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

In this paper we argue that a worker-centric scheduler design is more desirable for data-intensive applications in Grid environments. Previous research on task-centric scheduling for data-intensive applications has identified that reusing the data present in a Grid site improves performance. However, task-centric scheduling bears two problems - unbalanced task assignments and premature scheduling decisions. On the contrary, both of these problems can be avoided by using worker-centric scheduling, thus worker-centric scheduling leads to a simpler scheduler design and better performance. Therefore, we propose a series of worker-centric scheduling strategies for data-intensive applications and evaluate, with a real application (Coadd), how each strategy performs compared to a task-centric one. Our results show that worker-centric strategies improve the performance in terms of makespan and bandwidth usage.

1. Introduction

Data-intensive Grid applications are the class of applications that run on distributed Grid sites and access large amounts of datasets. Since these datasets range from several terabytes to petabytes [3], it is impractical to replicate all the data at every execution site, where a site refers to a single cluster. Instead, a data-intensive Grid application is divided into many small tasks, so that each site can execute a task with only a subset of data. Examples of data-intensive Grid applications can be found in many scientific domains such as Physics, Earth science, and Astronomy.

In a data-intensive Grid application, data is frequently transferred and replicated from one site to other, since execution of a task requires a subset of data to be acquired beforehand. Therefore, it is important for a Grid scheduler to be aware of this characteristic in order to reduce the time wasted for file transfers. Schedulers that do not exploit this characteristic are known to perform poorly with data-intensive Grid applications. [4][10][13][14].

Fortunately, many data-intensive Grid applications exhibit data sharing among different tasks, which gives an opportunity to reduce the number of redundant file transfers. This type of applications include data mining, image processing, genomics [14], and spatial processing applications which consist of tasks that process overlapping regions [10]. In fact, many researchers have exploited the data sharing characteristic of Grid data-intensive applications in the context of scheduling and application-specific workflow planning [4][10][13][14].

While previous studies by Casanova et al.[4], Ranganathan et al.[13] and Santos-Neto et al.[14] design schedulers that exploit data sharing and successfully demonstrate their benefits over traditional schedulers, their design is *task-centric* (i.e., each host receives tasks passively from a scheduler). Task-centric scheduling suffers from two major issues. First, there is a possibility of unbalanced task assignments, resulting in some sites being overloaded with tasks. Second, conditions of a host at the time of scheduling can be different from the conditions of the host at the time of execution because each task normally waits in the host's task queue.

We argue that having each host request a task only when it is idle addresses both of these issues. We call this approach as *worker-centric* scheduling. Since the resource suppliers (i.e. workers) are frequently overloaded, as observed in a Grid like PlanetLab [11], we believe that the scheduling needs to be driven by resource suppliers rather than resource demanders (i.e. tasks).

In this paper, we present a series of worker-centric scheduling strategies to demonstrate the advantages of worker-centric scheduling over task-centric scheduling for data-intensive Grid applications. In our strategies, each worker requests a task to the global scheduler only when it is idle. Upon receiving the request, the global scheduler iterates over the list of tasks and finds the "best" one for the worker according to its metric in use. We experiment with three different metrics that mainly consider the data sharing characteristic of data-intensive Grid applications, and aim to (1) maximize the chance of reusing the data, and (2) to minimize

[*]This work was supported in part by NSF CAREER grant CNS-0448246.

the number of file transfers. Our simulation results with a real application, *Coadd* (Sloan Digital Sky Survey southern-hemisphere coaddition [10][15]), indeed show that worker-centric scheduling gives better performance than task-centric scheduling in many scenarios.

The rest of the paper is organized as follows. In Section 2, we present background information. Section 3 presents the detailed problems of task-centric scheduling and advantages of worker-centric scheduling. Section 4 presents our basic algorithm and various metrics that we consider. Section 5 presents our simulation results and Section 6 discusses relates work. Section 7 concludes our paper.

2. Background and Basics

In this section, we first present the characteristics of data-intensive applications to motivate the scheduling problem. Next we present our system model and introduce the terms that we use in this paper. We then discuss two types of schedulers, task-centric and worker-centric, and discuss the issues of schedulers for data-intensive applications.

2.1. Characteristics of Data-Intensive Applications

Although the characteristics of data-intensive applications are well-known in the literature [4] [7] [10] [13] [14], we discuss them here to motivate the problem. As a real example, we use one particular application, *Coadd* (Sloan Digital Sky Survey southern-hemisphere coaddition [10][15]) in our discussion.

In general, tasks in a data-intensive application access a large set of files, thus data transfer time is the dominating factor in the entire execution time (i.e. data-intensive applications are network-bound [7][14]). In addition, the tasks have a high degree of data sharing among them, which gives an opportunity to reuse data in local storages.

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. [10] that when it was run on Grid3 [2] with over 30 sites and 4500 CPUs, it took roughly 70 days to completion. One of the reasons for the observed long completion time was the large number of files necessary for each task. Finally, [10] states that these characteristics would also be expected in other spatial processing applications.

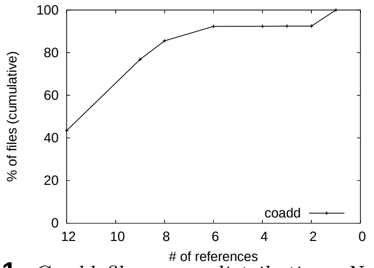

Figure 1. *Coadd file access distribution. Note that the x-axis is in decreasing order, so each point in the CDF represents the minimum number of files accessed and the y-axis is cumulative. x-axis shows the number of tasks that access a given file. Roughly 90% of files are accessed by 6 or more tasks.*

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. If we assume that each file is fixed at 5MB as in [10], 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.

2.2. System Model

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

1. A *job* is an application composed of multiple parallel *tasks*. Each task does not need to communicate with other tasks to proceed (i.e. a job is a Bag-of-Tasks [14]).
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 storages to be one combined storage.
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 can start executing a task only when all the files necessary for the task are present in the local data storage.
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. We assume that all files are equally-sized to simplify our discussion. However, all discussions in this paper are equally applicable to systems with varying file sizes, since the number of bytes is what matters.

We use the following terms throughout the paper.

- *Makespan*[12] is the total execution time of a job. This is the main metric for performance measurement.
- A task and a local storage (i.e. the data server at a site) are said to *overlap* with each other, when some files necessary for the task are already present in the local storage. We also use the term, *overlap cardinality*, to indicate the number of overlapping files.

2.3. Task- and Worker-Centric Schedulers

We consider two types of schedulers, namely, task-centric schedulers and worker-centric schedulers. This categorization is based on which side (worker or scheduler) is active when assigning a task.

We call a scheduler *task-centric*, when only the global scheduler is active in assigning tasks to workers. For a given set of tasks and a set of workers, it chooses the best match (based on its own metrics) between a worker and a task, and assigns the task to the worker. Each worker is passive and waits for a task to be assigned to itself by the scheduler. The worker usually has a task queue and executes the tasks in the queue one by one. Typical metrics that a scheduler uses are CPU load, network bandwidth, data overlap, etc.

On the contrary, we call a scheduler *worker-centric*, when a worker is active in assigning a task. The worker-centric scheduling 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. One example of the worker-centric scheduling is the traditional *workqueue* algorithm, which dispatches a task in FIFO order to an idle worker [6].

It is possible that a scheduler can keep track of idle workers and assign the next task only when a worker becomes idle. In this case, even though the scheduler assigns tasks actively, we would call it a worker-centric scheduler because it is semantically the same.

In short, the worker-centric scheduling makes scheduling decisions *only when* a worker becomes idle *for that specific worker.* Other scheduling approaches are considered task-centric. We will experimentally compare these two types of scheduling approaches in the context of data-intensive Grid applications.

2.4. Scheduling Issues for Data-Intensive Applications

Several previous studies have identified that reusing data in local storages gives a dramatic performance improvement for data-intensive applications [4][10][13][14]. Among others, studies by Ranganathan et al.[13] and Santos-Neto et al.[14] propose various task-centric scheduling strategies for data-intensive applications. Their studies suggest that making scheduling decisions based on data reuse indeed perform better than other scheduling decisions that consider various different metrics altogether. In their simplest form, strategies in both studies calculate the overlap cardinality (either the number of files or bytes) between all possible pairs of the files needed for a task and the files already present at a site. Then they assign each task with the maximum overlap cardinality to each site.

The reason why schedulers considering overlap cardinality work better is straightforward. As we state in Section 2.1 and show in Figure 1, (a) data transfer time is the dominating factor in the entire execution time of an data-intensive application (b) tasks have a high degree of data reference sharing among themselves. This strategy also works well in the real world because determining data location is relatively static and easy to obtain compared to dynamic metrics such as network bandwidth and CPU loads [14].

3. Task- vs. Worker-Centric Scheduling

Task-centric scheduling with data reuse bears two problems (1) unbalanced task assignments, and (2) latency between scheduling and execution. The first problem is avoidable by employing other mechanisms, for example, data replication and task replication [13][14]. However, the second problem is still present even with these mechanisms.

We first discuss these two problems of task-centric scheduling in detail.

3.1. Problems of Task-Centric Scheduling and Possible Solutions

Unbalanced Task Assignments As briefly mentioned by Ranganathan et al. [13], 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 [13] and task replication [14].

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 [13].

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 assigning is done, it waits until at least one worker becomes idle. Then the scheduler 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 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. [14]. 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.

Monitoring the load or the queue size of each CPU can also be used to avoid this problem. However, it is generally hard to obtain dynamic values like CPU loads. In addition, monitoring the queue size does not predict the future load because task run times are unpredictable a priori.

Long Latency between scheduling and execution This problem is caused by two reasons.

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 a storage is usually limited in size, it has to replace files at some point of time.

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 those files. This "premature scheduling decision" can cause performance degradation with smaller storage sizes as we show in Section 5.

3.2. Advantages of Worker-Centric Scheduling

Worker-centric scheduling does not suffer from the unbalanced task assignment problem because a worker requests a new task to the scheduler only when it is idle. 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 [13].

```
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)
```

Figure 2. *Pseudo-code of the basic algorithm. The global scheduler performs this algorithm whenever a worker requests a task.*

In addition, worker-centric scheduling has a short latency between scheduling and execution compared to task-centric scheduling. It makes scheduling decisions only when the worker is able to execute a task. Thus, it does not suffer from the premature scheduling decisions.

In the next section, we focus on worker-centric scheduling strategies and propose various metrics that consider data-reuse. We also show in Section 5 that worker-centric scheduling without additional mechanisms can achieve better performance in many scenarios than task-centric scheduling with additional mechanisms.

4. New Worker-Centric Scheduling Algorithms

In this section, we present our basic worker-centric scheduling algorithm, as well as various scheduling metrics that consider data-reuse for data-intensive applications.

4.1. Basic Algorithm

Our basic algorithm is shown in Figure 2. It is a worker-centric algorithm, thus there are one global scheduler and multiple workers, and each worker requests a task to the scheduler whenever it is idle. Upon receiving a request from a worker, the global scheduler calculates the weight of each and every task (*CalculateWeight()*) and chooses one to assign (*ChooseTask()*). We discuss two subprocedures *CalculateWeight()* and *ChooseTask()* in the next section.

4.2. *CalculateWeight()*

CalculateWeight() can calculate each task's weight based on three different alternatives - *Overlap*, *Rest*, and *Combined*. Before further discussion, we need to define the following terms and conditions:

- T: the set of all tasks that the scheduler has in its queue.
- F_t: the set of overlapping files between task t and the requesting worker.
- $|t|$: the total number of files required by task t.
- r_i: the number of past references of the file i at the local storage of the requesting worker.

- 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 related to the *overlap* metric, but is different from it.

3. *Combined*: For this metric, each worker keeps the number of past references for each file from prior tasks. 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, 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, 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 minimizes the number of files that need to be transferred as well as to prefer workers that accessed the same files in the past.

4.3. *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. For example, suppose worker h may be a better candidate to execute task t than worker h', but worker h' requests a task right before worker h becomes idle. In this case, the scheduler will assign task t to worker h' rather than h.

To take these types of scenarios into account, we use randomization when choosing a task through *ChooseTask(n)*. *ChooseTask(n)* takes two steps. First, it chooses a set, T_n, of best n tasks (i.e. tasks with n largest values calculated by *CalculateWeight()*) among all tasks. 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, $P_t = \frac{CalculateWeight(t)}{\sum_{k \in T_n} CalculateWeight(k)}$.

Table 1. *Default parameters for experiments*

capacity of each data server	6000 files
number of workers per site	1
number of sites	10
file size	25 MB

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

Total number of files	53390
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

If $n \geq 2$, it is a randomized approach. If $n = 1$, it is a deterministic approach that chooses the best task.

4.4. Complexity

The worst-case complexity of our basic algorithm with different metrics is $O(T{*}I)$, where T is the number of tasks and I is the worst-case number of files that a task needs. Task-centric strategies used in [13] and [14] compare all pairs of tasks and sites, thus the complexity of their algorithms is $O(T{*}I{*}S)$ where S is the number of sites. Our algorithm is more efficient because we do not assume any knowledge (a priori or otherwise) about other workers.

5. Evaluation

5.1. 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 [9]. Fro comparison, we also implement *storage affinity* [14], a task-centric scheduling with data reuse and task replication.

We vary the following four parameters in our experiments:

- capacity of each data server
- file size
- number of workers per site
- number of sites

The default values for the parameters can be found in Table 1, used unless otherwise noted.

Our main workload is *Coadd* (Sloan Digital Sky Survey southern-hemisphere coaddition [10][15]). 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. The workload characteristics are shown in Figure 3 and Table 2.

5.2. Simulation Environment

We use 5 different topologies with 90 sites each generated with Tiers topology generator [8]. Tiers is a

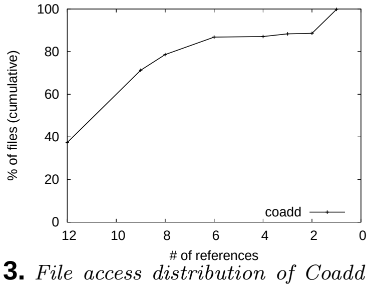

Figure 3. *File access distribution of Coadd with 6,000 tasks. Note that the x-axis is in decreasing order and the y-axis is cumulative. Roughly 85% of files are accessed by 6 or more tasks. The overall characteristics are very similar to the original Coadd except it is a scaled-down version.*

structural topology generator that generates hierarchical cluster topologies. 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 is negligible. Each worker's computation capacity (in MFLOPS) is chosen randomly from top500 list [1] and is 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.

5.3. 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 [14]. 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)*. 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()*, but only 1 and 2 give good results. Thus, we only show the results of $n = 1$ and 2.

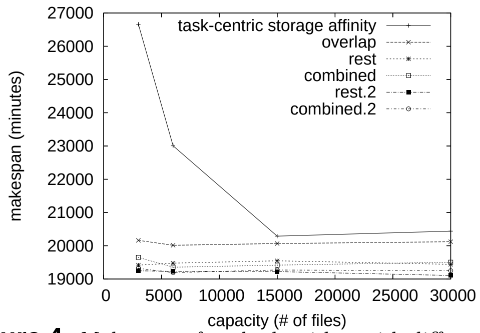

Figure 4. *Makespan of each algorithm with different capacities of 3000, 6000, 15000, and 30000 files.*

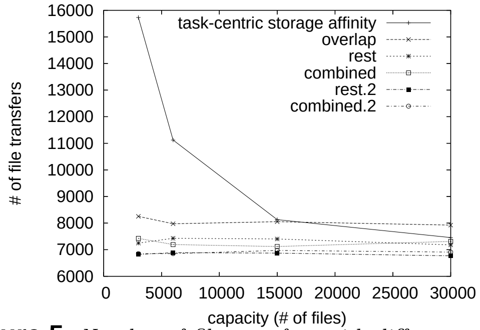

Figure 5. *Number of file transfers with different capacities*

5.4. Capacity per Data Server

Figure 4 shows the makespan (i.e. total execution time) of each algorithm with different capacities of 3000, 6000, 15000, and 30000 files. Randomized algorithm, *rest.2* and *combined.2* perform the best in all cases, which confirms that it avoids sub-optimal scheduling decisions described in Section 4.3. *storage affinity* has a negative performance impact with smaller capacities because of premature scheduling decisions as discussed in Section 3.1. However, the performance becomes comparable to worker-centric scheduling as the storage size increases.

Figure 4 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, *overlap* usually has higher number of file transfers than other metrics.

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 3000 files can actually give similar performance as a storage with, say, 30000 files.

5.5. Number of Hosts per Site

Figure 6 shows the makespan of each algorithm with different numbers of workers at a site. *combined.2* per-

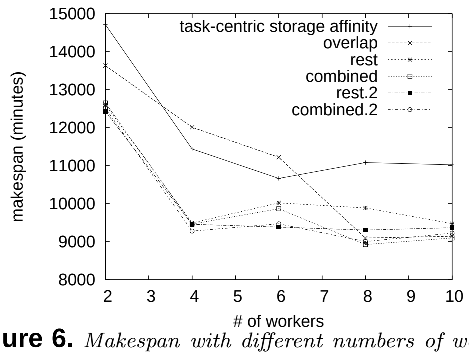

Figure 6. *Makespan with different numbers of workers at a site.*

Table 3. *Result of the* rest *metric at a site with 4 workers, 6 workers, and 8 workers. All numbers are averages. 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

forms the best mostly, which shows that minimizing file transfers as well as considering past references help 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! 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, Table 3 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 3) associated number of file transfers.

As is shown in Table 3, for the *rest* metric, 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 exhibit their worst makespan at different points.

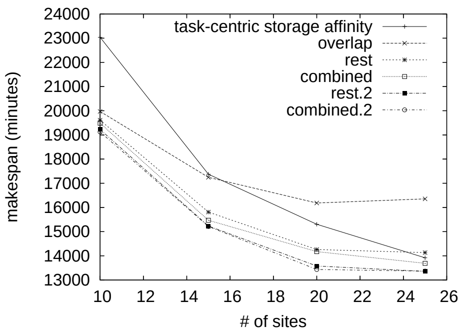

Figure 7. *Makespan with different numbers of sites*

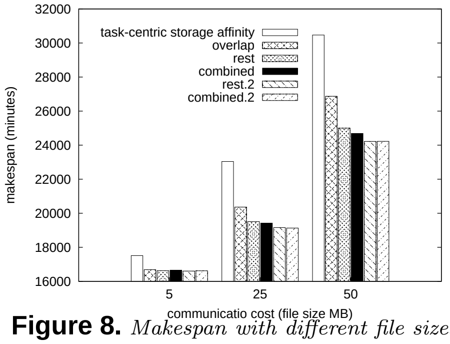

Figure 8. *Makespan with different file sizes*

5.6. Number of Sites

Figure 7 shows the makespan of each algorithm with different numbers of sites. Generally, 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 help to reduce the makespan. Randomized algorithms perform better than deterministic algorithms, which again shows that it avoids suboptimal scheduling decisions described in Section 4.3.

5.7. File Size

Figure 8 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 other scenarios shown before.

6. Related Work

Spatial Clustering[10] 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. Tasks in a cluster are then assigned to workers that sit on a same site. The great reduction in file transfers causes an important decrease in execution time. Two drawbacks to this approach are that (1) it cannot handle new jobs arriving asynchronously and (2) it is application specific.

Storage Affinity[14] 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. In this way it finds the task+site pair with largest data affinity (common bytes), which is chosen as the next schedule. 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[5] scheduling heuristic.

Decoupling data scheduling from task scheduling was proposed in [13]. 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. Note that execution time improves with the data replication schemes due to the data set popularity distribution, which was set to be geometric –not all grid applications will have the same characteristic.

A pull-based scheduler –that can also be categorized as worker-centric– is proposed in[16]. 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. The workers implement an iterative algorithm to improve the scheduler's buffer size estimation. This work completely ignores data transfer time, and requires knowledge of CPU speed and memory size in all workers.

7. Conclusion

We argued that worker-centric scheduling is more desirable than task-centric scheduling for data-intensive applications. We base our argument on two problems of task-centric scheduling, namely, unbalanced task assignments and premature scheduling decisions. We proposed various metrics 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 in all the scenarios we consider.

References

[1] Top 500 list. *http://www.top500.org*.

[2] Ian T. Foster et al. The Grid2003 Production Grid: Principles and Practice. In *Proc. of HPDC-13*, 2004.

[3] W. E. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. T. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Secure, efficient data transport and replica management for high-performance data-intensive computing. *CoRR*, cs.DC/0103022, 2001.

[4] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proc. of SC*, 2000.

[5] H. Casanova, D. Zagorodnov, F. Berman, and A. Legrand. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *9th Heterogeneous Computing Workshop*, 2000.

[6] W. Cirne, F. Brasileiro, J. Sauv, N. Andrade, D. Paranhos, E. Santos-Neto, and R. Medeiros. Grid Computing for Bag of Tasks Applications. In *Proc. Third IFIP I3E*, September 2003.

[7] D. P. da Silva, W. Cirne, and F. V. Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In *Proc. of Euro-Par 2003*, 2003.

[8] M. B. Doar. A Better Model for Generating Test Networks. In *Proc. of Globecom*, 1996.

[9] A. Legrand, L. Marchal, and H. Casanova. Scheduling Distributed Applications: the SimGrid Simulation Framework. In *Proc. of CCGrid*, 2003.

[10] L. Meyer, J. Annis, M. Mattoso, M. Wilde, and I. Foster. Planning Spatial Workflows to Optimize Grid Performance. *Technical Report, GriPhyN 2005-10*, 2005.

[11] S. Muir. Seven Deadly Sins of Distributed Systems. In *Proc. of WORLDS*, 2004.

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

[13] K. Ranganathan and I. T. Foster. Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications. In *Proc. of HPDC-11*, 2002.

[14] E. Santos-Neto, W. Cirne, F. V. Brasileiro, and A. Lima. Exploiting Replication and Data Reuse to Efficiently Schedule Data-Intensive Applications on Grids. In *Proc. of JSSPP*, 2004.

[15] V. Sekhri. Lessons Learned on Summer 04 Grid SDSS Coadd. *https://www.darkenergysurvey.org/the-project/simulations/sdss-grid-coadd/summer-04-grid-coadd.*

[16] S. Viswanathan, B. Veeravalli, D. Yu, and T. G. Robertazzi. Design and Analysis of a Dynamic Scheduling Strategy with Resource Estimation for Large-Scale Grid Systems. In *Proc. of GRID*, 2004.