Evolving Toward the Perfect Schedule: Co-scheduling Job Assignments and Data Replication in Wide-Area Systems Using a Genetic Algorithm

Thomas Phan[1], Kavitha Ranganathan[2], and Radu Sion[3]

[1] IBM Almaden Research Center
phantom@us.ibm.com
[2] IBM T.J. Watson Research Center
kavithar@us.ibm.com
[3] Stony Brook University
sion@cs.stonybrook.edu

Abstract. Traditional job schedulers for grid or cluster systems are responsible for assigning incoming jobs to compute nodes in such a way that some evaluative condition is met. Such systems generally take into consideration the availability of compute cycles, queue lengths, and expected job execution times, but they typically do not account directly for data staging and thus miss significant associated opportunities for optimisation. Intuitively, a tighter integration of job scheduling and automated data replication can yield significant advantages due to the potential for optimised, faster access to data and decreased overall execution time. In this paper we consider data placement as a first-class citizen in scheduling and use an optimisation heuristic for generating schedules. We make the following two contributions. First, we identify the necessity for co-scheduling job dispatching and data replication assignments and posit that simultaneously scheduling both is critical for achieving good makespans. Second, we show that deploying a genetic search algorithm to solve the optimal allocation problem has the potential to achieve significant speed-up results versus traditional allocation mechanisms. Through simulation, we show that our algorithm provides on average an approximately 20-45% faster makespan than greedy schedulers.

1 Introduction

Traditional job schedulers for grid or cluster systems are responsible for assigning incoming jobs to compute nodes in such a way that some evaluative condition is met, such as the minimisation of the overall execution time of the jobs or the maximisation of throughput or utilisation. Such systems generally take into consideration the availability of compute cycles, job queue lengths, and expected job execution times, but they typically do not account directly for data staging and thus miss significant associated opportunities for optimisation. Indeed, the impact of data and replication management on job scheduling behaviour has largely remained unstudied. In this paper we investigate mechanisms that simultaneously schedule both job assignments and data replication and propose an optimised co-scheduling algorithm as a solution.

This problem is especially relevant in data-intensive grid and cluster systems where increasingly fast wide-area networks connect vast numbers of computation and storage resources. For example, the Grid Physics Network [10] and the Particle Physics Data Grid [18] require access to massive (on the scale of petabytes) amounts of data files for computational jobs. In addition to traditional files, we further anticipate more diverse and widespread utilisation of other types of data from a variety of sources; for example, grid applications may use Java objects from an RMI server, SOAP replies from a Web service, or aggregated SQL tuples from a DBMS.

Given that large-scale data access is an increasingly important part of grid applications, it follows that an intelligent job-dispatching scheduler must be aware of data transfer costs because jobs must have their requisite data sets in order to execute. In the absence of such awareness, data must be manually staged at compute nodes before jobs can be started (thereby inconveniencing the user) or replicated and transferred by the system but with the data costs neglected by the scheduler (thereby producing sub-optimal and inefficient schedules). Intuitively, a tighter integration of job scheduling and automated data replication potentially yields significant advantages due to the potential for optimised, faster access to data and decreased overall execution time. However, there are significant challenges to such an integration, including the minimisation of data transfers costs, the placement scheduling of jobs to compute nodes with respect to the data costs, and the performance of the scheduling algorithm itself. Overcoming these obstacles involves creating an optimised schedule that minimises the submitted jobs' time to completion (the "makespan") that should take into consideration both computation and data transfer times.

Previous efforts in job scheduling either do not consider data placement at all or often feature "last minute" sub-optimal approaches, in effect decoupling data replication from job dispatching. Traditional FIFO and backfilling parallel schedulers (surveyed in [8] and [9]) assume that data is already pre-staged and available to the application executables on the compute nodes, while workflow schedulers consider only the precedence relationship between the applications and the data and do not consider optimisation, e.g. [13]. Other recent approaches for co-scheduling provide greedy, sub-optimal solutions, e.g. [4] [19] [16].

This work includes the following two contributions. First, we identify the necessity for co-scheduling job dispatching and data replication and posit that simultaneously scheduling both is critical for achieving good makespans. We focus on a massively-parallel computation model that comprises a collection of heterogeneous independent jobs with no inter-job communication. Second, we show that deploying a genetic search algorithm to solve the optimal allocation problem has the potential to achieve significant speed-up results. In our work we observe that there are three important variables within a job scheduling system, namely the job order in the global scheduler queue, the assignment of jobs to compute nodes, and the assignment of data replicas to local data stores. There exists an optimal solution that provides the best schedule with the minimal makespan, but the solution space is prohibitively large for exhaustive searches. To find the best combination of these three variables in the solution space, we provide an optimisation heuristic using a genetic algorithm. By representing the three variables in a "chromosome" and allowing them to compete and evolve, the algorithm naturally converges towards an optimal (or near-optimal) solution.

We use simulations to evaluate our genetic algorithm approach against traditional greedy algorithms. Our experiments find that our approach provides on average an approximately 20-45% faster makespan than greedy schedulers. Furthermore, our work provides an initial promising look at how fine-tuning the genetic algorithm can lead to better performance for co-scheduling.

This paper is organised in the following manner. In Section 2 we discuss related work. We describe our model and assumptions in Section 3, present our genetic algorithm methodology in Section 4 and present the results of our simulation experiments in Section 5. We conclude the paper in Section 6.

2 Related Work

The need for scheduling job assignment and data placement together arises from modern clustered deployments. The work in [24] suggests I/O communities can be formed from compute nodes clustered around a storage system. Other researchers have considered the high-level problem of precedence workflow scheduling to ensure that data has been automatically staged at a compute node before assigned jobs at that node begin computing [7] [13]. Such work assumes that once a workflow schedule has been planned, lower-level batch schedulers will execute the proper job assignments and data replication. Our work fits into this latter category of job and data schedulers.

Other researchers have looked into the problem of job and data co-scheduling, but none have considered an integrated approach or optimisation algorithms to improve scheduling performance. The XSufferage algorithm [4] includes network transmission delay during the scheduling of jobs to sites but only replicates data from the original source repository and not across sites. The work in [19] looks at a variety of techniques to intelligently replicate data across sites and assign jobs to sites; the best results come from a scheme where local monitors keep track of popular files and preemptively replicate them to other sites, thereby allowing a scheduler to assign jobs to those sites that already host needed data. However, this work only considers jobs that use a single input file and assumes homogeneous network conditions. The Close-to-Files algorithm [16] assumes that single-file input data has already been replicated across sites and then uses an exhaustive algorithm to search across all combinations of compute sites and data sites to find the combination with the minimum cost, including computation and transmission delay. The Storage Affinity algorithm [21] treats file systems at each site as a passive cache; an initial job executing at a site must pull in data to the site, and subsequent jobs are assigned to sites that have the most amount of

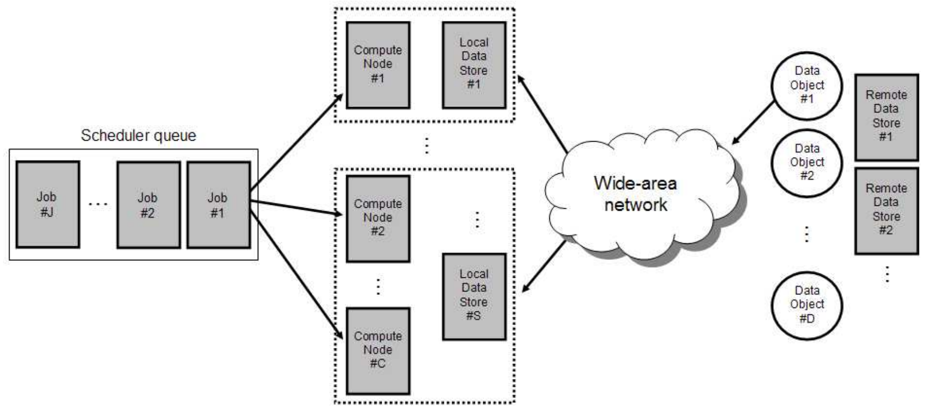

Fig. 1. A high-level overview of a job submission system in a generalised distributed grid. Note that although our work can be extended to multiple LANs containing clusters of compute nodes and local data stores (as is depicted here), for simplicity in this paper we consider only a single LAN.

needed residual data from previous application runs. The work in [5] decouples jobs scheduling from data scheduling: at the end of periodic intervals when jobs are scheduled, the popularity of needed files is calculated and then used by the data scheduler to replicate data for the next set of jobs, which may or may not share the same data requirements as the previous set.

Although these previous efforts have identified and addressed the problem of job and data co-scheduling, the scheduling is generally based on decoupled algorithms that schedule jobs in reaction to prior data scheduling. Furthermore, all these previous algorithms perform FIFO scheduling for only one job at a time, resulting in typically locally-optimum schedules only. On the other hand, we suggest a methodology to provide simultaneous co-scheduling in an integrated manner using global optimisation heuristics. In our work we execute a genetic algorithm that converges to a schedule by looking at the jobs in the scheduler queue as well as replicated data objects at once. While other researchers have looked at global optimisation algorithms for job scheduling [3] [22], they do not consider job and data co-scheduling. In the future, we plan to use simulations to compare the performance and benefits of our genetic algorithm with the other scheduling approaches listed above.

3 Job and Data Co-scheduling Model

Consider the scenario illustrated in Figure 1 that depicts a typical distributed grid or cluster deployment. Jobs are submitted to a centralised scheduler that queues the jobs until they are dispatched to distributed compute nodes. This scheduler can potentially be a meta-scheduler that assigns jobs to other local schedulers (to improve scalability at the cost of increased administration), but in our work we consider only a single centralised scheduler responsible for assigning jobs; in future work we look to extend this model to a decentralised meta-scheduling system.

The compute nodes are supported by local data stores capable of caching read-only replicas of data downloaded from remote data stores. These local data stores, depending on the context of the applications, can range from web proxy caches to data warehouses. We assume that the compute nodes and the local data stores are connected on a high-speed LAN (e.g. Ethernet or Myrinet) and that data can be transferred across the stores. (The model can be extended to multiple LANs containing clusters of compute nodes and data stores, but for simplicity we assume a single LAN in this paper.) Data downloaded from the remote store must cross a wide-area network such as the Internet. In the remainder of this paper, we use the term "data object" [23] to encompass a variety of potential data manifestations, including Java objects and aggregated SQL tuples, although its meaning can be intuitively construed to be a file on a file system.

Our model relies on the following key assumptions on the class of jobs being scheduled and the facilities available to the scheduler:

- √ The jobs are from a collection of heterogeneous independent jobs with no inter-job communication. As such, we do not consider jobs with parallel tasks (e.g. MPI programs).
- √ Data retrieved from the remote data stores is read-only. We only consider the class of applications that do not write back to the remote data store; for these applications, computed output is typically directed to the local file system at the compute nodes, and such output is commonly much smaller and negligible compared to input data.
- √ The computation time required by a job is known to the scheduler. In practical terms, when jobs are submitted to a scheduler, the submitting user typically assigns an expected duration of usage to each job [17].
- √ The data objects required to be downloaded for a job are known to the scheduler and can be specified at the time of job submission.
- √ The local data stores are assumed to have enough secondary storage to hold all data objects. In a more realistic setting of limited storage, a policy like LRU could be implemented for storage management.
- √ The communication cost for acquiring this data can be calculated for each job. The only communication cost we consider is transmission delay, which can be computed by dividing a data object's size by the bottleneck bandwidth between a sender and receiver. As such, we do not consider queueing delay or propagation delay.
    - If the data object is a file, its size is typically known to the job's user and specified at submission time. On the other hand, if the object is produced dynamically by a remote server, we assume that there exists a remote API that can provide the approximate size of the object. For example, for data downloads from a web server, one can use HTTP's HEAD method to get the requested URI's size prior to actually downloading it.

- The bottleneck bandwidth between two network points can be ascertained using known techniques [12] [20] that typically trade off accuracy with convergence speed. We assume such information can be periodically updated by a background process and made available to the scheduler.

√ Finally, we do not include arbitrarily detailed delays and costs in our model (e.g. database access time, data marshalling, or disk rotational latency), as these are dominated by transmission delay and computation time.

Given these assumptions, the lifecycle of a submitted job proceeds as follows. When a job is submitted to the queue, the scheduler assigns it to a compute node (using a traditional load-balancing algorithm or the algorithm we discuss in this paper). Each compute node maintains its own queue from which jobs run in FIFO order. Each job requires data objects from remote data stores; these objects can be downloaded and replicated on-demand to one of the local data stores (again, using a traditional algorithm or the algorithm we discuss in this paper), thereby obviating the need for subsequent jobs to download the same objects from the remote data store. In our work we associate a job to its required data objects through a Zipf distribution. All required data must be downloaded before a job can begin, and objects are downloaded in parallel at the time that a job is run. (Although parallel downloads will almost certainly reduce the last hop's bandwidth, for simplicity we assume that the bottleneck bandwidth is a more significant concern.) A requested object will always be downloaded from a local data store, if it exists there, rather than from the remote store. If a job requires an object that is currently being downloaded by another job executing at a different compute node, the job either waits for that download to complete or instantiates its own, whichever is faster based on expected download time maintained by the scheduler.

Intuitively, it can be seen that if jobs are assigned to compute nodes first, the latency incurred from accessing data objects may vary drastically because the objects may or may not have been already cached at a close local data store. On the other hand, if data objects are replicated to local data stores first, then the subsequent job executions will be delayed due to these same variations in access costs. Furthermore, the ordering of the jobs in the queue can affect the performance. For example, if job A is waiting for job B (on a different compute node) to finish downloading an object, job A blocks any other jobs from executing on its compute node. Instead, if we rearrange the job queue such that other shorter jobs run before job A, then these shorter jobs can start and finish by the time job A is ready to run. (This approach is similar to backfilling algorithms [14] that schedule parallel jobs requiring multiple processors.) The resulting tradeoffs affect the makespan.

With this scenario as it is illustrated in Figure 1, it can be seen that there are three independent variables in the system, namely (1) the ordering of the jobs in the global scheduler's queue, which translates to the ordering in the individual queue at each compute node, (2) the assignment of jobs in the queue to the individual compute nodes; and (3) the assignment of the data object replicas to the local data stores. The number of combinations can be determined as follows:

- $\surd$ Suppose there are J jobs in the scheduler queue. There are then $J!$ ways to arrange the jobs.
- $\surd$ Suppose there are C compute nodes. There are then C^J ways to assign the J jobs to these C compute nodes.
- $\surd$ Suppose there are D data objects and S local data stores. There are then S^D ways to replicate the D objects onto the S stores.

There are thus $J! \cdot C^J \cdot S^D$ different combinations of these three assignments. Within this solution space there exists some tuple of {job ordering, job-to-compute node assignment, object-to-local data store assignment} that will produce the minimal makespan for the set of jobs. However, for any reasonable deployment instantiation (e.g. J=20 and C=10), the number of combinations becomes prohibitively large for an exhaustive search.

Existing work in job scheduling can be analysed in the context presented above. Prior work in schedulers that dispatch jobs in FIFO order eliminate all but one of the $J!$ job orderings possible. Schedulers that assume the data objects have been preemptively assigned to local data stores eliminate all but one of the S^D ways to replicate. Essentially all prior efforts have made assumptions that allow the scheduler to make decisions from a drastically reduced solution space that may or may not include the optimal schedule.

The relationship between these three variables is intertwined. Although they can be changed independently of one another, adjusting one variable will have an adverse or beneficial effect on the schedule's makespan that can be counter-balanced by adjusting another variable. We analyse this interplay in Section 5 on results.

4 Methodology: a Genetic Algorithm

With a solution space size of $J! \cdot C^J \cdot S^D$, the goal is to find the schedule in this space that produces the shortest makespan. To achieve this goal, we use a genetic algorithm [2] as a search heuristic. While other approaches exist, each has its limitations. For example, an exhaustive search, as mentioned, would be pointless given the potentially huge size of the solution space. An iterated hill-climbing search samples local regions but may get stuck at a local optima. Simulated annealing can break out of local optima, but the mapping of this approach's parameters, such as the temperature, to a given problem domain is not always clear.

4.1 Overview

A genetic algorithm (GA) simulates the behaviour of Darwinian natural selection and converges toward an optimal solution through successive generations of recombination, mutation, and selection, as shown in the pseudocode of Figure 2 (adapted from [15]). A potential solution in the problem space is represented as a chromosome. In the context of our problem, one chromosome is a schedule that

```
Procedure genetic algorithm
{
  t = 0;
  initialise P(t);
  evaluate P(t);
  while (! done)
  {
        alter P(t);
        t = t + 1;
        select P(t) from P(t - 1);
        evaluate P(t);
  }
}
```

Fig. 2. Pseudocode for a genetic search algorithm. In this code, the variable t represents the current generation and P(t) represents the population at that generation.

consists of string representations of a tuple of {queue order, job assignments, object assignments}.

Initially a random set of chromosomes is instantiated as the population. The chromosomes in the population are evaluated (hashed) to some metric, and the best ones are chosen to be parents. In our context, the evaluation produces the makespan that results from executing the schedule of a particular chromosome. The parents recombine to produce children, simulating sexual crossover, and occasionally a mutation may arise which produces new characteristics that were not present in either parent; for simplification, in this work we did not implement the optional mutation. The best subset of the children is chosen, based on an evaluation function, to be the parents of the next generation. We further implemented elitism, where the best chromosome is guaranteed to be included in each generation in order to accelerate the convergence to an optimum, if it is found. The generational loop ends when some criteria is met; in our implementation we terminate after 100 generations (this value is an arbitrary number, as we had observed that it is large enough to allow the GA to converge). At the end, a global optimum or near-optimum is found. Note that finding the global optimum is not guaranteed because the recombination has probabilistic characteristics.

Using a GA is naturally suited in our context. The job queue, job assignments, and object assignments can be intuitively represented as character strings, which allows us to leverage prior genetic algorithm research in how to effectively recombine string representations of chromosomes (e.g. [6]).

It is important to note that a GA is most effective when it operates upon a large collection of possible solutions. In our context, the GA should look at a large window of jobs at once in order to achieve the tightest packing of jobs into a schedule. In contrast, traditional FIFO schedulers consider only the front job in the queue. The optimising scheduler in [22] uses dynamic programming and considers a large group of jobs called a “lookahead,” on the order of 10-50 jobs. In our work we call the collection of jobs a snapshot window. The scheduler takes this snapshot of queued jobs and feeds it into the scheduling algorithm.

Our simulation thus only models one static batch of jobs in the job queue. In the future, we will look at a more dynamic situation where jobs are arriving even as the current batch of jobs is being evaluated and dispatched by the GA. In such an approach, there will be two queues, namely one to hold incoming jobs and another to hold the latest snapshot of jobs that had been taken from the first queue. Furthermore, note that taking the snapshot can vary in two ways, namely by the frequency of taking the snapshot (e.g. at periodic wallclock intervals or when a particular queue size is reached) or by the size of the snapshot window (e.g. the entire queue or a portion of the queue starting from the front).

4.2 Workflow

The objective of the genetic algorithm is to find a combination of the three variables that minimises the makespan for the jobs. The resulting schedule that corresponds to the minimum makespan will be carried out, with jobs being executed on compute nodes and data objects being replicated to data stores in order to be accessed by the executing jobs. At a high level, the workflow proceeds as follows:

i. Jobs requests enter the system and are queued by the job scheduler.
ii. The scheduler takes a snapshot of the jobs in the queue and gives it to the scheduling algorithm.
iii. Given a snapshot, the genetic algorithm executes. The objective of the algorithm is to find the minimal makespan. The evaluation function, described in subsection 4.5, takes the current instance of the three variables as input and returns the resulting makespan. As the genetic algorithm executes, it will converge to the schedule with the minimum makespan.
iv. Given the genetic algorithm's output of an optimal schedule consisting of the job order, job assignments, and object assignments, the schedule is executed. Jobs are dispatched and executed on the compute nodes, and the data objects are replicated on-demand to the data stores so they can be accessed by the jobs.

4.3 Chromosomes

As mentioned previously, each chromosome consists of three strings, corresponding to the job ordering, the assignment of jobs to compute nodes, and the assignment of data objects to local data stores. We can represent each one as an array of integers. For each type of chromosome, recombination and mutation can only occur between strings representing the same characteristic. The initial state of the GA is a set of randomly initialised chromosomes.

Job ordering. The job ordering for a particular snapshot window can be represented as a queue (vector) of job unique identifiers. Note that the jobs can have their own range of identifiers, but once they are in the queue, they can

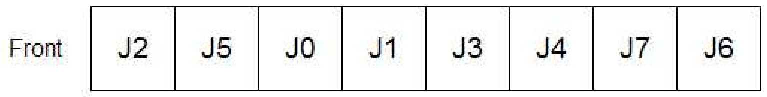

Fig. 3. An example queue of 8 jobs.

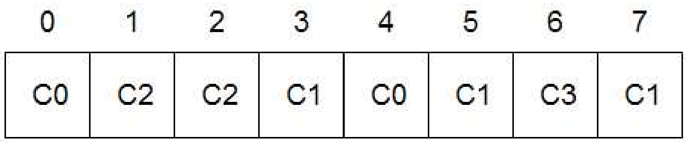

Fig. 4. An example mapping of 8 jobs to 4 compute nodes.

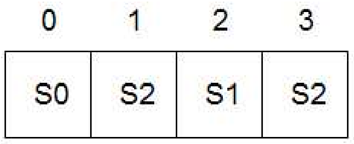

Fig. 5. An example assignment of 4 data objects to 3 local data stores.

be represented by a simpler range of identifiers going from job 0 to J-1 for a snapshot of J jobs. The representation is simply a vector of these identifiers. An example queue is shown in Figure 3.

Assignment of jobs to compute nodes. The assignments can be represented as an array of size J, and each cell in the array takes on a value between 0 and C-1 for C compute nodes. The i^{th} element of the array contains an identifier for the compute node to which job i has been assigned. An example assignment is shown in Figure 4.

Assignment of data object replicas to local data store. Similarly, these assignments can be represented as an array of size D for D objects, and each cell can take on a value between 0 and S-1 for S local data stores. The i^{th} element contains an integer identifier of the local data store to which object i has been assigned. An example assignment is shown in Figure 5.

4.4 Recombination and Mutation

Recombination is applied only to strings of the same type to produce a new child chromosome. In a two-parent recombination scheme for arrays of unique elements, we can use a 2-point crossover scheme where a randomly-chosen contiguous subsection of the first parent is copied to the child, and then all remaining items in the second parent (that have *not* already been taken from the first parent's subsection) are then copied to the child in order [6]. In a uni-parent mutation scheme, we can choose two items at random from an array and reverse the elements between them, inclusive. Note that in our experiments, we did not implement the optional mutation scheme, as we wanted to keep our GA as simple as possible in order to identify trends resulting from recombination. In the future we will explore ways of using mutation to increase the probability of finding global optima. Other recombination and mutation schemes are possible (as well as different chromosome representations) and will be explored in future work.

4.5 Evaluation Function

A key component of the genetic algorithm is the evaluation function. Given a particular job ordering, set of job assignments to compute nodes, and set of object assignments to local data stores, the evaluation function returns the makespan calculated deterministically from the algorithm described below. The rules use the lookup tables in Table 1. We note that the evaluation function is easily replaceable: if one were to decide upon a different model of job execution (with different ways of managing object downloads and executing jobs) or a different evaluation metric (such as response time or system saturation), a new evaluation function could just as easily be plugged into the GA as long as the same function is executed for all the chromosomes in the population.

At any given iteration of the genetic algorithm, the evaluation function executes to find the makespan of the jobs in the current queue snapshot. The pseudocode of the evaluation function is shown in Figure 6. We provide an overview of this function here.

The evaluation function considers all jobs in the queue over the loop spanning lines 6 to 37. As part of the randomisation performed by the genetic algorithm at a given iteration, the order of the jobs in the queue will be set, allowing the jobs to be dispatched in that order.

In the loop spanning lines 11 to 29, the function looks at all objects required by the currently considered job and finds the maximum transmission delay incurred by the objects. Data objects required by the job must be downloaded to the compute node prior to the job's execution either from the data object's source data store or from a local data store. Since the assignment of data object to local data store is known during a given iteration of the GA, we can calculate the transmission delay of moving the object from the source data store to the assigned local data store (line 17) and then update the NAOT table entry corresponding to this object (lines 18-22). Note that the NAOT is the next available time that the object is available for a final-hop transfer to the compute node regardless of the local data store. The object may have already been transferred to a different store, but if the current job can transfer it faster to its assigned store, then it will do so (lines 18-22). Also note that if the object is assigned to a local data store that is on the compute nodes' LAN, then the object must still be transferred across one more hop to the compute node (see line 23 and 26).

```
int evaluate(Queue, ComputeNodeAssignments, DataStoreAssignments)
{
    makespan = 0;
    clock = getcurrenttime();

    foreach job J in Queue
    {
        // This job J is assigned to compute node C.

        maxTD = 0; // maximum transmission delay across all objects
        foreach object Oi required by this job
        {
            // This data object O resides originally in Ssource and is
            //   assigned to Sassigned.

            // calculate the transmission delay for this object
            TD = SIZE(Oi) / BANDWIDTH(Ssource, Sassigned);
            if ((clock+TD) < NAOT(Oi))
            {
                NAOT(Oi) = clock + TD;
                // file transfer from Ssource to Sassigned would occur here
            }
            finalHopDelay = SIZE(Oi) / BANDWIDTH(Sassigned, C); // optional

            // keep track of the maximum transmission delay
            maxTD = MAX(maxTD, NAOT(Oi) + finalHopDelay);

            // file transfer from Sassigned to compute node C would occur here
        }

        startComputeTime = NACT(C)+ maxTD;
        completionTime = startComputeTime + COMPUTE(J, C);
        NACT(C) = MAX(NACT(C), completionTime);

        // keep track of the largest makespan across all jobs
        makespan = MAX(makespan, completionTime);
    }
    return makespan;
}
```

Fig. 6. Evaluation function for the genetic algorithm.

Lookup table	Comment
REQUIRES (Job J_i, DataObject O_i)	1 if Job J_i requires/accesses Object O_i.
COMPUTE (Job J_i, ComputeNode C_i)	The time for Job J_i to execute on compute node C_i.
BANDWIDTH (Site a, Site b)	The bottleneck bandwidth between two sites. The sites can be data stores or compute nodes.
SIZE (DataObject O_i)	The size of object O_i (e.g. in bytes).
NACT (ComputeNode C_i)	Next Available Compute Time: the next available time that a job can start on compute node C_i.
NAOT (Object O_i)	Next Available Object Time: the next available time that an object O_i can be downloaded.

Table 1. Lookup tables used in the GA's evaluation function.

Lines 31 and 32 compute the start and end computation time for the job at the compute node. Line 36 keeps track of the largest completion time seen so far for all the jobs. Line 38 returns the resulting makespan, i.e. the longest completion time for the current set of jobs.

5 Experiments and Results

To show the effectiveness of the GA in improving the scheduling, we simulated our GA and a number of traditional greedy FIFO scheduler algorithms that dispatch jobs (to random or to least-loaded compute nodes) and replicate data objects (no replication or to random local data stores). We used a simulation program developed in-house that maintains a queue for the scheduler, queues for individual compute nodes, and simulation clocks that updates the simulation time as the experiments progressed. We ran the simulations on a Fedora Linux box running at 1 Ghz with 256 MB of RAM.

5.1 Experimental Setup

Our aim is to compare the performance of different algorithms to schedule jobs. Since all the algorithms use some randomisation in their execution, it was important to normalise the experiments to achieve results that could be compared across different schemes. We thus configured the algorithm simulations to initially read in startup parameters from a file (e.g. the jobs in the queue, the job assignments, the object assignments, etc.) that were all randomly determined beforehand. All experiments were performed with three different initialisation sets with ten runs each and averaged; the graphs represent this final average for any particular experiment. The experimental parameters were set according to values shown in Table 2.

Experimental parameter	Comment
Queue size	Varies by experiment; 40-160
Number of compute nodes	Varies; 5-20
Number of local data stores	Varies; 5-20
Number of remote data stores	20
Number of data objects	50
Data object popularity	Based on Zipf distribution
Average object size	Uniformly distributed, 50-1500 MB
Average remote-to-local store bandwidth	Uniformly distributed, 700-1300 kbps
Average local store-to-compute node bandwidth	Uniformly distributed, 7000-13000 kbps
GA: number of parents	Varies; typically 10
GA: number of children	Varies; typically 50
GA: number of generations	100

Table 2. Experimental parameters

The simulations use a synthetic benchmark based on CMS experiments [11] that are representative of the heterogeneous independent tasks programming model. Jobs download a number of data objects, perform execution, and terminate. Data objects are chosen based on a Zipf distribution [1]. The computation time for each job is kD seconds, where k is a unitless coefficient and D is the total size of the data objects downloaded in GBytes; in our experiments k is typically 300 (although in subsection 5.2 this value is varied).

5.2 Results

We first wanted to compare the GA against several greedy FIFO scheduling algorithms. In the experiments the naming of the algorithms is as follows:

√ Genetic algorithms (2 variations):
- all varying: the genetic algorithm with all three variables allowed to evolve
- rep-none: the genetic algorithm with the job queue and the job assignments allowed to evolve, but the objects are not replicated (a job must always download the data object from the remote data store)

√ Greedy algorithms (2x2 = 4 variations):
Job dispatching strategies
- jobs-LL: jobs are dispatched in FIFO order to the compute node with the shortest time until next availability
- jobs-rand: jobs are dispatched in FIFO order to a random compute node

Data replication strategies
- rep-none: objects are not replicated (a job must always download the data object from the remote data store)
- rep-rand: objects are replicated to random local data stores

Makespans for Various Algorithms In this experiment, we ran the six algorithms with 20 compute nodes, 20 local data stores, and 100 jobs in the queue. Two results, as shown in Figure 7, can be seen. First, as expected, data placement/replication has a strong impact on the resulting makespan. Comparing the three pairs of experiments that vary by having replication activated or deactivated, namely (1) GA all varying and GA rep-none, (2) Greedy, jobs-LL, rep-none and Greedy, jobs-LL, rep-rand, and (3) Greedy, jobs-rand, rep-none and Greedy, jobs-rand, rep-rand, we can see that in the absence of an object replication strategy, the makespan suffers. Adding a replication strategy improves the makespan because object requests can be fulfilled by the local data store instead of by the remote data store, thereby reducing access latency as well as actual bandwidth utilisation (this latter reduction is potentially important when bandwidth consumption is metered).

The second result from this experiment is that the GA with all varying parameters provides the best performance of all the algorithms. Its resulting makespan is 22% faster than the best greedy algorithm (Greedy, jobs-LL, rep-rand) and 47% faster than the worst greedy algorithm (Greedy, jobs-rand, rep-none). To better explain the result of why the GA is faster than the greedy algorithm, we ran another experiment with 5 compute nodes and 5 local data stores, as shown in Figure 8.

As can be seen, the performance of the GA is comparable to that of the greedy algorithms. This result is due to the fact that with the reduced number of compute nodes and local data stores, the solution space becomes smaller, and both types of algorithms become more equally likely to come across an optimum solution. If we restrict our attention to just the assignment of the 100 jobs in the queue, in the previous experiment with 20 compute nodes there are 20^{100} possible assignments, whereas with 5 compute nodes there are only 5^{100} possible assignments, a difference in the order of 10^{60}. With the larger solution space in the previous experiment, the variance of makespans will be larger, thus allowing the GA to potentially find a much better solution. It can be seen that in these scenarios where the deployment configuration of the grid system contains a large number of compute nodes and local data stores, a GA approach tends to compute better schedules.

Effect of Queue Length In this experiment we ran the same application but with varying numbers of jobs in the queue and with 20 compute nodes and 20 local data stores; Figure 9 shows the results. For conciseness, we show only the best GA (GA all varying) and the best greedy algorithm (Greedy, jobs-LL, rep-rand). It can be seen that the GA performs consistently better than the greedy algorithm, although with an increasing number of jobs in the queue, the difference between the two algorithms decreases. We suspect that as more jobs are involved, the number of permutations increases dramatically (from 40! to 160!), thereby producing too large of a solution space for the GA to explore in 100 generations. Although in the previous subsection we observed that increasing the solution space provides a more likely chance of finding better solutions,

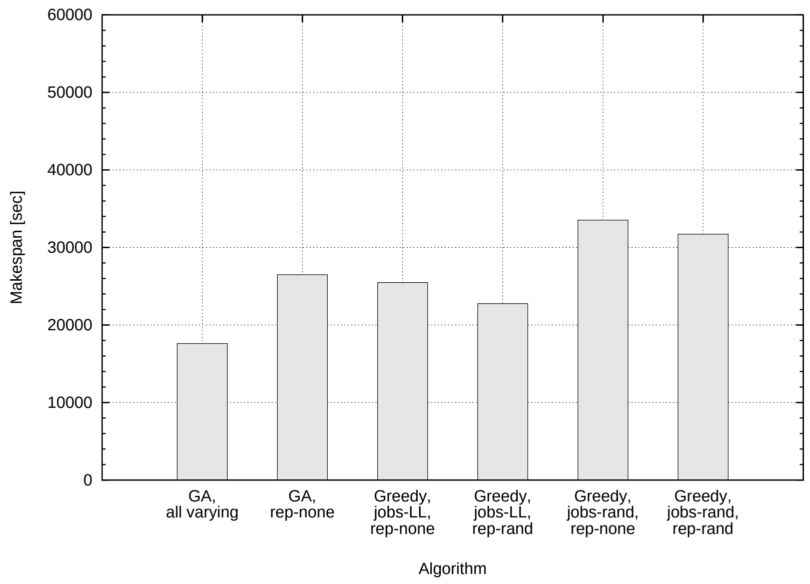

Fig. 7. Makespans for various algorithms using 20 compute nodes and 20 local data stores.

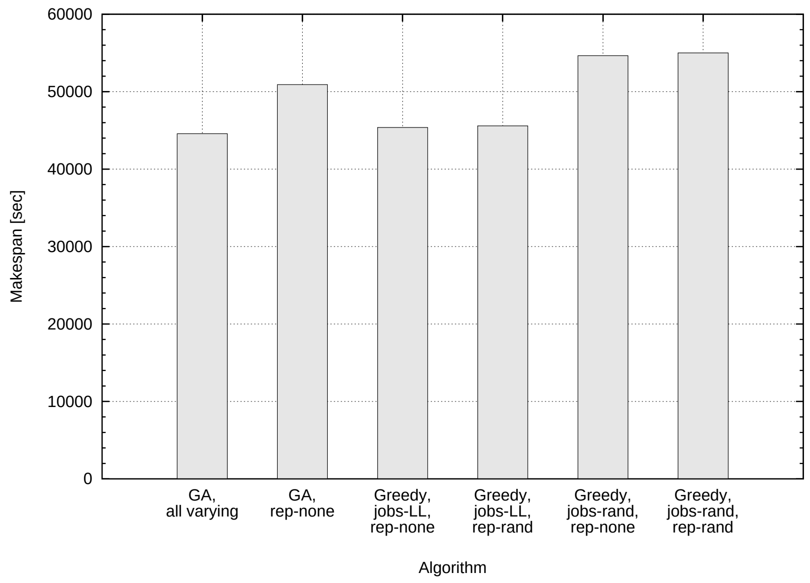

Fig. 8. Makespans for various algorithms using 5 compute nodes and 5 local data stores.

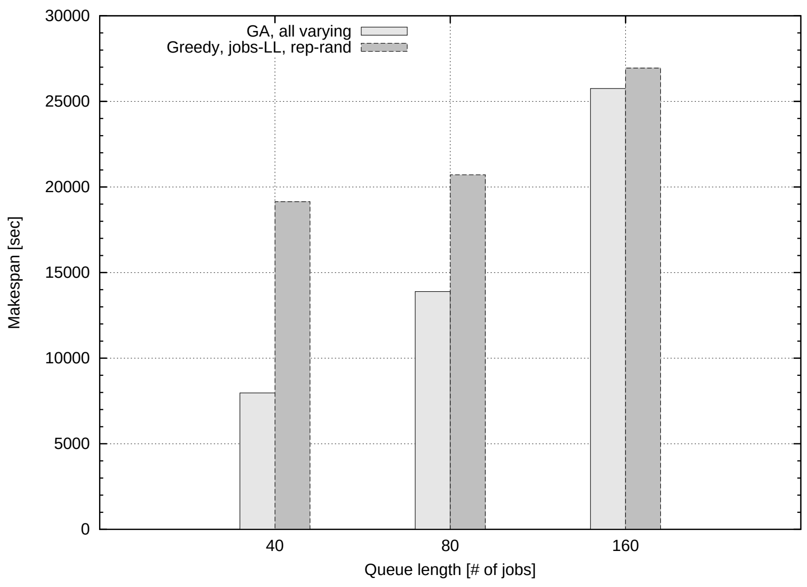

Fig. 9. Makespans for different queue lengths.

we conjecture that there is a trade-off point somewhere; we are continuing to investigate this issue.

Effect of Computation Ratio Coefficients In previous experiments we set the computation coefficient to be 300 as mentioned in subsection 5.1. In Figure 10 we show the effect of changing this value. With a smaller coefficient, jobs contain less computation with the same amount of communication delay, and with a larger coefficient, jobs contain more computation. As can be seen, as the coefficient increases, the difference between the GA and the greedy algorithms decreases. This result stems from the fact that when there are more jobs with smaller running times (which includes both computation and communication), the effect of permuting the job queue is essentially tantamount to that of backfilling in a parallel scheduler: when a job is delayed waiting, other smaller jobs with less computation can be run before the long job, thereby reducing the overall makespan.

Effect of Population Size In Figure 11 we show the effect of population size on the makespan produced by the GA. In all previous experiments, we had been running with a population comprising 10 parents spawning 50 children per generation. We can change the population characteristics by varying two parameters: the number of children selected to be parents per generation and the ratio of parents to children produced. The trend shown in the figure is that as the population size increases, there are more chromosomes from which to choose,

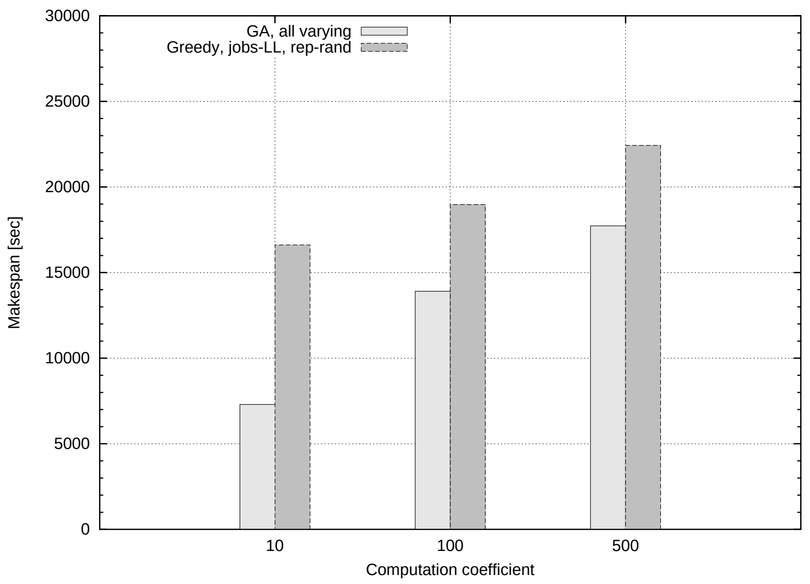

Fig. 10. Makespans for different computation coefficients.

thereby increasing the probability that one of them may contain the optimum solution. As expected, the best makespan results from the largest configuration in the experiment, 50 parents and a ratio of 1 parent to 50 children.

However, this accuracy comes at the cost of increased running time of the algorithm. As the population size increases, the time to execute the evaluation function on all members increases as well. As can be seen in Figure 12, the running time accordingly increases with the population size. This tradeoff of running time against the desire to find the optimal solution can be made by the scheduler's administrator. For completeness, we note that the greedy algorithms typically executed in under 1 second. While this performance is faster than that of the GA, this distinction is dwarfed by the difference between the makespans produced by greedy algorithms and the GA; as was shown in Figure 4 for this benchmark, the makespan difference can be on the order of thousands of seconds.

6 Conclusion and Future Work

In this paper we looked at the problem of co-scheduling job dispatching and data replication in wide-area distributed systems in an integrated manner. In our model, the system contains three variables, namely the order of the jobs in the global scheduler queue, the assignment of jobs to the individual compute nodes, and the assignment of the data objects to the local data stores. The solution space is enormous, making an exhaustive search to find the optimal tuple of these three variables prohibitively costly. In our work we showed that a genetic

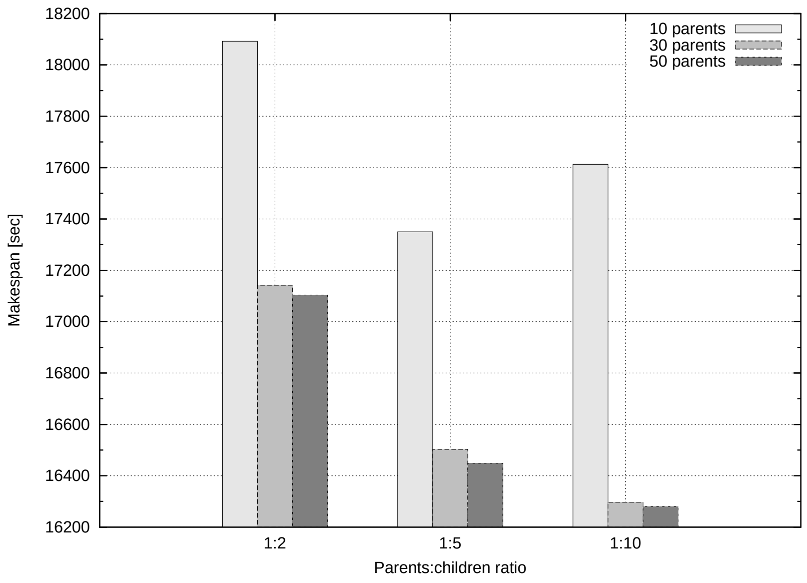

Fig. 11. Makespans for different GA ratios of parents to children.

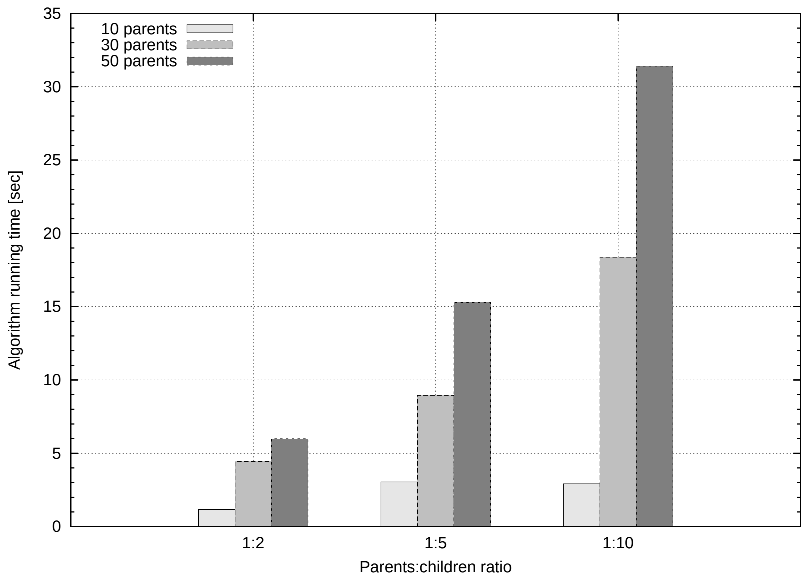

Fig. 12. GA running times for different ratios of parents to children.

algorithm is a viable approach to finding the optimal solution. Our simulations show our implementation of a GA produces a makespan that is 20-45% faster than traditionally-used greedy algorithms.

For future work, we plan to do the following:

- √ More comprehensive comparisons. We look to simulate other approaches that can be used to perform co-scheduling, including those found in the related work section as well as other well-known scheduling algorithms, such as traditional backfilling, shortest-job-first, and priority-based scheduling.
- √ Handling inaccurate estimates. Our evaluation function used in the GA relies on the accuracy of the estimates for the data object size, bottleneck bandwidth, and job computation time. However, these estimates may be extremely inaccurate, leading the GA to produce inefficient schedules. In the future we will look into implementing a fallback scheduling algorithm, such as those in the related work, when the scheduler detects widely fluctuating or inaccurate estimates. Additionally, we will research different evaluation functions and metrics that may not be dependent on such estimates.
- √ Improved simulation. We plan to run a more detailed simulation with real-world constraints in our model. For example, we are looking at nodal topologies, more accurate bandwidth estimates, and more detailed evaluation functions that consider finer-grained costs and different models of job execution.
- √ More robust GA. Alternative genetic algorithm methodologies will also be explored, such as different representations, evaluation functions, alterations, and selections. Furthermore, we conjecture that since all three variables in the chromosome were independently evolved, there may be conflicting interplay between them. For instance, as the job queue permutations evolves to an optimum, the job assignments may have evolved in the opposite direction; the latter situation might occur because the job queue evolution has a greater impact on the evaluation function. In the future we will look into ways of hashing all three variables into a single string for the chromosome so that there will be reduced interplay.

7 Acknowledgments

We would like to thank the anonymous paper reviewers for their invaluable comments and insight.

References

1. L. Adamic. "Zipf, Power-laws, and Pareto – a ranking tutorial," `www.hpl.hp.com/research/idl/papers/ranking/ranking.html`
2. T. Baeck, D. Fogel, and Z. Michalewicz (eds). "Evolutionary Computation 1: Basic Algorithms and Operators," Institute of Physics Publishing, 2000.
3. T. Braun, H. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hengsen, and R. Freund. "A Comparison of Eleven

Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," Journal of Parallel and Distributed Computing, vol. 61, no. 6, June 2001.

4. H. Casanova, A. Legrand, D. Zagorodnov, F. Berman. "Heuristics for Scheduling Parameter Sweep Applications in Grid Environments," In *Proceedings of the 9th Heterogeneous Computing Workshop*, May 2000.
5. A. Chakrabarti, D. R. A., and S. Sengupta. "Integration of Scheduling and Replication in Data Grids," In *Proceedings of the International Conference on High Performance Computing*, 2004.
6. L. Davis. "Job Shop Scheduling with Genetic Algorithms," In *Proceedings of the International Conference on High Performance Computing*, 1985.
7. E. Deelman, T. Kosar, C. Kesselman, and M. Livny. "What Makes Workflows Work in an Opportunistic Environment?" Concurrency and Computation: Practice and Experience, 2004.
8. D. Feitelson. "A Survey of Scheduling in Multiprogrammed Parallel Systems," *IBM Research Report RC 19790 (87657)*, 1994.
9. D. Feitelson, L. Rudolph, and U. Schwiegelshohn. "Parallel Job Scheduling – A Status Report," In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, 2004.
10. The Grid Physics Project. `www.griphyn.org`
11. K. Holtman. "CMS Requirements for the Grid," In *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics*, 2001.
12. N. Hu, L. Li, Z. Mao, P. Steenkiste, and J. Wang. "Locating Internet Bottlenecks: Algorithms, Measurements, and Implications," In *Proceedings of SIGCOMM*, 2004.
13. T. Kosar and M. Livny. "Stork: Making Data Placement a First Class Citizen in the Grid," In *Proceedings of IEEE International Conference on Distributed Computing Systems*, 2004.
14. D. Lifka. "The ANL/IBM SP Scheduling System," In *Job Scheduling Strategies for Parallel Processing*, Lecture Notes on Compute Science, Springer-Verlag 1995.
15. Z. Michaelewicz and D. Fogel. How to Solve It: Modern Heuristics, Springer-Verlag, 2000 ,
16. H. Mohamed and D. Epema. "An Evaluation of the Close-to-Files Processor and Data Co-Allocation Policy in Multiclusters," In *Proceedings of the IEEE International Conference on Cluster Computing*, 2004.
17. A. Mu'alem and D. Feitelson. "Utilization, Predictability, Workloads,and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling," IEEE Transactions on Parallel and Distributed Systems, June 2001.
18. The Particle Physics Data Grid, `www.ppdg.net`
19. K. Ranganathan and I. Foster. "Computation Scheduling and Data Replication Algorithms for Data Grids," Grid Resource Management: State of the Art and Future Trends, J. Nabrzyski, J. Schopf, and J. Weglarz, eds. Kluwer Academic Publishers, 2003.
20. V. Ribeiro, R. Riedi, and R. Baraniuk. "Locating Available Bandwidth Bottlenecks," *IEEE Internet Computing*, September-October 2004.
21. E. Santos-Neto, W. Cirne, F. Brasileiro, and A. Lima. "Exploiting Replication and Data Reuse to Efficiently Schedule Data-Intensive Applications on Grids," In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, 2004.

22. E. Schmueli and D. Feitelson. "Backfilling with Lookahead to Optimize the Packing of Parallel Jobs," Springer-Verlag Lecture Notes in Computer Science, vol. 2862, 2003.
23. H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman, and B. Tierney. "File and Object Replication in Data Grids," In *Proceedings of the 10th International Symposium on High Performance Distributed Computing*, 2001.
24. D. Thain, J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. "Gathering at the Well: Creating Communities for Grid I/O", In Proceedings of Supercomputing, 2001.