

New Worker-Centric Scheduling Strategies for Data-Intensive Grid Applications^{*}

Steven Y. Ko, Ramsés Morales, and Indranil Gupta

Department of Computer Science
University of Illinois, Urbana-Champaign
Urbana, IL 61801
{sko, rvmorale, indy}@cs.uiuc.edu

Abstract. Distributed computations, dealing with large amounts of data, are scheduled in Grid clusters today using either a *task-centric* mechanism, or a *worker-centric* mechanism. Because of the large data sets, the execution time is bounded by the cost of data transfer. In this paper, we introduce new worker-centric scheduling strategies that are novel in that they aim to implicitly exploit the locality of interest in order to reduce the cost of data transfer. Many Grid applications are characterized by such a locality of interest, *i.e.*, a file is often accessed by multiple tasks and, more importantly, a set of files that are accessed by one task are also likely to be accessed together by other tasks. Our new deterministic, as well as probabilistic, scheduling algorithms implicitly exploit this feature to improve running time. Our experiments are done with traces of a real Grid application (*Coadd*), and show that our algorithms are able to achieve utilization of over 90%, while reducing makespan significantly compared to task-centric approaches.

Keywords worker-centric scheduling, task-centric scheduling, data-intensive applications, Grid environments

1 Introduction

Data-intensive Grid applications are the applications that run on distributed Grid sites and are characterized by their access of large amounts of data sets. In attempting to minimize the execution time for such applications, schedulers of the Grid application are hampered by the sheer size of the data sets involved. While these data sets are mostly read-only and predefined, their size ranges from several terabytes to petabytes [1]. Examples of such data-intensive Grid applications can be found in many scientific domains such as Physics, Earth science, and Astronomy, *e.g.*, [2, 3].

At run time, this large scale of the data sets makes it impractical to replicate all the data at every execution site, where the term “site” refers to a cluster of client machines (“workers”). Instead, the typical approach to structuring such a data-intensive Grid application (*i.e.*, the “job”) is to partition the execution code into several small “tasks”, and to divide up the data into several disjoint pieces, each of which we call a “file”. Thus, each task requires a specific subset of

^{*} This work was supported in part by NSF CAREER grant CNS-0448246 and in part by NSF ITR grant CMS-0427089.

the files that constitute the job data, and a site begins the execution of a given task by retrieving all those required files.

When running a data-intensive Grid application across a collection of several sites, one of the most challenging problems is the design of a (global) Grid scheduling algorithm. Specifically, since the cost of data transfer is a major bottleneck for the execution time [2, 4–6], the main goal of the (global) scheduling algorithm becomes assigning tasks to sites in such a way as to reduce the frequency and amount of data transfer [4–6]. Fortunately, many data-intensive Grid applications exhibit *locality of interest*, *i.e.*, a file is often accessed by multiple tasks and also, a set of files that are accessed by one task are also likely to be accessed together by other tasks [7] (note: we will also use *data-sharing* whenever appropriate).

Our analysis of *Coadd* (Sloan Digital Sky Survey southern-hemisphere coaddition [2, 3]) (explained in detail in Section 2.1) also shows the locality of interest in data-intensive Grid applications. There is a significant number of files accessed by multiple tasks (Figure 1(a)) and there is a large number of tasks that access the same set of files during their execution (Figure 1(b)). This locality of interest gives an opportunity to reduce the numbers of both redundant file transfers and file replicas, and is present in wide variety of applications including data mining, image processing, genomics [4], and spatial processing applications which consist of tasks that process overlapping regions [2].

Previously, locality of interest has been exploited for scheduling and workflow planning in Grid data-intensive applications. Casanova *et al.* [6], Ranganathan *et al.* [5] and Santos-Neto *et al.* [4] successfully demonstrated the benefits of their locality-aware schedulers over traditional schedulers. However, the scheduler design in all the mentioned papers is *task-centric*, *i.e.*, the global scheduler assigns a task to a worker, without considering whether or not the worker can start executing the task immediately after the task assignment.

We observe that such task-centric scheduling suffers from two major issues when dealing with data-intensive applications. First, there is a possibility of unbalanced task assignments, resulting in some sites being overloaded with tasks. Second, conditions at a site during scheduling time of a task may be different from the conditions at the site during execution of the task, because each task usually waits in the site’s (or worker’s) task queue for a while.

We argue that an alternative *worker-centric* scheduling [8, 9], where a scheduling decision to a worker is made only when the worker can start executing the task immediately, is amenable to approaches that exploit locality in file accesses, and addresses both of these issues. In worker-centric scheduling, the times of task assignment to a worker are determined solely by the worker’s preference based on its local criteria, *e.g.*, by using policies based on local CPU load, site queue length, time of the day, etc. The task execution begins as soon as the task arrives at the worker. The scheduling problem then becomes the one of designing a global scheduler that assigns the best possible as-yet-unscheduled task to the “best” worker, based on such characteristics as the files already present at the worker’s site, and the data required by the unscheduled tasks.

There are two options for implementing worker-centric scheduling strategies - either (1) workers could *pull* tasks from a task repository associated with the global scheduler, when the worker’s local policies allow it to do so; or (2) the global scheduler could *push* tasks out to workers, depending on the worker’s preference. We consider only the pull variant ((1) above) since it is simpler and more practical. Henceforth in this paper, whenever we use the term “worker-centric”, we will be referring to *only* the pull variant of the worker-centric algorithm.

In this paper, we present the first (to the best of our knowledge) worker-centric scheduling strategies that implicitly exploit the locality of interest in data-intensive Grid applications. We then demonstrate the advantages of worker-centric scheduling over task-centric scheduling for data-intensive Grid applications through experiments. In our worker-centric strategies, each worker requests a task from the global scheduler when convenient to the worker. Upon receiving this request, the global scheduler iterates over the list of as-yet-unscheduled tasks and finds the best task to assign to the worker. The “best” task could be selected according to a variety of metrics, which we discuss later in detail.

We propose three different metrics that consider the different aspects of locality of interest in data-intensive Grid applications, and aim to: (1) maximize the chance of reusing the data, and (2) to minimize the number of file transfers. Our simulation results with *Coadd* confirm that worker-centric scheduling gives better performance than task-centric scheduling in many scenarios. We select *Coadd* for all our experiments in this paper because (1) it is difficult to obtain Grid application traces, and (2) *Coadd* is a real Grid application used by several research organizations [2, 3] and it shows many typical characteristics of data-intensive Grid applications. Thus, we believe that our results will hold for many other data-centric Grid applications.

It is important to note that our Grid model is general, and *not* intended to specifically target production Grids such as Grid2003 [10]. Rather, we use the term “Grid” as a generic model, where a set of cooperating sites (a cluster of workers) can be used to execute a job (which consists of tasks sharing read-only data). Also, our scheduling strategies focus only on scheduling data-sharing tasks within a single large job (application), instead of multiple disconnected jobs injected into the system by different users. However, for realistic evaluation, we do simulate the presence of background jobs running concurrently with our main Grid job in our experiments in Section 4.

The rest of the paper is organized as follows. In Section 2, we present background information including the detailed problems of task-centric scheduling and advantages of worker-centric scheduling. Section 3 presents our basic algorithm and various metrics that we consider. Section 4 presents our simulation results and Section 5 discusses related work. Section 6 concludes our paper.

2 Background and Basics

In this section, we motivate the scheduling problem by presenting the characteristics of data-intensive applications. We then elaborate on the two types of schedulers mentioned: task-centric and worker-centric. Lastly, we discuss scheduling issues for data-intensive applications.

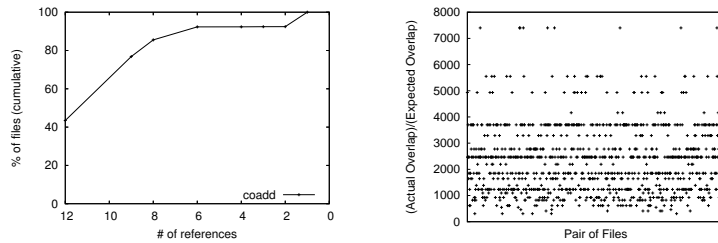


Fig. 1. (a) *Coadd* file access distribution. Note that the x-axis is in decreasing order, so each point in the CDF represents the minimum number of files accessed. (b) Locality of interest in *Coadd*.

2.1 Characteristics of Data-Intensive Applications

We discuss characteristics of data-intensive applications here to motivate the problem. As a real example, we use one particular application, *Coadd* (Sloan Digital Sky Survey southern-hemisphere coaddition [2, 3]) in our discussion.

In general, tasks in a data-intensive application access a large set of files, thus data transfer time significantly affects the entire execution time (*i.e.* data-intensive applications are network-bound [4, 11]). In addition, the tasks have a high degree of data-sharing among them, which gives an opportunity to reuse data in local storage [2, 4–6, 11].

For example, *Coadd* is a spatial processing application that has 44,000 tasks accessing 588,900 files in total. It is reported by Meyer *et al.* [2] that when it was run on Grid3 [10] with over 30 sites and 4,500 CPUs, it took roughly 70 days to complete. One of the reasons for the observed long completion time was the large number of files necessary for each task. Meyer *et al.* [2] state that these characteristics would also be expected in other spatial processing applications.

Our analysis of *Coadd* indeed confirms the characteristics of data-intensive applications. In *Coadd*, each task accesses a different number of files ranging from 36 to 181, and approximately 124 files on average. Moreover, roughly 90% of files are accessed by 6 or more tasks, as shown in Figure 1(a). If we assume that each file is fixed at 5MB as in [2], then the total size of all the files is roughly 2.8TB, and each of 44,000 tasks potentially requires 620MB of data transfer on average and up to 905MB in the worse case for each execution. Considering the number of tasks and size of data transfers, it is desirable to reduce the redundant file transfers.

To show locality in *Coadd*, we first pick 1,000 sample pairs of files (say, A and B) accessed by *Coadd* tasks. We then plot the ratio between the actual number of tasks accessing both files, and the expected number of tasks accessing the same files. Figure 1(b) shows the result. The former (the actual number, say, C) is directly counted from our *Coadd* workload, and the later (the expected number) is derived from $\frac{a}{T} \times \frac{b}{T} \times T$, where T is the total number of tasks, and a, b are the numbers of tasks accessing A and B , accordingly. The Y-axis shows $C / (\frac{a}{T} \times \frac{b}{T} \times T)$. As we can see, the values are much larger than 1, which means that the number of tasks that access the same pair of files is much larger than statistically expected.

2.2 System Model

Before comparing task-centric to worker-centric solutions, we present our system model. We assume that:

- 1) A *job* is defined as an application composed of multiple parallel *tasks*. Each task does not need to communicate with other tasks in order to proceed (*i.e.*, a job is a Bag-of-Tasks [4]). However, tasks do share read-only files (data). These files are provided a priori along with the job specification.
- 2) There are multiple sites. Each site has at least one computation server or *worker* (and possibly multiple workers), and one data server to store data locally. We further assume that there is only one *data server* (or *local storage*) per site. If there are multiple data servers at a site, we consider all these data servers as combined storage. Storage size at a site is limited.
- 3) The data server of a site receives all file requests from the workers in the same site, and sends batch file requests for the missing files to the external file server. The data server processes requests one by one. This is more efficient than simultaneous requests, given the bandwidth limits.
- 4) Each task issues exactly one batch file request.
- 5) A worker starts executing a task by transferring all the files necessary for the task to the local data storage. After the transfer is over, the worker begins the actual computation of the task.
- 6) There is one external (global) scheduler that contains information about all tasks and gives tasks out on-demand to workers. Also, there is an external file server that has all the files necessary for all tasks, and hands them out to data servers on-demand.
- 7) Intra-site communication costs are negligible compared to inter-site communication costs.
- 8) In order to simplify our exposition, we will henceforth assume that all files are equally-sized. However, all our algorithms can be easily extended to variable sized files, by modifying the considered metrics to reflect the data size rather than the number of files.

We use the following two terms throughout the paper:

- 1) *Makespan* [12] is the total execution time of the job in consideration. This is the main metric for performance measurement.
- 2) *Utilization* of worker *A* is defined as, $(\text{total computation time of } A) / (\text{total execution time of } A)$.
- 3) A task and a local storage (*i.e.* the data server at a site) are said to *overlap* with each other, when at least one file necessary for the task is already present in the local storage. We use the term, *overlap cardinality*, to indicate the number of overlapping files.

The main goals for a scheduling algorithm are then to: (1) reduce the makespan, (2) reduce the number of files transferred to sites, and (3) increase the utilization at workers.

2.3 Task-Centric and Worker-Centric Schedulers

We elaborate two types of schedulers, namely, task-centric schedulers and worker-centric schedulers. Figure 2 shows an illustration of worker-centric and task-centric scheduling. In essence, this categorization is based on whether or not a

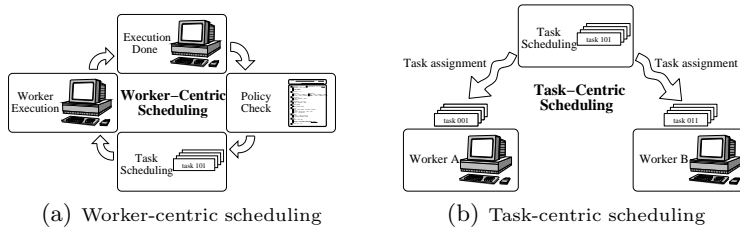


Fig. 2. An illustration of worker-centric and task-centric scheduling

scheduling strategy considers immediate task execution of a worker after a task assignment.

Concretely, a scheduler is *worker-centric*, if the task assignment to a worker is done when the worker can start executing the task immediately. As mentioned before, we consider only the pull-based variant of worker-centric scheduling and the term “worker-centric” refers to this pull-based variant of worker-centric scheduler throughout the paper. This variant has each worker *pull* a task from a task repository associated with the global scheduler, when its *local policies* allow it. These local policies may be a function of CPU load, free RAM space, time of day, etc. For instance, a site could have a policy that Grid jobs are executed only over night or at a specific time of the day. Another policy might state that a site could execute Grid jobs only when the average CPU load has been below a specified threshold for a while. This architecture is similar to a server-client architecture - a worker requests a task to the scheduler, and the scheduler finds the “best” task for the worker according to a set of metrics and local policies of the worker. One example of this type of worker-centric strategies is the traditional *workqueue* algorithm, which dispatches a task in FIFO order to an idle worker [13].

On the contrary, a scheduler is *task-centric*, if a task assignment is done without considering whether or not the worker can execute the task immediately. For a given set of tasks and a set of workers, the global scheduler chooses the best match (based on its certain metrics other than immediate task execution) between workers and tasks, and assigns each task to the best worker. Each worker has a task queue and executes the tasks in the queue one by one; an empty queue means the corresponding worker is not executing tasks for that job. Typical metrics used by schedulers are CPU load, network bandwidth, data overlap, etc. For example, scheduling strategies in [5] and storage affinity-based schemes [11] are task-centric.

Since our focus in this paper is to show the effectiveness of worker-centric scheduling in exploiting locality compared to task-centric scheduling, we do not discuss various policies of worker-centric scheduling further. In Section 4, we first evaluate our task-centric and worker-centric strategies using a simple policy called *always available* - a worker requests a task from job X immediately after it finishes the previous task from the same job X. Later, to consider the effect of slowdown due to background CPU load, we experimentally study the effect of local jobs at individual workers (which might be submitted by local users or

through other schedulers) - these background jobs run concurrently with tasks of the Grid job under consideration.

2.4 Scheduling Issues for Data-Intensive Applications

Several previous studies have identified that reusing data in local storage gives a dramatic performance improvement for data-intensive applications [2, 4–6]. Among others, studies by Ranganathan *et al.* [5] and Santos-Neto *et al.* [4] propose various task-centric scheduling strategies for data-intensive applications. Their studies suggest that making scheduling decisions based on data reuse indeed improve performance over other scheduling strategies that consider various different metrics altogether. Broadly, both types of strategies calculate and use the overlap cardinality (either the number of files or bytes) between all possible task-site pairs, in order to make the scheduling decisions.

The reason why schedulers considering overlap cardinality work better is intuitive. As we state in Section 2.1 and show in Figure 1(a), (a) data transfer time significantly affects the entire execution time of a data-intensive application, and (b) tasks have a high degree of data-sharing among themselves. This strategy also works well in the real world because data location is relatively static and easy to obtain compared to dynamic metrics such as network bandwidth and CPU loads [4].

2.5 Problems of Task-Centric Scheduling and Possible Solutions

We observe two problems from task-centric scheduling strategies. These problems are significant because data replication and task replication [4, 5] never address the second problem, although the first problem can be avoided by both mechanisms.

1) **Unbalanced Task Assignments:** As mentioned by Ranganathan *et al.* [5], task-centric scheduling with data reuse has the problem of overloading certain sites with popular files. Since the overlap cardinality is the primary metric when assigning a task, workers with popular files may be assigned more tasks than the workers with less popular files. Since this problem is inherent in task-centric scheduling, other mechanisms need to be used to avoid the problem, *e.g.*, data replication [5] and task replication [4].

With data replication, the system keeps track of the popularity of each file. If a file’s popularity exceeds the pre-determined threshold, it is replicated to other sites. Thus, data replication helps to distribute the load of sites with popular files [5].

Task replication can also help to distribute the unbalanced load caused by popular files. With task replication, the scheduler first distributes its tasks according to the overlap cardinality. Once the initial assignment is done, the scheduler waits until at least one worker becomes idle. Then it picks a task already assigned to a worker and replicates it to the idle worker. If one of the workers finishes the task, the other worker cancels the task. The process is repeated whenever there is an idle worker. This strategy, called *storage affinity*, is proposed and evaluated by Santos-Neto *et al.* [4]. They show that a task-centric scheduler with data reuse and task replication performs better than other scheduling strategies with dynamic information such as CPU loads and available

bandwidth.

2) **Long Latency between scheduling and execution:** Task-centric scheduling typically has long latency between scheduling and execution. The following two reasons cause this problem - (1) Since each worker accepts tasks passively from the scheduler and stores received tasks in its queue, there is latency between task assignment time and the actual execution time. (2) Since storage at a site is limited in size, some files required by a task may have been replaced by other required files between the scheduling and execution times of the task.

Therefore, it is possible that a worker was assigned a task because it had some files needed by the task, but at the time of execution, the worker might no longer have some of those files. This “premature scheduling decision” can cause performance degradation with small storage sizes as we show in Section 4.

2.6 Advantages of Worker-Centric Scheduling

In comparison to the above approach, worker-centric scheduling does not suffer from the unbalanced task assignment problem because a worker requests a new task to the scheduler only when its local policies allow it to execute a task. This means that it is not necessary to have other mechanisms to resolve the issue. Therefore, a worker-centric scheduler only needs to consider its scheduling metric, which leads to a simpler scheduler design.

In fact, both data replication and task replication are orthogonal mechanisms to improve performance in worker-centric schedulers. Thus, they might help the performance of worker-centric schedulers, but are not necessary. However, task-centric schedulers *require* other mechanisms because unbalanced task assignment caused by popular files actually *hurts* the performance of task-centric schedulers [5].

In addition, worker-centric scheduling has short latency between scheduling and execution compared to task-centric scheduling. This arises because w.r.t. a worker, this is a *just-in-time* scheduling policy. Each worker executes a task as soon as the task has arrived at the worker. Thus, it does not suffer from the premature scheduling decisions.

In Section 3, we focus on worker-centric scheduling strategies and propose various metrics that consider data-reuse. We also show in Section 4 that worker-centric scheduling without additional mechanisms can achieve better performance in many scenarios than task-centric scheduling with additional mechanisms.

3 New Worker-Centric Scheduling Algorithms

In this section, we present our new worker-centric scheduling algorithms that attempt to exploit locality by considering data-reuse during scheduling.

3.1 Basic Algorithm

Our basic algorithm is shown in Figure 3. It is a worker-centric algorithm, with one global scheduler and multiple sites, each containing multiple workers. Upon receiving a request from a worker, the global scheduler calculates the weight of each as-yet-unscheduled task (*CalculateWeight()*) and chooses the best task to assign to the requesting worker (*ChooseTask()*). Notice that worker requests are processed sequentially. *CalculateWeight()* and *ChooseTask()* take into account

```

while(forever):
    req = GetNextRequest()
    if taskQueue is empty:
        wait for a task
    for each task t in taskQueue:
        CalculateWeight(t)
    t = ChooseTask(n)
    ReturnRequest(t)

```

Fig. 3. Pseudo-code of the basic algorithm. The global scheduler performs this algorithm whenever a worker requests a task.

the set of files already at the worker’s site, and the set of files required by the worker, thus attempting to exploit locality. These are detailed next.

As mentioned in Section 2.2, for simplicity of exposition, we restrict our discussion to tasks that share equally-sized files. However, our algorithms can easily be extended to varied file sizes by merely considering a “file block” (instead of a file) as a unit of sharing among tasks.

3.2 CalculateWeight()

CalculateWeight() calculates a weight for each each task in order to exploit the locality of file access. This weight can be calculated via one of three possible metrics - *Overlap*, *Rest*, and *Combined*. Before further discussion, we need to define the following terms and conditions:

- 1) T : the set of all unscheduled tasks that the scheduler currently has in its queue.
- 2) F_t : the set of overlapping files between task t and the data storage at the site of the requesting worker.
- 3) $|t|$: the total number of files required by task t .
- 4) r_i : the number of past references of the file i at the local storage (*i.e.* data server) of the requesting worker, *i.e.*, the number of previously completed tasks at the site that accessed file i .
- 5) Task t is said to be *better* than task t' , when $CalculateWeight(t) > CalculateWeight(t')$

Now we consider three metrics that could be used by the scheduler.

- 1) *Overlap*: This metric is the overlap cardinality (discussed in Section 2.2). It counts the number of files that are needed by the given task and are already present in the local storage of the requesting worker. Thus, $|F_t|$ is the overlap cardinality. Intuitively, the goal of this metric is to maximize the chance of reusing the data already stored in the local storage of the requesting worker. As mentioned before, this metric is the primary metric of task-centric scheduling strategies in the previous studies.
- 2) *Rest*: This metric is the inverse of the number of files that need to be transferred in order to execute the given task, *i.e.*, $rest_t = \frac{1}{|t| - |F_t|}$. Intuitively, the goal of this metric is to minimize the number of files that need to be transferred. This is a complement of *overlap* metric conceptually.
- 3) *Combined*: For this metric, each data server keeps for each file the number of past references, *i.e.*, the number of previously completed tasks at the site that have accessed the file. It combines these past references and *rest* using an

equation defined as follows. We define ref_t to be the total references of all the overlapping files of task t at the worker’s site, *i.e.*, $ref_t = \sum_{i \in F_t} r_i$. Now, let $totalRef$ be the sum of all ref_t over all t in T (w.r.t. the requesting worker’s site), *i.e.*, $totalRef = \sum_{t \in T} ref_t$. Also, let $totalRest$ be the sum of all $rest_t$ over all t in T , *i.e.*, $totalRest = \sum_{t \in T} rest_t$. Then, $combined_t = \frac{ref_t}{totalRef} + \frac{totalRest}{rest_t}$. Intuitively, this metric attempts to exploit locality of file access, and thus minimize both the number of files that need to be transferred as well as to prefer workers that accessed the same files in the past.

3.3 ChooseTask()

Since the scheduler greedily assigns a task to a worker based on the value of $CalculateWeight()$, there is some possibility of sub-optimal assignments. One reason for this is the sequential nature of such worker-centric scheduling. For example, suppose worker h is a better candidate to execute task t than worker h' , but worker h' requests a task right before worker h requests a task. In this case, the scheduler will assign task t to worker h' rather than h . This can happen quite often especially for data-intensive applications - since file transfer time is usually long after a task assignment, the global scheduler can receive a number of requests from different workers during the transfer. So it is possible that a better worker comes by while the previously-assigned worker has not even started processing, *i.e.*, it is still awaiting the file transfer to complete.

To take these types of scenarios into account, we use randomization when choosing a task through $ChooseTask(n)$. $ChooseTask(n)$ then executes two steps. First, it chooses a set, T_n , of the best n tasks among all tasks (*i.e.*, tasks with n largest values calculated by $CalculateWeight()$), where n is a parameter. Second, it chooses one task among the best n tasks with a probability proportional to the $CalculateWeight()$ values. Thus the probability of choosing task t is,

$$P_t = \frac{CalculateWeight(t)}{\sum_{k \in T_n} CalculateWeight(k)}.$$

If $n \geq 2$, this is a randomized approach. If $n = 1$, this is a deterministic approach that greedily chooses the best task. Notice that this procedure, in combination with $CalculateWeight()$, attempts to implicitly exploit the locality of file access.

3.4 Reducing Communication Cost

In order to make the scheduling decision for a requesting worker, we assumed above that global scheduler has all the necessary information about files currently stored at the requesting worker’s site, namely, (1) names of files that the data server is currently storing, and (2) the reference count for each of these files. In other words, we assumed that the global scheduler implicitly maintains a *reference table*, as shown in Figure 4. In this table, there is one column per file in the job, and one row per site in the Grid. Each entry (i, j) specifies “reference count” for file j at site i . The reference count denotes the past references of file j at site i and also shows the presence of file j at site i .

There are two efficiency sub-problems that need to be addressed: how to maintain this table efficiently, and how to keep it updated with minimal network bandwidth overhead. The first sub-problem is addressed by having the

File ID Site ID	file0	file1	file2
site0	N / A	12	6
site1	N / A	N / A	N / A
site2	10	1	5
site3	8	N / A	1
site4	N / A	N / A	1

Fig. 4. A reference table example. Each entry contains a reference counter. We use N/A to indicate that the entry is not present for the sake of demonstration.

global scheduler maintain a local hash table per site (row in the reference table), containing the names of files currently stored at that site along with their reference counts. File names are the keys for this data structure. Notice that lookup, insertion and deletion into this hash table are each $O(1)$ on expectation.

The bandwidth problem is addressed by piggybacking each task-requesting message, from a worker to the global scheduler, with the set of file names that have been replaced at the data server of the worker’s site *since the last request from the same site, i.e.*, the list of names of files that were eliminated from the site’s data server since its last request. The global scheduler deletes these file names from the hash table for that site. Then, once it makes the requested scheduling decision for the worker, the new files required by the assigned task are inserted into this hash table and the corresponding reference counts are initialized to 1. For all other files that are already present at the site and required by the assigned task, the global scheduler increments corresponding reference counts by 1. In this way, the communication between the worker to the global scheduler is reduced to only once per request no matter how many files are added and/or deleted from the site’s data server.

This approach is very efficient for our considered cases. In spite of file-sharing across tasks, each task in our observed data-intensive applications typically accesses a relatively small number of files compared to the total number of files for a given application. For example, in the *Coadd* traces, no task accesses more than 181 files out of a total of 588,900, in spite of data-sharing. This also means that at most 181 files are replaced between two consecutive requests. Thus, assuming file names are 4 bytes each, the additional information piggybacked along with a worker request is at most 724 bytes in size, which is reasonably small.

3.5 Complexity

If $|T|$ is the number of currently waiting tasks, and $|I|$ is the maximal number of files required by any task, then the total communication complexity of our algorithm arises out of the per-request piggybacked information as described in the previous section - this is $O(|I|)$ per task assigned to a worker. Similarly, the computation complexity is $O(|I| + |T| \times |I|)$ per task assigned to a worker, with the first term accounting for the hash table operations, and the second one for the scheduler’s operation itself. This is $O(|T| \times |I|)$, and more efficient than task-centric strategies used by Ranganathan *et al.* [5] and Santos-Neto *et al.* [4], which compare all pairs of tasks and sites. Their complexity is $O(|T| \times |I| \times |S|)$ (where $|S|$ is the total number of sites), even assuming the use of a hash table similar to that described in the previous section. Our approach is more efficient

because we do not assume any knowledge (a priori or otherwise) about sites other than the requesting worker’s.

4 Evaluation

In this section, we present our evaluation of worker-centric scheduling strategies and discuss the results.

4.1 Simulation Overview

To demonstrate the advantages of worker-centric scheduling over task-centric scheduling, we implement our basic algorithm with three metrics on the SimGrid simulator [14]. For comparison, we also implement *storage affinity* [4], a task-centric scheduling with data reuse and task replication.

We vary five main parameters in our experiments - (1) capacity of each data server, (2) number of workers per site, (3) computation time, (4) number of sites, and (5) file size. The default values for these parameters are summarized in Table 1, and used in our experiments unless otherwise noted. However, we vary each of these 5 parameters in our experiments to see the effects of different values. Throughout the experiments, the computation time of each task is linear to the number of files (*i.e.*, $(\text{number of files}) * (\text{unit computation cost})$).

Table 1. *Default parameters for experiments*

Unit computation cost	1,000 MFLOPS
capacity of each data server	6,000 files
number of workers per site	1
number of sites	10
file size	25 MB

Our main workload is *Coadd* (Sloan Digital Sky Survey southern-hemisphere coaddition [2,3]). As mentioned before, *Coadd* is a spatial processing application that has 44,000 tasks accessing 588,900 files in total. We use only the first 6,000 tasks of *Coadd* to finish our experiments in a reasonable amount of time. A total of 53,390 files are accessed by these 6,000 tasks. More workload characteristics are shown in Table 2. Although we only use the first 6,000 tasks, our workload characteristics remain similar to Figure 1(a).

4.2 Simulation Environment

Network Configuration: We use 5 different topologies, each with 90 sites, generated with Tiers topology generator [15]. Tiers is a structural topology generator that generates hierarchical cluster topologies. We use Tiers because it is well-supported by SimGrid, the simulator we use in our experiments. Only a subset of 90 sites are used in each experiment. For each topology, there are one global scheduler and one global file server which stores all the files. At each site, there are 30 workers and 1 data server. All 30 workers and the data server in a site share outgoing links to the global scheduler and the file server. Intra-site communication cost (cause by bandwidth and latency) is negligible. Inter-site communication cost is determined by underlying network links generated by Tiers. Each path between two sites consists of multiple network links, and the bandwidth and latency of each of these links determine the inter-site communication cost. Table

Table 2. *Characteristics of Coadd with 6,000 tasks*

Total number of files	53,390
Max number of files needed by a task	101
Min number of files needed by a task	36
Average number of files needed by a task	78.4327

3 summarizes the average and standard deviation of bandwidth values between a site to the file server for each topology. Each worker’s computation capacity (in MFLOPS) is chosen randomly from top500 list [16] and is uniformly divided by 100, since most of the 500 machines are too powerful. Each experiment is performed with 5 different topologies and the results are averaged over the 5 runs.

Table 3. *Average bandwidth and standard deviation between a site and the file server*

	Avg (MB/s)	Std dev
Topology 0	4.418	5.416
Topology 1	4.631	6.734
Topology 2	3.858	2.599
Topology 3	3.432	1.432
Topology 4	3.932	2.778

Background Jobs: We perform our experiments with background jobs as well as without background jobs. We use background jobs to evaluate the performance of different strategies in the presence of competing applications running on each site. Since a site is typically shared by different schedulers and local users, this gives us a more realistic setting.

We simulate background jobs through varying each worker’s CPU load. A worker is always executing a task for the Grid job in question, but in addition it is also running background jobs. The background jobs thus slow down the execution of the task at the worker. The load due to these background jobs is simulated as follows: at each worker, once every 5 minutes, the background CPU load is picked as a floating-point number uniformly at random between 0 to 100. This becomes the worker’s background load over the next 5 minutes. Considering that the total job execution time in our simulations is $O(\text{tens to hundreds of days})$, we consider the granularity of 5 minutes to be fine-grained enough to capture dynamics of background jobs.

4.3 Algorithms

We compare the following 6 different algorithms. The first algorithm is task-centric; the rest are worker-centric.

- 1) *task-centric storage affinity* : The task-centric scheduling with data reuse and task replication [4]. This is a deterministic algorithm.
- 2) *overlap* : Our basic algorithm with the *overlap* metric. This is a deterministic algorithm.
- 3) *rest* : Our basic algorithm with the *rest* metric. $n = 1$ for *ChooseTask*(n).

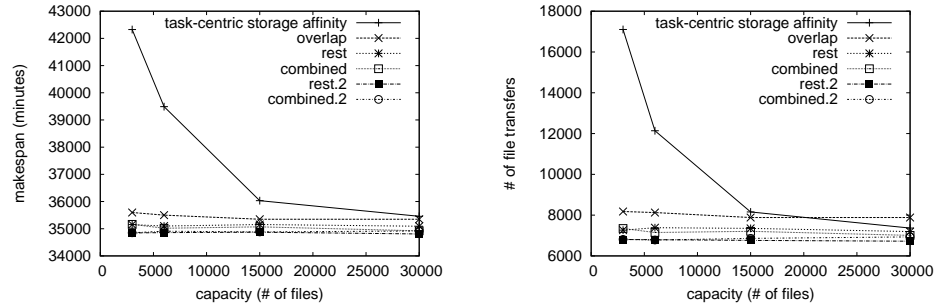


Fig. 5. (a) Makespan and (b) file transfers of each algorithm with different capacities of 3,000, 6,000, 15,000, and 30,000 files (with background jobs).

This is a deterministic algorithm.

4) *combined* : Our basic algorithm with the *combined* metric. $n = 1$ for *ChooseTask*(n). This is a deterministic algorithm.

5) *rest.2* : Our basic algorithm with the *overlap* metric. $n = 2$ for *ChooseTask*(n). This is a randomized algorithm.

6) *combined.2* : Our basic algorithm with the *overlap* metric. $n = 2$ for *ChooseTask*(n). This is a randomized algorithm.

We have tried different values of n for *ChooseTask*(n), but only 1 and 2 give good results. Thus, we only show the results of $n = 1$ and 2.

4.4 Capacity per Data Server

Figure 5(a) shows the makespan (*i.e.* total execution time) of each algorithm with different capacities of 3,000, 6,000, 15,000, and 30,000 files in the presence of background jobs. We do not present the results without background jobs since the performance characteristics are similar. Randomized algorithms, *rest.2* and *combined.2*, perform the best in all cases, which confirms that it avoids sub-optimal scheduling decisions described in Section 3.3. *Storage affinity* has a negative performance impact with smaller capacities because of premature scheduling decisions as discussed in Section 2.5. However, the performance becomes comparable to worker-centric scheduling as the storage size increases.

Figure 5(a) also shows the importance of considering the number of files that actually need to be transferred. Among the worker-centric strategies, *overlap* performs worse than other metrics because it does not explicitly consider the number of file transfers, while other metrics do. As we can see in Figure 5(b), *overlap* usually has higher number of file transfers than other metrics. Overall, the randomized algorithms appear to perform the best (*i.e.*, *rest.2* and *combined.2*).

The makespan of each metric in worker-centric scheduling shows steady behavior because the working set of a *Coadd* task is not big. As is shown in Table 2, a task needs 101 files at most, and roughly 78 files on average. Thus, a storage with 3,000 files can actually give similar performance as a storage with, say, 10,000 files.

Figure 6(a) shows the average utilization of each worker (accounting for both the main Grid job and the background jobs). For *task-centric storage affinity*, the

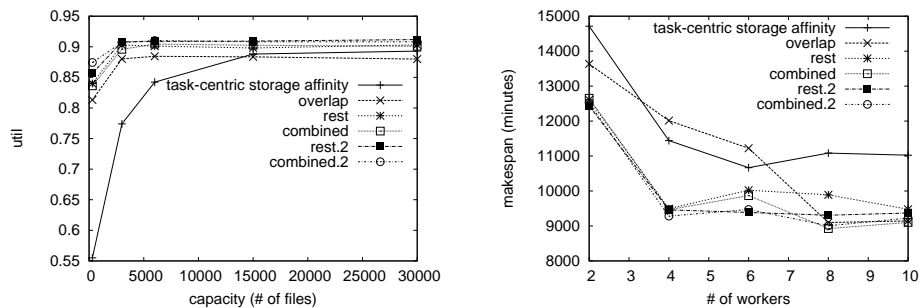


Fig. 6. (a) Average utilization at worker, with different capacities of 3,000, 6,000, 15,000, and 30,000 files (with background jobs) (b) Makespan with different numbers of workers at a site (without background jobs)

low utilization with the capacity of 3,000 files means that the greedy approach requests files more often than other strategies. This behavior shows (1) that randomized decisions can be better than taking what looks as the “best” decision at some particular time and, again, that (2) the *task-centric storage affinity* suffers from premature scheduling decisions.

Due to the lack of space, we do not present the utilization results without background jobs here. However, the utilization of each worker with background jobs is slightly higher than that of each worker without background jobs. There are two factors contributing to this result. The first factor is obviously background jobs running on each worker. The second factor is that it takes more time for a worker to finish a task with background jobs. Thus, the utilization goes higher with background jobs.

4.5 Number of Workers per Site

Figure 6(b) shows the makespan of each algorithm with different numbers of workers at a site. *combined.2* performs the best mostly, which shows that minimizing file transfers as well as considering past references helps to reduce the makespan. Overall, worker-centric scheduling metrics perform well with smaller numbers of workers, but *storage affinity* performs well with larger numbers of workers. Also, randomized algorithms that consider the number of file transfers perform better than others.

The makespan of each algorithm flattens as the number of workers increases. In some cases, the performance is worse with more workers (in Figure 6(b))! We can understand the reason behind this behavior with two factors that contribute to the makespan. First, as the number of workers increases at a site, the contention at the data server of the site increases. Since the data server processes each request one by one so as to minimize the redundant file transfers (as mentioned in Section 2.2), this contention is unavoidable. This factor has a negative impact on the makespan (*i.e.* increases it). On the contrary, as the number of workers increases, the number of files that can be shared by the workers also increases. This factor has a positive impact on the makespan. The interaction of these two factors results in different behaviors of different algorithms.

To validate the reason, Figure 7(a) shows the number of file transfers per worker and Figure 7(b) shows the corresponding utilization. It shows that the

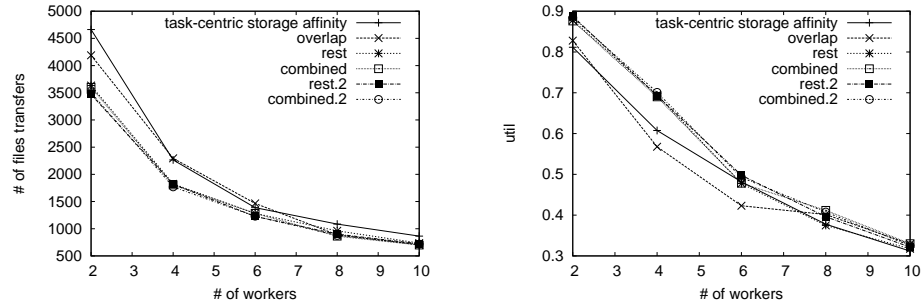


Fig. 7. (a) Average number of file transfers per worker with different numbers of workers at a site (b) Average worker utilization, with different numbers of workers at a site. We only show the results without background jobs, since the presence of background jobs does not show any different behavior.

Table 4. Result of the rest metric at a site with 2 workers, 4 workers, 6 workers, and 8 workers. All numbers are averages per worker. Note that rest shows the worst makespan with 6 workers at a site.

	waiting time (hrs)	transfer time (hrs)	# of file transfers
2 workers	3.59	30.35	3998.5
4 workers	40.32	45.45	2086.5
6 workers	98.35	33.85	1335.17
8 workers	75.93	18.81	906.38

average number of file transfers per worker decreases as the number of workers increases. Thus, it shows that good file-sharing is achieved intra-site as the number of workers increases. In addition, Table 4 shows the result of the *rest* metric at one particular site with 2, 4, 6, and 8 workers. It shows (1) average waiting time that a file request spends at the data server’s waiting queue, (2) transfer time that it takes to transfer all the files from the external file server to the data server, and (3) associated number of file transfers.

In the case of 2 workers in Table 4, the contention at each data server and the file server is very low compared to other settings, simply because there are fewer workers. Thus, the waiting time and the transfer time are rather small even though the number of file transfers is high.

We can reason why the performance is sometimes worse with more workers with the data of 4 workers, 6 workers, and 8 workers. If we look at the data in this range, both the average number of file transfers and the average transfer time decrease as the number of workers increases, but the average waiting time peaks at 6 workers. This means that the reduced transfer time is not enough to compensate the increased competition at the data server for *rest* with 6 workers at a site. For the same reason, other algorithms sometimes exhibit a worse makespan with more workers.

4.6 Effect of Computation Time

With our default parameter values in Table 1, the average utilization per worker is usually more than 90%, which means that each worker spends most of its time on computation. Thus, we perform an experiment with smaller values of

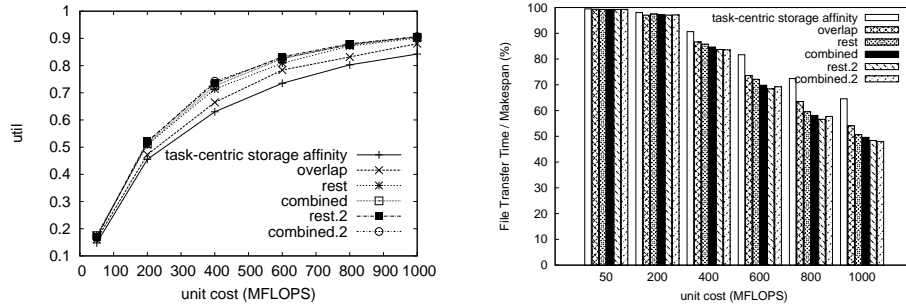


Fig. 8. (a) Average utilization per worker, and (b) total file transfer time compared to makespan, for different unit computation costs of 50, 200, 400, 600, 800, and 1,000 MFLOPS

unit computation time in order to understand how different computation-to-communication ratios affect the behavior of each strategy. As mentioned before, the computation time of each task is linear to the number of files that it needs to process, *i.e.*, $(\text{computation time}) = (\text{number of files}) * (\text{unit computation cost})$. We vary the unit computation cost in this experiment.

Figure 8(a) and Figure 8(b) show that our experiment covers a wide range of communication-to-computation ratio. As shown in Figure 8(a), the utilization of each worker (*i.e.*, $(\text{total computation time of the worker}) / (\text{total execution time of the worker})$) varies from roughly 0.2 to 0.9. Also, Figure 8(b) shows that the file transfer time (*i.e.*, communication time) takes from roughly 50% to almost 100% of the entire makespan. Thus, our experiment covers a wide range of communication-to-computation ratio, and still captures the characteristic of long communication time in data-intensive applications. Although Figure 8 shows the results without the presence of background jobs, the overall behavior remains similar even with background jobs. Note that file transfer time does not directly contribute to worker utilization as in Figure 8. The reason is because computation is parallelized, and hence, most workers are busy with doing computation even when the file server transfers files. This explains a seemingly inconsistent behavior of Figure 8, in which the file transfer time takes roughly 50% with the unit computation cost of 1,000 MFLOPS in Figure 8(b), even when the average utilization of each worker is roughly 90% in Figure 8(a).

Figure 9(a) shows the makespan (with background jobs) of each algorithm in percentile scale using task-centric storage affinity as a baseline comparison. We do not present the results without background jobs since they exhibit similar behaviors. Overall, we observe that the performance trend remains similar across different strategies even with various computation-to-communication ratios. Worker-centric strategies perform better than the task-centric storage affinity in terms of makespan. In the best case, worker-centric *rest* takes roughly 28% less makespan time than *task-centric storage affinity*. Also, the gap between task-centric storage affinity and other strategies generally becomes wider as the unit computation cost decreases. This is an expected behavior since file transfer time becomes more dominating in total execution time as the unit computation cost decreases.

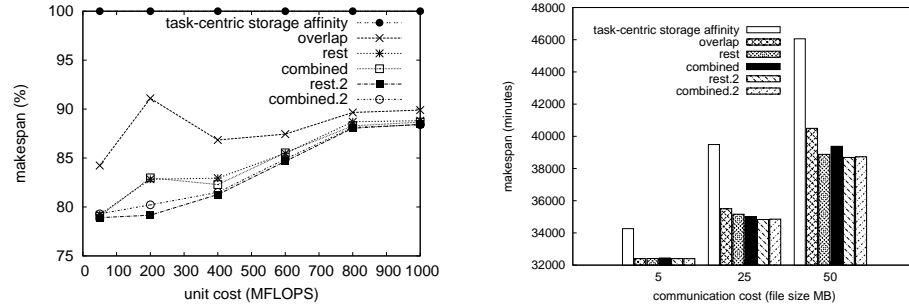


Fig. 9. (a) Makespan (percentile) of each algorithm with different unit computation costs of 50, 200, 400, 600, 800, and 1,000 MFLOPS (b) Makespan with different file sizes (both with background jobs)

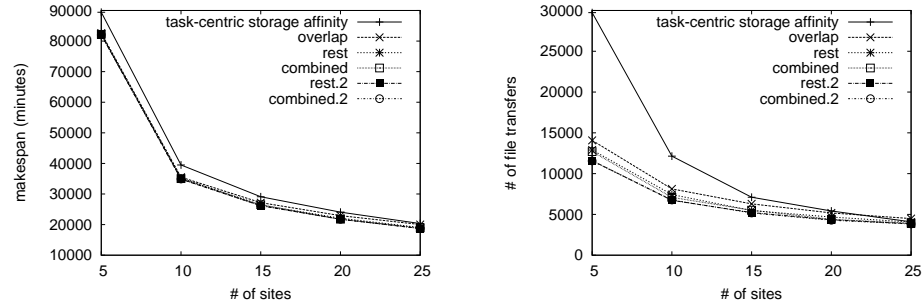


Fig. 10. (a) Makespan with different numbers of sites (b) Number of file transfers with different numbers of sites (both with background jobs)

4.7 Number of Sites

Figure 10(a) shows the makespan of each algorithm with different numbers of sites and Figure 10(b) shows the number of file transfers accordingly. Generally, the makespan of each algorithm reduces as the number of sites increases, as expected. *combined.2* performs the best, which again confirms that minimizing file transfers as well as considering past references helps to reduce the makespan. In the best case, *combined.2* takes roughly 17% less makespan time than *task-centric storage affinity*. Randomized algorithms perform better than deterministic algorithms, which again shows that it avoids sub-optimal scheduling decisions described in Section 3.3.

4.8 File Size

Figure 9(b) shows the makespan of each algorithm with different file sizes. We choose small (5MB), middle (25MB), and large (50MB) file sizes. The makespan grows almost linearly as the file size grows. Since all algorithms consider files as the primary metric, various file sizes do not result in dramatically different behaviors. *combined.2* shows the best performance just like many other scenarios shown before. The general behavior remains the same even in the presence of background jobs.

5 Related Work

Spatial Clustering [2] creates a task workflow based on the spatial relationship of files in the input data set. It improves data reuse and diminishes file transfers

by clustering together tasks with high input-set overlap. Two drawbacks to this approach are that (1) it cannot handle new jobs arriving asynchronously, and (2) it is application specific.

Storage Affinity [4] also addresses file reuse for data-intensive applications. The algorithm computes a data affinity value for each task, for each site, according to the input set of each task and the data currently stored at a site’s networked storage. To address inefficient CPU assignments, they propose replicating tasks, also based on the storage affinity. The algorithm shows improved makespan and good data reuse, specially when compared to the XSufferage [17] scheduling heuristic.

Decoupling data scheduling from task scheduling was proposed by Ranganathan *et al.* [5]. The work evaluates four simple task scheduling mechanisms and three simple data scheduling mechanisms. Best results are obtained when a task is scheduled to a site that has a good part of its input data already in place, combined with proactive replication of a popular input data-set to a random/least-loaded site.

A pull-based scheduler is proposed by Viswanathan *et al.* [8]. It employs an Incremental Based Strategy, where a scheduler determines how to fraction a job among available workers, based on worker’s computing speed and estimated buffer. This work completely ignores data transfer time, and requires knowledge of CPU speed and memory size in all workers.

Rosenberg *et al.* [9] study global scheduling strategies in the Grid-like environments theoretically. Their scheduling strategies focus mainly on *DAGs of tasks*, where tasks are inter-dependent and pre-ordered, and the dependency structure follows DAG (Directed Acyclic Graph). Although they discuss *pull* and *push* strategies, their studies do not assume (1) data-intensive applications (transfer time, storage capacity, data correlation, etc), (2) data-sharing, and (3) task-independence. Thus, the issues are not related to our work.

6 Conclusion and Future Work

We argued that worker-centric scheduling is more desirable than task-centric scheduling to exploit locality of interest present in data-intensive applications. We base our argument on two problems of task-centric scheduling, namely, un-balanced task assignments and premature scheduling decisions. We proposed various metrics, both deterministic and randomized, that can be used with worker-centric scheduling and found that metrics considering the number of file transfers generally give better performance over metrics considering the overlap between a task and a storage. We also found that worker-centric scheduling algorithms achieve better or comparable performance to task-centric scheduling, with the randomized approaches performing best. Our future work includes quantifying how much data-sharing is required for our algorithms to be effective, and using multiple applications to evaluate the performance of our algorithms.

References

1. Allcock, W.E., Bester, J., Bresnahan, J., Chervenak, A.L., Foster, I.T., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D., Tuecke, S.: Secure, efficient data transport

- and replica management for high-performance data-intensive computing. CoRR **cs.DC/0103022** (2001)
2. Meyer, L., Annis, J., Mattoso, M., Wilde, M., Foster, I.: Planning Spatial Workflows to Optimize Grid Performance. Technical Report, GriPhyN 2005-10 (2005)
 3. Sekhri, V.: Lessons Learned on Summer 04 Grid SDSS Coadd. <https://www.darkenergysurvey.org/the-project/simulations/sdss-grid-coadd/summer-04-grid-coadd>
 4. Santos-Neto, E., Cirne, W., Brasileiro, F.V., Lima, A.: Exploiting Replication and Data Reuse to Efficiently Schedule Data-Intensive Applications on Grids. In: Proc. of JSSPP. (2004)
 5. Ranganathan, K., Foster, I.T.: Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications. In: Proc. of HPDC-11. (2002)
 6. Casanova, H., Obertelli, G., Berman, F., Wolski, R.: The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In: Proc. of SC. (2000)
 7. Iamnitchi, A., Doraimani, S., Garzoglio, G.: Filecules in High-Energy Physics: Characteristics and Impact on Resource Management. In: Proc. of HPDC-15. (2006)
 8. Viswanathan, S., Veeravalli, B., Yu, D., Robertazzi, T.G.: Design and Analysis of a Dynamic Scheduling Strategy with Resource Estimation for Large-Scale Grid Systems. In: Proc. of GRID. (2004)
 9. Rosenberg, A.L., Yurkewych, M.: Guidelines for scheduling some common computation-dags for internet-based computing. *IEEE Transactions on Computers* **Vol. 54, No. 4** (April 2005)
 10. Ian T. Foster et al. The Grid2003 Production Grid: Principles and Practice. In: Proc. of HPDC-13. (2004)
 11. da Silva, D.P., Cirne, W., Brasileiro, F.V.: Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In: Proc. of Euro-Par 2003. (2003)
 12. Pinedo, M.: *Scheduling: Theory, Algorithms and Systems*. Second edn. Prentice Hall, New Jersey, USA (August 2001)
 13. Cirne, W., Brasileiro, F., Sauv, J., Andrade, N., Paranhos, D., Santos-Neto, E., Medeiros, R.: Grid Computing for Bag of Tasks Applications. In: Proc. Third IFIP I3E. (September 2003)
 14. Legrand, A., Marchal, L., Casanova, H.: Scheduling Distributed Applications: the SimGrid Simulation Framework. In: Proc. of CCGrid. (2003)
 15. Doar, M.B.: A Better Model for Generating Test Networks. In: Proc. of Globecom. (1996)
 16. Top 500 list. <http://www.top500.org>
 17. Casanova, H., Zagorodnov, D., Berman, F., Legrand, A.: Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In: 9th Heterogeneous Computing Workshop. (2000)