Co-Scheduling of Computation and Data on Computer Clusters

Alexandru Romosan, Doron Rotem, Arie Shoshani and Derek Wright [*]
Lawrence Berkeley National Laboratory,
University of California,
Berkeley, California 94720

Abstract

Scientific investigations have to deal with rapidly growing amounts of data from simulations and experiments. During data analysis, scientists typically want to extract subsets of the data and perform computations on them. In order to speed up the analysis, computations are performed on distributed systems such as computer clusters, or Grid systems. A well-known difficult problem is to build systems that execute the computations and data movement in a coordinated fashion. In this paper, we describe an architecture for executing co-scheduled tasks of computation and data movement on a computer cluster that takes advantage of two technologies currently being used in distributed Grid systems. The first is Condor, that manages the scheduling and execution of distributed computation, and the second is Storage Resource Managers (SRMs) that manage the space usage and content of storage systems. This is achieved by including the information about the availability of files on the nodes provided by SRMs into the advertised information that Condor uses for the purpose of matchmaking. The system is capable of dynamically load balancing by replicating popular files on idle nodes. To confirm the feasibility of our approach, a prototype system was built on a computer cluster. Several experiments based on real work logs were performed. We observed that without replication compute nodes are underutilized and job wait times in the scheduler's queue are longer. This architecture can be used in wide-area Grid systems since the basic components are already used for the Grid.

[*]Visiting LBNL from the Computer Sciences Department, University of Wisconsin

1 Introduction

It is typical of scientific investigations to have two phases: the data generation phase, and the data analysis phase. The data generation phase is usually the result of running a large simulation or the collection of data from experiments. Using modern computer systems the amount of data generated in the data generation phase is massive, in the order of terabytes to petabytes. In the data analysis phase, the scientist typically wants to extract a subset of the data based on some criteria. For example, a simulated climate modeling dataset may have data over the entire globe, with multiple height levels for tens of variables, such as temperature, humidity, wind velocity, etc. This large simulation dataset is usually stored on some mass storage system, such as IBM's High Performance Storage System (HPSS). During the analysis phase, a scientist may want to select only temperature over the equator for sea-surface level for 100 years. This requires some way of selecting the files that contain the relevant data, downloading the relevant files to the analysis system and processing only these files.

Another example of the need for co-scheduling of compute and data movement involves Particle Physics data mining and analysis of detector data. The Data Acquisition System in these detectors records information about collision events between particle beams. The information is stored in multiple files, where each file contains information about thousands of such events. Typical analysis of data involves searching for rare and interesting processes and is performed in multiple phases involving classification and summarization. In addition, the same data files may be shared simultaneously by several different groups of scientists with different interests. In general, the problem discussed here is that of effective scheduling of a collection of jobs, each requiring one or more input files to run on a group of servers. Each server in the cluster may have one or more compute slots and a disk cache that can hold some fraction of the data files needed as input for the analysis. A given job can be scheduled on a selected server if: (i) the server has at least one available compute slot; (ii) all the data files needed by the job are available on the disk cache at that server.

This introduces the problem of scheduling data movement in coordination with scheduling of computation on a cluster, and the software systems to execute and monitor the schedules. Data movement may be cheap (from one server on the cluster to another) or expensive (from a remote archive).

The above two phases of operations reflect the manner in which most scientific applications run. To speed the data analysis phase, the analysis is partitioned into parallel jobs, and distributed to multiple compute systems. In a Grid environment the compute systems are distributed over the wide area network, and the data sources are usually on remote storage systems. In order to perform the parallel analysis, the data have to be moved to the compute nodes, and the jobs scheduled on these nodes. There are Grid middleware components designed to schedule compute jobs on distributed nodes, and components designed to manage storage and move files between nodes. However, there are currently no components that perform co-scheduling the data and the computation, in part because of the complexity of such middleware systems. Developing a real practical system to perform co-scheduling is one of the most difficult challenges in the Grid domain. We address this challenge in this paper.

We describe a system that was developed to perform co-scheduling of data and computation by taking advantage of two technologies used in distributed Grid systems. The first is Condor, that manages the scheduling and execution of distributed computation, and the second is Storage Resource Managers (SRMs) [1] that manage the space usage of storage systems and the dynamic content of the storage. In order to have a controlled experimental environment, the system was developed on a small cluster of workstations that do not share memory, and have their own independent attached disk. In order to achieve co-scheduling, some modifications to Condor and SRMs had to be made, but as will be discussed next, we achieved coordination between these systems with relatively modest enhancements.

1.1 Organization of paper

The rest of the paper is organized as follows. In Section 2 we explain the main contributions of this work emphasizing the ability to build a complex co-scheduling system by using existing mature components. In Section 3 we describe the architecture of the co-scheduling system and the software modules developed for this project. In Section 4 we describe the replication algorithms used for evaluating our system. In Section 5 the data used in our experiments and the test environment are described and the performance results are analyzed. Finally, in Section 6 some conclusions and future work are presented.

1.2 Related Work

The development of the co-schedueling system facilitated tests on a real co-scheduling system with real logs (taken from a high-energy physics experiment). Most of the previous work in this area is based on simulations.

In [2] several replication algorithms are examined and evaluated using simulations. The results in that paper show that data aware scheduling on the grid results in significant improvements in job response time. In [3] the authors describe simulation of a Grid system and evaluation of different file replication algorithms. The authors also examined various cache replacement policies. The research in [4] describes a system for treating data transfer events as real jobs. The system allows checkpointing and monitoring of data transfers. This work is complementary to our work as we can use such a system (called Stork) for scheduling data movement while using SRMs to keep track of cache contents and enforce caching policies. Another approach to perform replication management on the grid uses economic models [5] where some incentive is offered to resource owners for contributing and sharing resources, and motivates resource users to think about tradeoffs between the processing time (e.g., deadline) and computational cost (e.g., budget), depending on their QoS requirements. In [6] the authors use an auction protocol for selecting the optimal replica of a data file. The work in [7] and [8] deals with prediction functions to make informed decisions about pre-fetching of data.

2 Main Contributions

Job scheduling systems are very complex and take a large effort to implement and support. While there are examples of such systems that are available commercially or as open source products, these systems manage scheduling of compute slots only, not the co-scheduling of data with the compute slots. Examples of such packages include SUN's Grid Engine software, Load Sharing Facility (LSF), Portable Batch System (PBS), and Condor (from the University of Wisconsin, http://www.cs.wisc.edu/condor).

Developing a system that can co-schedule compute and data resources is a very large undertaking. It took years to perfect the systems that perform only compute-slot scheduling. We address in this work the possibility of using existing software components to tackle this complex challenge. Our starting point was to select a job scheduling system and a storage management system, and design a combined co-scheduling system without making major changes to these systems. The main contribution of this paper is in the methodology and architectural design that succeeded to bring this co-scheduling system into fruition.

The key to this success was the flexibility of the existing systems we chose to work with. In particular, the Condor system is designed to perform matches between jobs and worker nodes based on an open-ended description of what to match on. Thus, we could easily extend the descriptions to include sets of files that a worker node has at any one time. This was complemented by the flexibility of Storage Resource Managers to manage their content dynamically (i.e. using automatic caching policies), and their ability to keep files for jobs that are scheduled to be matched as well as remove unneeded files after the jobs finish.

We encountered one issue that could be a barrier to the scaling of the co-scheduling system. The issue was of advertising the data files that a node currently has. Since disk caches of worker nodes can be very large and contain thousands of files, providing this information to the component that performs matches may overwhelm the scheduler. While one can design more efficient schedulers with smart indexing technology, there was no certainty that this will scale, and it would require a serious enhancement of the existing system. Our solution, instead, is to provide each node with a component that extracts the information of the requested files only, so that it can be advertised in the *classAd* along with the node's resources. This component would then ask the Storage Resource Manager to provide information about which of these files it has. This modification to the advertising component solved this potential problem, and made our design practical and scalable.

We now have a real system where various algorithms that were previously tested only by simulations can now be tested in a realistic environment. Furthermore, while we implemented this architecture on a cluster, it is straight-forward to adapt this design to wide-area Grid systems, because the components are already functional as Grid middleware.

3 Architecture

To achieve co-scheduling, the scheduler must have information on the content of each machine's disk cache, as well as the availability of compute-slots on each machine. The problem is one of matching each job to the machine that has the files needed by the job. This is achieved by providing the Condor scheduler information on the dynamic content in the disk caches in the compute nodes. This information is provided by the SRMs that reside on each computing node. We accomplish this by extending the standard mechanism used by Condor to describe a worker node to also include information on the files available on the node.

The advantages of using Condor and SRMs are numerous. Both of these systems are open-source. Condor provides job scheduling using an extensible "match-making" technology, as well as initiation and monitoring of jobs on the compute nodes. SRMs provides dynamic storage allocation, with the ability to "pin" and "release" files to ensure that they stay on the disk cache when they are needed. SRMs also have their own local policies for removing files that were released and are not needed. The combination of these two technologies is a fast and efficient way to implement and test the co-scheduling setup.

The architecture and the components we used to achieve co-scheduling are shown in Figure 1. As can be seen, the master node has three parts of the Condor system: i) *condor_schedd* (the scheduler daemon) that is responsible for scheduling jobs and keeping their state information, ii) the *condor_collector* that collects and organizes all the information about nodes in the form of *classAds* (classified advertisements); the *classAds* contain information about compute-slots in the nodes, and their hardware and software capabilities, and iii) the *condor_negotiator* whose function is to find a match for each scheduled job; the match is done by finding a compute-slot that has at least the capabilities required by the job being matched.

The other part of the Condor system is a component, called the *condor_startd* (start daemon), whose function is to start jobs running on a node, to collect information on the node capabilities and generate the *classAd*, and to monitor the progress of the job. If the job completes successfully, the *condor_startd* advertises the availability of the slot by issuing a new *classAd*. If the job is interrupted and was not completed, it communicates with *condor_schedd* to schedule the job again. As can be seen from Figure 1, the *condor_startd* was installed on every worker node.

A disk version of an SRM, called a DRM, developed at LBNL (http://sdm.lbl.gov/srm) is also installed on every worker node. Its function is to manage the disk cache associated with the node. That includes allocating space for every file that has to be moved into the disk cache, keeping track of popular files (so called "hot files" that are accessed multiple times), and removing unwanted files ("cold" files). The DRM performs this function by "pinning" a file as soon as the file was advertised as required by a job to be scheduled to run on that node, and by releasing the file as soon as the job is finished. Note that a file may be pinned multiple times if multiple jobs are using it, and the DRM keeps track of that as well. Finally, the DRM also initiates file transfers from the mass storage system we use, by communicating with a version of an SRM, called a Hierarchical Storage Manager (HRM) that can request file staging out of the mass storage system (we use HPSS). The DRM can also request a file from a neighbor node if it is asked to do so.

As mentioned above, the technique of achieving co-scheduling is to add in the *classAd* the information

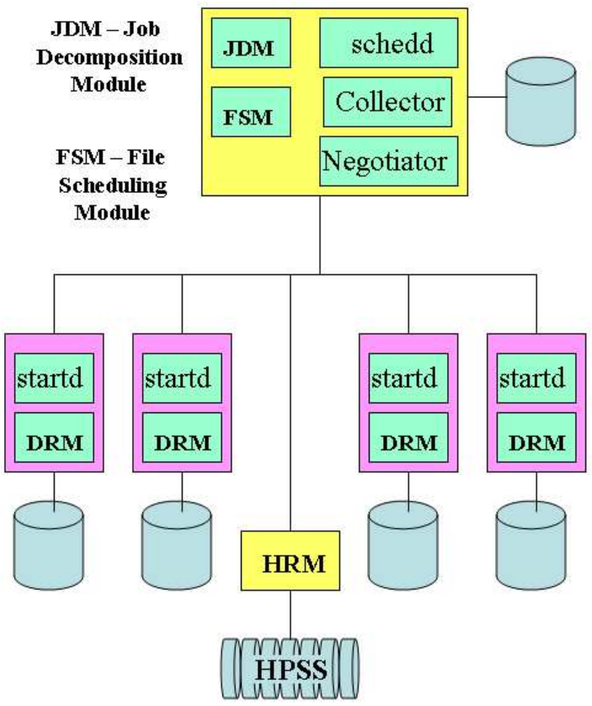

Figure 1. The architecture of the co-scheduling system using Condor and DRM components.

on what files are available on each node. However, this can cause a major problem, since a node can have thousands of files, and the *classAd* will become too expensive to manage and match against. Our solution to this problem is to put in the *classAds* only files that are *relevant* to the pending jobs. The ability to achieve this solution was critical to the practical success of this co-scheduling problem. We identified two components that are needed to achieve this functionality and the co-scheduling operation: the Job Decomposition Module (JDM) and the File Scheduling Module (FSM). Both of these components are located on the master node. We explain their functionality next.

The JDM is the component which accepts jobs submitted by clients. Each job consists of an executable, a set of input files and, optionally, a set of output files. The JDM parses this information and decomposes each job into one or more jobs each requesting a single file †. The JDM performs the following tasks: i) decomposing all incoming jobs dynamically; ii) generating a list of files that is the union of all requested files; iii) communicating with each *condor_* *startd* and provide them with this list; iv) providing the FSM with a list of jobs to be scheduled with Condor's *condor_schedd*; v) keeping track of completed jobs; and vi) providing the client with information on the progress of the job, as well as when it completes.

The FSM was designed to interact with the Condor system. It is responsible for the following actions: i) schedule with each DRM the files that it should acquire; ii) schedule all jobs with Condor's *condor_schedd*; iii) monitor the progress of jobs by inquiring from Condor which jobs are being delayed; iv) analyze the reasons for the delays and issue replication requests to DRMs; iv) decide when pre-staging of files from HPSS is warranted, and v) notify the JDM when a job completes. As can be seen, the algorithms for optimizing the co-scheduling belong in the FSM.

There were relatively small changes required to accomplish the coordination between Condor and the DRMs. The main change required from the *condor_startd* is the ability to accept a list of files from the JDM and invoke the DRM to find which of these files the DRM has. *Condor_startd* then includes these files in the *classAd* it advertises. The DRM had to be modified to provide a response to an inquiry that amounts to "which of these files do you currently have?".

Taking advantage of the fairly complex middleware systems developed over many years by making relatively modest modifications was the reason for our success in developing the co-scheduling system. This provided us with a real environment to explore the behavior of the system under different scheduling strategies. We describe in Section 5 some of the results achieved so far by running experiments on this system.

3.1 Information Flow

Figure 2 describes the information flow between the components of the co-scheduling system. For ease of explanation the steps are labeled in the logical order of flow, however, in reality these steps may repeat and run asynchronously. The steps are as follows:

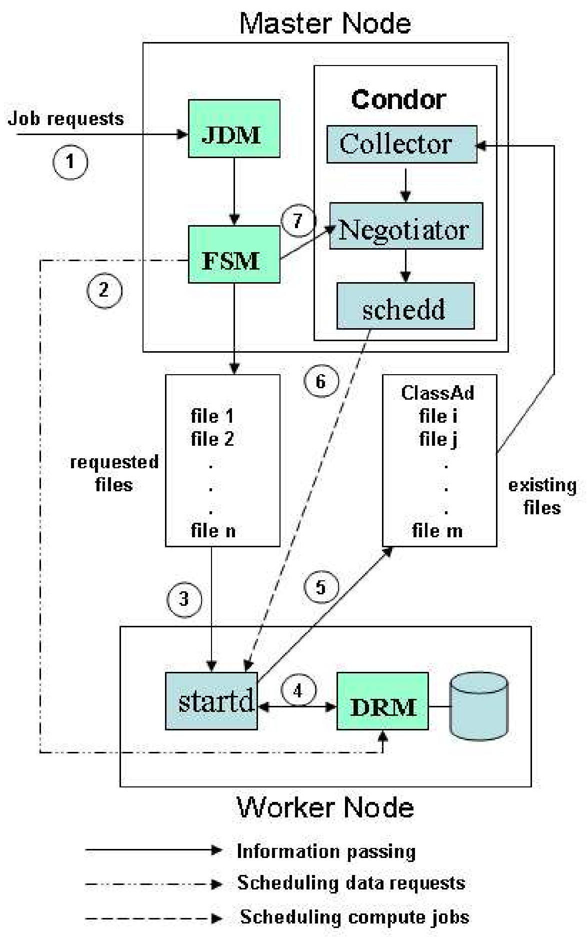

Figure 2. The steps of the co-scheduling system

1. Job requests arrive to the Job Decomposition Module (JDM), are parsed and decomposed to multiple smaller jobs, and are passed to the File Scheduling Module (FSM).

2. The FSM schedules data requests to the DRMs according to the scheduling algorithm it uses. The simplest one is round robin.

3. The FSM composes a list of all requested files that have not been processed yet. It extracts the information on job completion from Condor logs. This list of files is passed to *condor_startd*. The FSM also submits the jobs to Condor.

4. *condor_startd* communicates with its local DRM to find out which of the files in the requested-

†We note that in HEP applications it is always possible to run an analysis job on a single file at a time because processing of collision events contained in files are independent of each other. However, in other applications it may be necessary to specify the subset of files that are needed concurently to execute the analysis job.

file list it actually has. These are refered to as existing-files in the figure.

5. *condor_startd* puts the existing-files into a *classAd* that it passes to the *condor_collector*.

6. Condor finds a match for an available compute-slot on a node that has the file needed by the job and schedules that job.

7. The FSM checks with the *condor_negotiator* for jobs in queue. If there are free compute-slots, it chooses a file to replicate based on the length of time the jobs requesting it have been waiting in the queue (for details see section 4).

This iterative process is performed continuously when new jobs arrive, or when the monitoring thread triggers a replication action. The files in the DRMs stay in the cache until space is needed. The DRMs currently use a least-recently-used caching algorithm.

4 Scheduling Algorithms

In typical particle physics analysis applications a job that requires n files can be decomposed into n smaller jobs each requiring one of the files. This increases the opportunity for parallelism since after performing decomposition, some subset of the n jobs can be scheduled to run in parallel on different servers. According to this simple model, we assume that the original jobs have been decomposed and each of the resulting jobs requires exactly one file. The scheduling problem we are considering here consists of the following inputs:

- a set of files $F = \{f_1, f_2, f_3, \ldots, f_n\}$
- a set of queued jobs $J = \{j_1, j_2, \ldots, j_m\}$ each requesting a single file from F
- a set of worker nodes $N = \{N_1, N_2, \ldots, N_k\}$ where each node N_i is associated with one or more compute-slots and a cache $C(N_i)$ that contains some subset of the files in F.

Let $f(j_i)$ be the file requested by job j_i. The job j_i can be assigned to run on a node N_k (j_i is matched to N_k in Condor terminology) if (a) N_k has an available compute slot and (b) the cache $C(N_k)$ contains the file $f(j_i)$. Among the many possible scheduling algorithms we chose to evaluate two basic algorithms. These are described below:

scheduling with no-replication. This algorithm retains at most one copy of a file on the cluster. It copies a file, f_i, from remote storage to a node N_k (selected in a round-robin fashion) only if f_i is requested by a job and cannot be found in any of the caches of the worker nodes. As long as f_i is not purged from the cache $C(N_k)$, any subsequent jobs that request the file f_i will be automatically matched with N_k by Condor once N_k has a free compute slot. In order to avoid purging f_i from the cache $C(N_k)$ prematurely, it is pinned by the system as long as there are jobs waiting for it in the queue.

scheduling with replication. Files may be replicated in the cluster across multiple nodes. A replication decision is made whenever there are jobs waiting in the queue and there are available compute-slots in the system. The selection of which file to replicate next is determined by a weight function $w(f_i)$ computed as follows: For each file f_i requested by one or more jobs in the queue, $w(f_i)$ is equal to the total time these jobs have been waiting for it. The file with the maximal $w(f_i)$ is then replicated on one the worker nodes with a free compute slot chosen at random. The rationale behind this replication algorithm is that "popular" files should be available on multiple nodes for better load-balancing of the system and jobs that have been waiting for a long time should get some priority to avoid starvation.

The first algorithm (scheduling with no-replication) attempts to minimize file transfers from the mass storage system across the network and also saves on disk storage requirements. It may be attractive in situations where file transfers and disk storage costs are relatively expensive resources. The second algorithm (scheduling with replication), tends to move more files but reduces queue waiting times and improves system utilization. We chose to experiment first with these two basic algorithms since they are simple to implement and do not introduce excessive overhead costs on the system. We are planning to evaluate more elaborate algorithms that take into account node capacities, different replication costs (remote vs. local) using mathematical optimization techniques.

5 Experimental Results

5.1 Description of physics analysis environment and data characteristics

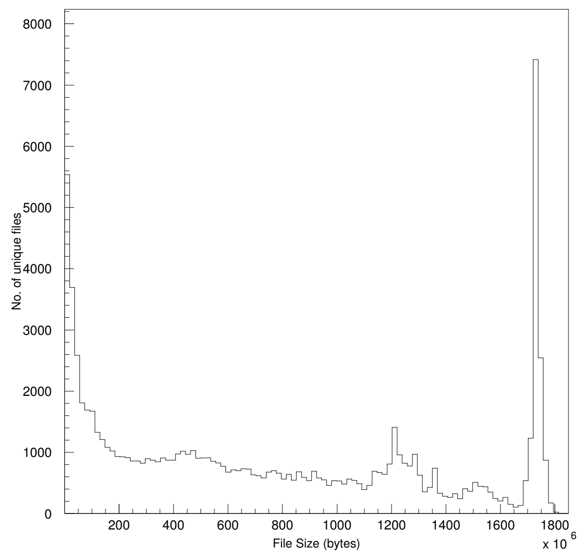

Figure 3. Unique file size distribution

We experimented with data from BaBar, which is a High Energy Physics experiment with over 600 world-wide collaborators (http://www.slac.stanford.edu/BFROOT). The data for this experiment is stored on tapes at the Stanford Linear Accelerator Collider (SLAC) managed by IBM's Mass Storage System HPSS storing over 1.3 petabytes of data on about 13,000 tapes managed by 6 StorageTek tape silos. To deliver data to jobs in a reasonable time the system is currently backed by 160TB of disk cache implemented on thousands of physical disks bound into large arrays managed by Sun's Solaris 9 UFS. The analysis jobs run on a cluster of hundreds of nodes accessing the persistent data through a high performance data server [9]. The work logs used in our paper were extracted from trace data produced by the data server. The raw trace data contained, for each job, only its job id, the files accessed by the job, and the time of access of each file. We had to match this information with the file characteristic data stored in a Oracle database at SLAC in order to get file size information. We analyzed trace logs taken from October 1 to October 26, 2004. During this time interval 504,493 jobs were submitted requesting a total of 2,028,541 files, 86,378 of which were unique. Figure 3 shows the size distribution of the 86,378 unique files. Note that the maximum size of a file is close to 2GB, due to file system limitations. There is also a significant number of files of size less than 200MB. The large files most likely represent raw detector data, whereas the smaller sized files are in most cases filtered data ("skims") for the purpose of user analysis.

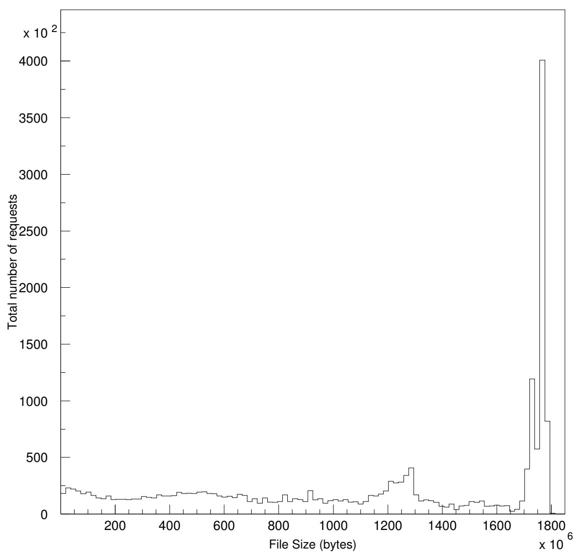

Figure 4. Requested file size distribution

In Figure 4 we plot the number of file requests as a function of file size. We note that the large files appear in more requests as compared to the small files. One possible explanation for this access pattern is the automated running of "skimming" which use the raw data files as input to produce the user-analysis files.

5.2 Description of the cluster environment

We ran our experiment on a cluster of 8 single and 1 dual CPU 1.5GHz AMD Athlon processors, each with a 20GB disk cache and 2GB of RAM for a total of 10 compute slots. The cluster nodes are on a GigaBit network. We installed one DRM on each machine.

Condor software and the FSM were installed on the dual processor. The data files needed for the analysis jobs were stored on the HPSS mass storage system at LBNL. We observed that the average transfer rate from the HPSS system was 15MB/s.

5.3 Experimental setup and performance results

Each experiment consisted of a sample of 1500 jobs from the pool of jobs presented above. Assuming the duration of a typical analysis job is proportional to the size of the input, we simulated a job by running 100 empty for-loops per each byte requested. We experimented with different job arrival rates, and for the purpose of this study we chose an interval of 60 seconds between job submissions based on the average job execution time to match the job arrival rate to the job service rate. Shorter arrival intervals would lead to a saturation of computing resources, while a very long arrival interval would underutilize the cluster. Based on these parameters, each experiment took about 27 hours of continuous running time. Furthermore, we had to set the Condor configuration parameters to much smaller values than the default values. For example, the default resource matching negotiation cycle, whose default value is 300 seconds was lowered to 60 seconds to make the matchmaking more responsive to our job arrival and data transfer rates.

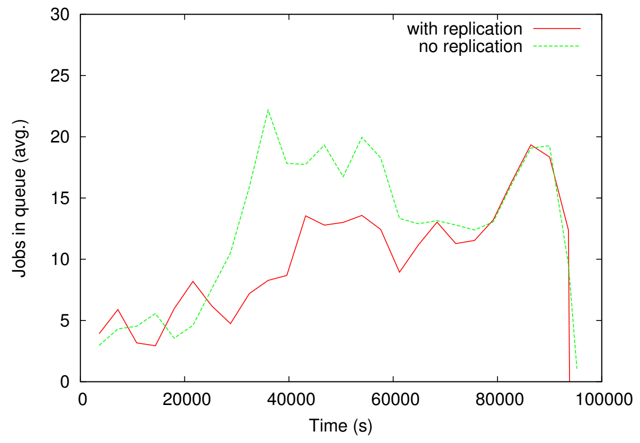

Figure 5. Average number of queued jobs

In Figure 5 we compare the two algorithms in terms of the number of jobs waiting to be matched in the Condor queue during the running time of the experiment. As expected, the number of jobs in the queue under the *with-replication* algorithm is almost always smaller than that of the *no-replication* algorithm. This is due to the fact that the *with-replication* algorithm removes jobs from the queue as soon as compute slots become available.

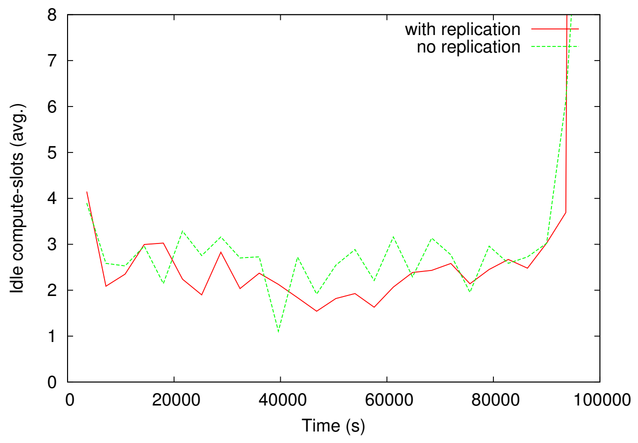

Figure 6. Average number of idle compute-slots

In Figure 6 we compare the system utilization under the two algorithms by counting the number of idle compute-slots during the running time of the experiment. Existence of idle compute-slots while there are jobs waiting in the Condor queue represent wasted system resources. Again the *with-replication* algorithm achieves better system utilization and shows almost always fewer idle compute-slots as compared with the *no-replication* algorithm.

Next we looked at the average time that jobs waited in the Condor queue under both algorithms. Waiting time in the queue is calculated as the number of seconds between the time a job arrives at the system until the time it is matched by Condor with some compute-slot and submitted for processing. The first important observation here is that the *with-replication* algorithm has a maximum waiting time of 4000 seconds whereas the no-replication has a maximum waiting time of 25000 seconds. This represents a dramatic improvement (a factor of 6) in terms of worst case behaviour. The mean waiting time of the *with-replication* algorithm is also better (by about 25%).

6 Conclusions and future plans

The main accomplishment described in this paper is the ability to put together a co-scheduling system from

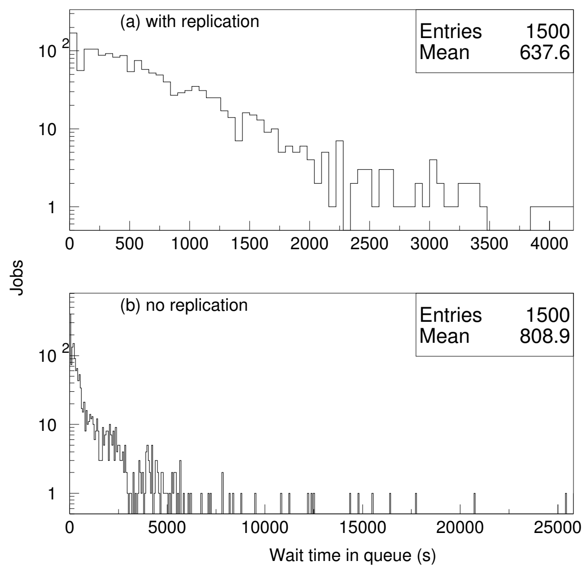

Figure 7. Waiting time in queue

existing Grid middleware components that were designed to manage compute and storage separately, by making relatively small enhancement to these systems. The problem of optimizing the behavior of these systems is the task of an external File Scheduling Module that makes file replication choices based on monitoring information of queues from the Condor scheduler. File replication is facilitated by making requests to the DRMs on each node. Garbage collection from the nodes disk caches are performed by the DRM based on usage policies. Thus, the tasks of scheduling, executing, monitoring, file movement, and garbage collection are performed by the existing Condor and DRM systems.

Future work includes trying out various optimization algorithms that are typically based on approximation results from scheduling theory [10] as computing optimal solutions for the co-scheduling problem is known to be NP-complete [11]. While we performed simulations to compare several algorithms, it is important to verify the performance on real systems. Furthermore, the next step will be to implement and test this combined co-scheduling system on Grid testbeds. Since the components we used on the cluster are general Grid components we believe that applying them in a real Grid system will be straightforward. The real challenge will be to measure and understand performance in an uncontrolled environment such as the Grid.

Acknowledgement

We thank Ekow Otoo from LBNL for his early participation in this project, and John Bent from the University of Wisconsin for pointing out relevant work. This work was supported by the Director, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under contract No. DE-AC03-76SF00098.

References

[1] A. Shoshani, A. Sim, and J. Gu, "Storage resource managers: Essential components for the grid," in *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, 2003.

[2] K. Ranganathan and I. Foster, "Decoupling computation and data scheduling in distributed data-intensive applications," in *Proc. 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11 2002)*, Edinburgh, Scotland, 23-26 July, 2002, pp. 352–358.

[3] W. Bell, D. Cameron, L. Capozza, A. Millar, K. Stockinger, and F. Zini, "Simulation of dynamic grid replication strategies in optorsim," in *Proc. Grid Computing - GRID 2002, Third International Workshop*, Baltimore, MD, USA, November 18 2002, pp. 46–57.

[4] T. Kosar and M. Livny, "Scheduling data placement activities in grid," University of Wisconsin - Madison Computer Sciences Department, Tech. Rep. UW-CS-TR-1483, July 2003.

[5] R. Buyya, H. Stockinger, J. Giddy, and D. Abramson, "Economic models for management of resources in peer-to-peer and grid computing," in *Proceedings of the SPIE International Conference on Commercial Applications for High-Performance Computing*, Denver, USA, August 20-24 2001.

[6] W. H. Bell, D. G. Cameron, A. P. M. Ruben Carvajal-Schiaffino, K. Stockinger, and F. Zini, “Evaluation of an economy-based file replication strategy for a data grid,” in *International Workshop on Agent based Cluster and Grid Computing at CCGrid 2003, Tokyo, Japan*. IEEE Computer Society, 2003.

[7] L. Capozza, K. Stockinger, and F. Zini, “Preliminary evaluation of revenue prediction functions for economically-effective file replication,” CERN Geneva, Switzerland, Tech. Rep. DataGrid-02-TED-020724, July 2002.

[8] J. B. Weissman, “Predicting the cost and benefit of adapting data parallel applications in clusters,” *J. Parallel Distrib. Comput.*, vol. 62, no. 8, pp. 1248–1271, 2002.

[9] J. Becla and D. L. Wang, “Lessons learned from managing a petabyte.” in *CIDR*, 2005, pp. 70–83.

[10] M.Pinedo., *Scheduling: Theory, Algorithms, and Systems*. Prentice Hall, 2001.

[11] J. Bent, D. Rotem, and A. Romosan, “Coordination of data movement with computation scheduling on a cluster,” LBNL, Tech. Rep. LBNL-55591, July 2002.