Scheduling in Heterogeneous Grid Environments: The Effects of Data Migration

Hongzhang Shan, Leonid Oliker
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720
{hshan,loliker}@lbl.gov

Warren Smith, Rupak Biswas
NASA Advanced Supercomputing Division
NASA Ames Research Center
Moffett Field, CA 94035
{wwsmith,rbiswas}@mail.arc.nasa.gov

***Abstract*— Computational grids have the potential for solving large-scale scientific problems using heterogeneous and geographically distributed resources. However, a number of major technical hurdles must be overcome before this goal can be fully realized. One problem critical to the effective utilization of computational grids is efficient job scheduling. Our prior work addressed this challenge by defining a grid scheduling architecture and several job migration strategies. The focus of this study is to explore the impact of data migration under a variety of demanding grid conditions. We evaluate our grid scheduling algorithms by simulating compute servers, various groupings of servers into sites, and inter-server networks, using real workloads obtained from leading supercomputing centers. Several key performance metrics are used to compare the behavior of our algorithms against reference local and centralized scheduling schemes. Results show the tremendous benefits of grid scheduling, even in the presence of input/output data migration — while highlighting the importance of utilizing communication-aware scheduling schemes.**

I. Introduction

One of the primary goals of grid computing [5], [6] is to provide shared access to geographically distributed heterogeneous resources in a transparent manner. There will be many benefits when this goal is fully realized, including the ability to execute applications whose computational requirements exceed local resources, and the reduction of job turnaround time through load balancing across multiple computing facilities. The development of computational grids and associated middleware has therefore been actively pursued in recent years. However, many technical hurdles stand in the way of achieving these objectives. Among the myriad research issues to be addressed is the problem of distributed resource management and job scheduling for computational grids. Although numerous researchers have studied this problem [1], [2], [4], [7], [12], [13], the effect of transferring input/output data files for the migrated jobs on overall grid performance has not been comprehensively analyzed, particularly the conditions under which the network communication cost begins to dramatically affect the job turnaround time. This is the focus of this paper.

In our previous work [12], we developed a grid scheduling architecture that consisted of autonomous local schedulers and distributed grid schedulers. The local schedulers schedule access to individual computer systems, while the grid schedulers (paired with local schedulers) send jobs to the corresponding local schedulers and/or migrate jobs between grid schedulers. We also proposed several algorithms for transferring jobs between grid schedulers. These algorithms strive to move jobs when wait times at the compute servers rise above or fall below specific thresholds. The migration strategies determine whether to send and/or receive jobs using the resource requirements of each job, the availability of computational resources, and the actual performance of the computer systems. For completeness, the scheduling architecture and migration algorithms are described in Sections II and III, respectively. However, the volume of input/output data across the limited network bandwidth between systems had not been taken into consideration in our prior work. Here, we extend our algorithms to include these important factors. The goal is to examine how the performance advantages of a computational grid are impacted as these parameters are varied.

Scheduling algorithms that consider data transfer overhead have been proposed; however, they are primarily suited for data-intensive applications where the data is shared among multiple jobs or used repeatedly. Ranganathan and Foster [10] developed an algorithm in which job scheduling and data movement are decoupled. It includes two components: an external scheduler that selects the destination for the job, and a dataset scheduler responsible for data replication. However, our simulation results show that serious performance degradation will occur for slower networks if data migration is not considered when scheduling jobs. Another strategy, called the Storage Affinity algorithm proposed by Santos-Neto et al. [11], tracks the location of data and replicates portions to produce schedules that avoid, as much as possible, large data transfers. This is suitable in environments where the data is frequently reused by different tasks.

XSufferage [3], by Casanova et al., also considers data location when scheduling tasks to exploit reuse. Compared with these techniques, our focus is quite different. Instead of investigating intelligent ways to reuse data, our objective is to understand the conditions under which the data transfer overhead must be considered in distributed job scheduling, and how the advantages of a grid environment are affected by variations in this parameter. We therefore assume no data reuse. All input files are resident on the host where the job is submitted, and all output files must be redirected to the same machine. This is true for most scientific applications.

We evaluate our grid scheduling algorithms by simulating compute servers, various groupings of servers into sites, and inter-server networks, and drive these simulations using real workloads derived from trace data gathered from leading supercomputing centers over the same time period. The experimental methodology is described in Section IV. We gather several key performance metrics, and use them to compare the behavior of our algorithms against reference local and centralized scheduling schemes. Our specific objective in this paper is to understand the effects of data migration.

Our results, presented in Section V, show that our best scheduling strategy delivers turnaround times that are 60% smaller than those without grid scheduling, even in the presence of input/output data migration. Alternatively, our algorithm can execute 40% more jobs in the grid environment and deliver the same turnaround times as in a non-grid scenario. Finally, for large data files (or slow networks), we find that it is imperative to consider data transfer times when making job migration decisions as results show an increase of up to 43x in job turnaround times if data migration overhead is ignored.

II. Grid Scheduling Architecture

We use a simple grid scheduling architecture, shown in Fig. 1, for evaluating our proposed job migration algorithms. The architecture is composed of distributed compute servers, local schedulers with local queues, and grid schedulers with grid queues. A new job is always submitted to the *grid scheduler* (GS) of the compute server with which it has an "affinity" and placed in the associated *grid queue* (GQ). The GS then analyzes the job's resource requirements after gathering local information from the corresponding *local scheduler* (LS) and remote information from its peer GSs. The LS provides data about the *local queue* (LQ) and the local compute server, while other GSs supply data about remote sites. Based on all this information, the GS determines whether to send the job from the GQ to its own LQ or to the LQ of another server through the grid middleware and appropriate GS. Once a job is placed in a LQ, the LS schedules it for execution on the compute server using

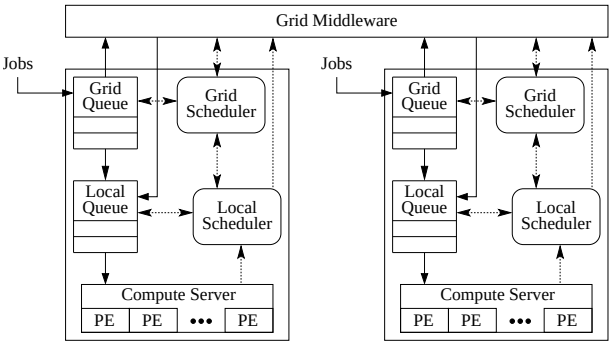

Fig. 1. Our grid scheduling architecture (solid arrows represent movement of jobs, dashed arrows represent transfer of information).

the local scheduling policy. One issue not addressed in this work is how, in practice, a GS locates other GSs. We expect to utilize traditional peer-to-peer (P2P) strategies that use centralized or distributed indices, and plan to examine it in detail at a later time.

There are other grid scheduling architectures that we could have adopted. A centralized scheme with a single scheduler for multiple computer systems might be a good choice for a relatively small set of servers at a single site, but the approach does not scale and is not fault tolerant. A hierarchy of grid schedulers organized into a tree where jobs flow up and down is an interesting approach [7], but we do not expect it to scale as well as a P2P strategy. A variation of our architecture combines pairs of local and grid schedulers into a single scheduler. This is starting to occur as vendors adopt a grid perspective to scheduling [8], [9], but these systems do not interoperate and are not yet widely used.

Another approach to grid scheduling is where users employ user-level schedulers to select the compute servers for submitting their applications [2]. This strategy is somewhat similar to our P2P method, the difference being that user-level grid schedulers seek to optimize the execution of jobs for a single user while our grid schedulers strive to optimize the execution of all jobs. We believe this distinction results in the P2P grid scheduling approach having potentially greater overall performance. In the end, we chose a P2P architecture with a grid scheduler co-located with each local scheduler. This strategy [12] gives us the best scalability, fault tolerance, and scheduling performance without requiring that sites replace their local schedulers.

III. Grid Scheduling Algorithms

This section presents the three distributed scheduling algorithms that are the subject of this work, and the two reference algorithms against which they are compared. The three distributed algorithms are *sender-initiated*, *receiver-initiated*, and *symmetrically-initiated*. All three operate in a P2P manner but use different strategies

for migrating jobs between grid schedulers. The first reference algorithm is *centralized* that uses a single grid scheduler interacting with all local schedulers. The second reference algorithm is *local* that has no grid schedulers and executes all jobs on the compute server where they are submitted. Details of all five methods are given in the following subsections.

A. Distributed Algorithms

Our three distributed algorithms are based on these common primary steps:

- A job j is submitted to a GS on compute server s_i and placed in the associated GQ.
- The GS queries the LS on s_i for the *approximate wait time* (*AWT*) of j. *AWT* is the amount of time LS estimates j, if submitted to it, will wait in LQ before beginning to execute. *AWT* is computed by simulating the local scheduling policy using the local jobs that are either running or waiting in LQ, and j. If LS cannot satisfy the resource requirements of j, an *AWT* of infinity is returned.
- The GS compares *AWT* for j against a threshold ϕ. If the *AWT* is less than ϕ, j is moved from GQ to LQ for execution on s_i. Otherwise, j is retained in GQ and one of the following three distributed job migration algorithms is invoked.

1) Sender-Initiated: In the sender-initiated (S-I) strategy, the GS sends the resource requirements of j to its peers. In this study, we only consider the CPU and run time requirements of each job; however, this can be easily extended to an arbitrary number of resource constraints. In response to the query, each peer GS returns the *approximate turnaround time* (*ATT*) for j and the *resource utilization* (*RU*) of the associated compute server. If a peer GS does not respond within a specified time limit due to traffic congestion or machine failure, it is simply ignored for that request.

ATT is an estimate of the amount of time it will take to complete a job. The *ATT* for j on compute server s_f initially submitted to s_i is derived as follows:

$$\begin{aligned} \textit{ATT}(j, s_f) = \max(&\textit{AWT}(j, s_f), \textit{ADT}(j_{in}, s_i, s_f)) + \\ &\textit{ERT}(j, s_f) + \textit{ADT}(j_{out}, s_f, s_i)). \end{aligned}$$

Before j begins to execute, it must wait in a LQ and transfer input data to s_f. $\textit{AWT}(j, s_f)$ is the approximate wait time of j on s_f while $\textit{ADT}(j_{in}, s_i, s_f)$ is the *approximate data transfer time* of j's input data j_{in} from s_i to s_f. We assume these activities can be performed simultaneously, so the maximum of the two constraints determines when j can begin executing. The job then runs on s_f with an *expected run time* of $\textit{ERT}(j, s_f)$, and the output data j_{out} is transferred back to s_i in time $\textit{ADT}(j_{out}, s_f, s_i)$. Note that *ERT* can vary from one compute server to another depending on their architectural designs and program characterizations. We simplify the calculation of *ERT* by assuming that run time is only related to the clock frequency of the compute server.

RU is the fraction of the computer server that is currently being utilized. We assume our compute servers have multiple CPUs that are space shared so we calculate *RU* as the fraction of CPUs assigned to jobs.

Based on all collected information, the GS at server s_i where j is initially submitted calculates its local *ATT* and compares it against the values from each peer that responded. If the local *ATT* is within a factor τ of the minimum *ATT*, j is scheduled for execution on s_i; otherwise, j is migrated (the migration threshold τ acts as a gate to discourage excessive job movement). In case multiple machines respond with *ATT* values that are within a small tolerance ϵ, the server with the lowest *RU* is chosen to accept j. This heuristic process attempts to minimize the user's time-to-solution, while using system utilization as a tiebreaker. We found this approach to be more effective than simply relying on *ATT*. The job is then sent to the LQ (by way of its partner GS and LS) on the winning compute server. Note that once a job enters a LQ, it is scheduled and run based exclusively on the policy of the LS, and cannot migrate to another site.

2) Receiver-Initiated: The receiver-initiated (R-I) algorithm takes a more passive approach to job migration. Here, each compute server periodically checks its own *RU* at time interval σ. If the RU is below a threshold δ, the system volunteers itself for receiving jobs by informing other machines of its low utilization. Once a peer GS at server p receives this information, it checks its GQ to see if any jobs are waiting to be scheduled. If so, the resource requirements of the first job are sent to the volunteer server. The underutilized system then responds with the job's *ATT*, as well as its own *RU*. If the *ATT* of the volunteer system is lower than that of p (or if the local and remote *ATT*'s are within the tolerance ϵ but the *RU* of the volunteer is smaller), the job is transferred to the LQ of that system. Otherwise, the job continues to wait in the GQ until either its local *AWT* falls below ϕ (examined at time interval σ), or an available machine again volunteers its services.

3) Symmetrically-Initiated: Unlike S-I and R-I, the symmetrically-initiated (SY-I) algorithm works in both active and passive modes. As with the R-I strategy, each machine periodically checks its own *RU* and broadcasts a message if it is underutilized. The difference occurs when the local *AWT* of a job exceeds ϕ but no underutilized machine volunteers its services. In the R-I approach, the job passively sits in the GQ while waiting for a volunteer, periodically checking its local *AWT* at each σ time interval. However, the SY-I algorithm immediately switches to active mode and sends out a request using the S-I strategy. The main differences in the

three job migration algorithms therefore lie in the timing of the job transfer request initiations and the destination choice for those requests.

B. Reference Algorithms

We use two scheduling algorithms as baseline references for comparison. The centralized strategy has a single GS and represents a performance target for our distributed scheduling approaches. The local algorithm, on the other hand, performs no job migration and represents a traditional non-grid scheduling environment.

1) Centralized: In the centralized algorithm, all jobs are submitted to a single GS which does not have an affinity to a specific compute server. The GS is responsible for making global decisions and assigning each job to a specific machine. It tracks the status of each job and maintains current information on all available resources, allowing it to compute *ATT* and *RU* without any communication. When a job is submitted, the GS selects the optimal server (based on *ATT* and *RU*) and migrates the job to that system. Although communication-free resource awareness is unrealistic, this strategy allows us to model the potential gain of a centralized architecture. However, the model is impractical as it constitutes a single point of failure and thus suffers from a lack of reliability and fault tolerance. Additionally, this approach has severe scalability problems that may result in a performance bottleneck for large-scale grid environments.

2) Local: In the local scheduling algorithm, there are no GSs. All jobs are submitted to LSs and executed on the compute server associated with each LS. This approach represents how scheduling is currently being performed and we use it to demonstrate the benefits of grid scheduling algorithms. The local scheduling policy for all strategies is first-come-first-serve with backfilling.

IV. Methodology

We evaluate our grid scheduling algorithms by simulating resources and jobs. Our environment models the submission of workloads to grid schedulers, the operation of grid and local schedulers, the transfer of job input/output data between compute servers, the network bandwidth contention, and the execution of jobs on compute servers. During these simulations, we gather performance information so that we can compare the various grid scheduling algorithms. Our specific objective in this paper is to understand the effects of data migration.

A. Resource Configurations

We simulate seven actual compute servers in our simulations. These systems are located at Lawrence Berkeley National Laboratory, NASA Ames Research Center, Lawrence Livermore National Laboratory, and San Diego Supercomputer Center. All seven machines are either cache-coherent SMP clusters or NUMA shared memory systems, interconnected by a fast proprietary network. Both architectures partition CPUs into nodes for management purposes, and the current practice is to allocate each node to a single application so that applications do not interfere with each other. We therefore used this allocation approach in our simulation environment.

We wanted to use 12 servers to give us more flexibility for grouping them into sets; so we duplicated five of the seven systems. The systems were then split into 3, 6, and 12 sets to simulate machines grouped at 3, 6, and 12 different sites. Each site has an equal number of machines with equivalent total computational power. The characteristics of these systems and the sites to which they are assigned are shown in Table I.

TABLE I
CONFIGURATIONS OF THE COMPUTE SERVERS AND THEIR ASSIGNMENT TO SITES

Server ID	# of Nodes	CPUs/ Node	Clock (MHz)	Site Locator		
				3 sites	6 sites	12 sites
S_1	184	16	375	0	0	0
S_2	305	4	332	1	1	1
S_3	144	8	375	2	3	2
S_4	256	4	600	1	0	3
S_5	32	2	250	2	2	4
S_6	128	4	400	2	5	5
S_7	64	2	250	2	5	6
S_8	144	8	375	1	2	7
S_9	256	4	600	0	4	8
S_{10}	32	2	250	0	1	9
S_{11}	128	4	400	0	3	10
S_{12}	64	2	250	1	4	11

We also simulate the networks connecting the servers. We assume that all systems at a single site share a local network and that each of these networks is connected to every other local network (at remote sites) using a point-to-point connection. When we model the transfer of a job's input/output data, we simulate the use of a local network on the sending side, a point-to-point inter-site network, and a local network on the receiving side. Any of these three networks can constrain the end-to-end data migration bandwidth. We assume that all data transfers share the network bandwidth equally, and network contention occurs when multiple data transfers simultaneously utilize a given network path. In our model, each site network has a peak bandwidth of 800 Mb/s, while 40 Mb/s is available from each point-to-point network. This represents a gigabit Ethernet LAN and a relatively high-performance WAN.

For the experiments reported here, we make two simplifying assumptions. First, we assume that program performance is linearly related to CPU speed. Second, even though the systems we are simulating are not all binary compatible, we assume that users have compiled

their codes for each of the heterogeneous platforms. We plan to relax both these assumptions in future work.

B. Job Workloads

We base our job workloads on trace data obtained from schedulers on the seven compute servers (see Table I), each recorded from March through May of 2002. Five other traces were gathered from a subset of the systems but recorded from September through November of 2002. Unfortunately, these 12 traces do not include the input or output data volume for each job, because this information is typically not available to local schedulers. We therefore added synthetic input/output data sizes to each job in the workloads.

We assume that the data volume is correlated to the amount of work (number of CPUs multiplied by run time) performed by each job. To somewhat randomize this, we set the input data size for job j using a Gaussian distribution with mean $\mu_j = B \times cpus_j \times runtime\_secs_j$ and standard deviation $\sigma_j = \frac{\mu_j}{3}$, where B is the number of Kbytes for each unit of work. Using anecdotal observations, our best estimate for B is 1 Kb for each CPU second the application executes. This set of workloads is denoted as B (for baseline), and their characteristics are shown in Table II. In all cases, we assume that the output data volume is five times the input data size.

TABLE II
WORKLOAD CHARACTERISTICS FOR SET B (W_i IS SUBMITTED TO SERVER S_i)

Workload ID	Time Period (Start–End)	# of Jobs	Avg. Input Size (MB)
W_1	03/01/02–05/31/02	59,623	312.7
W_2	03/01/02–05/31/02	22,941	300.8
W_3	03/01/02–05/31/02	16,295	305.0
W_4	03/01/02–05/31/02	8,291	237.3
W_5	03/01/02–05/31/02	10,543	28.9
W_6	03/01/02–05/31/02	7,591	236.1
W_7	03/01/02–05/31/02	7,251	86.5
W_8	09/01/02–11/30/02	27,063	293.0
W_9	09/01/02–11/30/02	12,666	328.3
W_{10}	09/01/02–11/30/02	5,236	29.3
W_{11}	09/01/02–11/30/02	11,804	226.5
W_{12}	09/01/02–11/30/02	6,911	53.7

For comparison, we also create other workload sets when B is 0, 0.1, 10, and 100 Kb (referred to as $0B$, $0.1B$, $10B$, and $100B$, respectively). Note that these sets can also be interpreted in terms of network bandwidth variability. For example, workload set B is equivalent to $0.1B$, but with a network that is an order of magnitude slower (80 Mb/s LAN and 4 Mb/s WAN). Set B is also equivalent to $10B$, but with a network that is an order of magnitude faster (8 GB/s LAN, 400 Mb/s WAN). A similar relationship also exists for B, $10B$, and $100B$. The set $0B$ is a special case where the jobs have no input/output files or the network has infinite bandwidth.

C. Performance Metrics

We use several key metrics to evaluate the effectiveness of our grid scheduling and job migration algorithms, and to capture the effects of data movement. These metrics are also used to compare performance with the local and centralized job scheduling schemes.

Since individual users and system administrators often have different (and possibly conflicting) demands, no single measure can comprehensively capture overall grid performance. From the users' perspective, key measures of grid performance include *Average Response Time* (*ART*) and *Average Wait Time* (*AWT*). These are computed as follows (N is the total number of jobs):

$$ART = \frac{1}{N} \sum_{j \in Jobs} (ET_j - QT_j)$$

$$AWT = \frac{1}{N} \sum_{j \in Jobs} (ST_j - QT_j)$$

where QT_j, ST_j, and ET_j are the times when job j is queued to the grid, when it starts execution, and when it ends. The response (or turnaround) time is probably the single most important measure for an individual submitting a job; however, the wait time is also critical to users even though it is usually beyond their control. In our results, we actually present normalized values of *AWT* and *ART* (*NAWT* and *NART*, respectively) with respect to the local scheduling approach.

A system administrator, on the other hand, may be more interested in maximizing the utilization of the available computational resources at the site. Thus, we present *Weighted Grid Utilization* (*UTIL*), which measures the overall ratio between consumed and available computational resources across a grid. It is computed as:

$$UTIL = \frac{\sum_j (ET_j - ST_j) \times CPU_j \times Clock_j}{(ET_{last} - QT_{first}) \times \sum_m CPU_m \times Clock_m}$$

where $(ET_{last} - QT_{first})$ is the duration of the entire simulation; CPU_j and $Clock_j$ are the number of processors used by job j and their clock speed; and CPU_m and $Clock_m$ are the number of processors in machine m and their clock speed.

The *Fraction of Jobs Migrated* (*FOJM*) allows us to determine if there is any relationship between the number of jobs transferred and the performance of the scheduling algorithms. This metric is defined as:

$$FOJM = \frac{Number\ of\ Jobs\ Transferred}{Total\ Number\ of\ Jobs}$$

However, *FOJM* can be misleading as it captures only the number of migrated tasks, but does not consider their data requirements. We therefore also measure the

Fraction of Data Volume Migrated (*FDVM*) to help determine if the amount of data transferred by a scheduling algorithm affects its performance. It is defined as:

$$FDVM = \frac{\sum_k (InputSize_k + OutputSize_k)}{\sum_j (InputSize_j + OutputSize_j)}$$

where the numerator is a summation only over migrated jobs but the denominator is a summation over all jobs.

The final performance metric is the *Data Migration Overhead* (*DMOH*), which is defined as:

$$DMOH = \frac{Total\ Data\ Migration\ Time}{\sum_j (ET_j - QT_j)}$$

In other words, *DMOH* is the fraction of job response time that is spent moving data, across all jobs. The metric basically captures the overhead of grid scheduling.

Note that scheduler performance, measured by any metric, is highly dependent on the workload. For example, we would not expect an underloaded grid to derive much benefit from a smart distributed scheduler in terms of efficiency, as there may not be much room for improvement. In our experiments, we use moderately heavy workloads that were derived from real trace data collected at leading supercomputer centers.

V. Results

This section presents and analyzes the simulation results of our job migration algorithms using the performance metrics described in Section IV-C. We performed an initial set of experiments and determined that the optimal value for the migration threshold τ is 1.4. This means that a job is not migrated unless its local approximate turnaround time (*ATT*) is 40% more than the *ATT* on a remote system. The other parameters are set as follows: threshold for waiting $\phi = 1$ min, *ATT* comparison tolerance $\epsilon = 0.01$, interval for checking workload $\sigma = 5$ mins, and utilization threshold $\delta = 0.7$.

A. Scheduling Policy

Figure 2 compares the performance of our three distributed job scheduling algorithms with the centralized and local schemes for workload set B using 12 sites (i.e. one server per site). Results show the S-I algorithm minimizes both the normalized average response and wait times (*NART* and *NAWT*, respectively). Furthermore, S-I has a *NART* that is only 1% greater than that for the centralized approach but 60% less than the local scheme. In terms of *NART*, the other two distributed algorithms (R-I, SY-I) are also comparable with centralized, and 50% better than local. Thus, our scheduling algorithms perform much better than the local approach and are almost as effective as the impractical centralized strategy.

While job turnaround time is the most important metric to optimize, we can make other interesting observations from Fig. 2. First, our grid scheduling approaches

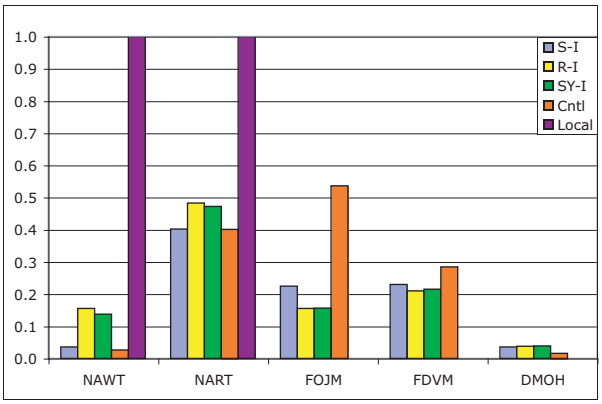

Fig. 2. Performance comparison of our job scheduling techniques for workload set B using 12 sites.

reduce *NAWT* by as much as 25x, which also improves *NART*. Second, an inverse relationship exists between the migration (*FOJM*, *FDVM*) and the timing (*NAWT*, *NART*) metrics. This indicates for the data volume associated with these workloads, there is sufficient network bandwidth so that more aggressive job migration is rewarded by lower response and wait times. This is also evident from the very small fraction of response time that is spent moving data (*DMOH*) for workload set B. We therefore examine the effects of varying the data volume associated with each job in the next set of experiments.

B. Data Volume

Figure 3 compares the effects of our different assumptions of input/output data size per job (or equivalently, network bandwidth characteristics), using 12 sites. Since the S-I algorithm was established to be the best distributed scheduling strategy, we only present its results for the remainder of the paper. Recall that workload set B is our best estimate of data size per job. Results show that sets $0B$, $0.1B$, and B have almost identical

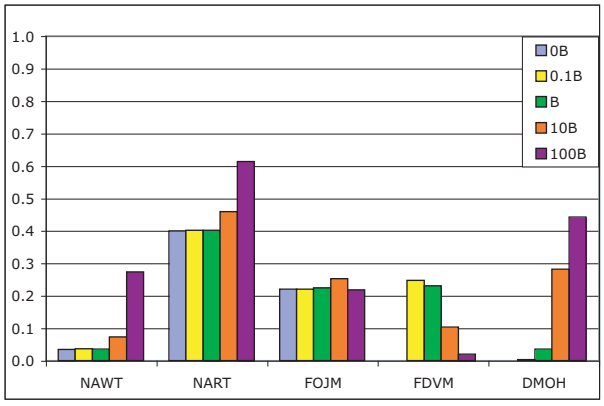

Fig. 3. Comparison of the effects of job data sizes (for different workload sets) using 12 sites and the S-I scheduling algorithm.

response and wait times, while sets $10B$ and $100B$ have increasingly larger values. For example, *NAWT* for $100B$ is almost 8x that for B, while the associated *NART* is 50% higher. This is explained by examining the fraction of response time spent moving data. Observe that *DMOH* for $0.1B$ and B is very low, but jumps to 28% and 44% for $10B$ and $100B$, respectively.

Another observation that can be made from Fig. 3 is that as the amount of data per job increases, the fraction of data volume migrated (*FDVM*) across the network decreases. This is expected due to the fact that while the total data volume increases, the available network bandwidth remains fixed. The *FOJM* metric is somewhat inconsistent (highest for $10B$ instead of $100B$); however, such anomalies occur because it refers to the number of jobs migrated without considering the data size associated with each job.

C. Number of Sites

Figure 4 shows the effects of varying the number of sites while having 12 compute servers spread across the sites. Observe that for the $0.1B$ workload set, the number of sites does not have a significant impact on any of the five metrics. This is due to the relatively small volume of data associated with each job. For $10B$, however, changing the number of sites has a noticeable effect. Decreasing the number of sites causes the available network bandwidth between systems to increase, thus the average wait and response times (*NAWT* and *NART*) drop slightly. Also, as expected, with fewer sites, the fraction of data volume migrated (*FDVM*) increases and the fraction of response time spent moving data (*DMOH*) decreases. Indeed, *DMOH* decreases by 40% when going from 12 sites to 3 sites.

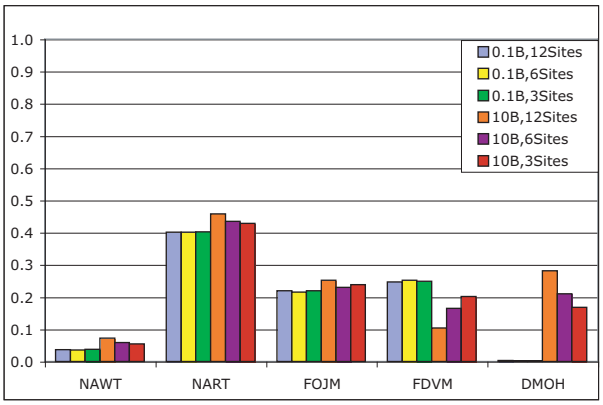

Fig. 4. Effects of varying the number of sites with a total of 12 compute servers and the S-I scheduling algorithm.

D. Communication-Oblivious Scheduling

We next examine the effects of migrating jobs without considering their input/output data that would need to be transferred over the network. The performance of this approach is presented in Fig. 5. Comparing these results with those presented in Fig. 4 reveal that for the $0.1B$ and B workload sets, ignoring the volume of data to be migrated does not significantly affect any of our metrics. However, for $10B$, there is a very large performance impact of not considering data movement when migrating jobs. The average response time (*NART*) for 12 sites is over 6x greater than just performing local scheduling, and almost 14x larger than our S-I algorithm that does consider the data movement overhead. The impact is even bigger (28x) for the normalized average wait time (*NAWT*), which increases from 0.07 to 2.1. The same general trend holds for 3 sites except that the effects are more pronounced: 40x and 43x for *NART* and *NAWT*, respectively, compared to communication-aware scheduling (see Fig. 4).

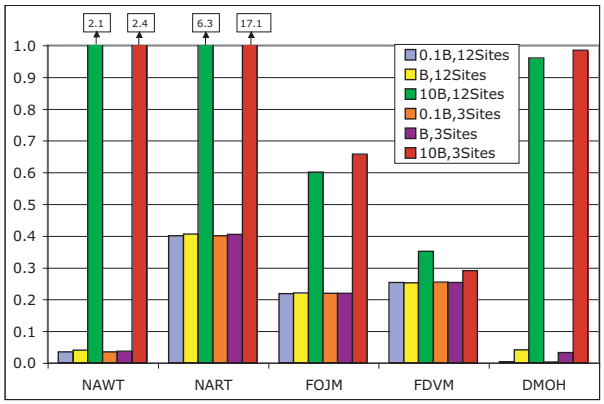

Fig. 5. Effects of not considering input/output data size while scheduling using the S-I algorithm.

Another interesting observation for workload $10B$ is that most of the job response time is spent transferring data. This is captured by the *DMOH* metric which is 96% and 98% for 12 and 3 sites, respectively (only 4% and 3% for B). As a side-effect (data not shown), the number of blocked jobs with allocated resources unable to execute because their data have not arrived increases from practically 0% using communication-aware scheduling, to 16% when job migration decisions ignore the impact of data-transfer overhead.

E. Increased Workload

One of the benefits of grid scheduling is that the distributed system can handle a larger number of jobs without degrading the performance seen by users. The data presented in Fig. 6 shows the effects of increasing the number of jobs in workload set B for the same time period. We constructed these workloads by duplicating certain jobs. For example, to obtain a workload that is 125% heavier than the baseline, we duplicated every fourth job in the trace files. Observe that almost 40%

more jobs can be handled by the S-I grid scheduler while maintaining the same *NART* and *NAWT* values as for the local scheduling approach. In other words, a user (on average) will not see an increase in his/her job's response and wait times while the system processes 40% more jobs. This increases the weighted grid utilization (*UTIL*) from 66% to 93%. However, like any scheduling system, there is a fine line: when the number of jobs is increased by 45%, *NART* and *NAWT* grow precipitously, exceeding the base case by factors of 3.5x and 2.4x.

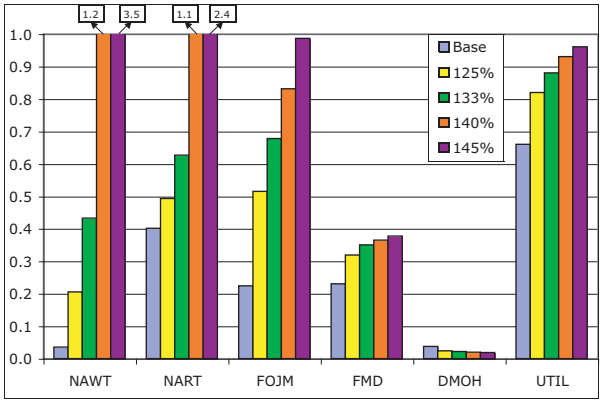

Fig. 6. Effects of increasing the number of jobs in workload set *B* when using 12 sites and the S-I scheduling algorithm.

VI. Conclusions and Future Work

One of the primary goals of grid computing is to provide shared access to geographically distributed heterogeneous resources in a transparent manner. Among the many open research issues is the problem of distributed resource management and job scheduling for computational grids. Our previous work addressed this challenge by defining and evaluating a grid scheduling architecture and several job migration algorithms. In this study, we focused on understanding the impact of data migration under a variety of demanding grid conditions.

We evaluated our grid scheduling algorithms by simulating compute servers, various groupings of servers into sites, and inter-server networks, using workloads derived from real trace data collected at leading supercomputing centers. Several key performance metrics were used to compare the behavior of our algorithms against reference local and centralized scheduling schemes.

Results showed the tremendous benefits of grid scheduling. Our best distributed strategy, sender-initiated, reduced average turnaround times by 60% compared with the local approach, even in the presence of input/output data migration. Alternatively, our algorithm can execute 40% more jobs in the grid environment and deliver the same turnaround times as in a non-grid scenario. Finally, for large data files (or slow networks), we found that it is critical to consider data transfer overhead when making job migration decisions as results show an increase of up to 43x in average turnaround times using a communication-oblivious scheduling algorithm.

There are several areas that we plan to explore in future. We wish to study the scalability of our grid scheduling algorithms, including problems such as resource discovery and fault tolerance. We plan to incorporate more realistic grid environments such as true server heterogeneity and multiple resource requirements. Finally, we would like to compare our peer-to-peer approach with other competing strategies such as hierarchical and combined local-grid schedulers.

Acknowledgments

The authors would like to thank LBNL, LLNL, NASA Ames, and SDSC for providing the batch job trace files. This work was supported by the Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098 and the NASA Computing, Information and Communications Technology Program.

References

[1] M. Arora, S.K. Das, and R. Biswas. A de-centralized scheduling and load balancing algorithm for heterogeneous grid environments. In *ICPP Workshop on Scheduling and Resource Management for Cluster Computing*, pages 499–505, 2002.

[2] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing '96*, 1996.

[3] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *9th Heterogeneous Computing Workshop*, pages 349–363, 2000.

[4] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. On advantages of grid computing for parallel job scheduling. In *2nd International Symposium on Cluster Computing and the Grid*, pages 39–46, 2002.

[5] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, CA, 1999.

[6] Global Grid Forum. http://www.gridforum.org.

[7] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. Evaluation of job-scheduling strategies for grid computing. In *1st International Workshop on Grid Computing*, volume LNCS 1971, pages 191–202, 2000.

[8] Load Sharing Facility. http://www.platform.com/products/LSFfamily.

[9] Portable Batch System. http://www.pbspro.com.

[10] K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *11th International Symposium for High Performance Distributed Computing*, 2002.

[11] E. Santos-Neto, W. Cirne, F. Brasileiro, and A. Lima. Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids. In *10th Workshop on Job Scheduling Strategies for Parallel Processing*, 2004.

[12] H. Shan, L. Oliker, and R. Biswas. Job superscheduler architecture and performance in computational grid environments. In *SC2003*, 2003.

[13] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan. Distributed job scheduling on computational grids using multiple simultaneous requests. In *11th International Symposium for High Performance Distributed Computing*, 2002.