Exploiting Replication and Data Reuse to Efficiently Schedule Data-intensive Applications on Grids

Elizeu Santos-Neto, Walfredo Cirne, Francisco Brasileiro, Aliandro Lima
Universidade Federal de Campina Grande
http://www.ourgrid.org
{elizeu,walfredo,fubica,aliandro}@dsc.ufcg.edu.br

Abstract

Data-intensive applications executing over a computational grid demand large data transfers. These are costly operations. Therefore, taking them into account is mandatory to achieve efficient scheduling of data-intensive applications on grids. Further, within a heterogeneous and ever changing environment such as a grid, better schedules are typically attained by heuristics that use dynamic information about the grid and the applications. However, this information is often difficult to be accurately obtained. On the other hand, although there are schedulers that attain good performance without requiring dynamic information, they were not designed to take data transfer into account. This paper presents Storage Affinity, a novel scheduling heuristic for bag-of-tasks data-intensive applications running on grid environments. Storage Affinity exploits a data reuse pattern, common on many data-intensive applications, that allows it to take data transfer delays into account and reduce the makespan of the application. Further, it uses a replication strategy that yields efficient schedules without relying upon dynamic information that is difficult to obtain. Our results show that Storage Affinity may attain better performance than the state-of-the-art knowledge-dependent schedulers. This is achieved at the expense of consuming more CPU cycles and network bandwidth.

1 Introduction

Each year more data are generated and need to be processed [1]. Currently, there are many scientific and enterprise applications that deal with a huge amount of data [2][3][4]. These applications are called *data-intensive*. In order to process large datasets, these applications typically need a high performance computing infrastructure. Fortunately, since the data splitting procedure is easy and each data element can often be processed independently, a solution based on data parallelism can often be employed.

Task independence is the main characteristic of parallel *Bag-of-Tasks* (BoT) applications [5][6]. A BoT application is composed of tasks that do not need to communicate to proceed with their computation. In this work, we are interested in the class of applications which has both BoT and *data-intensive* characteristics. We have named it *processors of huge data* (pHd). Shortly, pHd = *BoT* + *data-intensive*. There are innumerous important applications that fall in this category.

This is the case, for instance, of data mining, image processing, and genomics.

Due to the independence of their tasks, BoT applications are normally suitable to be executed on grids [7][5]. However, since resources in the grid are connected by wide area network links (wan), the bandwidth limitation is an issue that must be considered when running pHd applications on such environments. This is particularly relevant for those pHd applications that present a data reutilization pattern. For these applications, the data reuse pattern can be exploited to achieve better performance. Data reutilization can be either among tasks of a particular application or among a succession of applications executions. For instance, in the visualization process of quantum optics simulations results [4] it is common to perform a sequence of executions of the same parallel visualization application, simply sweeping some arguments (e.g. zoom, view angle) and preserving a huge portion of the data input from the previous executions.

There exists some algorithms that are able to take data transfer into account when scheduling pHd applications on grid environments [8][9][10][11]. However, they require knowledge that is not trivial to be accurately obtained in practice, especially on a widely dispersed environment such as a computational grid [7]. For ex-

ample, XSufferage [9] uses information about the CPU and network loads, as well as the execution time of each task on each machine, all of which must be known a priori, to perform the scheduling.

On the other hand, for *CPU-intensive* BOT applications, there are schedulers that do not use dynamic information, yet achieve good performance (e.g. Workqueue with Replication - WQR [12][13]). They use replication to tolerate inefficient scheduling decisions taken due to the lack of accurate information about both the environment and the application. However, these schedulers were conceived to target CPU-intensive applications and thus data transfers are not taken into account by them.

In this paper we introduce *Storage Affinity*, a new heuristic for scheduling PHD applications on grids. Storage Affinity takes into account the fact that input data is frequently reused either by multiple tasks of a PHD application or by successive executions of the application. It tracks the location of data to produce schedules that avoid, as much as possible, large data transfers. Further, it reduces the effect of inefficient task-processor assignments via the judicious use of task replication.

The rest of the paper is organized in the following way. In the next section we present the system model that is considered in this work. In Section 3, we present the Storage Affinity heuristic as well as other heuristics used for comparative purposes. In Section 4, we evaluate the performance of the discussed schedulers. Section 5 concludes the paper with our final remarks and a brief discussion on future perspectives.

2 System Model

This section formally describes the problem investigated and also provides the terminology used in the rest of the paper.

2.1 System Environment

We consider the scheduling of a sequence of jobs[1] over a grid infrastructure. The grid G is formed by a collection of sites. Each site is comprised of a number of processors, which are able to run tasks, and a single data server which is able to store input data required in the execution of a task, and output data generated by the execution of a task. More formally:

$$G = \{site_1, \ldots, site_g\}, g > 0, \text{and } site_i = P_i \cup \{S_i\},$$

where P_i is the non-empty set of processors at $site_i$ and S_i is the data server at $site_i$. We assume that the resources owned by the various sites are disjoint, *i.e.* $\forall i, j, i \neq j, site_i \cap site_j = \varnothing$.

[1] We use the terms job and application interchangeably.

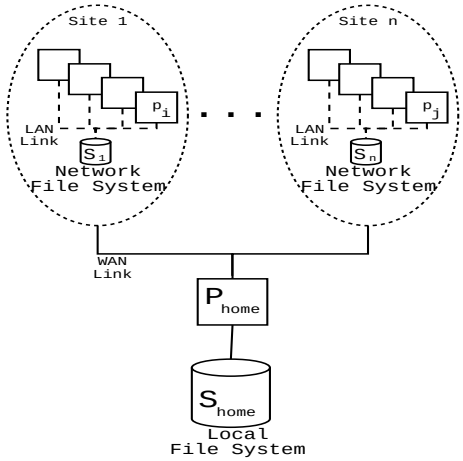

Figure 1. The system environment model

Processors belonging to the same site are connected to each other through a high bandwidth local area network, whose latency is small and throughput is large when compared to those experienced by the wide area networks that interconnect processors belonging to different sites. Because of this assumption we only consider one data server per site, *i.e.* the collection of data servers that may be present in a site is collapsed into a single data server.

We define two sets to encompass all processors (P_G) and all data servers (S_G) present in a grid G. That is to say:

$$P_G = \bigcup_{1 \leq i \leq |G|} P_i, \quad \text{and} \quad S_G = \bigcup_{1 \leq i \leq |G|} \{S_i\}.$$

We assume that the user spawns the execution of applications from a *home* machine that does not belong to the grid ($p_{home} \notin G$). Further, we assume that before the first execution of an application, all its input data are stored at the local file system of the home machine (S_{home}). The bandwidth is shared equally among the transfers initiated by the user in the *home* machine. Figure 1 illustrates the assumed environment.

2.2 Application

Job J_j, $j > 0$, is the j^{th} execution of the application J. A job is composed by a non-empty collection of tasks, in which each task is defined by two datasets: the input and the output datasets. Formally:

$$J_j = \{t_1^j, \ldots, t_n^j\}, n > 0, \text{and } t_t^j = (I, O), I \cup O \neq \varnothing,$$

where $t_t^j.I$ and $t_t^j.O$ are the input and the output datasets of task t_t^j, respectively.

For each data server $S_i, S_i \in S_G$, let $ds_j(S_i)$ be the set of data elements that are stored at S_i before the execution of the j^{th} job was started, and let $ds(S_{home})$ be the set of data elements that are stored at the home machine. We define D_j as the set of all data elements that are available to be taken as input by the j^{th} job to be executed. D_j is given by:

$$D_j = ds(S_{home}) \cup \left\{ \bigcup_{S_i \in S_G} ds_j(S_i) \right\}.$$

We have that $t_t^j.I \subseteq D_j$ and after the execution of the j^{th} job, the set of available data elements D_{j+1} is given by the union of D_j with all data elements that have been output by J_j:

$$D_{j+1} = D_j \cup \left\{ \bigcup_{1 \leq t \leq |J_j|} t_t^j.O \right\}.$$

We also define the input dataset of the entire application as the union of the input dataset of each task in the job. It is expressed by:

$$J_j.I = \bigcup_{k=1}^{|J_j|} t_k^j.I$$

2.3 Job Scheduling and Performance Metrics

A schedule Σ_j of the job J_j comprises the schedule of each one of the tasks that form J_j. The schedule of a particular task t_t^j of J_j specifies the processor that is assigned to execute t_t^j. Note that it is possible for the same processor to be assigned to more than one task. Formally,

$$\Sigma_j = \{p_1^j, \ldots, p_n^j\}, n = |J_j|, p_t^j \in P_G, 0 \leqslant t \leqslant n$$

We assume that a task can only access the data server at the same site of the processor on which the task is running. Consequently, if all data elements in the dataset $t_t^j.I$ are not already stored at S_i, the absent data elements must be first transferred to S_i before the execution of t_t^j can be started at p_t^j. Thus, after t_t^j is executed at $p_t^j \in site_i$, S_i will have stored all data elements in the dataset $t_t^j.O$.

We measure the application execution time to evaluate the efficiency of a scheduling. Thus, the heuristic we propose in this paper and also the others that we discuss, all have a common goal, which is to minimize this metric. The application execution time, normally referred as its *makespan* [14], is the time elapsed between the moment the first task is started until the earliest moment in which all tasks have finished their execution.

3 Scheduling Heuristics

Despite the fact that PHD applications are suitable to run on computational grids, the efficient scheduling of these applications on grid environments is not trivial. The difficulty in scheduling PHD application is twofold. The first problem relates to the very nature of PHD applications, which must deal with a huge amount of data. The issue here is that the application overall performance is greatly affected by the large data transfers that occur before the execution of tasks. The second problem is related to obtaining accurate information about the performance resources will deliver to the application. Despite the fact that this information is typically not available a priori, they are input for most available schedulers. In fact, there has been a great deal of research on predicting future CPU and network performance as well as application execution time [15][16][17][18][19]. As the results of these efforts show, this is by no means an easy task. To complicate matters further, the lack of central grid control poses an obstacle for deploying resource monitoring middleware.

We can observe that the difficulty in obtaining dynamic information and the impact of large data transfers have been individually attacked. Therefore, we comment two scheduling heuristics that deal with these problems separately, Workqueue with Replication (WQR) [12] and XSufferage [9]. We also introduce our approach to address the two PHD scheduling problems together.

3.1 Workqueue with Replication

The WQR scheduling heuristic [12] has been conceived to solve the problem of obtaining precise information about the future performance tasks will experience on grid resources. Initially, WQR is similar to the traditional Workqueue scheduling heuristic. Tasks are sent at random to idle processors and when a processor finishes a task, it receives a new task to execute. WQR differs from Workqueue when a processor becomes available and there is no waiting task to start. At this point, Workqueue would just wait for all tasks to finish. WQR, however, starts replicating the tasks yet running. The result from a task comes from the first replica to finish. After the first replica completes, all other replicas are killed.

The idea behind the task replication is to improve the application performance by increasing the chances

of running a task on a fast/unloaded processors. WQR achieves good performance for CPU-intensive application [12] without using any kind of dynamic information about processors, network links or tasks. The drawback is that some CPU cycles are wasted with the replicas that do not complete. Moreover, WQR does not take data transfers into account, what results in poor performance for PHD applications, as we shall see in Section 4.

3.2 XSufferage

XSufferage [9] is a knowledge-based scheduling heuristic that deals with the impact of large data transfers on PHD applications running on grid environments. XSufferage is an extension of the *Sufferage* scheduling heuristic [20]. *Sufferage* prioritizes the task that would "suffer" the most if not assigned to the processor that fastest runs it. How much a task would suffer is gauged by its *sufferage value*, which is defined as the difference between the best and the second best completion time for the task.

The main difference between XSufferage and Sufferage algorithms is the sufferage value determination method. In XSufferage, the sufferage value is calculated using the *site-level* task completion times. The *site-level* completion time of a given task is the minimum completion time achieved among all processors within the site. The *site-level sufferage* is the difference between the best and second best *site-level* completion times of the task. The other difference is that XSufferage considers input data transfers in the calculation of the completion time of the task, thus, differently from Sufferage, it requires information about network available bandwidth as input.

The algorithm input is a job J_j and a grid G. The algorithm traverses the set J_j until it finds the task t_t^j with the highest sufferage value. This task is assigned to the processor that has presented the earliest completion time. This action is repeated until all tasks in J_j are scheduled.

The rationale behind XSufferage is to consider the data location when performing the task-to-host assignments. The expected effect is the minimization of the impact of unnecessary data transfers on the application makespan. The evaluation of XSufferage shows that avoiding unnecessary data transfers indeed improves the application's performance [9]. However, XSufferage calculates sufferage values based on the knowledge about CPU loads, network bandwidth utilization and task execution times. In general, this information is not easy to obtain.

input : G, J_j

output : $\Sigma_j \cup \Sigma_r$

while $(J_j \neq \varnothing)$ **do**
- Get (t_t^j) which $SA(t_t^j)$ is the largest.
- Schedule (t_t^j) to a processor at $site_i$.
- $J_j \leftarrow J_j - t_t^j$
- **if** $(\forall p \in P_G, p$ *is* busy) **then**
  - $waitForATaskCompletionEvent()$
- **end**

end

$J_r \leftarrow$ **getAllRunningTasks**()

while $(J_r \neq \varnothing)$ **do**
- Remove from (J_r) which:
  - $SA(t_t^j, site_i) = 0$
  - **getReplicationDegree**$(t_t^r) > Degree_{min}$
- Get the (t_t^r) which $(SA(t_t^r, site_i))$ is the largest.
- Schedule replica $((t_t^r)^d)$ to a processor at $site_i$.
- **if** $(\forall\, p \in P_G \mid p$ is busy) **then**
  - **waitForATaskCompletionEvent**()
  - **killAllReplicasOfTheCompletedTask**()
- **end**
- $J_r \leftarrow$ **getAllTasksRunning**()

end

Algorithm 1: Storage Affinity scheduling heuristic

3.3 Storage Affinity

Storage Affinity was conceived to exploit data reutilization to improve the performance of the application. Data reutilization appears in two basic flavors: *inter-job* and *inter-task*. The former arises when a job uses the data already used by (or produced by) a job that executed previously, while the latter appears in applications whose tasks share the same input data. More formally, the *inter-job* data reutilization pattern occurs if the following relation holds:

$$(j < k) \wedge ((J_j.I \cup J_j.O) \cap J_k.I \neq \varnothing)$$

On the other hand, the *inter-task* data reutilization pattern occurs if this other relation holds:

$$\bigcap_{t=1}^{|J_j|} t_t^j.I \neq \varnothing$$

In order to take advantage of the data reutilization pattern and improve the performance of PHD applications, we introduce the *storage affinity* metric. This metric determines *how close* to a site a given task is. By *how close* we mean *how many bytes* of the task input dataset are already stored at a specific site. Thus, *storage affinity* of a task to a site *is the number of bytes within the task*

input dataset that are already stored in the site. Formally, the *storage affinity* value between t_t^j and $site_i$ is given by:

$$SA(t_t^j, site_i) = \sum_{d \in (t_t^j.I \cap ds_j(S_i))} |d|$$

in which, $|d|$ represents the number of bytes of the data element d.

We claim that information about data size and data location can be obtained a priori without difficulty and loss of accuracy, unlike, for example, CPU and network loads or the completion time of tasks. For instance, this information can be obtained if a data server is able to answer the requests about *which* data elements it stores and *how large* is each data element. Alternatively, an implementation of a Storage Affinity scheduler can easily store a history of previous data transfer operations containing the required information.

Naturally, since *Storage Affinity* does not use dynamic information about the grid and the application which is difficult to obtain, inefficient *task-to-processor* assignments may occur. In order to circumvent this problem, Storage Affinity applies *task replication*. Replicas have a chance to be submitted to faster processors than those processors assigned to the original task, thus increasing the chance of the task completion time be decreased.

Algorithm 1 presents *Storage Affinity*. Note that this heuristic is divided in two phases. In the first phase Storage Affinity assigns each task $t_t^j \in J_j$ to a processor $p \in G$. During this phase, the algorithm calculates the highest storage affinity value for each task. After this calculation, the task with the largest storage affinity value is chosen and scheduled. This continues until all tasks have been scheduled. The second phase consists of task replication. It starts when there are no more waiting tasks and there is, at least, one available processor. A replica could be created for any running task. Considering that the replication degree of a particular task is the number of replicas that have been created for the task, whenever a processor is available, the following criteria are considered to choose the task to be replicated: i) the task must have a positive storage affinity with the site that has an available processor; ii) the current replication degree of the task must be the smallest among all running tasks; and iii) the task must have the largest storage affinity value among all remaining candidates. When a task completes its execution, the scheduler kills all remaining replicas of the task. The algorithm finishes when all the running tasks complete. Until this occurs the algorithm proceeds with replications.

4 Performance Evaluation

In this section we analyze the performance of Storage Affinity, comparing it against WQR and XSufferage. We have decided to compare our approach to these heuristics because WQR represents the state-of-the-art solution to circumvent the dynamic information dependence, whereas XSufferage is the state-of-the-art for dealing with the impact of large data transfers. We have used simulations to evaluate the performance of the scheduling algorithms. These simulations were validated by performing a set of real-life experiments (see Section 4.5).

Since the performance attained by a scheduler is strongly influenced by the workload [21] [22][23], we have designed experiments that cover a wide variety of scenarios. The scenarios vary in the heterogeneity of both the grid and the application, as well as the application granularity (see Section 4.2). Our hope was not only to identify which scheduler performs better, but also to understand how different factors impact their performance.

4.1 Grid Environment

Each task has a computational cost, which expresses how long the task would take to execute in a dedicated reference processor. Processors may run at different speeds. By definition, the reference processor has $speed = 1$. So, a processor with $speed = 2$ runs a 100-second task in 50 seconds (when dedicated). Since the computational grid may comprise processors acquired at different points in time, grids tend to be very heterogeneous (i.e. their processors speed may vary widely). In order to investigate the impact of grid heterogeneity on scheduling, we consider four levels of grid heterogeneity, as shown in Table 1[2]. Thus, for heterogeneity $1x$, we always have $speed = 10$, and the grid is homogeneous. On the other hand, for heterogeneity $8x$, we have maximal heterogeneity, with the fastest machines being up to 8 times faster than the slowest ones. Note that, in all cases, the average speed of the machines forming the grid is 10.

The grid power is the sum of the speed of all processors that comprise the grid. For all experiments we fixed the grid power to $1,000$. Since the speed of processors are obtained from the Processor Speed Distributions, a grid is "constructed" by adding one processor at a time until the grid power reaches $1,000$. Therefore, the average number of processors in the grid is 100. Processors are distributed over the sites that form the grid in equal

[2]Note the $U(x, y)$ denotes the uniform distribution in the $[x, y]$ range.

Grid Heterogeneity	Processor Speed Distributions
$1x$	$U(10, 10)$
$2x$	$U(6.7, 13.4)$
$4x$	$U(4, 16)$
$8x$	$U(2.2, 17.6)$

Table 1. Grid heterogeneity levels and the distributions of the relative speed of processors

proportions. Similarly to Casanova et al [9], we assume that a grid has $U(2, 12)$ sites.

For simplicity, we assume that the data servers do not run out of disk space (i.e., we do not address data replacement policies in the present work). As previously indicated, we neglect data transfers within a site. Inter-site communication is modeled as a single shared $1Mbps$ link that connects the *home* machine to several sites. It important to highlight that, in the model, $1Mbps$ is the maximum bandwidth that an application can use in the wide area network. However, the connections are frequently shared among several applications, thus, the limit is often not achieved by a particular application. We used NWS [24] real traces to simulate contention for both CPU cycles and network bandwidth. For example, a processor of $speed = 1$ and $availability = 50\%$ runs a 100-second task in 200 seconds.

4.2 pHD Applications

In pHD applications, the application execution time is typically related to the size of the input data. The explanation for this fact is quite simple. The more data there is to process, the longer the tasks take to complete. In fact, there are pHD applications whose cost is completely determined by the size of the input data. This is the case, for example, of a scientific data visualization application, which processes the whole input data to produce the output image [4]. There are other applications that have the cost influenced, but not completely determined, by the size of the input data. This is the case of a *pattern search* application, in which the size of the input data of each task determines an upper bound for the cost of the task, not the cost of the task itself. We simulated both kinds of applications.

The total size of the input data of each simulated application was fixed in $2GBytes$. Based on experimental data available [4], we were able to convert the amount of input data processed by each task of the visualization application into the time (in seconds) required to process the data, which is its computational cost. We have used the same proportionality factor (1.602171 ms/KByte) to calculate the computational cost of the pattern search application, as a function of the amount of data actually processed by its tasks. To determine the computational cost of each task of the pattern search application, we used an uniform distribution $U(1, UpperBound)$, in which $UpperBound$ is the computational cost to process the entire input of a particular task.

We also wanted to analyze how the relation between the average number of tasks and the number of processors in the grid would impact the performance of a schedule. Note that when both application and grid sizes are fixed, this relation is inversely proportional to the average size of the input data of the tasks that comprise the application, i.e. the *application granularity*. We have considered three application groups that are defined by the following application granularity values: $3MBytes$, $15MBytes$ and $75MBytes$.

The tasks that comprise the application can vary in size. Therefore, to simulate this variation, we introduced an application heterogeneity factor. The heterogeneity factor determines how different are the sizes of the input data elements of the tasks that form the job, and consequently their costs. The size of the input data are taken from the uniform distribution $U(AverageSize \times (1 - \frac{H_a}{2}), AverageSize \times (1 + \frac{H_a}{2}))$, in which $AverageSize \in \{3MBytes, 15MBytes, 75MBytes\}$ and $H_a \in \{0\%, 25\%, 50\%, 75\%, 100\%\}$.

4.3 Simulation Setting and Environment

A total of $3,000$ simulations were performed, with half of them for each type of application (Visualization and Pattern Search). As we shall see in Section 4.4.1, $3,000$ simulations make for good precision of results (in 95% confidence interval). Each simulation consists of a sequence of 6 executions of the same job. Those 6 executions are repeated for each of 3 analyzed scheduling heuristics (Workqueue with Replication, XSufferage and Storage Affinity). Therefore, we have 18000 makespan values for each scheduling heuristic analyzed.

Our simulation tool has been developed using an adapted version of the Simgrid toolkit [25]. The Simgrid toolkit provides the basic functionalities for the simulation of distributed applications on grid environments. Since the set of simulations is itself a BoT application, we have executed it over a grid composed of 107 machines distributed among five different administrative domains (LSD/UFCG, Instituto Eldorado, LCAD/UFES, UniSantos and GridLab/UCSD). We have used the MyGrid middleware [5] to execute the simulations.

4.4 Simulation Results

In this section we show the results obtained in the simulations of the scheduling heuristics and discuss their statistical validity. We also analyze the influence of the application granularity, as well as the heterogeneity of both the grid and the application on the performance of the application scheduling.

4.4.1 Summary of the results

Table 2 presents a summary of the simulation results. It is possible to note that, in average, Storage Affinity and XSufferage achieve comparable performances. Nevertheless, the standard deviation values indicate that the makespan presents a smaller variation when the application is scheduled by Storage Affinity when compared to the other two heuristics. It is important to explain that the *resource wasting* percentage is given by: $\frac{TimeConsumedByKilledReplicas}{TimeConsumedByFinishedTasks}$. Obviously, we do not report any wasting values in Table 2 for XSufferage because this heuristic does not apply any replication strategy, consequently it does not kill any running task.

Makespan (seconds)	*Storage Affinity*	WQR	*XSufferage*
Mean ($\overline{x}$)	14377	42919	14665
Standard deviation (σ)	10653	24542	11451
CPU Wasting	*Storage Affinity*	WQR	*XSufferage*
Mean ($\overline{x}$)	59.243%	1.0175%	–
Standard deviation (σ)	52.715%	4.1195%	–
Bandwidth Wasting	*Storage Affinity*	WQR	*XSufferage*
Mean ($\overline{x}$)	3.1937%	130.88%	–
Standard deviation (σ)	8.5670%	135.82%	–

Table 2. Summary of simulation results

In order to evaluate the precision and confidence of the summarized means presented in Table 2, we have determined the 95% *confidence interval* [26] for the population mean (μ) based on those values in Table 2. That is, using the sample mean, standard deviation and the sample size (number of makespan values) we estimate the confidence intervals, as shown in Table 3.

Heuristic	95% *Confidence interval*
Storage Affinity	$14241 < \mu < 14513$
Workqueue with Replication	$42547 < \mu < 43291$
XSufferage	$14498 < \mu < 14832$

Table 3. 95% confidence intervals for the mean of the makespan for each heuristic.

Since the width of the confidence interval (w) is relatively small compared to the results (see Table 4), we feel that we have performed enough simulation to obtain a good precision in the results.

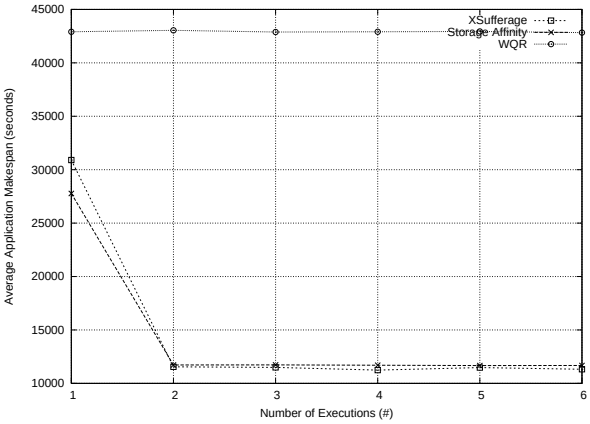

Figure 2. Summary of the performance of the scheduling heuristics

Heuristic	w	% with respect the makespan
Storage Affinity	330	2.2%
Workqueue with Replication	330	2.2%
XSufferage	850	2%

Table 4. Width of the confidence intervals and proportion with respect the mean

In Figure 2 we show the average application makespan and in Figure 3 we present the resource waste for all performed simulations with respect to all heuristics analyzed. The results show that both data-aware heuristics attain much better performance than WQR. This is because data transfer delays dominate the makespan of the application, thus not taking them into account severely hurts the performance of the application. In the case of WQR, the execution of each task is always preceded by a costly data transfer operation (as

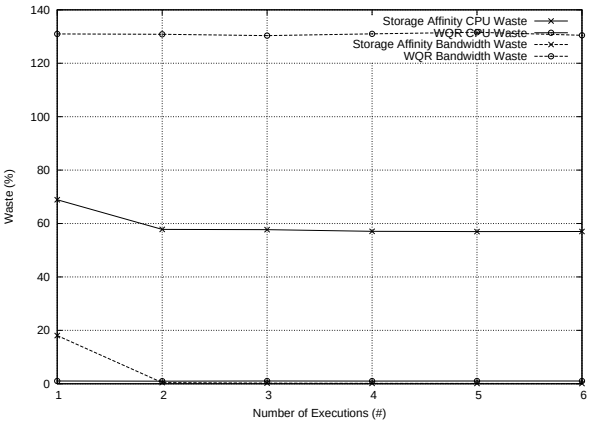

Figure 3. Summary of resource waste

can be inferred from the large bandwidth and small CPU waste shown in Figure 3). This impairs any improvement that the replication strategy of WQR could bring. On the other hand, the replication strategy of Storage Affinity is able to cope with the lack of dynamic information and yield a performance very similar to that of XSufferage. The main inconvenience of XSufferage is the need for knowledge about dynamic information, whereas the drawback of Storage Affinity is the consumption of extra resources due to its replication strategy (an average of 59% of extra CPU cycles and a negligible amount of extra bandwidth). From this result we can state that the Storage Affinity task replication strategy is a feasible technique to obviate the need for dynamic information when scheduling PHD applications, although at the expenses of consuming more CPU.

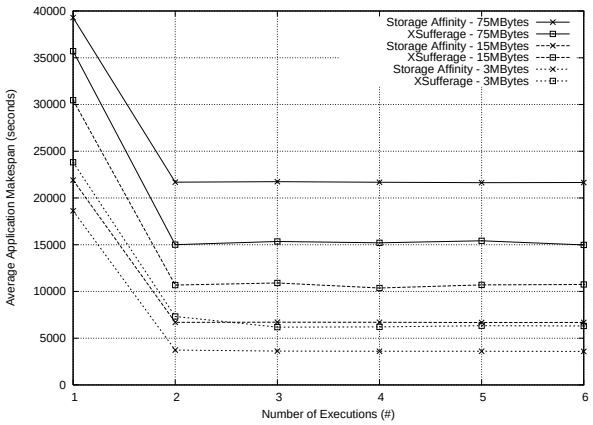

Figure 4. Impact of application granularity

4.4.2 Application granularity

Next, we investigate the impact of application granularity on the application scheduling performance. In Figure 4 we can see the influence of the three different granularities on the data-aware schedulers. From the results presented we conclude that no matter the heuristic used, smaller granularities yield better performance. This is because smaller tasks allow greater parallelism. We can further observe that XSufferage achieves better performance than Storage Affinity only when the granularity of the application is $75Mbytes$. This is because the larger a particular task is, the bigger its influence in the makespan of the application. Thus, the impact of a possible inefficient task-host assignment for a larger task is greater than that for a smaller one. In other words, the replication strategy of Storage Affinity is more efficient when circumventing the effects of inefficient task-host assignments when the application granularity is small. Nevertheless, for PHD applications, it is normally possible - and quite easy - to reduce the application granularity by converting a task with a large input into several tasks with smaller input datasets. The conversion is performed by simply slicing large input datasets into several smaller ones. It is important to note that there is a trade off here, because the scheduling overhead when running on real environments.

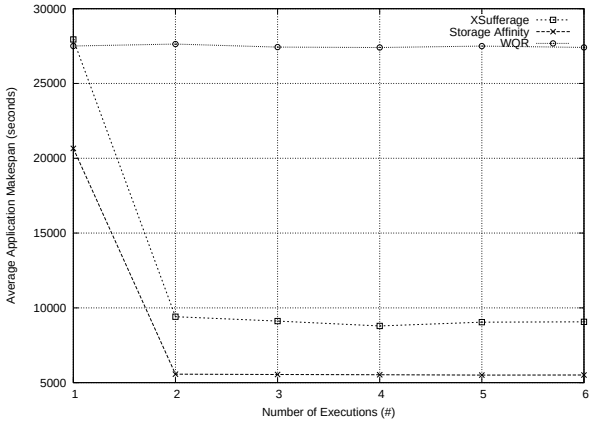

Figure 5. Performance of the heuristics with respect to the granularities $3Mbytes$ and $15MBytes$

Given the above discussion, we show in Figure 5 the values for the makespan of the applications, considering only the granularities $3Mbytes$ and $15Mbytes$. For these simulations, Storage Affinity outperforms XSufferage by 42%, in average. Further, as can be seen in Figure 6, the percentage of CPU cycles wasted is reduced from 59% to 31%, in average. We emphasize that reducing the application granularity is a good policy as smaller tasks yields more parallelism (see Figure 4).

4.4.3 Application type

In order to analyze the influence of the different characteristics of PHD applications on the application makespan and the resource waste, we have considered two types of applications (see Section 4.2). The results show that the behavior of the heuristics has not been affected by the different characteristics of the application. On the other hand, we found out that the waste of resources was affected by the type of application considered. Figure 9 and Figure 10 show the results attained. Recall that in the data visualization application, the computational cost of the task is completely determined by the size of its input dataset. Since Storage Affinity prioritizes the task with the largest *storage affinity* value, it means that the largest tasks are scheduled first. Therefore, task replication only starts when most

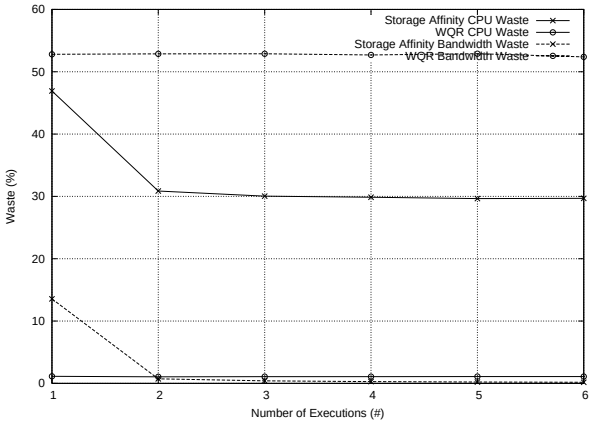

Figure 6. Resource waste with respect to the granularities $3Mbytes$ and $15MBytes$

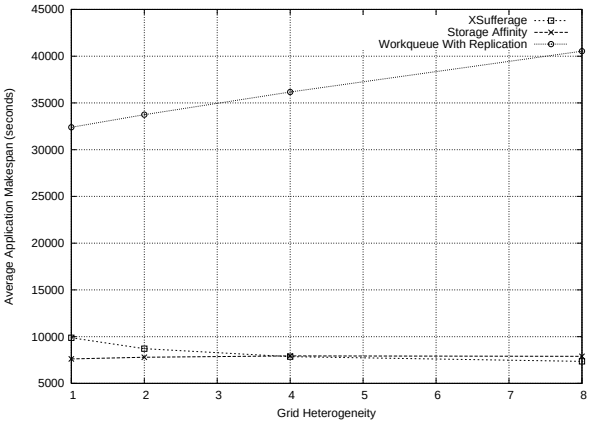

Figure 8. Grid Heterogeneity impact for Pattern Search application

of the application has already been executed. In the case of the pattern search application, the computational cost of the tasks is not completely determined by the size of the input dataset of the task, thus proportionally large tasks can be scheduled at later stages in the execution of the application. Therefore, replication may start when a large portion of the application is still to be accomplished, and consequently more resources are wasted to improve the application makespan.

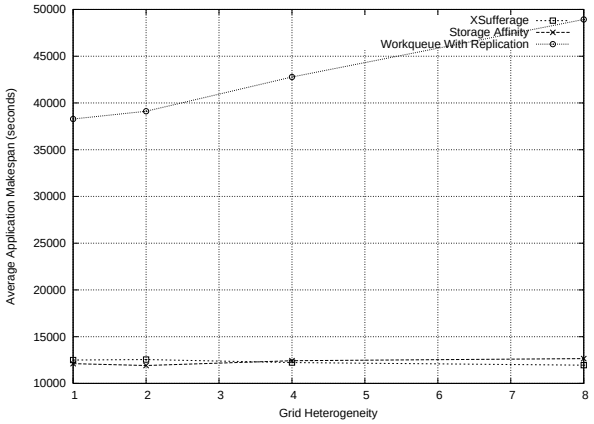

Figure 7. Grid Heterogeneity impact for Scientific Visualization application

4.4.4 Grid and Application heterogeneity

Finally, we have analyzed the impact of the heterogeneity of the grid and the application in both Scientific Visualization and Pattern Search applications.

In Figure 7 and Figure 8 we can see how the heterogeneity of the grid influences the makespan of both types of applications, considering the three heuristics discussed. It is possible to see that the two data-aware heuristics are not greatly affected by the variation of the grid heterogeneity. It is not surprising that XSufferage presents this behavior, given that it uses information about the environment. However, Storage Affinity shows that its replication strategy circumvents the effects of the variations of the speed of processors in the grid, even without using information about the environment. WQR is influenced a lot by the grid heterogeneity variation, we can see that increasing the grid heterogeneity the application makespan get worse.

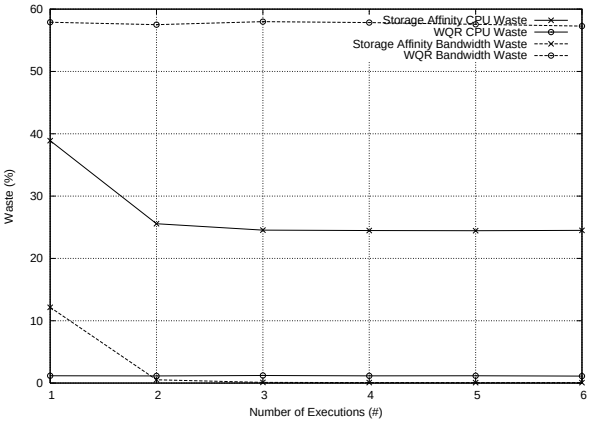

Figure 9. Resource waste considering the Scientific Visualization Application

Storage Affinity and XSufferage present a similar behavior with respect to application heterogeneity. Both heuristics show good tolerance to the variation of the application heterogeneity. In Figure 11 and Figure 12

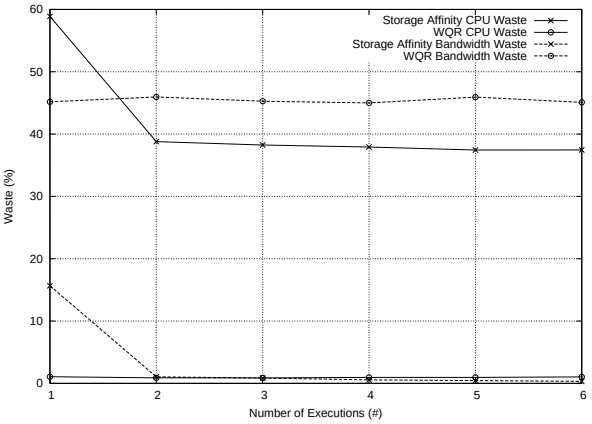

Figure 10. Resources wasted considering the Pattern Search Application

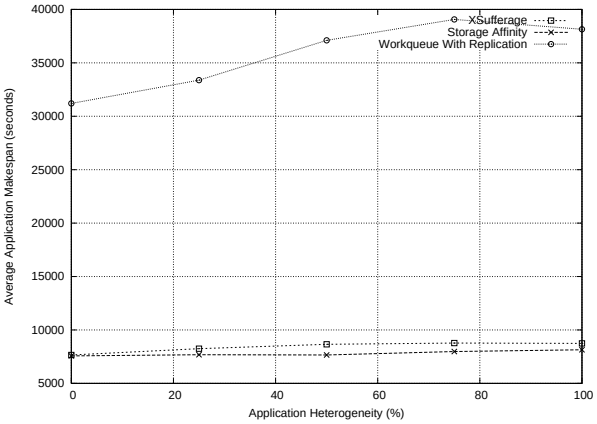

Figure 12. Application heterogeneity impact for Pattern Search application

we observe that the application makespan presents a tiny fluctuation for both types of application (Visualization and Search).

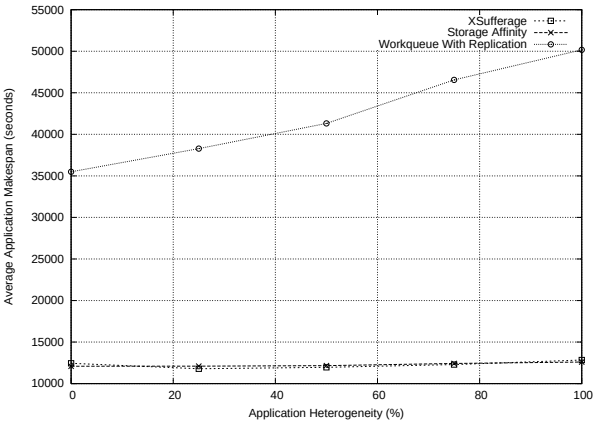

Figure 11. Application heterogeneity impact for Scientific Visualization application

4.5 Validation

In order to validate our simulations, we have conducted some experiments using a prototype version of *Storage Affinity*. The *Storage Affinity* prototype has been developed as a new scheduling heuristic for MyGrid [5, 27].

The grid environment used in the experiments was comprised by 18 processors located at 2 sites (Carcara Cluster/LNCC - Teresópolis, Brazil and GridLab/UCSD - San Diego, USA). The home machine (p_{home}) was located at the Laboratório de Sistemas Distribuídos/UFCG - Campina Grande, Brazil. It is important to highlight that during the experiments the resources were shared with other applications.

With respect to the application, we have used BLAST [2]. BLAST is an application that searches a given sequence of characters into a database. These characters represent a protein sequence and the database contains several identified sequences of proteins. The application receives two parameters: a database and a sequence of characters to be searched. The database size is of the order of many GBytes, but it can be sliced into many slices of few MBytes. On the other hand, the size of the sequence of the characters to be searched does not surpass 4KBytes.

The application was composed of 20 tasks. Each task of the application receives a slice of 3MBytes of a large database downloaded from the BLAST site [28] and a sequence of characters smaller than 4KBytes. Since the simulations have been focused on applications that present *inter-job* data reuse pattern (Section 3.3), we have set the application to present the same data reuse pattern. Most of the input of each task was reused (the database), while a minor part of the input (the search target of few KBytes) has changed between executions.

4.5.1 Methodology

Two scheduling heuristics have been analyzed in the experiments: *Storage Affinity* and *Workqueue with Replication*. We did not use *XSufferage* due to the very lack of deployed monitoring infrastructure that could provide resource load information. On the other hand, MyGrid already has a version of *Workqueue with Replication* heuristic available.

In order to minimize the effect of the grid dynamism

on the results, the experiment consisted of *back-to-back* executions of the two scheduling heuristics (i.e. we did intermixed the experiments of both scheduling heuristics). Following this approach, 11 experiments have been executed. Each experiment consisted of 4 successive executions of the same application for each scheduling heuristic, thus adding to a total of 88 application executions.

4.5.2 Results

In Figure 13 we present the average of the application makespan for each scheduling heuristic. Figure 14 contains the simulation of the scenario used in the experiment. The results show that both *Storage Affinity* and *Workqueue with Replication* present the same overall behavior noticed in the simulations. However, the experiment does differ from the simulation in some aspects. One is the greater fluctuation the makespan values in the experiment. This is due to the high level of heterogeneity of the grid environment, and to the fact that we were ran much fewer cases than we simulated.

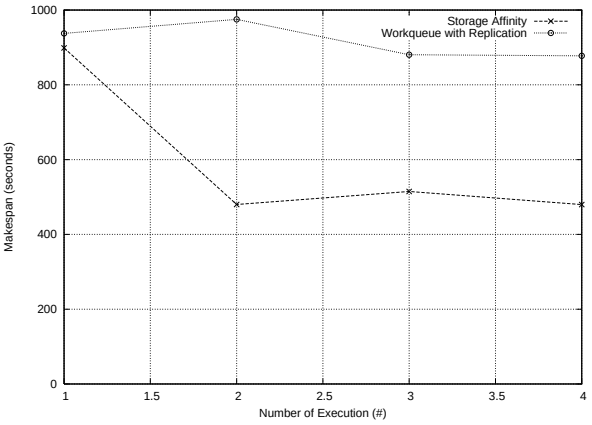

Figure 13. Real application execution using prototype version of schedulers

We can also observe a difference between the makespan in the experiment results and the simulated scenario. We believe there are two reasons for this discrepancy. First, we could not collect CPU and network loads experienced during the real life experiments. The standard NWS logs we used instead. Therefore, the grid scenario is not quite the same for the simulation and the experiments. Second, the *Storage Affinity* prototype always queries the sites to obtain information on the existence and size of files. This costly remote operation was not modeled in the simulator. However, since the scheduler itself is the responsible for transfering files to the sites, this information can be cached locally, thus greatly reducing the need for remote invocations during the execution of *Storage Affinity*. We are currently implementing such caching strategy and expect such a modification to greatly reduce the discrepancy between simulation and experimentation.

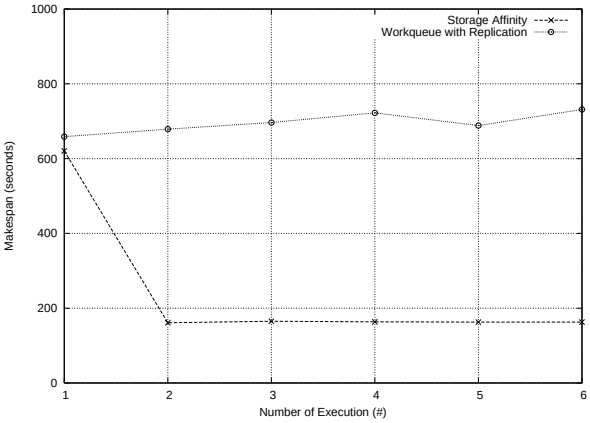

Figure 14. Simulation of the scenario used in the experiments

5 Conclusions and future work

In this paper we have presented Storage Affinity, a novel heuristic for scheduling PHD on grid environments. We have also compared its performance against that of two well-established heuristics, namely: XSufferage [9] and WQR [12]. The former is a knowledge-centric heuristic that takes data transfer delays into account, while the latter is a knowledge-free approach, that uses replication to cope with inefficient task-processor assignments, but does not consider data transfer delays. Storage Affinity also uses replication and avoids unnecessary data transfers by exploiting a data reutilization pattern that is commonly present in PHD applications. In contrast with the information needed by XSufferage, the data location information required by Storage Affinity is trivially obtained, even in grid environments.

Our results show that taking data transfer into account is mandatory to achieve efficient scheduling of PHD applications. Further, we have shown that grid and application heterogeneity have little impact in the performance of the studied schedulers. On the other hand, the granularity of the application has an important impact on the performance of the two data-aware schedulers analyzed. Storage Affinity is outperformed by XSufferage only when application granularity is large. However, the granularity of PHD applications can be easily reduced to levels that make Storage Affinity outper-

form XSufferage. In fact, independently of the heuristic used, the smaller the application granularity the better the performance of the scheduler (at least the granularity size which corresponds to an overhead starts to dominate the execution time). In the favorable scenarios, Storage Affinity achieves a makespan that is in average 42% smaller than XSufferage. The drawback of Storage Affinity is the waste of grid resources due to its replication strategy. Our results show that the wasted bandwidth is negligible and the wasted CPU can be reduced to 31%.

As future work, we intend to investigate the following issues: i) the impact of the inter-task data reutilization pattern on application scheduling; ii) disk space management on data servers; iii) the emergent behavior of a community of Storage Affinity schedulers competing for shared resources; and iv) the use of introspection techniques for data staging [29] to provide the scheduler with information about data location and disk space utilization. Finally, we are about to release a stable version of Storage Affinity within the MyGrid middleware [5, 27]. We hope that practical experience with the scheduler will help us to identify aspects of our model that need to be refined.

References

[1] P. Lyman, H. R. Varian, J. Dunn, A. Strygin, and K. Swearingen, "How much information?." http://www.sims.berkeley.edu/research/projects/how-much-info-2003, October 2003.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 1, no. 215, pp. 403–410, 1990.

[3] G. Group, "http://www.griphyn.org," 2002.

[4] E. L. Santos-Neto, L. E. F. Tenório, E. J. S. Fonseca, S. B. Cavalcanti, and J. M. Hickmann, "Parallel visualization of the optical pulse through a doped optical fiber," in *Proceedings of Annual Meeting of the Division of Computational Physics (abstract)*, June 2001.

[5] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauvé, F. A. B. da Silva, C. O. Barros, and C. Silveira, "Running bag-of-tasks applications on computational grids: The mygrid approach," in *Proceedings of the ICCP'2003 - International Conference on Parallel Processing*, October 2003.

[6] J. Smith and S. K. Shrivastava, "A system for fault-tolerant execution of data and compute intensive programs over a network of workstations," in *Lecture Notes in Computer Science*, vol. 1123, IEEE Press, 1996.

[7] I. Foster and C. Kesselman, eds., *The Grid: Blueprint for a Future Computing Infrastructure*. 1999.

[8] O. Beaumont, L. Carter, J. Ferrante, and Y. Robert, "Bandwidth-centric allocation of independent task on heterogeneous plataforms," in *Proceedings of the Internetional Parallel and Distributed Processing Symposium*, (Fort Lauderdale, Florida), April 2002.

[9] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, "Heuristics for scheduling parameter sweep applications in grid environments," in *Proceedings of the 9th Heterogeneous Computing Workshop*, (Cancun, Mexico), pp. 349–363, IEEE Computer Society Press, May 2000.

[10] M. Faerman, R. W. A. Su, and F. Berman, "Adaptive performance prediction for distributed data-intensive applications," in *Proceedings of the ACM/IEEE SC99 Conference on High Performance Networking and Computing*, (Portland, OH, USA), ACM Press, 1999.

[11] K. Marzullo, M. Ogg, A. R. amd A. Amoroso, A. Calkins, and E. Rothfus, "Nile: Wide-area computing for high energy physics," in *Proceedings 7th ACM European Operating Systems Principles Conference. System Support for Worldwide Applications*, (Connemara, Ireland), pp. 54–59, ACM Press, Sept. 1996.

[12] D. Paranhos, W. Cirne, and F. Brasileiro, "Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids," in *Proceedings of the Euro-Par 2003: International Conference on Parallel and Distributed Computing*, (Klagenfurt,Austria), August 2003.

[13] Z. M. Kedem, K. V. Palem, and P. G. Spirakis, "Efficient robust parallel computations (extended abstract)," in *ACM Symposium on Theory of Computing*, pp. 138–148, 1990.

[14] M. Pinedo, *Scheduling: Theory, Algorithms and Systems*. New Jersey, USA: Prentice Hall, 2nd edition, August 2001.

[15] A. Downey, "Predicting queue times on space-sharing parallel computers," in *Proceedings of 11th International Parallel Processing Symposium (IPPS'97)*, April 1997.

[16] R. Gibbons, "A historical application profiler for use by parallel schedulers," *Lecture Notes in Computer Science*, vol. 1291, pp. 58–77, 1997.

[17] W. Smith, I. Foster, and V. Taylor, "Predicting application run times using historical information," *Lecture Notes in Computer Science*, vol. 1459, pp. 122–142, 1998.

[18] R. Wolski, N. Spring, and J. Hayes, "Predicting the CPU availability of time-shared unix systems on the computational grid," in *Proceedings of 8th International Symposium on High Performance Distributed Computing (HPDC'99)*, August 1999.

[19] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. F. Gryniewicz, and Y. Jim, "An architecture for a global internet host distance estimation service," in *Proceedings of IEEE INFOCOM*, 1999.

[20] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *Journal of the ACM (JACM)*, vol. 24, no. 2, pp. 280–289, 1977.

[21] D. Feitelson and L. Rudolph, "Metrics and benchmarking for parallel job scheduling," in *Job Scheduling Strategies for Parallel Processing* (D. Feitelson and L. Rudolph, eds.), vol. 1459, pp. 1–24, Lecture Notes in Computer Science, Springer-Verlag, 1998.

[22] D. G. Feitelson, "Metric and workload effects on computer systems evaluation," *Computer*, vol. 36(9), pp. 18–25, September 2003.

[23] V. Lo, J. Mache, and K. Windisch, "A comparative study of real workload traces and synthetic workload models for parallel job scheduling," in *Job Scheduling Strategies for Parallel Processing* (D. Feitelson and L. Rudolph, eds.), vol. 1459, pp. 25–46, Lecture Notes in Computer Science, Springer Verlag, 1998.

[24] R. Wolski, N. T. Spring, and J. Hayes, "The network weather service: a distributed resource performance forecasting service for metacomputing," *Future Generation Computer Systems*, vol. 15, no. 5-6, pp. 757–768, 1999.

[25] H. Casanova, "Simgrid: A toolkit for the simulation of application scheduling," in *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, (Brisbane Australia), May 2001.

[26] J. L. Devore, *Probability and Statistics for Engineering and The Sciences*, vol. 1. John Wiley and Sons, Inc., 2000.

[27] "Mygrid site." http://www.ourgrid.org/mygrid.

[28] "Blast webpage." http://www.ncbi.nlm.nih.giv/BLAST.

[29] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, IEEE Computer Society Press, Nov. 2000.