Multi-Replication with Intelligent Staging in Data-Intensive Grid Applications

Yuya Machida [#1], Shin'ichiro Takizawa [#2], Hidemoto Nakada [*#3], Satoshi Matsuoka [#$4]

[#] *Tokyo Institute of Technology*
2-12-1 Ookayama, Tokyo, 152-8550, Japan
[1] machida@matsulab.is.titech.ac.jp
[2] takizawa@matsulab.is.titech.ac.jp
[4] matsu@is.titech.ac.jp

[*] *National Institute of Advanced Industrial Science and Technology (AIST)*
1-1-1 Umezono, Tsukuba, 305-8568, Japan
[3] hide-nakada@aist.go.jp

[$] *National Institute of Informatics*
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430, Japan

***Abstract*— Existing data grid scheduling systems handle huge data I/O via replica location services coupled with simple staging, decoupled from scheduling of computing tasks. However, when the application/workflow scales, we observe considerable degradations in performance, compared to processing within a tightly-coupled cluster. For example, when numerous nodes access the same set of files simultaneously, major performance degradation occurs even if replicas are used, due to bottlenecks that manifest in the infrastructure. Instead of resorting to expensive solutions such as parallel file systems, we propose alleviating the situation by tightly coupling replica and data transfer management with computation scheduling. In particular we propose three techniques: (1) dynamic aggregation and O(1) replication of data-staging requests across multiple nodes using a multi-replication framework, (2) replica-centric scheduling — data re-use and time-to-replication as compute scheduling metrics on the grid and (3) overlapped execution of data staging and compute bound tasks. Early benchmark results implemented in our prototype Condor-like grid scheduling system demonstrate that the techniques are quite effective in eliminating much of the overhead in data transfers in many cases.**

I. Introduction

Recent workloads in data-intensive applications on the Grid such as HEP, astronomy, life-science, etc., consist of huge data sets that require abundant computational power to process them. For example, a typical BLAST [1] job requires hours of aggregate compute time over hundreds of tasks and processors for multi-dimensional, complex sequence comparisons over gigabyte of data sets, typically submitted as an grid workflow batch-job where the scheduling system allocates the numerous parallel subjobs onto appropriate compute hosts. There, data in the file or database on the grid are either fetched as a remote file from a shared file system, or simply remotely staged to the scheduled machine using copying methods such as GridFTP underneath a replica location service (RLS).

However, with increasing size of the data sets and processing required, simple shared file system or staging could result in a significant overhead and under-utilization of the grid infrastructure as we exemplify. Compare the execution time and efficiency when we submit 80 BLAST tasks to 16 compute hosts managed by our Condor-like scheduling system Jay (described later), so that each compute host executes 5 tasks on average, and they all use the same genomic database file. We compare the number of running tasks under the following 3 settings (details given in Section IV).

Shared file system:
: Use a single NFS (Network File System) server node to emulate a shared grid file system where the database file is stored.

Simple staging:
: The database file is stored on the local disk on the submit host, and then staged each time by a `scp` prior to each execution. This emulates simple replica staging techniques employed by most current data grids.

Ideal setting:
: We store the database file onto the local disks of all compute hosts so that data will always be locally accessible for minimum overhead.

Figure 1 is the time chart of the number of running the tasks for the three settings. Compared to the ideal setting, the shared file system and simple staging both suffer significant overhead, resulting in inefficient utilization of the computing resources. This is due to the concurrent access to the I/O node becoming the bottleneck. Figure 2 focuses on simple staging, where the hatched area represents tasks performing data transfer, as opposed to the dark shaded area representing tasks actually computing. We observe a cyclic pattern where each node alternates between data transfer phase with very little CPU utilization, and the compute phase where there is very little data transfer.

The root of this problem is that, in the current data grid each data storage element as well as its associated network

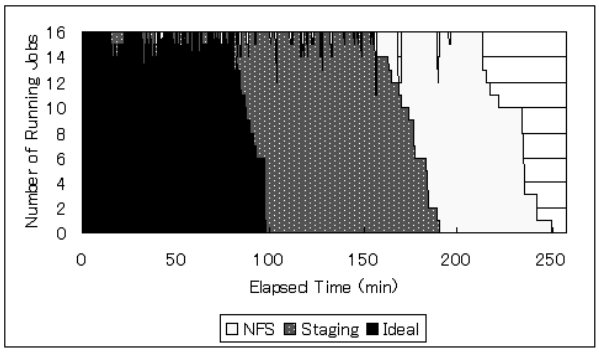

Fig. 1. Number of Running Jobs under Shared FSS, Simple Staging and Ideal Settings

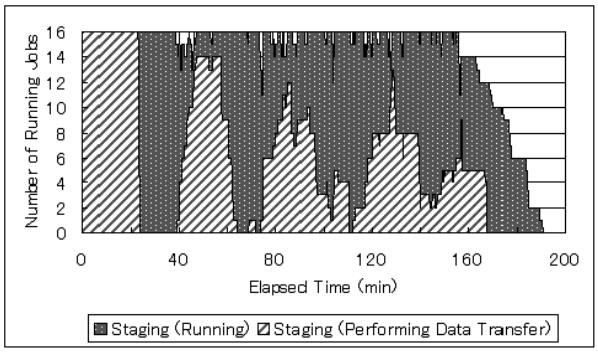

Fig. 2. Number of Running Jobs and Transferring Jobs under Simple Staging

for transfer may not be very scalable across the board in the grid infrastructure. A single file may be accessed and staged repeatedly; even if multiple files are accessed, because of observational or computational affinity they may emanate from a single fileserver, having to share disk, I/O, and most importantly, limited network bandwidth. This results in significant data transfer latency, which the current grid resource schedulers either ignore, or mostly force the users to cope with themselves as we see in the related work section.

For tightly coupled large scale cluster environments, low latency networking and parallel file systems such as PVFS [2] or LUSTRE [3] on I/O-intensive data farm of cluster nodes will help to alleviate the problem. On a data grid without such "brute-force" solution available, we claim that we can still solve large parts of the problem by techniques whereby we achieve *tight coupling between the grid compute resource scheduling and data transfer scheduling of the replica management system*. More concretely, we present three techniques where such tight coupling is achieved transparently to effectively overlap the computation and communication, achieving almost 100% utilization in compute resources. Although scheduling computation and data transfer collaboratively with scheduling has been attempted in various out-of-core parallel computing settings, our technique is on grids is more dynamic in that it is reliant on dynamic scheduling within a workflow of a single job involving multiple tasks, as well as across multiple workflows of different jobs.

We designed and implemented a prototype by coupling a Condor [4], [5], [6]-like batch scheduling system Jay with the proposed techniques. The result of several benchmarks show that our system demonstrates high utilization efficiency and scalability in the data grid compared to traditional methods, and is comparable expensive solutions on a tightly coupled cluster.

The rest of this paper is organized as follows. In Section II, we propose three techniques for highly utilizing data grid resources. In Section III, we describe our prototype implementation. In Section IV, we discuss the results on our cluster system. Finally, Section VI presents our conclusions and summarizes future works.

II. Proposal—Three Techniques for High Utilization of Data Grid Resources based on Tight Coupling of Compute Resource Scheduling

In order to resolve the low utilization of resources in data grids, we propose three techniques to tightly couple replica staging and computation scheduling, so that replicated data are effectively reused, and that computation and communication are effectively overlapped. The first technique reuses cached files, and when no cache exists facilitates O(1) efficient replication over the when the same dataset is reused; the second technique accommodates replication time-to-completion as the computation scheduling metric; the third technique co-schedules a separate compute-intensive job to a node during staging of a data-intensive job to maximize utilization. Collectively, we demonstrate through experiments on our prototype replica management and scheduling system that, for realistic usage scenarios the utilization approaches close to ideal where no staging needs to be performed, due to all data being local, and/or there is a massive shared parallel file system attached to low-latency, high-bandwidth network as is the case for tightly-coupled clusters:

- Dynamic aggregation and O(1) replication of data-staging requests across multiple nodes using a multi-replication framework
  There are several effective multi-replication algorithms that facilitate parallel replication without excessive I/O or network load, if the transfer occurs simultaneously. We automatically detect when the same file is being staged to jobs that are invoked (near-) simultaneously, and will achieve efficient O(1) parallel replication according to network topology for aggregated data staging. As an implementation, we combine automatic detection at the replica manager, fast inter-site, grid-level file copying via scp/GridFTP, and efficient and fault-tolerant O(1) intra-site copying by our Dolly+ tool.
- Replica-Centric Scheduling — data re-use and time-to-replication as compute scheduling metrics on the grid
  When our target shared data set is huge, the scheduler should effectively schedule computing tasks to nodes to minimize delays incurred by data access. This includes re-scheduling tasks that will utilize the same file (typical

in task farming style data grid applications) to the same node to eliminate staging, and when staging will be required, employing time to replication completion as the compute resource selection metric. In our implementation, the scheduler works tightly with the replica manager to keep track of what files have been replicated where, which file will be requested by the application, and network measurement metrics, which will manifest in the Condor-style classad[7] to determine appropriate resource matching.

- Overlapped Execution of Data Staging and Compute Bound Tasks
  Even if we achieve maximum efficiency, idle cycles in the compute nodes may not be avoidable due to high replication costs, which in turn result from to slow networks, etc. Since we target a practical grid environment where there will be a task mix within and across workflows from different users, we mix and match scheduling of compute bound tasks and data-intensive tasks so that potential idle cycle slots due to data transfer are filled by a compute bound task. When transfer finishes, our system suspends and/or migrates running tasks and starts data-intensive tasks of higher priority. Similarly, when outstaging of data starts then a compute bound tasks aggressively exploit this and may start or resume their execution. In our implementation, tasks can be characterized so that the scheduler could know whether a task is compute or data bound, expressed in terms of the classad description, and also monitored so that the scheduler will know which phase the data-intensive application is in (in/out staging of data, or performing the actual computation).

The overall summary of our proposed system is depicted in Figure 3. The RMS manages the locations of the files. Each host knows which local files are being managed by the RMS, and sends the host and file information to the scheduler. When the user submits a job that involves a data-intensive task that reads a large file F (as described in the classad), the scheduler will try to allocate the task to an idle compute host which has previously staged F and cached it. If there are no idle hosts with F cached, the scheduler allocates the task to the most opportune host taking into account the available network bandwidth of the compute host and the potential replica host. The compute host then actually inquires the best replica location of F and requests its transfer to the RMS. When multiple tasks request the same file F nearly at the same time, this is detected by the RMS, and for nodes that need replication, O(1) multi-replication transfer is organized and invoked automatically to best exploit the local bandwidth of sites and clusters. If there are any pending, fairly short, compute-bound tasks (again, as described by its classad), their expected job completion time is matched with expected transfer completion time of in- and out- staging on the compute hosts where the data-intensive jobs have been already allocated, and if they are a “good” match then the compute-bound tasks are aggressively allocated to exploit the

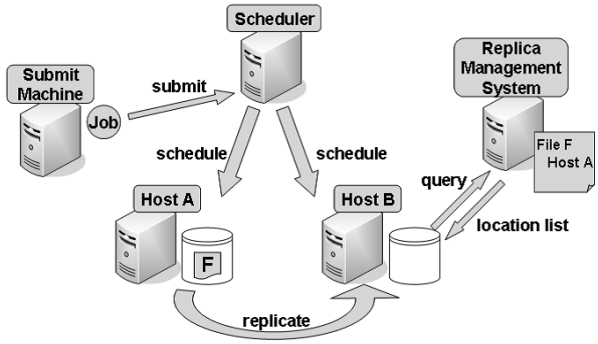

Fig. 3. System Overview

available idle cycle. When data transfer finishes, compute-bound jobs are either suspended or migrated out to a different host. Finally, F is registered to the RMS so that the compute node can now serve as a potential replica host (provided that the user guarantees in his submission file that F is read-only).

III. Prototype Implementation of the Techniques in the Grid Scheduling System “Jay”

We implemented a prototype grid scheduling and RMS system by coupling several components. For the scheduling system we employed and extended the scheduler and the compute host management in the Jay system, which is our simplified Java clone subset of Condor, with some additional Grid extensions such as direct GSI support. Although being a subset for prototyping purposes, Jay nevertheless supports the same key concepts and mechanisms of Condor such as classads, we expect much of our results to be directly applicable to Condor and other similar Grid job scheduling systems. We combined Jay with our multi-replication framework [8] we have developed as a replica management system based on Globus RLS-like services coupled with Dolly+[9] multi-threaded fast O(1) and fault-tolerant replication service. Here we describe an overview of the multi-replication framework and how scheduling is performed interactively with the replica management system.

A. Overview of the Multi-Replication Framework

Our multi-replication framework is a RMS which couples RLS and multicast data transfer system, utilizing threaded peer-to-peer techniques to conduct effectively O(1) parallel replication without imposing excessive load on the network and/or the I/O node. It selects the best source host from which the data set are transferred to the requesting host automatically. It registers the location of the replica which has been already transferred for subsequent data transfers. Moreover it can aggregate requests for the same file by using application-level multicast in a part of the data-transfer. The followings are the mechanism of the source host selection and the data transfer.

1) Source Host Selection Mechanism: The source host selection consists of a Replica Location Service (RLS) and a Replica Selector.

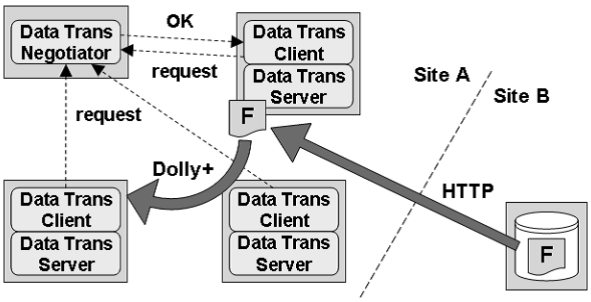

Fig. 4. A Data Transfer Mechanism

The RLS provides a service which is composed of a RLS server and a RLS client. The RLS server manages the locations of the files in the replica catalog database and handles its operation requests from the RLS client. The entries in the database use the standard logical name and physical location pairing, but embody several other attribute information regarding the hosts where the replica is located, used for selecting the best replica among all replicas from which the file will be transferred. The host info includes various network info such as observed network bandwidth, network RTT, etc. that are periodically updated.

2) Data Transfer Service: The overview of the data transfer service is depicted in Figure 4. The data transfer negotiator in the service can aggregate the requests for the same file by grouping the requests emanating from multiple hosts of the same site in the grid in a dynamic fashion. The grouping can be done in several ways; we currently employ a simple algorithm where each file request is cached for a certain adjustable time duration measured by a timer, and successive requests will refresh the timer. The actual data transfer occurs when the timer expires. When the cost metric dictates that multi-replication is advantageous, a representative node in the site is selected, to which the file is transferred using P2P copying such as scp or GridFTP.

Once the file is fetched, the data transfer servers that exists on every compute and data hosts transfers the file collectively within the grid site using the application-level multicast tool Dolly+[9] that employs threaded O(1) peer-to-peer copying algorithm. In effect, data are handed off in a pipelined fashion, utilizing the available network bandwidth of modern routers and switches effectively. As such the transfer time $T(n)$ is $T_S + T_w + T_l$, where T_S is the startup time overhead including the grouping timer expiration, T_w is the WAN inter-site transfer time, and T_l is the LAN inter-site transfer time to a single node, and is independent of the number of nodes, n.

B. Implementation of Replica-Centric Scheduling

With Jay, matchmaking [10] is performed by the central manager to decide the job allocation according to the rank value of the classads and their matches, in the same manner as is with Condor.

To accommodate replica-centric scheduling, location information of the replica files has to be reflected onto the classad; to achieve this the central manager queries the RLS and automatically adds value signifying the properties pertinent to data location (measured network RTT, bandwidth etc. of the host holding the data) of the (potential) replica files to the rank value used by matchmaking. Also added are other data-relevant info such as the size of the file being requested as well as the available disk space of the compute node, in order to allow the scheduler to avoid allocating the job to a host with disk capacity too small to store the data for the task.

If the compute node happens to be caching the data, and still has cycles left to process the data (i.e., if the host is an SMP), and then the replication cost is effectively set to zero to maximize the rank value of the host.

We describe how our prototype system works in more detail. Data-intensive applications are executed on our system in the following fashion as depicted in Figure 5. When a user wants to submit a job involving a task which uses the data registered in the RLS server, the user specifies its logical file name in `transfer_replica_files` in the submission file as in Figure 6. The `$(Replica_Files)` supplies the physical paths corresponding to the logical names specified by the user.

1) The startd daemon periodically sends a classad describing not only the host machine information such as the OS type/version or CPU load average, but also the replica information listing a the information pertinent to replication cost (network RTT, bandwidth etc., of the host holding the replica) of the files registered in the RLS server. (Because the number of registered replicas may be huge, filtering is performed so that only the data either cached in the compute host and/or has been recently named in the submission file and later reused, are announced) The schedd also periodically sends the classads describing the jobs which are scheduled and still actively running.
2) The central manager periodically allocates the tasks to the appropriate host by matchmaking with the received classads from the submission host. When the central manager receives a task which uses the data sets registered in the RLS server, it adds additional cost metric value that represents the replication cost to the rank value computed in matchmaking. Finally the central manager notifies the match to the allocated host which had the maximum match value, and to the job submission host.
3) The schedd spawns a shadow process when receiving the match notification from the central manager. The shadow sends a classad describing the task to the startd on the allocated compute host. When the startd receives the classad, it examines whether if could start the job, and the rest proceeds in the same manner as Condor, where startd spawns a stater process which in turns executes the task if possible.
4) If the received task uses data registered to the RLS server, then data staging with possible multi-replication starts as described in the earlier sections.

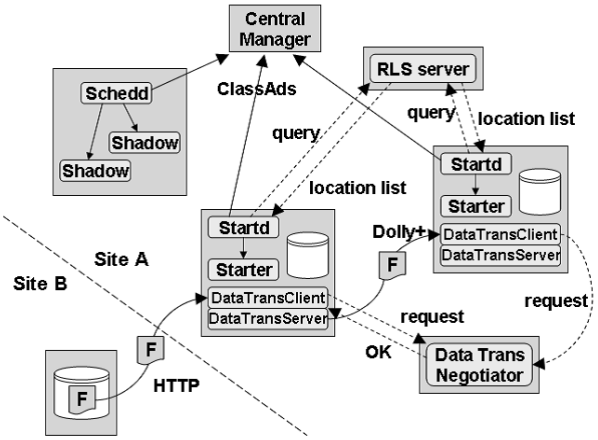

Fig. 5. Execution Flow of Data-Intensive Tasks

```
executable = application
input      = input.$(Process)
output     = output.$(Process)
error      = error.$(Process)
arguments  = $(Replica_Files)
transfer_replica_files = data1,data2
queue 100
```

Fig. 6. A example of a Submission File

C. Implementation of Overlapped Execution of Data Staging and Compute Bound Tasks

A compute host constantly monitors the running task to determine whether it is in/out staging the data, or performing the actual computation. This monitored job state is reflected in the classad published to the central manager, along with the resources it embodies (such as the number of CPUs) that will allow determination of whether a given host will have superficially been allocated tasks to fill the available compute resources, but in practice the CPUs remain underutilized because during in/out staging of data. If the size of the file to be staged as well as the effective network bandwidth is also available, we obtain an estimate of the duration of staging, and thus how long the CPUs will remain mostly idle.

The central manager, once given such classads, considers the available CPUs and the duration of availability, and attempts to schedule compute-intensive tasks that would likely fall into such duration onto such hosts. Conversely, if the compute bound task is executing, and then data-intensive task could be scheduled to pre-stage the input data.

In our current prototype implementation, we use a simple criterion where a task is data-intensive if the size of data used in the task exceeds a certain threshold in matchmaking. This threshold can be set on a host-by-host basis, to compensate for the data processing rate according to the performance of the machine. We are currently investigating a more flexible metric to distinguish data and compute "boundness" of a given task, but the current experiments do demonstrate that even such a simple metric works well under certain settings.

TABLE I
SPECIFICATION OF THE PRESTOIII CLUSTER

CPU	Opteron 242
Memory	2GBytes
OS	Linux 2.4.30
Network	1000Base-T

Also, currently the data-intensive tasks take priority once the actual computing phase starts over compute bound tasks. Thus, for example if a data-intensive task is scheduled first, and then the compute bound task is scheduled later in an overlapping fashion, but data transfer happens to finish earlier than expected, then the compute bound task is suspended or migrated to allow for the data-intensive task to proceed. This is due to the observation that, data replication may be costly, as replicas may be fetched from remote sites, whereas most migration opportunities are available within local sites with fast networks to migrate the task state in a rapid fashion. Even if task suspension is only available, it is still worthwhile to let the data-intensive task proceed first as most compute bound tasks are part of a large farming task, and progress of a single particular task within the job may have very minor effect on the overall job completion time.

IV. PERFORMANCE EVALUATION

We performed comparative execution of the sample BLAST data-intensive application depicted in Section 1 to our prototype. We launched a RLS server and the data server on one node, and registered the location of the database file on the RLS server. We also designated one node as a central manager, another as a submit machine, and the remaining $n(n = 4, 8, 16, 32)$ node(s) as compute hosts with specs as described in I. We subsequently submitted $5n$ tasks that execute BLAST with 5 nucleotide query sequences against nucleotide database nt.

The average execution time as well as the number of tasks running for $(n = 16)$ are illustrated in Figure 7 and 8. They show that our system performs much superior to simple file sharing and staging techniques, and in fact is almost equivalent to the ideal setting. Moreover it shows that our system is very scalable since performance remains constant regardless of the number of nodes. This result could be achievable with expensive "brute-force" infrastructures, but in our case our I/O system is a standard PC with nominal I/O bandwidth.

Figure 9 categorizes the nodes in our proposal as either performing data transfer or actually running. As one can observe, data transfer takes place only once and very efficiently thanks to the detection of multi-replication opportunity and effectively conducting O(1) group data transfer. Moreover we can see that the data transfer is avoided by reusing already-created replicas, compared to Figure 2.

To evaluate the scaling of multi-replication, we increased the number of compute hosts from 2 to 16 nodes. On initial startup, a genomic sequence database of approximately 3 GBytes is staged to every node. With our system, the startup time is 3.23 minutes on 2 nodes (30 MBytes/s) and 6.4 minutes

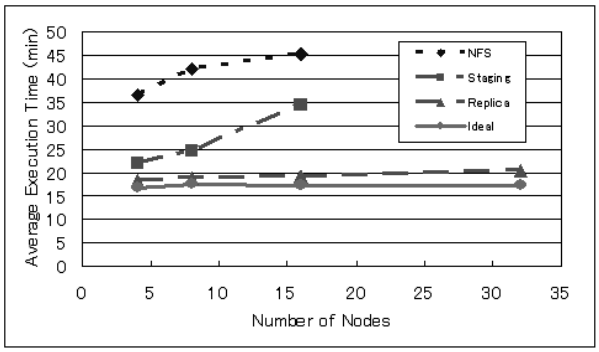

Fig. 7. Average Execution Time of Tasks

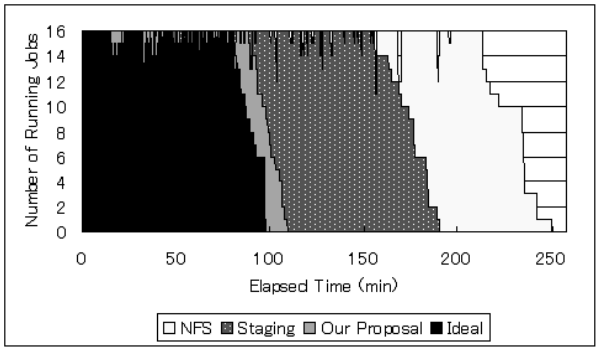

Fig. 8. Number of Running Tasks with Our Proposed Technique

on 16 nodes (120 MBytes/s aggregate). By comparison, raw NFS read on the same file system registered only 12 MBytes/s. Overall, our technique improved the total execution time by 57.5 % over NFS and 44.3 % over the simple staging.

To evaluate overlapped execution of data and compute bound tasks, we measured the throughput in a controlled task mix, where on 8 compute hosts we submit 8 compute-bound jobs (simple calculation of π using the Monte Carlo method) after submitting 40 data-intensive BLAST job described earlier. Figures 10 and 11 show the results. The former takes approximately 111 minutes to complete due to idle cycles caused by data transfer, while the latter takes only 88 minutes to complete. The figures show that this is caused by lower

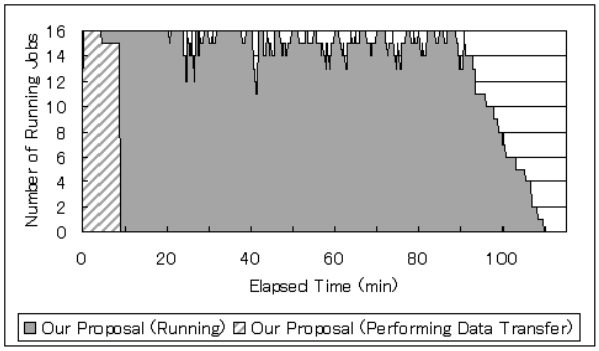

Fig. 9. Breakdown of Tasks in Different Phases with Our Proposed Technique

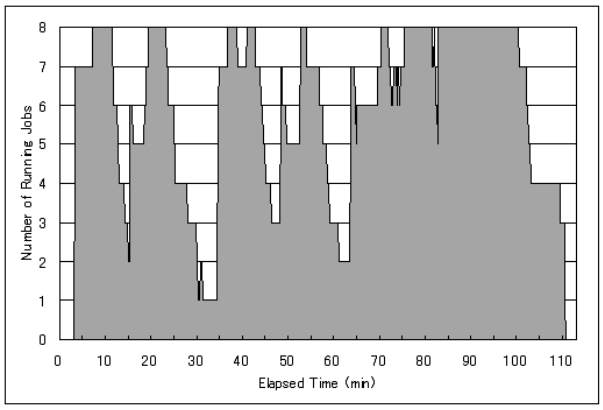

Fig. 10. Number of Running Tasks without Overlapped Execution

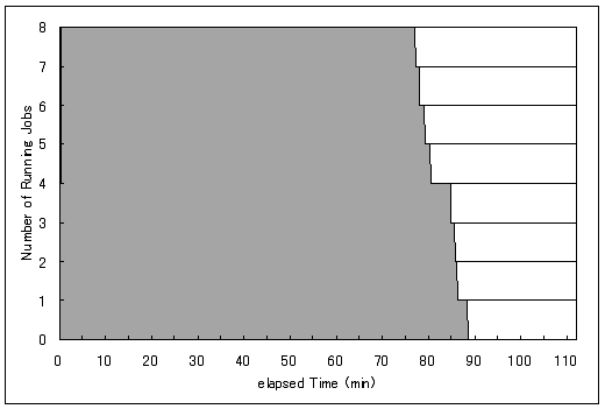

Fig. 11. Number of Running Tasks with Overlapped Execution

CPU utilization of the former, while in the latter utilization is almost constantly 100%.

V. Related Work

Parallel file systems such as PVFS, LUSTRE, and GPFS [11] provide scalable data access, and some have been extended for use on the Grid. All requires some (semi-)dedicated sets of I/O nodes that form a data farming cluster to facilitate fast, expensive bandwidth access to backend storage. Data are typically striped at a very low level for parallelism, with very little control of where the actual data would be located versus the clients of the data. Although it is possible to stage data using them with resource brokers such as SRB, they are rather expensive solutions. Gfarm [12] is a Grid file system that facilitates owner computes rule to localize computation with its data. Gfarm also corresponds parallel processes to its data in a much more deliberate fashion compared to conventional parallel file systems, including replication to balance I/O load. However, it does assume a set of clusters on a Grid of similar nature to conventional parallel file systems, and moreover, does not perform intricate staging of data from multiple file systems, etc.

Batch-Aware Distributed File System (BAD-FS) [13] is a distributed file system that exposes internal control to an

external scheduler to facilitate execution of data-intensive workloads on both wide- and local-area clusters. However, selection of the data location from replicas must be done manually by the user, as well as requiring users to have complete a-priori knowledge of the flow of data in a particular workflow, compromising scalability.

Stork [14] schedules data placement tasks on the grid, and hides failures in network, storage system, and middleware from user applications. A Stork user describes data placement request in a classad explicitly in the classad. Stork is used with DAGMan [15], a meta- (workflow) scheduler for Condor; DAGMan manages dependencies between Condor and Stork jobs, where computational tasks are submitted to Condor and data placement tasks to Stork according to the order in a DAG file [16]. Stork itself does not help to alleviate the problem of degradation of I/O performance when numerous nodes share the same data sets, without brute-force methods such as parallel file systems as described above. Moreover various techniques that require dynamic decisions that tightly couple computation and data transfer/placement is difficult with Stork, because Condor and Stork schedule the jobs independently from each other. It may be profitable to integrate the features of Stork into Condor, and apply our techniques so that expensive storage infrastructures could be eliminated.

Replica management services on the Grid serve as low-level building blocks for building systems that improve access latency, data locality and robustness, etc. The data management service for the data replication is provided by the Globus Toolkit [17], EU DataGrid (EDG) project [18], etc. The data management components of the Globus Toolkit are composed of RLS (RLS)[19], GridFTP [20] and RFT (Reliable File Transfer). Reptor [21] plugs into various components easily and interacts with the RLS (RLS), Replica Metadata Catalog Service (RMC) and the Replica Optimization Service (ROS), which selects the best replica allowing for the access cost. These services usually assume point-to-point communication and file transfer protocol, and do not cope with performance bottlenecks when multiple nodes access the I/O nodes simultaneously. Although several work [22], [23], [24] evaluates automatic replica placement algorithms with respect to common data grid workloads, they also do not account for concentration of data access to a single data or an I/O node, nor consider possibility of efficient parallel replication algorithms to alleviate the situations, the key element of our proposal. Also, the scheduling of data and compute bound tasks versus data placement are not necessarily considered in concert, again the key to achieving high efficiency in our proposal.

VI. Conclusion and Future Work

We proposed three techniques that improves resource utilization of data-intensive application on the data grid, and demonstrated on our prototype implementation that, at least under common application situations, the system will perform almost as good as "ideal" situations where all the data are local to the compute nodes, without resorting to high-cost, centralized storage systems and parallel file systems. The techniques are simple enough to be implemented as extensions to scheduling and classads of Condor, a proven system in data grid setting already. As such, we believe our technique will be applicable to wide range of data grid systems quite effectively.

As future work, we also plan to optimize execution by considering the treatment of outstaged data, as well as apply the techniques to real Condor and test it under a realistic and comprehensive data grid application environment.

References

[1] "NCBI BLAST," http://www.ncbi.nlm.nih.gov/BLAST/.

[2] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*. Atlanta, GA: USENIX Association, 2000, pp. 317–327. [Online]. Available: citeseer.ist.psu.edu/article/carns00pvfs.html

[3] "lustre," http://www.lustre.org/.

[4] "Condor Project Homepage," http://www.cs.wisc.edu/condor/.

[5] M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," *Proceedings of 8th International Conference of Distributed Computing Systems*, pp. 104–111, 1988.

[6] D. Epema, M. Livny, R. Dantzig, X. Evers, and J. Pruyne, "A Worldwide Flock of Condors: Load Sharing among Workstation Clusters," *Journal on Future Generations of Computer Systems*, vol. Vol. 12, 1996.

[7] M. Livny, R. Raman, and T. Tannenbaum, "Mechanisms for High Throughput Computing," *SPEEDUP Journal*, vol. Vol. 11, no. 1, 1997.

[8] S. Takizawa, Y. Takamiya, H. Nakada, and S. Matsuoka, "A Scalable Multi-Replication Framework for Data Grid," *Proceedings of the 2005 International Symposium on Applications and the Internet (SAINT 2005 Workshops)*, January 2005.

[9] A. Manabe, "Disk cloning program 'dolly+' for system management of pc linux cluster," *Computing in High Energy Physics and Nuclear Physics*, 2001.

[10] R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed Resource Management for High Throughput," *Proceedings of 7th IEEE International Synposium on High Performance Distributed Computing*, July 28-31 1998.

[11] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. of the First Conference on File and Storage Technologies (FAST)*, Jan. 2002, pp. 231–244. [Online]. Available: citeseer.ist.psu.edu/schmuck02gpfs.html

[12] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi, "Grid datafarm architecture for petascale data intensive computing," *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, pp. 102–110, 2002.

[13] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny, "Explicit Control in a Batch-Aware Distributed File System," in *Proceedings of the First USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, March 2004.

[14] T. Kosar and M. Livny, "Stork: Making Data Placement a First Class Citizen in the Grid," *Proceedings of 24th IEEE International Conference on Distributed Computing Systems (ICDCS2004)*, 2004, tokyo, Japan.

[15] "Directed Acyclic Graph Manager," http://www.cs.wisc.edu/condor/dagman.

[16] G. Kola, T. Kosar, and M. Livny, "A Fully Automated Fault-tolerant System for Distributed Video Processing and Off-site Replication," *In the 14th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV2004)*.

[17] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *Internatinal Journal of Supercomputer Applications*, vol. Vol. 11, no. 2, pp. 115–128, 1997.

[18] "EU DataGrid project," http://www.eu-datagrid.org/.

[19] A. L. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf, "Performance and scalability of a replica location service," *The Thirteenth IEEE International Symposium on High-Performance Distributed Computing*, 2004.

[20] W. Allcock, J. Bresnaham, I. Foster, L. Liming, J. Link, and P. Plaszczac, "Gridftp update january 2002," *Globus Project Technical Report*, 2002.

[21] D. Bosio, J. Casey, A. Frohner, L. Guy, P. Kunszt, E. Laure, S. Lemaitre, L. Lucio, H. Stockinger, K. Stockinger, W. Bell, D. Cameron, G. McCance, P. Millar, J. Hahkala, N. Karlsson, V. Nenonen, M. Silander, O. Mulmo, G.-L. Volpato, G. Andronico, F. DiCarlo, L. Salconi, A. Domenici, R. Carvajal-Schiaffino, and F. Zini, "Next-Generation EU DataGrid Data Management Services," in *Computing in High Energy Physics (CHEP 2003)*, La Jolla, CA, March 2003.

[22] K. Ranganathan and I. Foster, "Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications," in *Proceedings of 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, Edingurgh, Scotland, July 2002.

[23] ——, "Identifying Dynamic Replication Strategies for High Performance Data Grids," in *Proceedings of International Workshop on Grid Computing*, Denver, CO, November 2002.

[24] A. Takefusa, O. Tatebe, S. Matsuoka, and Y. Morita, "Performance Analysis of Scheduling and Replication Algorithms on Grid Datafarm Architecture for High-Energy Physics Applications," in *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*, June 2003, pp. 34–43.