Dynamic Co-Scheduling of Distributed Computation and Replication

Huadong Liu, Micah Beck, and Jian Huang

Abstract — **Ideas of scientific research occur spontaneously. To enable a group of collaborating but geographically separated researchers to effectively investigate and share their spontaneous hypotheses as well as research findings on the fly, we are interested in developing the infrastructural tools that allow a distributed data intensive computing environment to be shared in an interactive manner, as opposed to a batch mode of operation. However, without advanced reservation, it is difficult to assure a a certain level of performance on a large number of shared and heterogeneous servers. To achieve scalable parallel speedups in this scenario, especially with large distributed datasets, we must closely integrate the management of computation and runtime data movement. Unfortunately, such co-scheduling of computation and replication in the wide area is still not well understood in practice. In this paper, we first define the canonical scheduling problem for datasets distributed with k-way replication in the wide area. We then develop a dynamic co-scheduling algorithm that integrates the scheduling of computation and data movement. Using time-varying visualization as the driving application, we demonstrate, on 80 non-dedicated and heterogeneous nodes, that our co-scheduling approach improves not only application performance but also server utilization at a very reasonable cost.**

Index Terms — **Data replication, distributed computing, load balancing, scheduling, task farming, wide area storage**

I. Introduction

To facilitate today's collaborative scientific research, large datasets must frequently be shared among and analyzed by a team of geographically separated scientists, spread across the wide area network. Assuming similar performance, scientists would very likely prefer to use a system operated in an interactive manner, as opposed to a batch system. Scientists would then be able to efficiently explore and share spontaneous hypotheses or research ideas.

When designing infrastructural tools for widespread use by collaborating computational scientists, there are several possible alternatives. Among those, we would like to focus on an un-orchestrated distributed environment where computation and storage resources are not reserved beforehand, but, rather, co-scheduled at runtime. Using time-varying visualization of large datasets as a driving application, we hope to show that this approach provides a convenient utility to the user community.

There are several technical hurdles to achieving the above goal. For instance, in a distributed system composed of heterogeneous servers shared by a community without explicit orchestration, it would be hard to have all the involved servers produce a guaranteed level of performance. On such a shared distributed infrastructure, traditional parallel algorithms would need to be adapted with some special methods of scheduling. In this paper, we describe a systematic study of the co-scheduling algorithm that considers both computation and storage aspects simultaneously in a distributed system. We demonstrate, using our co-scheduling algorithm, that it is already feasible to visualize time-varying simulation datasets using a large number of un-reserved servers to achieve scalable performance as well as fault-tolerance. Most of these servers have heavy but dynamically varying workloads.

It is natural for parallel implementations to partition a dataset and the overall computing job accordingly. To improve throughput and reliability in case of individual failure in the wide area, the partitions are often replicated among *k* different servers, i.e. *k*-way replication [23]. Due to limited storage capacity, transport latency and maintenance overhead, the value of *k* is typically small. The programming paradigm of these parallel implementations is usually master-slave, also known as task-farming [26]. In the master-slave paradigm, the master is responsible for distributing tasks among a farm of slaves and collecting partial results. Each slave simply waits for a task from the master, computes the partial result and sends it back. Considering that most computations work on local data, tasks are only assigned to slaves that have the corresponding partitions. The master has to explicitly initiate any data movement between slaves before task assignment.

To get the best performance out of non-dedicated servers, dynamic management for computation and data replication has to be tightly coupled. Computation management involves the assignment of parallel tasks, while data replication management deals with data movement between selected servers. Both scheduling of computation and scheduling of replication aim at maximizing server utilization and minimizing application execution time. While scheduling of computation improves server utilization by distributing tasks intelligently to optimize load balancing among servers, scheduling of replication moves partitions around so that work assigned to each server is proportional to its performance as observed by the application.

Manuscript received November 30, 2005. This work was supported in part by NSF under Grant CNS-0437508.

Huadong Liu and Micah Beck (corresponding author) are with the Logistical Computing and Internetworking Laboratory, Department of Computer Science, University of Tennessee, Knoxville, TN, USA.

Jian Huang is with the SeeLab, Department of Computer Science, University of Tennessee, Knoxville, TN, USA.

Email: {hliu, mbeck, huangj}@cs.utk.edu

Many well-known middleware systems have been developed over the past few years to implement task-farming applications but scheduling strategies are still open research issues [12]. Many job scheduling and data management techniques have been proposed in the literature. However, few previous works have examined co-scheduling of computation and replication for operating on *k*-way replicated data in the wide area.

In this paper, we propose an integrated scheduling algorithm for both computation and replication. It adaptively measures server performance in terms of computation power and data transfer rate. This information is used to dynamically assign tasks to servers and direct data movements among them to achieve the best server utilization, minimizing application execution time. In addition, our co-scheduling algorithm is novel in runtime data movement schemes that use the deadline based partial download from multiple sources. User provided knowledge of the application such as computation complexity also contributes to an effective scheduling.

We have successfully run a large-scale volume visualization application on 80 distributed heterogeneous servers. Compared with the conventional work-queue scheduling algorithm, our co-scheduling algorithm improves both application execution time and server utilization by more than 30%. We note that none of the servers were reserved or under a controlled workload.

The remainder of the paper is organized as follows. Section II reviews related work in job scheduling and data replication. In Section III, we define the scheduling problem for wide area replicated datasets. In Section IV, we provide the details of our dynamic co-scheduling algorithm to solve the problem. Section V presents the experimental results. We conclude our work and point to future research directions in Section VI.

II. Related Work

Job scheduling in a dynamic, heterogeneous, distributed computing environment has been extensively studied [8, 12, 13, 15, 18, 20]. Casanova et al. [12] propose an adaptive scheduling algorithm for task farming applications. The algorithm adapts the length of the job queue to the underlying computing fabric according to constant computation throughput measurement. Desprez et al. [15] describes algorithms that compute an optimal placement of replicas prior to job execution. What distinguishes our work is that we consider dynamic data replication an important part of the scheduling problem.

Work on downloading wide area replicated data includes [4, 21]. Plank et al. [21] describe the progress-driven redundancy algorithm that uses the work-queue model to monitor the progress of each download and retry a download if it progresses too slowly. Allen et al. [4] proposes an alternative by using NWS [27] predictions to select the best server to download. We extend the scheduling of download to arbitrary computation and actively make fresh replicas at runtime.

The research most relevant to the algorithm presented in this paper is [14, 22]. Ranganathan et al. [22] evaluate several scheduling and replication strategies in a two-level scheduling framework. Chakrabarti et al. [14] propose the Integrated Replication and Scheduling Strategy to iteratively improve application performance. In stead of using simulations, we evaluate our algorithm in a real computing environment and prove that our result is close to the optimal.

III. Scheduling Problem for Replicated Datasets

In a distributed environment where shared resources cannot be brought under the control of a single global scheduler, the application must be scheduled by the user or by some middleware agent [9, 11]. For the latter case, the middleware agent itself can be viewed as a client. Figure 1 shows a typical structure of task parallel applications on shared datasets that are partitioned and replicated at distributed servers. We assume that every server is capable of handling both computation and data movement requests. Each user accesses and analyzes datasets independently without knowing activities of other users.

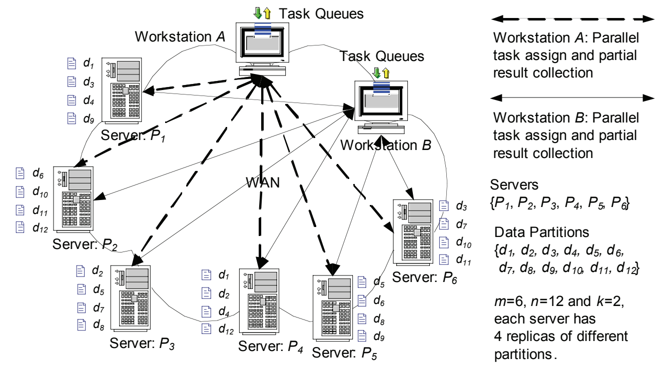

Task Queues
Workstation A
Task Queues
Workstation B
WAN
d1 d3 d4 d9
Server: P1
d6 d10 d11 d12
Server: P2
d2 d5 d7 d8
Server: P3
d1 d2 d4 d12
Server: P4
d5 d6 d8 d9
Server: P5
d3 d7 d10 d11
Server: P6
Workstation A: Parallel task assign and partial result collection
Workstation B: Parallel task assign and partial result collection
Servers {P1, P2, P3, P4, P5, P6}
Data Partitions {d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12}
m=6, n=12 and k=2, each server has 4 replicas of different partitions.

Figure 1. A typical structure of task parallel applications on replicated datasets

Before the discussion of various job scheduling algorithms, we define the scheduling problem for wide area replicated datasets. Suppose we have:

1. A collection of computational servers $\{P_1,P_2,\ldots,P_m\}$ where m is the number of servers. P_i is described by b_i and c_i. Bandwidth b_i represents the bandwidth between P_i and the client. Computational power c_i defines how fast a partition can be processed for an application. For convenience, both b_i and c_i are measured in megabytes per second. If a server has multiple processors (e.g. an SMP machine), c_i is the aggregation of all processors that can contribute to the computation. When several users contend for resources, b_i and c_i are a fraction of physical resources that are delivered to the user.
2. A large dataset that is partitioned into $\{d_1,d_2,\ldots, d_n\}$. n is the number of partitions and s_j is the size of d_j. We define $\delta_i^j = 1$ if d_j is on P_i, $\delta_i^j = 0$ otherwise. Partitions are distributed with k-way replication ($k \geq 1$), i.e. each partition is replicated on k out of m randomly selected servers. Formally, $\sum_{i=1}^{m} \delta_i^j = k$ for each partition d_j. As a result, each server has $n \times k/m$ partitions.
3. An application (e.g. parallel rendering) that is able to make use of the entire collection of partitions in parallel. Thus, we have a set of independent computational tasks $\{T_1,T_2,\ldots,T_n\}$. We assume that d_j is the only partition required by task T_j. We further assume that execution time

of T_j is proportional to $f(s_j)$ and the output size of T_j is $g(s_j)$. $f(x)$ is known as the complexity function and $g(x)$ is often constant or linear. They are application specific and usually required for an effective application level scheduling [9]. If $\delta_i^j = 1$ and T_j is assigned to P_i, the time required to complete T_j can be formulated as $f(s_j)/c_i + g(s_j)/b_i$. $f(s_j)/c_i$ is the time required for computation and $g(s_j)/b_i$ is the time spent on communication. Since the required partition already resides on the target server when a task is assigned, we assume that communication time is solely the time to receive the output. Although many effective techniques such as pipelining can be employed to overlap computation and communication between successive tasks, we assume that they are not overlapped in our model.

4. (Optional) A set of data movement tasks M_{ij} that makes a fresh copy of d_j on P_i. To exploit the fact that there are multiple replicas of d_j, data is downloaded from multiple sources. Thus, the time required to perform M_{ij} is $s_j / (\sum_{r=1}^{m} b_{ir} \times \delta_r^j)$, where b_{ir} represents the bandwidth between P_i and P_r. $\sum_{r=1}^{m} b_{ir} \times \delta_r^j$ is the aggregate bandwidth to P_i from all sources that have d_j.

To mitigate resource contention on shared servers with heavy load, we assign at most one computational task to a server for each application at a time. Data movement tasks can co-exist with a computational task because a good mix of CPU-bound and I/O-bound processes can actually improve system throughput. However, due to process scheduling, too many concurrent data movement tasks can slow down computational tasks, especially in a non-dedicated system. Thus, the number of active data movement tasks per application at each server is also set to be one. The number of simultaneous downloads could be k because we have k replicas.

Suppose each server P_i runs for time t_i and all servers start at the same time, then the execution time of an application would be $\max_{i=1}^{m} t_i$, which is the time required for the last server to finish its assigned tasks. For a given application, the shortest execution time occurs when all servers can be kept doing useful work and they all finish roughly at the same time.

Given that the dataset is replicated throughout a wide area network, does there exist a scheduling of computation and data movement tasks such that the execution time of an application over the entire partitions is minimal? This is the scheduling problem of replicated datasets we will explore in this paper. Formally, let $\sigma_i^j = 1$ denote that task T_j is assigned to server P_i. A schedule is a set of $\sigma_i^j \in \{0,1\}$, $i \in [1...m]$ and $j \in [1...n]$, such that $\forall j \sum_{i=1}^{m} \sigma_i^j \geq 1$. $\forall j \sum_{i=1}^{m} \sigma_i^j \geq 1$ mandates that each task T_j must be assigned to at least one server. If $\delta_i^j = 1$ and $\sigma_i^j = 1$, T_j can be immediately assigned to P_i as long as there is no other active task on P_i. However, if $\delta_i^j = 0$ and $\sigma_i^j = 1$, a copy of d_j must be moved to P_i before T_j can be assigned to P_i. Note that, $\delta_i^j = 1$ does not necessarily imply $\sigma_i^j = 1$ because we have r replicas to choose from and a fresh replica can be made at runtime when necessary. The best schedule should satisfy that $\max_{i=1}^{m} \sum_{j=1}^{n} \sigma_i^j \times (f(s_j)/c_i + g(s_j)/b_i)$ is minimal assuming that no fault happens after the schedule is made. For the last server, time spent on explicit data movement completely overlaps with computation, thus it is not included in the formula.

IV. Co-Scheduling of Computation and Replication

Shared datasets are typically replicated in a heterogeneous environment and accessed by geographically distributed users with competing goals. As a result, resource performance varies over time and is hard to predict. Experience with distributed applications indicates that adaptability is fundamental to achieving application performance in dynamic environments [8]. It is imperative for us to employ heuristics and dynamic load balancing to obtain a good approximation of the scheduling problem, while addressing fault-tolerance at the same time. We will first discuss the conventional work-queue scheduling of parallel tasks. Based on that, we present the co-scheduling algorithm in two steps: adaptive scheduling of computation and dynamic scheduling of replication.

A. Work-queue Scheduling

Work-queue scheduling [17] is a variation of the master-slave model. The master maintains a work queue and assigns tasks to available slaves. Each slave works on a task independently. On completion, it notifies the master that it is ready to receive the next task. As an alternative, the master can poll each slave periodically to see whether it can dispatch another unfinished task. In contrast to static scheduling [13] in which tasks are allocated to slaves before the application is started, work-queue scheduling attempts to deal with variability in resource performance and individual task workload by deferring task assignment decisions for as long as possible [24]. In work-queue scheduling, tasks are not distributed to slaves until they have finished a previously assigned task. In this way, fast slaves tend to deliver more tasks than slow slaves over time.

Algorithm 1 illustrates a scheduling of parallel tasks over replicated datasets by applying the work-queue scheduling. To avoid data movement, tasks are only assigned to servers that have the required partitions, i.e. task T_j is assigned to P_i only if $\delta_i^j = 1$.

Algorithm 1: Work-queue scheduling over replicated datasets

```
1  while not IsEmpty($Q$)
2      foreach available server $P_i$ do
3          DeQueue($T_f$, $Q$),  $T_f$ has been finished by $P_i$
4          $T_j$=GrabTask($P_i$, $Q$), $d_j$ is on $P_i$
5          AssignTask($T_j$, $P_i$)
```

In GrabTask, unassigned tasks have higher priority than assigned tasks. When there is no unassigned task that a fast server can do, it will try to help slow servers on assigned tasks whose partitions it holds. The algorithm is straightforward and theoretical work has proved that work-queue scheduling yields a good approximate solution to scheduling problems [19].

Even though it is very adaptive, the above algorithm ignores the fact that distributed servers have very diverse performance, which has two potential consequences. First, each task is performed by one server unless the server fails, the client scheduler times out or other servers that have the corresponding partition have no more tasks to do. If one server lags, the overall application cannot progress if the application (e.g. streaming, interactive visualization, etc.) needs ordered partial results. Second, each available server always picks the first unfinished task that it can do, which in some cases might be performed by a faster server. In this case, slow servers "steal" work from fast servers. When all candidate tasks on fast servers get depleted, they have to stop while slow servers still need to finish the tasks for which they hold the corresponding partitions.

Thus, there is a need for more sophisticated scheduling techniques that can perform adaptive resource selection and on-demand data movement. The following subsections describe our approach for designing and implementing such techniques.

B. *Adaptive Scheduling of Computation*

Our approach depends on discovering fast servers on the fly, assigning as many tasks to them as possible, and avoiding being stalled by slow or faulty servers. We devised three generic mechanisms for this purpose: (i) a dynamically ranked pool of servers, (ii) a two level priority queue of tasks and (iii) a competition avoidant task assignment scheme. This framework is very generic and can be applied to other distributed computing applications in general.

Each server P_i is ranked by its estimated time t_u^i to process a task of unit size u (e.g. 10MBytes). This measurement roughly reflects performance of the server delivered to an application. The less time a server needs to process the unit task, the higher rank this server has. Recall that $t_u^i = f(u)/c_i + g(u)/b_i$. Rather than a simple average, c_i is calculated from $c_i'+\rho\times(\tau-c_i')$, where c_i' is the previous value of c_i and τ is the most recent value. Similarly, $b_i=b_i'+\rho\times(\beta-b_i')$, where b_i' is the previous value of b_i and β is the most recent value of b_i. The parameter ρ is borrowed from machine learning [5]. It determines the influence of previous values, with the influence of outdated values tending towards zero over time. This technique causes the client to continuously adapt to the constantly changing resource performance [20].

When a server finishes a task, it returns the computation time t_c and the output. The client records the time t_s when it starts to receive the output and the time t_r when it finishes. With t_c, t_s and t_r, τ and β are formulated as $f(s_j)/t_c$ and $g(s_j)/(t_r-t_s)$ respectively. Note that both t_r and t_s are obtained from the local time service at the client. Although we can get a more accurate β by using the time that the server starts to send back the output, it requires time on both the client and servers to be closely synchronized, which is not very practical in a large distributed system.

A two-level priority queue maintains unfinished tasks. The higher priority queue (HPQ) contains tasks that are ready to be assigned and the lower priority queue (LPQ) contains tasks that have been assigned to one or more servers but not finished. If a task is assigned to an idle server, it is moved from HPQ to LPQ. Initially, only the first w tasks $\{T_1,T_2,\ldots,T_w\}$are placed in HPQ and task $T_x(x>w)$ can not be added until task T_{x-w} has been finished, where w is the size of the task window (TW). w controls how far out of order tasks can be finished. For example, if w=1, all tasks will be completed in order. In contrast, if w=n, every task is allowed to be finished out of order. In most cases, w is greater than m so that every server can contribute to the application. Figure 2 shows a snapshot of tasks in the two-level priority queue on the dataset as illustrated in Figure 1. The task window cannot move forward at this moment because server P_1 is still working on task T_3, which is the head of TW.

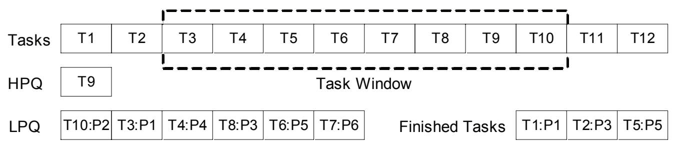


Figure 2: A snapshot of the two-level priority queue

Each task T_j in HPQ is keyed by $\min_{i=1}^{m} t_u^i \times \delta_i^j$, which is the minimum unit task process time t_u^i of all servers currently having partition d_j. This priority ranks new tasks by their likelihood to be finished by a fast server in terms of computational power and available bandwidth. Assume a task T_j in LPQ has been assigned to P_i. T_j is keyed by its estimated waiting time, which is the estimated execution time E_j=$f(s_j)/c_i$+$g(s_j)/b_i$ minus the time elapsed since start. E_j is static during task execution because b_i and c_i will not be updated until the task is completed. This priority ranks assigned tasks by its likelihood to finish soon. The client can dynamically sleep the minimum estimated waiting time to avoid busy wait. Tasks in both HPQ and LPQ are sorted by their keys in non-increasing order.

When the parallel computation starts, the client sequentially assigns each available server the first task in HPQ that it is able to perform, moving the task to LPQ. When P_i completes task T_j, b_i and c_i are updated. T_j is removed from LPQ also. If T_j is the first task in the task window, TW is moved forward by one, adding one more task to HPQ. Since b_i and c_i are adjusted, HPQ is resorted by the latest t_u^i as well. Then, there are three possible scenarios: (1) both HPQ and LPQ are empty, (2) there are unassigned tasks in HPQ, (3) there are unfinished tasks in LPQ. Case 1 signifies the completion of scheduling. In Case 2, every available server will be directly assigned the first task in HPQ that it can handle. In this way, slow servers do not compete for tasks with fast servers so that fast servers can be assigned as many tasks as possible. In Case 3, we would like unfinished tasks to be computed by additional servers (up to k-1, k is the number of replicas for each partition), which work in parallel with the server originally assigned for an unfinished task. These servers compete to finish the same task. Again, we assign the first task in LPQ to an available server that holds the required partition. This is the slowest task among all unfinished tasks that the server can help. If any of the duplicated tasks is done, others are aborted immediately.

C. Dynamic Scheduling of Replication

So far, our scheduling algorithm makes use of execution history to allocate tasks so that slow severs do not compete with fast servers for tasks. Fast servers can further help slow servers by repeating tasks on replicas. However, data placement is still static, i.e. there is no active data movement in the process of computing. There is the possibility that some partitions only reside on a set of slow servers. In that case, fast servers cannot help slow servers because they do not have the required partitions to work on.

One natural thought is to move partitions to fast servers before they become idle. In order to make sure that time spent on data movement does not exceed the profit that we gain from migrating the task, bandwidth information between servers needs to be acquired. However, this needs non-trivial setup and management of bandwidth estimation or prediction tools [16, 27]. Also, the information obtained is not always up to date. Instead of using existing tools to insert extra test traffic into the network and query for available bandwidth, we devised a partial download scheme with deadline for data movement between computational servers.

As parallel computation proceeds, the scheduler actively monitors tasks in HPQ that each server can perform. The maximum amount of work in HPQ a server P_i can contribute is $W_i = \sum_{j=1}^{n}(f(s_j)+g(s_j))$, $T_j \in$HPQ and $\delta_i^j = 1$. Recall that the shortest execution time occurs when all servers finish roughly at the same time. We calculate P_i's share of the unassigned tasks in HPQ, $S_i = S \times t_u^i / \sum_{j=1}^{m} t_u^j$. $S = \sum_{j=1}^{n}(f(s_j)+g(s_j))$, $T_j \in$HPQ, is the total amount of work left in HPQ. $t_u^i / \sum_{j=1}^{n} t_u^j$ is P_i's proportion of the unassigned work, according to its observed performance. Since the speeds of data processing and data transmission for each server are different, both W_i and S_i are rough estimations.

Once $W_i<S_i$, the scheduler tries to initiate a data movement task M_{ij}, moving data blocks from servers that have d_j to P_i. Since partitions are replicated and W_i increases with k, there is the possibility that no data movement is necessary at all ($W_i \geq S_i$). The scheduler starts from the first task in HPQ, which has the least likelihood to be finished by a fast server. To avoid always moving partitions out of the same set of slow servers, the task T_j should satisfy that sum of W_i of all servers that have d_j is above their aggregate share (the sum of all corresponding S_i). If this condition cannot be satisfied, the scheduler will try the next task in HPQ. Once a task is marked for data migration, the scheduler will skip it and assign the next task to an available server.

Before sending and receiving bits over the network, the profitability analysis is invoked to estimate the maximum data transfer time allowed, the deadline. For example, suppose T_8 has been picked to be migrated to P_1. Also assume that T_8 can also be performed by P_2 and P_3. We move d_8 only if $min(F_2, F_3) > (T_m+T_c)$, where T_m is the time for data movement, T_c is the time to compute T_8 on P_1, F_2 and F_3 are the time required to complete all remaining tasks (including T_8) on P_2 and P_3 respectively. The deadline of M_{18} is set to be $min(F_2, F_3)-T_c$.

After the deadline is calculated, the data movement task M_{ij} starts. We do not try to transfer the complete partition from the beginning. Instead, we try a small fraction p of the partition and see if it can be finished in p of the deadline. p is configured at runtime so that dynamics such as TCP slow start can be avoided. If the fractional transfer completes in p of the deadline, we proceed to move the rest of the data, otherwise, M_{ij} is aborted. Since the partition is replicated on k servers, the destination server takes advantage of downloading data from multiple sources by using the progressive driven redundancy algorithm [21]. When M_{ij} is done, key of T_j in HPQ is updated because a fast server can now work on it. Figure 3 illustrates the process of replication scheduling.

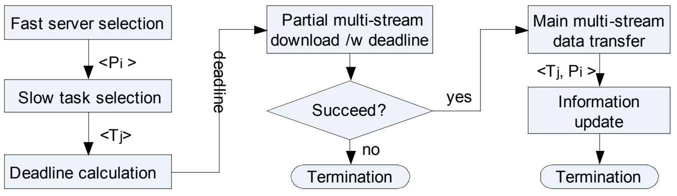


Figure 3: The process of dynamic scheduling of replication

V. Experiments

To investigate performance of our co-scheduling algorithm, we compare the wall clock execution time and server utilization with those of the basic work-queue scheduling and the adaptive scheduling of computation but without replication. Server utilization measures the efficiency of n servers allocated for an application. It is defined as the ratio of the time that n servers spent on doing useful work to the time those servers would be able to do useful work [18]. We run a massively parallel visualization application on 80 non-dedicated servers over the National Logistical Networking Testbed (NLNT) and PlanetLab [2]. Although these nodes are server-class machines, they are shared among a large user community. PlanetLab nodes are even virtualized as "slices" to enable large scale sharing. Loads on these nodes differ dramatically and vary over time. Figure 4 shows a snapshot of the one-minute load of 415 PlanetLab nodes starting from 15:50pm on Nov.16, 2005 on the left and the one-minute load of `pl1.cs.duke.edu` in 24 hours on the same day on the right. Load of the `duke` node is sampled every five minutes. The server was unavailable during 18:10 to 19:55, which happens frequently in a large distributed system. The raw data is gathered by the PlanetLab CoMon service [1].

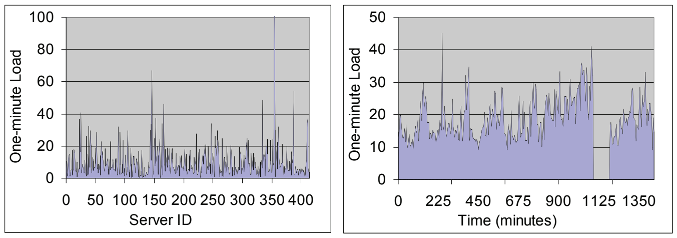
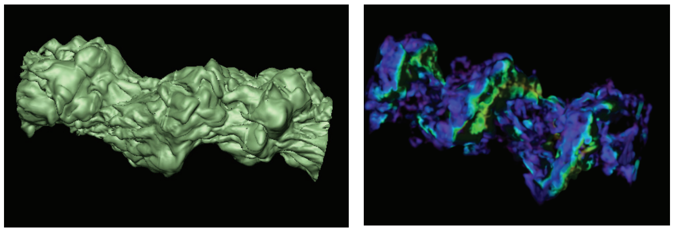


Figure 4: One-minute load of PlanetLab nodes

The visualization application does isosurface extraction and volume rendering on a time-varying dataset simulating a Jet shockwave with 100 time steps. The spatial resolution of each time step in the Jet dataset is 256×256×256. As a common practice in visualization, we compute the first derivative of the volume and store it with the scalar volume to accelerate the visualization process. This is necessary whether to compute per vertex normal on an extracted isosurface or volume render with shading effects. Every time step is partitioned into 8 partitions with spatial resolution 128×128×128 of 8.4MB in storage. There are in total 800 partitions, covering 100 time steps. Total size of the entire dataset is 6.7GB. These partitions are uploaded and augmented with *k* copies evenly on all participating servers. For example, using *k*=2, per-server storage is roughly 800×2×8.4/80=168MB. After all partitions are staged into the network, isosurface extraction or volumes rendering computations are spawned in parallel on distributed servers. We provide two sample images from the test runs in Figure 5. Note that volume rendering does a high quality image reconstruction, which consumes more CPU cycles than isosurface extraction for the Jet dataset.



(a) Isosurface extraction (b) Volume rendering

Figure 5: Two sample images of the Jet dataset

We setup NFU-enabled IBP depots on 80 randomly selected servers across North America from NLNT and PlanetLab. Most of them are PlanetLab nodes. The choice of IBP (Internet Backplane Protocol) is motivated by the integrated storage and computation service it provides and the authors' experience with that system. IBP implements a generic, best effort network storage service that can scale globally [6]. IBP storage is managed by servers called "depots", on which clients perform remote storage operations. IBP clients view a depot's storage resources as a collection of byte arrays. Clients initially obtain the use of a byte array by making a storage allocation on a depot.

The NFU (Network Functional Unit) is an extension to IBP, providing data transformation services for bytes stored in IBP allocations [7]. NFU operations are either static or dynamic. Static NFU operations are compiled and linked as part of an IBP depot. In contrast, dynamic NFU operations are mobile code that is executed or interpreted in a sandbox by a particular static NFU operation. The code that defines a dynamic NFU operation is stored in an IBP allocation and passed to the appropriate static operation as an argument. In our tests, both the isosurface extraction and volume rendering operations are deployed as dynamic operations.

In Figure 6 and Figure 7, we compare execution time and server utilization for volume rendering and isosurface extraction with *k*=2 and *k*=3 respectively. In both figures, i is the basic work-queue scheduling, ii is the adaptive scheduling of computation and iii is the co-scheduling of computation and replication. To maximize the differences, we use the maximum window size *w*=800. With each particular combination, 8 tests were run and only the average is reported. We note here that none of the servers were reserved or running with a controlled workload using the PlanetLab Sirius service [3]. Since conditions might change between one execution and the next due to resource contention, we ran one instance of each of the three scheduling algorithms back-to-back hoping that all three executions would enjoy similar conditions on average.

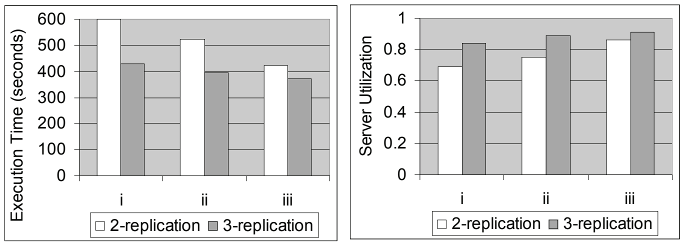


Figure 6: Performance of volume rendering with *w*=800

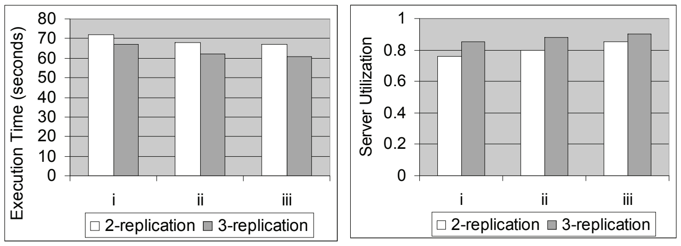


Figure 7: Performance of Isosurface extraction with *w*=800

In general, increasing the number of replicas, *k*, increases storage overhead on each server and consumes more network bandwidth when copying partitions between IBP depots during the data staging phase. Both isosurface extraction and volume rendering have shorter execution time and higher server utilization with 3-replication than with 2-replication for all the three scheduling algorithms. With a larger *k*, both fast and slow servers have more candidate partitions to work on, thus fast servers have more chances to help slow servers.

For the heavyweight volume rendering with *k*=2 and *w*=800, on average, the co-scheduling algorithm reduces execution time by 31% and increases server utilization by 32% at the cost of moving 56 partitions from the slow servers to fast servers, compared with the basic work-queue scheduling. For the lightweight isosurface extraction, in most cases, the cost of moving a partition out of *k* slow servers exceeds the profit gained from transferring the task to a fast server. We only see a slight improvement of execution time and server utilization for the co-scheduling algorithm over the adaptive scheduling of computation with k=2 and w=800 because of the overhead of vainly trying the deadline based data movement.

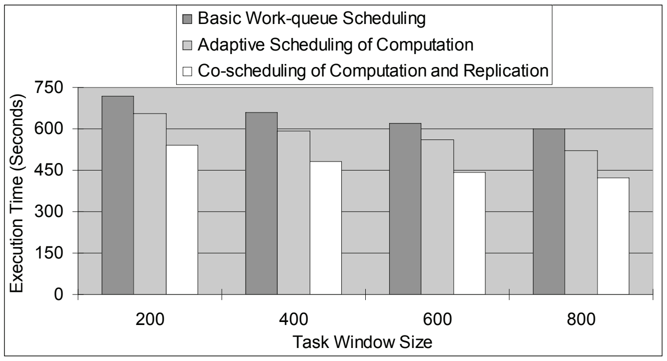


Figure 8: Execution time of volume rendering with different w

Size of the task window w also has a similar effect to execution time as k does. We plot execution time of volume rendering with different w for k=2 in Figure 8. By increasing w, the amount of duplicated tasks is reduced and the work completed by each server gets more proportional to its performance. For instance, suppose we have 4 tasks of unit size. They are replicated with k=2 on server P_1 and P_2. P_1 can finish a task in 30 seconds and P_2 can finish a task in 10 seconds. Initially, task T_1 was assigned to P_1 and task T_2 was assigned to P_2. If w is set to 2, then P_2 has to help P_1 after it finishes T_2. Thus, the number of duplicated tasks is 2 and total execution time is 40 seconds. In contrast, with w=4, P_2 can proceed to work on T_3 and T_4 without helping P_1. The number of duplicated tasks would be 0 and total execution time is 30 seconds.

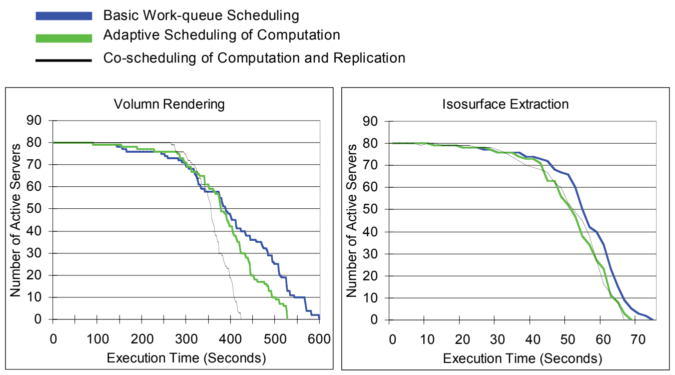


Figure 9: The variation of number of active servers over the time span of a typical run with k=2 and w=800

To better illustrate the dynamics of load balancing between the three scheduling algorithms, we plot the number of active servers during a typical execution using k=2 and w=800 in Figure 9. Server utilization is calculated as the area covered by the curve divided by the area of the bounding rectangle. Initially, all servers work on one of its 800×2/80=20 partitions. As tasks on a particular server are completed, the choice of the next task for this server becomes constrained. For volume rendering, in the basic work-queue scheduling, when tasks on the faster servers are eventually depleted, the slower servers still need to finish the tasks for which they hold the corresponding partitions. This explains why the basic work-queue scheduling has the least server utilization. Adaptive scheduling of computations improves server utilization by optimizing the task assignment process so that the fast servers can be assigned as many tasks as possible. With co-scheduling of computation and replication, fast servers are kept busy by moving extra tasks to them from slow servers.

For isosurface extraction, server utilizations with the three scheduling algorithms are better than their counterparts in volume rendering and they do not have too much difference. This has to do with process scheduling on servers that have a high average load, i.e. a large number of active processes are waiting in the ready queue for execution. Most operating systems schedule process execution by priority [25]. Linux (installed on all PlanetLab servers and more than 50% of NLNT servers) process scheduler keeps track of process execution and adjusts their priorities dynamically. Processes are assigned the highest priority initially. They are penalized by decreasing their priority for running a long time. Correspondingly, they are boosted by increasing their priority if they have been denied the use of the CPU for a long time [10]. Remember that the process doing isosurface extraction needs less CPU cycles. Thus, it is more likely to have a higher average priority than process doing volume rendering. As a result, servers that have high load tend to look faster when running lightweight computations than when running heavyweight computations. When all servers perform similarly fast, the system tends to have higher server utilization.

Understanding the relative performance between the three scheduling algorithms, we are further interested in knowing how close is the execution time obtained from the co-scheduling algorithm to the real optimal execution time. In order to calculate the optimal execution time, we need to find out the optimal schedule first. However, it is extremely difficult to figure out the optimal assignment of tasks, even if we know the performance of servers. Since each task must be assigned to one of the k servers that have the corresponding partition, there are k^{800} possible schedules in total. When data movement is considered, the scheduling problem is more complex.

Fortunately, tasks in our tests roughly have the same size. To obtain an estimation of the optimal execution time, we log the time taken for each server to complete a task and compute the average ($\overline{t_i}$) when all tasks are finished. Ideally, the optimal execution time occurs when all servers stop at the same time. We calculate the "super optimal" execution time as $800\times(1/\overline{t_x})/(\sum_{i=1}^{80}1/\overline{t_i})\times\overline{t_x}$ where $800\times(1/\overline{t_x})/(\sum_{i=1}^{80}1/\overline{t_i})$ is the number of tasks assigned to server P_x according to its performance. It does not matter which server's average task completion time is chosen for the calculation because all servers finish at the same time. We call it "super optimal" because $800\times(1/\overline{t_x})/(\sum_{i=1}^{80}1/\overline{t_i})$ is usually a fractional number, which is not true in real task assignment. Thus, we also calculate the "close optimal" execution time by rounding the number of tasks that each server is assigned. The execution time is formulated as $\max_{x=1}^{80} round(800 \times (1/\overline{t}_x)/(\sum_{i=1}^{80} 1/\overline{t}_i)) \times \overline{t}_x$. Execution time of the optimal scheduling should be somewhere between the "super optimal" and "close optimal". Using the co-scheduling algorithm, the average execution time of volume rendering with *k*=2 and *w*=800 is 1.07 times of the "close optimal" value and 1.16 times of the "super optimal" value, which we consider very close to the optimal execution time.

Knowing that it is not a rigorous comparison, but only to provide context, we note that the same isosurface extraction takes 22 minutes on one dedicated 2.8GHz P4 processor, whereas the volume rendering takes 1 hour and 19 minutes on the same processor.

VI. Conclusions and Future Work

In this paper, we have investigated the problem of scheduling jobs and data movement in a distributed environment with the goal of maximizing server utilization and minimizing application execution time. Toward this goal, we developed a dynamic co-scheduling algorithm that integrates the placement of jobs and data replication for wide area shared datasets.

We ran a large-scale volume visualization application on 80 distributed and heterogeneous servers to evaluate the co-scheduling algorithm. We came to the conclusion that even with a small number of replicas, the co-scheduling algorithm greatly improves both server utilization and application performance for computation intensive applications. We also demonstrated that the degree of data replication and size of the task window can affect performance of the algorithm.

In the future work, we plan to further experiment with a cutting edge 3TB supernova simulation dataset under multiple user access patterns to study how to optimize the original data distribution by utilizing new replicas made during previous executions. We also want to address whether the strict limit of one computational task per server can be loosened without causing conflicts between competing schedulers.

Acknowledgment

The authors would like to thank Yun Zhang for valuable discussion and feedback. The authors also acknowledge the PlanetLab project headed by Larry Peterson for depot access.

References

[1] CoMon: A Monitoring Infrastructure for PlanetLab. `http://comon.cs.princeton.edu/`.

[2] PlanetLab. `http://www.planet-lab.org/`.

[3] PlanetLab: Sirius Scheduler Service. `http://snowball.cs.uga.edu/~dkl/pslogin.php/`.

[4] M. S. Allen and R. Wolski. The livny and plank-beck problems: Studies in data movement on the computational grid. In *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 43, Washington, DC, USA, 2003. IEEE Computer Society.

[5] C. G. Atkeson, A. W. Moore, and S. Schaal. Locally weighted learning. *Artificial Intelligence Review*, 11(1-5):11-73, 1997

[6] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable network storage. In *SIGCOMM '02*, Pittsburgh, August 2002.

[7] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable programmable networking. *Computer Communication Review*, 33(4):328-339, 2003.

[8] F. D. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the grid using apples. *IEEE Trans. on Parallel and Distributed Systems*, 14(4):369-382, 2003.

[9] F. D. Berman, R.Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 39, 1996.

[10] D. Bovet and M. Cesati. *Understanding the Linux Kernel, Second Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[11] H. Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. Technical Report CS-96-328, Knoxville, TN 37996, USA, 1996.

[12] H. Casanova, M. Kim, J. S. Plank, and J. Dongarra. Adaptive scheduling for task farming with grid middleware. *International Journal of High Performance Computing*, 13(3):231-240, Fall 1999.

[13] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141-154, 1988.

[14] A. Chakrabarti, R. A. Dheepak, and S. Sengupta. Integration of scheduling and replication in data grids. In *International Conference on High Performance Computing (HiPC)*, December 2004.

[15] F. Desprez and A. Vernois. Simultaneous scheduling of replication and computation for data-intensive applications on the grid. Technical Report RR2005-01, Lyon, France, January.

[16] C. Dovrolis, P. Ramanathan, and D. Moore. Packet-dispersion techniques and a capacity-estimation methodology. *IEEE/ACM Transactions on Networking*, 12(6):963-977, 2004.

[17] T. Hagerup. Allocating independent tasks to parallel processors: an experimental study. *Journal of Parallel and Distributed Computing*, 47(2):185-197, 1997.

[18] E. Heymann, M. A. Senar, E. Luque, and M. Livny. Adaptive scheduling for master-worker applications on the computational grid. In *GRID '00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, pages 214-227, London, UK, 2000. Springer-Verlag.

[19] D. S. Hochbaum, editor. *Approximation algorithms for NP-hard problems*. PWS Publishing Co., Boston, MA, USA, 1997.

[20] A. Page, T. Keane, and T. J. Naughton. Adaptive scheduling across a distributed computation platform. In John P. Morrisson, editor, *Third International Symposium on Parallel and Distributed Computing*, pages 141-149, Cork, Ireland, July 2004. IEEE Computer Society.

[21] J. S. Plank, S. Atchley, Y. Ding, and M. Beck. Algorithms for high performance, wide-area distributed file downloads. *Parallel Processing Letters*, 13(2):207-224, June 2003.

[22] K. Ranganathan and I. T. Foster. Simulation studies of computation and data scheduling algorithms for data grids. *Journal of Grid Computing*, 1(1):53-62, 2003.

[23] R. Samanta, T. Funkhouser, and K. Li. Parallel rendering with k-way replication. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 75-84, Piscataway, NJ, USA, 2001. IEEE Press.

[24] G. Shao, R. Wolskiy, and F. D. Berman. Performance effects of scheduling strategies for master/slave distributed applications. Technical report, University of California, San Diego, 1998.

[25] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 2001.

[26] L. Silva and R. Buyya. *High Performance Cluster Computing: Programming and Applications*, volume 2, chapter Parallel Programming Models and Paradigms. Prentice Hall, NJ, USA, 1999.

[27] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5-6):757-768, 1999.