain, the user can specify a list of alternative generators sorted by preference order. The alternative generators usually perform equivalent processing using different programs or different versions of the same program. It enables users to use newer and faster programs when they work and switch to slower and more reliable programs when the faster ones fail. The pipeline generators are stored in a database, allowing them to be set up once and used repeatedly by multiple users.

Whenever a user wants to process some data, he specifies a list of pipeline identifiers, their priority, and the pipeline generators to be used. The user may also specify the domains to be used to execute these pipelines. Otherwise, the system uses only the local domain. The user can change the priority of the pipeline anytime.

The multi-domain scheduler determines the available CPU, storage, and network resources in each domain and uses that to dynamically assign pipelines to domains in the order of pipeline priority. It then invokes the appropriate pipeline generator and submits the resulting workflow to our workflow manager. The scheduler monitors the progress of executing pipelines and uses that to dynamically assign work. The net result is that each domain gets work according to its capability and users see a much shorter turn-around time.

To provide fault-isolation, we decouple data placement and computation, making data placement a full-fledged job. Thus, data placement failures do not result in computation failures and vice-versa. We schedule the data placement jobs taking into account end-host and storage server characteristics. Our concrete workflow DAG specifies two types of jobs: computation jobs and data placement jobs. Our workflow manager submits computation jobs to Condor-G and data placement jobs to Stork [17], our prototype data-placement scheduler.

We parse Stork and Condor-G job log files and store them in the database. We extended the methodology developed in our previous work [14], where we entered Condor log data into a relational database. This allows us to efficiently access past performance data to aid in future scheduling decisions.

The multi-domain scheduler is responsible for load balancing across domains. We built an analytical model, shown in figure 2, for calculating the throughput of a domain. We split the execution of a pipeline into three stages: stage-in, processing, and stage-out. The throughput of the stage-in process depends on the input network, source disk, and destination disk. We have profilers that calculate the data rate sustained by the source and destination disks. Using network bandwidth estimation tools [9,13], we estimate the input network bandwidth to the domain. The input data rate is the minimum of the source disk, destination disk, and network data rates. Similarly, we calculate the output data rate. Knowing the average input and output file sizes, we can calculate the throughput of the input network and output network. We needed to use a load dependent server to model the CPU because the throughput keeps increasing until the number of processing instances becomes equal to the number of idle processors. Space availability may limit the number of concurrent processing instances. Taking into account space availability, number of idle CPUs, and average processing time, we can estimate the throughput of the CPU resource. As we pipeline the three stages (overlap execution of these three stages), the overall throughput is the minimum of the throughput of the three stages. During the execution of each pipeline, the throughput of each stage is measured. We use these measurements to refine the model and get a better estimate of the overall throughput.

We assign pipelines to domains in proportion to their estimated throughput. The throughput estimate gets refined dynamically as the pipelines execute. In steady state, the work assigned to each domain is the same as the throughput of the domain. When failures occur, they affect the throughput of one or more stages and hence the overall throughput of the domain. Because the throughput drops, the multi-domain scheduler would assign new work proportional to the new throughput. If there is a backlog because of a drop in throughput, we do not assign new work until the backlog is cleared.

Towards the end of a run, certain domains may not have any pipelines to run while others may still have a backlog. The continuous dynamic estimation reduces the amount of backlog, but the actual amount depends on the scale of fluctuation and failures. For instance, towards the end if the number of available CPUs in a domain drops from hundred to five, there may be a backlog in that domain. We can safely assign unstarted processing to another domain. However, this is not work conserving, as we have to pay the cost for shipping the data to a different domain.

The system can run extra instances of the unfinished pipelines on the idle resources, killing all other instances when one completes. We must be careful that the final data transfers of two instances do not conflict, for example, by writing to the same file. We handle this with a namespace mapping technique (writing to a different directory and doing a move with the condition that the first domain that has successfully transferred the whole data wins). This may not be safe as some programs may modify permanent state outside the system's knowledge and control (e.g. directly modifying a central database). Simultaneously executing two instances of such a program may corrupt the shared data. Therefore, we require the user to explicitly enable this optimization. We provide the mechanism to reduce the completion time and let users specify policies on what they want to do: conserve work, amount of extra work to perform to reduce turn-around time, etc.

Towards the end, when there are fewer pipelines than available resources, we can use the analytical model and assign work to domains in such a way as to minimize the turn around time. This optimization requires further exploration.

One of the key issues that we wanted to address is the separation of system and application failures. To handle this we have developed a failure manager that can classify failures as transient or permanent and, consulting user-specified policies, handle each failure appropriately. We developed the failure manager from our earlier work on Phoenix [15], a fault-tolerant middleware layer. Details of the classification mechanism are described in [15]. If the user specifies alternative pipelines, the system switches to an alternative pipeline on application failure. At times, a data transfer may fail and if it is a transient failure (e.g.

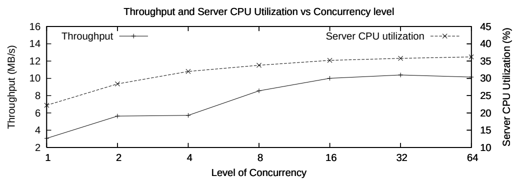

Figure 3: Shows how an agent can tune the concurrency level

network outage), the failure manager detects that and suggests a suitable strategy to Stork. For instance, if the transfer failed because of a network outage, the failure manager would ask Stork to try to resume the transfer with exponential back off between failures. If the transfer failed because of a protocol server crash, the failure manager would suggest an alternative protocol if that is available.

To handle concurrency issues, we have a concurrency manager. Users can restrict the number of concurrent pipelines both globally and within each domain. This may be necessary to prevent overloading of shared resources that our system is unaware of, such as a central database accessed by the user program. Such throttling does away the need to create artificial dependencies and improves throughput in the presence of failures.

Figure 3 shows a sample throughput and server CPU utilization with respect to concurrency level. An agent can monitor this and dynamically tune the concurrency level to achieve the user desired metric. In the simplistic case, the agent can tweak the concurrency level and set it to the minimum concurrency level to achieve the maximum throughput. When the server load becomes too high, it can lower the concurrency level to reduce the load.

Our system is fully backward compatible with DAGMan/Condor-G, used by most grid users [12]. If users want to use the old interface of the dependency manger, they would have to forgo multi-domain load balancing, setting priority between pipelines, and support for alternative pipelines, but they would still benefit from the other features. If they are willing to slightly change their setup and use our new interface, which is very similar to the existing one, they benefit from all our features.

5 Evaluation

5.1 NCSA DPOSS Image Processing

NCSA astronomers wanted to search for bright galaxies in the three terabyte DPOSS [8] dataset. Their two main requirements were fully automated processing and short turn around time. As we were building our system, we interacted with them and the generated interest resulted in collaboration.

The DPOSS dataset was stored in NCSA's UniTree [3] mass storage system. The astronomers had access to only a couple of shared dual Xeon processor machines and were in the process of acquiring resources to process the DPOSS data. They decided to try our system hoping to utilize idle

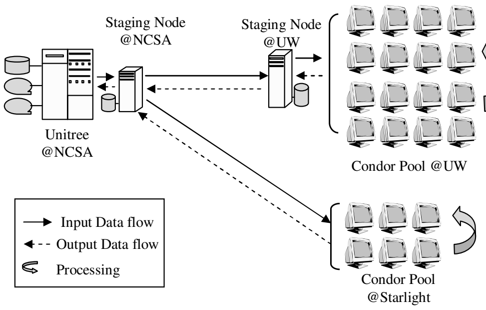

Figure 4: Shows the overview of our system to process DPOSS data stored in NCSA UniTree

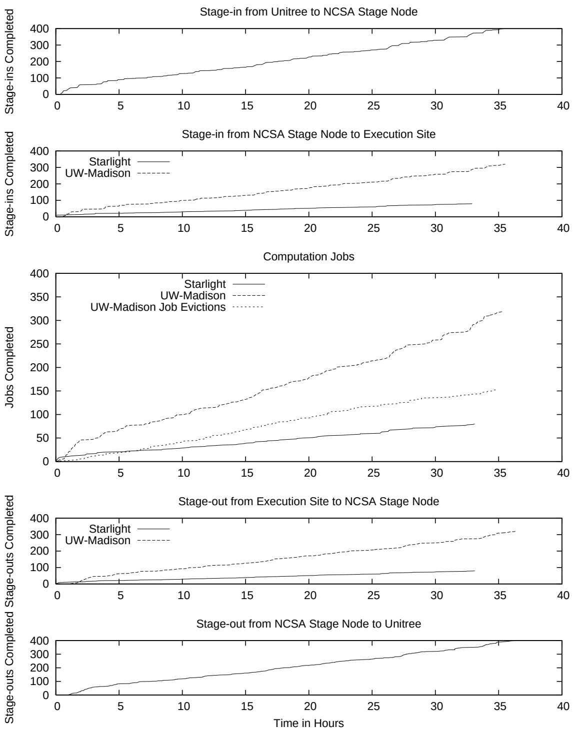

Figure 5: Throughput of the System

CPUs to perform their computation.

Processing time for the initial analysis of a single image varies from half-an-hour to two-and-a-half hours. After initial analysis, astronomers want more a more detailed processing, taking up to an order of magnitude longer, for some of the images. The wide area transfer took less than three minutes. The file transfer from the mass-storage server depended on whether the file was in the cache or on tape resulting in a high variance. This makes overlap of stage-in and processing, which our system performs, very beneficial.

We installed a prototype of our system (Figure 4). We used one of the dual Xeon machines as a staging node. We staged data from the NCSA mass-storage server to the local staging node, which had 20 GB of free disk space. Our plan was to use idle CPUs at the UW-Madison Computer Science Department and Starlight.

We had access to eight CPUs at Starlight and over a thousand CPUs at the UW-Madison Computer Science Department main condor pool. We had to use different strategies for the two domains. At UW-Madison, we used desktop CPUs, so an interactive user could evict our job when he started using the machine. In this case, directly transferring data to the compute node was not such a good idea, as an eviction would result in re-transfer over the wide-area network. We handled this by using a staging node at UW-Madison. At Starlight, the eviction was more static. There would be slots where users would use all the CPUs for experiments. Starlight lets us use the CPUs when there is no scheduled experiment. This meant that if an experiment started, we would be unable to get the Starlight CPUs for a long time.

We also used different protocols for the data transfer. Since Starlight nodes did not have GridFTP servers, we installed diskrouter [16] clients and used them. We used GridFTP for data transfer between the NCSA and UW-Madison staging nodes. Our system can handle any data transfer protocol and this multi-protocol setup validates that. Using a different protocol for each domain tests the system's domain specific pipeline capability.

Figure 5 shows the throughput of the processing of 400 images in a 36 hour period. Starlight had a consistent throughput of 2.4 jobs per hour until around 33 hours, when the machines became unavailable to us. UW-Madison had higher throughput but more fluctuation as well. The space constraints on the NCSA staging node created a backpressure and slowed down the stage-in to that node. The whole process was fully automated and did not require any human intervention. With some tune-ups, we were able to process the whole 3-terabyte DPOSS dataset in under a week, making this one of the fastest astronomy image processing systems.

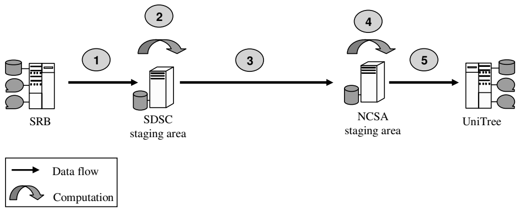

Figure 6: Shows the overview of our system to replicate and preprocess data stored in SRB server at SDSC

5.2 Data Set Replication and Preliminary Processing

We also evaluated our system on a data set replication and preliminary processing. The data set was residing at the SRB mass storage server at SDSC. We had to replicate that data set to the UniTree mass storage server at NCSA.

Since, the mass storage systems did not have a common interface, we had to use intermediate nodes to perform protocol translation. We also did some preliminary processing at SDSC and NCSA. Such combined data movement and processing are

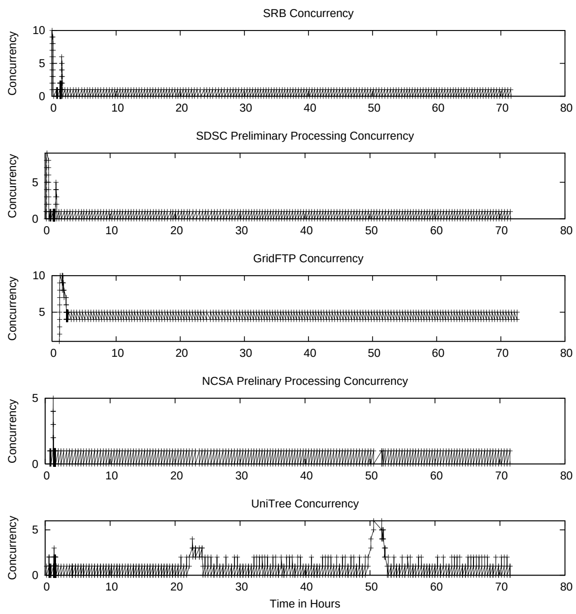

Figure 7: Shows the concurrency level of different stages of our system used to replicate and preprocess data stored in SRB server at SDSC

a common requirement for data set replication among collaborating sites. The preliminary processing could be just a simple checksum generation and verification or it could be generation of site-specific metadata to identify the contents of the dataset or any domain specific processing.

Figure 6 shows the process of moving 550 GB of data from SRB mass storage at SDSC to UniTree mass storage at NCSA using two stage nodes: one at SDSC and other at NCSA. We performed preliminary processing at the SDSC stage node and the NCSA stage node and transferred the results to UniTree. Figure 7 shows the concurrency level in the different stages of the system. We maintained a concurrency level of five for GridFTP to maximize the wide-area throughput. An SRB concurrency level of one was sufficient to sustain this wide area concurrency level. UniTree transfers have a higher concurrency level because we need to transfer the results of preliminary processing in addition to the source file. At around 50 hours, UniTree had a slowdown, possibly because of maintenance or a high priority job. The system continued operating and successfully transferred data. The system has backpressure, created by space limitation on the stage nodes, so a slowdown of one component would gradually slow down other components, preventing overfill of disks and the resultant failures.

6 Future Work

We are in the process of using the system across a larger number of sites. NCSA astronomers want to make our system the basis of their astronomy cyber infrastructure. They are planning to use it to process petabytes of data from Quest2 [21], CARMA [4], NOAO [23], NRAO [24], and LSST [20] datasets.

7 Conclusions

We have analyzed the problems with the existing distributed scheduling systems and used the result of this analysis to design our system. Our system considers data movement as full-fledged jobs, differentiates between permanent and transient failures, automatically recovers from transient failures, and dynamically adapts itself to the changing environment. We have shown how we used our system to successfully process three terabytes of DPOSS image data in under a week by using idle CPUs in desktops and commodity clusters in the UW-Madison Computer Science Department and Starlight, and presented the results of this study.

References

[1] E. Bertin. Sextractor – astronomical source extractor. http://terapix.iap.fr/rubrique.php?id_rubrique=91/, 2004.

[2] BioMagResBank. A repository for data from NMR spectroscopy on proteins, peptides, and nucleic acids. http://www.bmrb.wisc.edu/, 2004.

[3] M. Butler, R. Pennington, and J. A. Terstriep. Mass storage at NCSA: SGI DMF and HP UniTree. In *Proceedings of 40th Cray User Group Conference*, 1998.

[4] CARMA. The combined array for research in millimeter-wave astronomy. http://www.mmarray.org/, 2004.

[5] CMS. The Compact Muon Solenoid Project. http://cmsinfo. cern.ch/.

[6] Condor. The Directed Acyclic Graph Manager. http://www.cs.wisc.edu/condor/dagman, 2003.

[7] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. Pegasus: Planning for execution in grids. Technical Report 20, GriPhyN, 2002.

[8] S. G. Djorgovski, R. R. Gal, S. C. Odewahn, R. R. de Carvalho, R. Brunner, G. Longo, and R. Scaramella. The Digital Palomar Sky Survey (DPOSS). *Wide Field Surveys in Cosmology*, 1998.

[9] C. Dovrolis, P. Ramanathan, and D. Moore. What do packet dispersion techniques measure? In *INFOCOMM*, 2001.

[10] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *14th International Conference on Scientific and Statistical Database Management (SSDBM 2002)*, Edinburgh, Scotland, July 2002.

[11] J. Frey, T. Tannenbaum, I. Foster, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grid. In *Tenth IEEE Symposium on High Performance Distributed Computing*, San Francisco, CA, August 2001.

[12] R. Gardner. The Grid2003 production grid: Principles and practice. In *Proceedings of the Thirteenth IEEE Symposium on High Performance Distributed Computing*, Honolulu, Hawaii, June 2004.

[13] M. Jain and C. Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with tcp throughput. In *Proceedings of SIGCOMM*, Pittsburgh, PA, August 2002.

[14] G. Kola, T. Kosar, and M. Livny. A client-centric grid knowledgebase. In *Proceedings of Cluster 2004*, San Diego, CA, September 2004.

[15] G. Kola, T. Kosar, and M. Livny. Phoenix: Making data-intensive grid applications fault-tolerant. In *Proceedings of 5th IEEE/ACM International Workshop on Grid Computing*, 2004.

[16] G. Kola and M. Livny. Diskrouter: A flexible infrastructure for high Performance Large Scale Data Transfers. Technical Report 1484, University of Wisconsin-Madison Computer Sciences Department, 2003.

[17] T. Kosar and M. Livny. Stork: Making data placement a first class citizen in the grid. In *Proceedings of 24th IEEE International Conference on Distributed Computing Systems (ICDCS2004)*, Tokyo, Japan, March 2004.

[18] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, 1988.

[19] D. T. Liu and M. J. Franklin. Griddb: Data-centric services in scientific grids. In *The Second Workshop on Semantics in Peer-to-Peer and Grid Computing*, New York, May 2004.

[20] LSST. The large-aperture synoptic survey telescope. http://www.lsst.org/, 2004.

[21] A. Mahabal, S. G. Djorgovski, M. Graham, R. Williams, B. Granett, M. Bogosavljevic, C. Baltay, D. Rabinowitz, A. Bauer, P. Andrews, N. Morgan, J. Snyder, N. Ellman, R. Brunner, A. W. Rengstorf, J. Musser, M. Gebhard, and S. Mufson. The Palomar-Quest synoptic sky survey. 2003.

[22] NCBI. Blast project. http://www.ncbi.nlm.nih.gov/BLAST/.

[23] NOAO. National optical astronomy observatory. http://www.noao.edu/, 2004.

[24] NOAO. National rado astronomy observatory. http://www.tuc.nrao.edu/, 2004.

[25] D. Thain, J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Pipeline and batch sharing in grid workloads. In *Proceedings of the Twelfth IEEE Symposium on High Performance Distributed Computing*, Seattle, WA, June 2003.