

Automatic Data Placement and Replication in Grids *

Ying Ding, Ying Lu
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE 68588-0115
{yding, ylu}@cse.unl.edu

Abstract

Data grids provide geographically distributed storage and computing resources for large data-intensive applications. To support fast data access and processing, effective data management is essential. In this paper, we propose two novel algorithms on data placement and data replication. We evaluate the algorithms through intensive simulations. Results show that the new algorithms achieve significant improvement in facilitating load balancing, reducing job response time, and saving network bandwidth.

1 Introduction

In recent years, data-intensive scientific computations have become quite common in many disciplines such as bioinformatics, high energy, nuclear and particle physics. These applications require extreme-scale storage and computing resources. For example, the Compact Muon Solenoid (CMS) [26], a particle physics experiment associated with the Large Hadron Collider (LHC) [12] at CERN, has a long-term need to perform large-scale simulations and analysis. The CMS detector records physical events at a rate around 100 Hz, producing 100 MB raw data every second, i.e., several PB (petabytes) raw data every year. These periodically generated data need to be stored, accessed and analyzed by thousands of physicists around the world. To facilitate such experiments, data grids comprised of large storage and computing resources have been developed.

To achieve efficient data management in grids, two important issues need to be addressed: data placement and data replication. In a data grid, new data is often generated periodically. They need to be stored in proper sites of the grid for future processing. Currently, such data placement decisions are made either randomly or manually per user requests. However, without a careful control on the data placement, a site could get overloaded easily if it stores too

many popular data sets.

Effective data placement is complicated due to a number of factors. Data sets have different characteristics. They vary in size and popularity, where different data occupies a different amount of storage and is processed by a different number of jobs. As a result, storage and computing resources required by each data set are different. An intelligent data placement algorithm must take data heterogeneity into account. In addition, cluster sites of a data grid are often heterogeneous, meaning that they provide different amounts of storage and computing resources. These factors make it hard for users to decide how to place data properly. Therefore, we need a data placement algorithm that provides grid users automatic and intelligent data placement.

In current data grids, most data replication tools [23, 24, 7] focus on providing fast data transfer. These tools rely on users to decide where and what data to replicate. Again, due to the scale and complexity of the grids, it is prohibitively difficult to make good replication decisions manually. We need automatic data replication services in grids. As the size of a data grid increases, so does its dynamics. Resources can fail, be removed from or added to the grid frequently, causing capacity uncertainties. In addition, workloads have varied characteristics. Therefore, the data replication service has to be responsive and adaptive to such dynamic changes. Intensive research has been conducted on developing data replication algorithms. However, for most existing approaches, the performance adaptations are not self-tuned. They are designed and developed based on intuitions and are configured manually by experiments and trial-and-error methods [20, 21, 6, 4, 28, 15]. Upon system changes, if not reconfigured properly, their performance could become unpredictable. Therefore, we must investigate new approaches for self-tuning data replications.

In this paper, we address the aforementioned challenges in data grids. Particularly, we propose a data placement algorithm and a self-tuning data replication algorithm to facilitate load balancing in grids. The remaining of the paper is organized as follows. In Section 2, related work is presented. We describe the system model in Section 3 and the

*This work is funded by NSF grant No. 0720810.

proposed algorithms in Section 4. Section 5 discusses the simulations and we conclude the paper in Section 6.

2 Related Work

Most existing data management tools focus on providing fast data transfer. Examples include Phedex [23] for CMS, LDR [24] for LIGO (Laser Interferometer Gravitational Wave Observatory) [18] and DRS [7] for the Globus project [1]. These tools rely on users to decide what data to replicate and where to put replicas. However, as a grid scales up and its complexity increases, it becomes increasingly important that we have automatic data management services.

Previously, the replica placement has been defined as a file assignment problem (FAP) [10, 17], which aims to optimize a performance parameter (e.g., file transfer cost or response time) of a network system while satisfying the storage capacity constraint. Eswaran [11] has proved that the FAP is an NP-complete problem. Chu [8] formulates the problem as a nonlinear integer zero-one programming problem and solves the FAP to minimize the storage and transmission cost.

Liu et al. propose data placement algorithms in [29, 16]. Their work focuses on a tree topology in which requests can only be forwarded upwards towards the root node site. In a real-world data grid, the requests served on a site can be generated anywhere. Abawayjy [2] proposes a proportional share replica policy to place replicas in a grid with a tree topology. The goal is to make each replica serve approximately an equal number of requests. The results show significant improvement in system performance. However, this work only considers an ideal case where storage is sufficient and assumes all sites are homogenous. In this paper, we focus on a more general grid topology and take the heterogeneities of storage and computing resources into account.

Many researchers have investigated the data replication problem in grids, such as the work reported in [7, 6, 25, 28, 19]. Among the existing work, the most closely related is the research done by Ranganathan and Foster in [20, 21, 22], where the authors study the relationship between asynchronous data replication and job scheduling. The paper concludes that scheduling jobs where data sets are present performs the best, and then proposes several data replication algorithms to balance the system load.

The aforementioned algorithms are, however, threshold-based, where a data set with an access rate higher than a fixed threshold is deemed as popular and gets replicated from a heavily loaded site to another site. Although such algorithms were shown to improve performance in proto-

type systems, a limitation makes them difficult to be applied in real-world grid environments. It is a big challenge to set the popular data set threshold right. Too high a threshold will lead to insufficient data replications and result in a still unbalanced system. On the other hand, too low a threshold will trigger many unnecessary replications, which not only waste network bandwidth but may even lead to over-reactions that cause performance degradations. Moreover, the threshold needs to be adjusted for different workloads. A fixed threshold is inappropriate for dynamic environments.

Authors in [14, 3, 6, 4, 28, 15] propose different economic models to facilitate the data replication. These works note that it is important that intelligent data replication decisions be made based on expected performance gains. In this paper, we propose a self-tuning and goal-driven data replication algorithm that carries out just-enough data replications to eliminate system performance degradations.

3 System Model

This section describes the system model. We assume a typical data grid environment that is comprised of M cluster sites. Each site could have different computing and storage capacities. We use C_i and S_i to respectively denote the computing capacity and the storage capacity of the i^{th} site. Cluster sites are connected to each other and the bandwidth between site i and site j is denoted by B_{ij} .

Periodically, new data objects are generated and need to be replicated and stored in the cluster sites. A data object is a group of files that tend to be accessed together [27] and the accesses to different data objects are assumed to be independent. We assume N data objects are generated in a period and they vary in size and popularity. s_j is used to denote the object size. To model the data popularity, we define p_j as the probability that a request processes data object j , where $0 \leq p_j \leq 1$ and $\sum_{j=1}^N p_j = 1$. Data grid users (e.g., scientists) usually remain interested in the same types of data for a long time and they invoke the same applications to analyze the new data. As a result, the data access pattern is often repetitive [2]. Therefore, we assume that the popularity of a data object can be estimated using historical data access information.

To represent the site locations of the N data objects, the matrix X is used. If object j is stored in site i , x_{ij} is 1; otherwise x_{ij} is 0. We use m_j to denote the number of replicas for object j . Each site is assumed to store at most one replica of an object. Thus, m_j is also the number of sites that store object j . After all data objects and their replicas are placed and stored in the grid, users issue jobs to process the data. Since a data object is a group of files

that tend to be accessed together and the accesses to different data objects are assumed to be independent, each job only requests for one data object. As typical for many data grids [5], we assume that requests are submitted to a global computational scheduler, which decides and distributes requests to their proper execution sites. Let λ denote the job request rate for the data grid and λ_i be the request rate for site i . We have $\lambda = \sum_{i=1}^M \lambda_i$. In the grid, the sites are connected by limited bandwidth and the data objects are usually large (e.g., in multiple gigabytes). As a result, it takes significant time to transfer data objects between sites. Therefore, when scheduling requests, the global scheduler should consider the data locations. In this paper, we assume that an object-location-aware job scheduling [25] is adopted, where a job is always scheduled to a site that stores the requested data object. With this assumption, using λ_{ij} to denote the request rate for object j in site i , we have $\lambda_{ij} = 0$ when $x_{ij} = 0$. In addition, $\lambda_i = \sum_{j=1}^N \lambda_{ij}$.

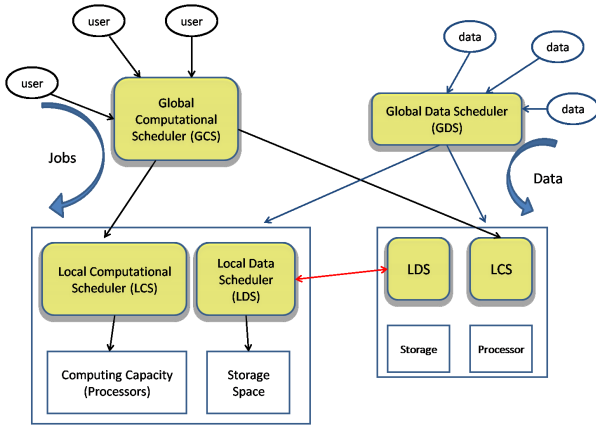


Figure 1: System Model

In Figure 1, we illustrate the system model. As we can see, it contains four resource management components: 1) *Global Data Scheduler (GDS)*, which replicates and distributes new data objects to the grid; 2) *Global Computational Scheduler (GCS)*, which dispatches jobs to cluster sites; 3) *Local Computational Scheduler (LCS)*, which decides how to schedule and execute jobs locally; and 4) *Local Data Scheduler (LDS)*, which changes the data replication by creating, deleting or transferring data replicas. To design and develop these components, job scheduling and data management algorithms are essential. In this paper, we focus on the design of data management algorithms, i.e., the data placement algorithm for GDS and the data replication algorithm for LDS.

4 Algorithms

The performance goal of our data management algorithms is to balance the workload across the data grid. The algorithms control data replica locations to achieve the goal. Since GCS adopts the object-location-aware job scheduling, a site will be overloaded with a huge number of requests if it stores too many popular objects. Therefore, our algorithms determine the proper locations for data replicas so that the resultant site request rates are around the target values. Intuitively, a grid is said to be balanced if job requests are distributed to sites in proportion to site computing capacities, which means that the ideal request rate for site i is,

$$\vec{\lambda}_i = \frac{C_i}{\sum_{k=1}^M C_k} \lambda. \quad (4.1)$$

However, because of system heterogeneities (i.e., data objects have different sizes and popularities, and site storage and computing capacities vary), the ideal rate is not always feasible. For instance, if a site has a large computing capacity but a relatively small storage space, the small number of objects stored in the site may not draw enough requests to fully utilize the site’s computing power. Therefore, a realistic goal is to make the actual rate approach the ideal rate. And for a grid, the *feasible* target rates need to be derived based on site resource configurations and object characteristics. To support our data management algorithms, a module is designed to compute the *feasible* target rates for the M sites, i.e., $\hat{\lambda}_i, i = 1, 2, \dots, M$. Subsequently, the data placement and data replication algorithms control replica locations to ensure that the actual request rates are close to the target values.

4.1 Data Placement Algorithm

We assume that periodically new data objects are generated to replace old objects. Upon the arrival of new objects, GDS invokes a data placement algorithm to decide which sites to store the objects. At that time, an object might be replicated so that its replicas will be stored in multiple sites. A random or a popularity-based algorithm [9] could be invoked to decide the number m_j of replicas for each object. In addition, GDS notifies LDS to delete the corresponding old objects. Instead of storing new data objects in the same locations as their old version or placing them randomly, our data placement algorithm considers site and object characteristics to make proper decisions.

We assume if a site changes its computing or storage capacities, GDS is notified of the change. Therefore, GDS has the latest information about site resources. In addition,

GCS is assumed to keep a log on the object access statistics. Therefore, by communicating with GCS, GDS could get and use the latest statistics to estimate the future data access pattern (i.e., the grid request rate λ and the object popularity p_j).

The data placement algorithm (Algorithm 1) first generates certain numbers of replicas for new data objects. Then it invokes an assignment module to decide replica locations. Finally, the replicas are distributed and stored in the chosen cluster sites.

Algorithm 1 DataPlacement

- 1: **for** each new object j **do**
 - 2: add m_j replicas of j to ReplicaList
 - 3: **end for**
 - 4: Assign(ReplicaList)
 - 5: place new replicas according to X
-

As mentioned, the performance goal of the data placement algorithm is to make the workload λ_i approach the ideal level, i.e., $\bar{\lambda}_i$ specified in Equation (4.1). The replica assignment algorithm (Algorithm 2) is designed to achieve the goal, which also computes the *feasible* target rate $\hat{\lambda}_i$ for each site.

Let S'_i be the unused storage space in site i . If new data objects are generated and sent to GDS at once to replace all old objects, S'_i is initialized to the site storage capacity S_i . If groups of new data arrive at different time, we invoke the algorithm every time a group of new data arrive, in which case S'_i is initialized to the size of site i 's unused storage space after the removal of corresponding old objects. Initially, matrix X records the locations of the remaining old objects (line 1).

Let $\bar{\lambda}_{ij}$ be the estimated value of λ_{ij} , the request rate for object j in site i . If site i does not store object j (i.e., $x_{ij} = 0$), GCS sends no object j 's request to site i and thus $\bar{\lambda}_{ij} = 0$. On the other hand, if site i stores object j , we assume that approximately the load for object j is shared evenly among its m_j replicas and therefore $\bar{\lambda}_{ij} = \frac{\bar{\lambda}_{oj}}{m_j}$, where $\bar{\lambda}_{oj} = p_j \lambda$ is the estimated request rate for object j (lines 2-11).

We call the list of sites that still have unused spaces the remaining sites list (SL). Let the total expected workload for SL sites be λ' (lines 12-19). Then, the ideal request rate for a site in SL is $\bar{\lambda}_i = \frac{C_i}{\sum_{k \in SL} C_k} \lambda'$. $\Delta \lambda_i$ denotes the difference between the ideal load $\bar{\lambda}_i$ and the expected load λ_i . As replicas are assigned to a site, its load is expected to increase. Thus, to make λ_i approach $\bar{\lambda}_i$, site i still needs $\omega_i = \frac{\Delta \lambda_i}{S'_i}$ additional load per storage unit (lines 20-25).

For a replica of object j , we use $\varphi_j = \frac{p_j \lambda}{m_j s_j}$ to represent its estimated load per storage unit. Since the higher ω_i , the

Algorithm 2 Assign(RL)

- initialize S'_i and X
 - 2: // Set $\bar{\lambda}_{ij}$ for remaining old objects
 - for** each site i **do**
 - 4: **for** each object j **do**
 - if** $x_{ij} = 0$ **then**
 - 6: $\bar{\lambda}_{ij} = 0$
 - else**
 - 8: $\bar{\lambda}_{ij} = \frac{p_j \lambda}{m_j}$
 - end if**
 - 10: **end for**
 - end for**
 - 12: // Set the remaining sites list (SL)
 - // and their expected workload (λ'):
 - for** each site i **do**
 - 14: $\bar{\lambda}_i = \sum_{j=1}^N \bar{\lambda}_{ij}$
 - if** $S'_i > 0$ **then**
 - 16: add site i to SL
 - end if**
 - 18: **end for**
 - $\lambda' = \lambda - \sum_{i \notin SL} \bar{\lambda}_i$
 - 20: **repeat**
 - for** each site $i \in SL$ **do**
 - 22: $\bar{\lambda}_i = \frac{C_i}{\sum_{k \in SL} C_k} \lambda'$
 - $\Delta \lambda_i = \bar{\lambda}_i - \lambda_i$
 - 24: $\omega_i = \frac{\Delta \lambda_i}{S'_i}$
 - end for**
 - 26: **while** TRUE **do**
 - sort RL in non-increasing order of $\frac{p_j \lambda}{m_j s_j}$
 - 28: remove the first replica from RL
 - and assume it is object j
 - // Site selection:
 - for** all sites i in SL with $S'_i > s_j$ && $x_{ij} = 0$ **do**
 - 30: choose site p whose w_p is the highest
 - end for**
 - 32: // Assign the replica to site p :
 - update variables: x_{pj} , S'_p , $\bar{\lambda}_{pj}$, $\bar{\lambda}_p$, $\Delta \lambda_p$, and ω_p
 - 34: **if** site p is full **then**
 - remove site p from SL
 - 36: $\hat{\lambda}_p = \sum_{j=1}^N \bar{\lambda}_{pj}$
 - $\lambda' = \lambda' - \hat{\lambda}_p$
 - 38: break the while loop
 - end if**
 - 40: **end while**
 - until** RL is empty
-

more load site i would like to get for every unit of its storage space, it is preferred that we distribute a data replica with large φ_j to a site with high ω_i . We sort new replicas in non-increasing order of φ_j and assign them following this order. First, the chosen site must have enough unused storage space. Second, the replica is not stored in the site. In addition, to assign a replica with large φ_j to a site with high ω_i , among all sites that satisfy the above two constraints, the site with the highest ω_i is chosen. We then make the assignment and update the site information (lines 26-33).

The above process continues until a site becomes full (i.e., $S'_i \approx 0$). The full site will then be excluded from further consideration. That is, for the next round of replica placement, the algorithm no longer takes this site into account (lines 34-35). For the remaining sites, their ideal shares of load are recomputed. Since some site may have a very small storage space and become full before achieving its ideal load $\vec{\lambda}_i$, its unsatisfied load $\Delta\lambda_i$ should be shared by the remaining sites in proportion to their computing capacities. The ideal loads for the remaining sites (i.e., $\vec{\lambda}_i, \forall i \in SL$) are therefore recomputed (lines 36-37 and 21-25). The *feasible* target rate for site i is computed when the site is expected to be full by the assignment, i.e., $\hat{\lambda}_i = \sum_{j=1}^N \bar{\lambda}_{ij}$ (line 36), which could be less than its ideal rate $\vec{\lambda}_i$. As we will see, the knowledge of the feasible target rate $\hat{\lambda}_i$ is particularly important for the data replication algorithm (Section 4.2).

For the next round, the algorithm considers all remaining replicas. The above replica assignment process is repeated until all replicas are considered (lines 20-41).

4.2 Data Replication Algorithm

Following the data placement algorithm described in the previous section, GDS is able to distribute new data objects in a way that facilitates load balancing. If resources in cluster sites do not change and the data access pattern remains the same, the load is balanced across the grid. However, composed of a large number of machines, a data grid is very dynamic, where resources can be removed from or added to it frequently, causing capacity uncertainties. In addition, jobs submitted to a grid have characteristics that vary over time. Such changes can cause the grid to become unbalanced. Therefore, when facing an unbalanced load, LDS needs to invoke a data replication algorithm to adjust replica locations and re-balance the load.

In this section, we propose a novel *self-tuning* data replication algorithm that is automatic, effective and efficient. For comparison, we first briefly discuss the *threshold-based* data replication algorithm commonly adopted for LDS. Like algorithms proposed by Ranganathan and Fos-

ter in [21, 22], a threshold-based algorithm keeps track of access rates of local data objects and when the computational load of a site exceeds a limit, popular data objects are replicated to remote sites so that loads can be evenly distributed. We call those algorithms threshold-based because they consider a data object to be popular when its access rate exceeds a fixed threshold. As explained in Section 2, it is, however, difficult if not impossible to properly choose a fixed threshold that works in dynamic grid environments.

To address the deficiency of threshold-based algorithms, we propose a self-tuning data replication algorithm. Following our new algorithm, LDS controls the degree of changes adaptively, makes necessary data replications and quickly re-balances the system.

Our self-tuning data replication algorithm aims to keep site request rates around their target values: $\hat{\lambda}_1, \hat{\lambda}_2 \cdots \hat{\lambda}_M$. In the previous section, we have described a replica assignment algorithm (Algorithm 2) as an essential module for placing new data objects. Since the algorithm also derives *feasible* target rates for a balanced grid, we could leverage it to guide the data replication. Upon system changes such as resource capacity changes or data access pattern changes, GDS invokes the replica assignment module (Algorithm 2) with an empty new replica list (i.e., $\text{Assign}(\emptyset)$) to re-calculate the *feasible* target rates of the cluster sites. Then, being notified of the new target rate $\hat{\lambda}_i$, LDS in site i invokes the self-tuning data replication algorithm. It is a distributed data replication algorithm, where LDSs collaborate with each other and control their replica locations so that the actual load λ_i is kept within a specified range around the target $\hat{\lambda}_i$. That is, $(1 - \beta)\hat{\lambda}_i \leq \lambda_i \leq (1 + \beta)\hat{\lambda}_i$ is ensured, where $0 < \beta < 1$ is a configurable parameter.

The algorithm pseudo code is presented in Algorithm 3. Upon receiving the new target rate $\hat{\lambda}_i$, the algorithm checks the actual rate of the site λ_i and compares them. If λ_i is greater than the upper bound $(1 + \beta)\hat{\lambda}_i$, the site is considered as heavily loaded; if λ_i is less than the lower bound $(1 - \beta)\hat{\lambda}_i$, the site is lightly loaded; otherwise, λ_i is within the specified range and site i is a balanced site (line 1). In addition, the difference between actual and target loads is also computed: $\Delta\hat{\lambda}_i = |\lambda_i - \hat{\lambda}_i|$. The algorithm aims to reduce a heavily loaded site h 's load by $\Delta\hat{\lambda}_h$ and to increase a lightly loaded site l 's load by $\Delta\hat{\lambda}_l$.

In the algorithm, when a site is diagnosed as heavily loaded, it starts a pair-establish protocol to pair with a lightly loaded site (line 2). After a heavily loaded site h and a lightly loaded site l become a pair, they collaborate with each other for the data replication. Unlike the threshold-based algorithms which blindly replicate "popular" data objects, our algorithm considers $\Delta\hat{\lambda}_h$ and $\Delta\hat{\lambda}_l$ and only replicates data objects that are necessary for achieving $\hat{\lambda}_h$ or $\hat{\lambda}_l$. To determine which data objects should be replicated from

site h to site l , the algorithm estimates the load change effect of data replications. Once the planned replications are deemed sufficient to achieve one of the targets $\hat{\lambda}_h$ or $\hat{\lambda}_l$, the replication from site h to site l stops.

Assuming a new data object j is going to be replicated from site h to site l , the resultant load changes on the two sites, $\Delta\lambda_h$ and $\Delta\lambda_l$, are estimated. Since the load changes are mainly caused by object j 's request rate changes, we assume $\Delta\lambda_h \approx \Delta\lambda_{hj}$ and $\Delta\lambda_l \approx \Delta\lambda_{lj}$. The algorithm estimates $\Delta\lambda_{hj}$ and $\Delta\lambda_{lj}$ using local information or information solicited from GCS. For instance, if GCS adopts a scheduling algorithm that considers not only object locations but also current site loads, i.e., the object-location-aware and least-loaded scheduling, a request for object j is sent to a site i that stores object j and has the smallest $\frac{\lambda_i}{C_i}$. Therefore, approximately, the request rate is inversely proportional to $\frac{\lambda_i}{C_i}$. As a result, since site h has a heavier load than site l , i.e., $\frac{\lambda_h}{C_h} > \frac{\lambda_l}{C_l}$, site h 's load reduction $\Delta\lambda_{hj}$ after object j is added to site l is estimated to be greater than $\frac{\lambda_{hj}}{2}$. To estimate site l 's load change $\Delta\lambda_{lj}$, we need to know the total request rate for object j . This is because after site l stores object j , it begins to share requests for object j with all other sites that have replicas of object j . Via soliciting information from GCS, LDS obtains the total request rate λ_{oj} for object j and $\Delta\lambda_l$ is estimated to be less than $\frac{\lambda_h/C_h}{\lambda_l/C_l + \lambda_h/C_h} \times \lambda_{oj}$.

Leveraging the procedure described above, LDS in site h decides which data objects to replicate. First, it generates a list (L_h) that includes the most popular data objects whose replications are expected to cause the workload of site h , to reduce below the target (lines 4-16). That is, $\sum_{j \in L_h} \Delta\lambda_{hj} \geq \Delta\hat{\lambda}_h$. The L_h list gives a super set of data objects that need to be replicated from site h to site l . Therefore, in the next step, the global request rate information is collected only for those objects in L_h list (line 17). In addition, LDS in site h communicates with LDS in site l to obtain the information of replicas in site l (line 18). Then, LDS in site h estimates $\Delta\lambda_{lj}$ and decides a list L_l of data objects to be replicated to site l (lines 19-31), where $L_l \subseteq L_h$, $\forall o_j \in L_l$ $x_{lj} = 0$, and $\sum_{o_j \in L_l} \Delta\lambda_{lj} \leq \Delta\hat{\lambda}_l$. This way, after replicating all objects in L_l (lines 32-33), the load on site l is expected to increase no more than $\Delta\hat{\lambda}_l$.

As designed, following the algorithm, LDS only replicates data objects considered necessary for achieving $\hat{\lambda}_l$ or $\hat{\lambda}_h$. It could happen that only one target rate is achieved. If the replications are not enough to convert the heavily load site h to a balanced site, the algorithm repeats and site h pairs with another lightly loaded site to replicate more data objects.

Algorithm 3 DataReplication($\hat{\lambda}_i$)

```

1: compare  $\hat{\lambda}_i$  with  $\lambda_i$  to determine site status
2: run the pairing protocol
3: // For a pair of sites  $h$  and  $l$ , do the following
   // computation on the heavily loaded site  $h$ 
4: add all local objects to a list  $L$ 
5: sort  $L$  in a non-increasing order of  $\lambda_{hj}$ 
6:  $\Delta\hat{\lambda}_h = |\lambda_h - \hat{\lambda}_h|$ 
7:  $\Delta\lambda_h = 0$ 
8: for each object  $j$  in  $L$  do
9:    $\Delta\lambda_{hj} = \frac{\lambda_{hj}}{2}$ 
10:   $\Delta\lambda_h = \Delta\lambda_h + \Delta\lambda_{hj}$ 
11:  if ( $\Delta\lambda_h \geq \Delta\hat{\lambda}_h$ ) then
12:    break for
13:  else
14:    add object  $j$  to  $L_h$  list
15:  end if
16: end for
17: collect from GCS  $\lambda_{oj}$  information for all  $L_h$  objects
18: collect  $\lambda_l$ ,  $\hat{\lambda}_l$  and  $X_l$  information from site  $l$ 
19:  $\Delta\hat{\lambda}_l = |\lambda_l - \hat{\lambda}_l|$ 
20:  $\Delta\lambda_l = 0$ 
21: for each object  $j$  in  $L_h$  do
22:  if ( $x_{lj} = 0$ ) then
23:     $\Delta\lambda_{lj} = \frac{\lambda_h/C_h}{\lambda_l/C_l + \lambda_h/C_h} \times \lambda_{oj}$ 
24:     $\Delta\lambda_l = \Delta\lambda_l + \Delta\lambda_{lj}$ 
25:    if ( $\Delta\lambda_l \geq \Delta\hat{\lambda}_l$ ) then
26:      break for
27:    else
28:      add object  $j$  to  $L_l$  list
29:    end if
30:  end if
31: end for
32: send  $L_l$  to site  $l$ 
33: site  $l$ : replicate  $L_l$  objects from site  $h$ , if space runs out,
    apply LRU replacement strategy. The single-replica objects
    (i.e.,  $o_j$  where  $m_j = 1$ ) are, however, kept in the
    storage.

```

5 Performance Evaluation

In this section, we evaluate the proposed algorithms through simulations. We develop a modular simulator based on the model described in Section 3. It contains four resource management components: Global Data Scheduler (GDS), Global Computational Scheduler (GCS), Local Computational Scheduler (LCS) and Local Data Scheduler (LDS). By choosing different algorithms for these components, we can simulate different data grid management mechanisms. We evaluate the proposed algorithms under a variety of system configurations and compare them with previous approaches. Simulation results show that our algorithms achieve much better performance on load balancing, average response time and network bandwidth cost. Next, we first introduce system configurations and then describe simulation results.

5.1 System Configurations

We simulate a data grid of 8 sites, connected by network links of 100 MB/s. The site storage spaces are assigned to be different, with a total of 160 TB. We assume that the site storage space is limited and no site can store all data objects. That is, $\forall i, S_i < \sum_{j=1}^N s_j$. To evaluate the effect of system heterogeneity, we generate four different configurations of site computing capacity (C_i): *AllSame*, where $\forall i, j, C_i = C_j$, *Prop*, where C_i is proportional to S_i , *In-Prop*, where C_i is inversely proportional to S_i , and *Random*, where C_i is randomly assigned. Among the four, a particular site configuration is chosen for each simulation.

We assume that 1,000 data objects are generated periodically. Object size follows a normal distribution with a mean of 50 GB and a standard deviation of 200 MB [13]. Data popularity follows a zipf [30] distribution and the transfer of a data object from one site to another incurs a bandwidth cost. The data transfer delay is simulated. That is, when data object j is replicated to site i , jobs requesting object j can only be sent to site i after the data transmission is complete. Each simulation lasts 50,000 seconds. Job request follows a poisson distribution with an average arrival rate of λ . To indicate how loaded a data grid is, we define a system load parameter: $\rho = \frac{\lambda}{\mu}$, where μ is the average service rate of the grid and is estimated based on the total computing capacity of the grid ($\sum_{i=1}^M C_i$). We simulate various system loads in our experiments. As system load ρ increases, so does the total number of jobs. The maximum number of jobs simulated is 100,000 when $\rho = 1.0$. Each job involves read-only accesses to a single data object, which is selected based on the data popularity. Job execution time, with an average of 1,000 seconds, is assumed to be proportional to the size of the requested object. In order to simulate the situ-

ation where grid users shift their interests from one group of data objects to another, we model data popularity changes. Two types of workload are simulated. For the first type of workload, called a *static workload*, the data popularity follows a zipf distribution with a constant parameter $\alpha = 0.68$. The second type of workload is called a *dynamic workload*, where data popularity follows a zipf distribution with parameter α changing from 0.68 to 0.4 during the simulation.

5.2 Evaluation: Data Placement

We first evaluate the data placement algorithm developed in Section 4.1. Since when making data placement decisions, the algorithm considers factors like object Popularity and site Storage space, we use *PS* to denote the algorithm. It is compared with a commonly adopted algorithm [21, 22, 3, 6] that randomly distributes replicas to cluster sites. We name this baseline strategy as the *Random* data placement algorithm. In this group of simulations, we choose the *Random* site configuration and simulate the *static workload*. Initially, each object has only one replica and GDS distributes these replicas to cluster sites. The object-location-aware and round-robin scheduling is adopted by GCS and the simple first-come, first-served scheduling is adopted by LCS, which executes n tasks simultaneously on n local computing nodes. For this group of simulations, LDS may apply an algorithm to replicate more objects to the sites. The performance is evaluated in both scenarios with and without replications.

Figure 2 shows the average job response time of the algorithms. We can see that our *PS* algorithm outperforms the *Random* algorithm by up to 200% when no data replication is applied and 100% with replications. Even with replications, the *Random* algorithm (i.e., *Random-Replication*) still performs worse than our *PS* algorithm without any replication (i.e., *PS-NO*). This is because our algorithm provides an intelligent data placement so that the system load is balanced from the beginning, optimizing the job response time. However, with a random data placement, the system is not balanced and it needs further adjustments (i.e., dynamic data replications) to reach a balanced state. Such a process, however, takes time and bandwidth, resulting in a longer response time and a higher bandwidth cost. Figure 3 shows the bandwidth costs resulted from dynamic data replications. From the figure, we can see when using our *PS* algorithm the bandwidth cost can be saved by up to 200%. These results demonstrate that a good data placement algorithm can improve performance significantly, even when dynamic data replications are applied.

We have carried out numerous other simulations with a different global scheduling algorithm (i.e., object-location-aware and least-loaded scheduling) and/or varied site con-

figures, where we obtain similar results [9].

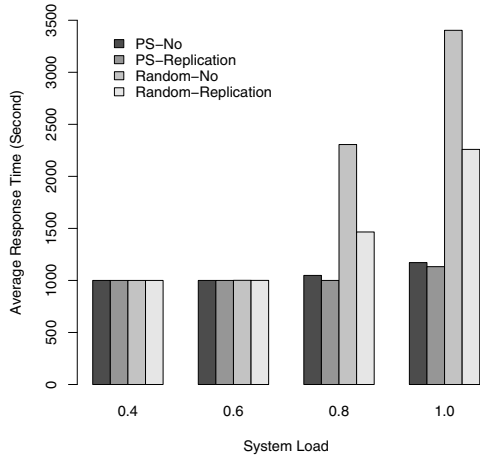


Figure 2: Average Response Time

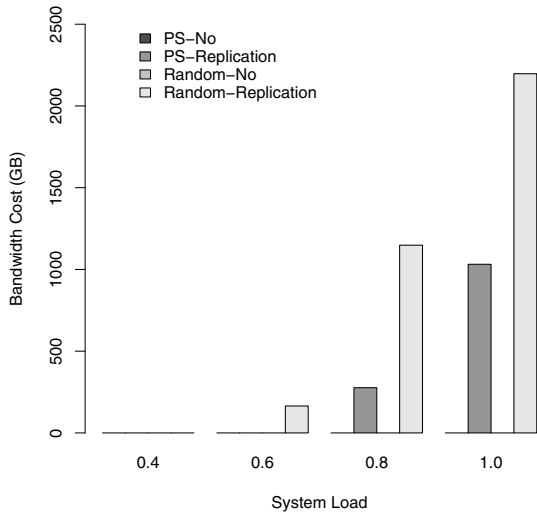


Figure 3: Bandwidth Cost

5.3 Evaluation: Data Replication

As described in Section 4.2, the data replication algorithm aims to making the real load λ_i close to the target load $\hat{\lambda}_i$. We, therefore, introduce a new metric called the *balance ratio* to measure the performance, which is defined as $br_i = \frac{|\hat{\lambda}_i - \lambda_i|}{\lambda_i}$ for site i . The *maximum* balance ratio of the grid is calculated as,

$$br_{max} = \max(br_1, br_2, \dots, br_M). \quad (5.2)$$

We simulate the self-tuning data replication algorithm developed in Section 4.2. The sampling period is set to be

300 seconds. That is, in every 300 seconds, the algorithm is invoked and if needed data objects are replicated. We evaluate how the algorithm performs with dynamic changes in the grid. Two types of changes are simulated: data popularity change and site computing capacity change. The performance goal is to keep λ_i within a specified range of its target $\hat{\lambda}_i$. That is, to ensure $(1 - \beta)\hat{\lambda}_i \leq \lambda_i \leq (1 + \beta)\hat{\lambda}_i$. In this group of simulations, we set $\beta = 10\%$ and as long as for any site i , λ_i and $\hat{\lambda}_i$ differ by less than 10%, the data grid is considered balanced. We thus measure the maximum balance ratio (Equation (5.2)) in every sampling period and check if the algorithm can successfully keep br_{max} below 10%. The average job response time and the bandwidth cost are also measured.

Dynamic Workload with No Capacity Change. We first use the *dynamic workload* to simulate data popularity changes. A *Random* site configuration is adopted. In GCS, the object-location-aware and least-loaded scheduling is chosen. GDS applies the *PS* data placement algorithm to distribute data objects. In LCS, the first-come, first-served scheduling is simulated. For comparison, we also simulate two baseline algorithms. The first baseline is a threshold-based algorithm proposed by Ranganathan and Foster in [22]. When a site becomes heavily loaded, the algorithm replicates popular data objects to a randomly-picked least-loaded site. It chooses popular objects based on a fixed request rate threshold. In this simulation, in order to make this baseline algorithm perform at its best, we have tuned the algorithm and selected a proper threshold for it. No dynamic replication is chosen as the other baseline.

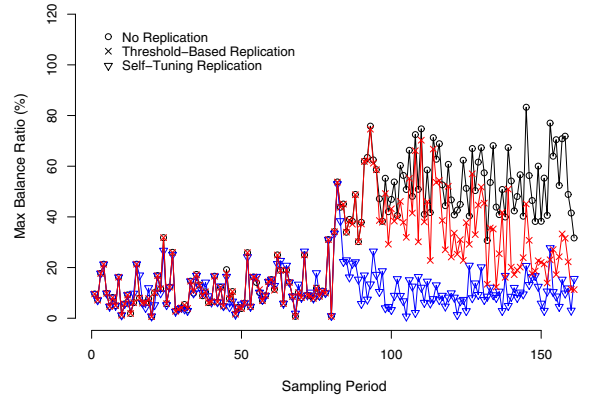


Figure 4: Dynamic Workload: Max Balance Ratio

Figure 4 shows the maximum balance ratios (br_{max}) achieved by the three algorithms. We can see that initially, br_{max} is around 10% for all three algorithms. This is because GDS adopts the *PS* data placement algorithm which has placed objects in a way that balances the load. Around

Algorithm	AverageResponseTime	BandwidthCost
No Replication	1454.26 sec	-
Threshold-Based	1253.87 sec	1911 GB
Self-Tuning	1011.71 sec	1853 GB

Table 1: Dynamic Workload: Response Time and Bandwidth Cost

the 70th sampling period, br_{max} increases rapidly due to the workload change. Some site experiences a large increase in its request rate because some unpopular data objects stored in the site suddenly become popular. The request rate increase leads to a larger balance ratio for the site. If no dynamic replication is applied, br_{max} remains high till the end of the simulation. When data replications are applied with either a threshold-based or a self-tuning algorithm, br_{max} is reduced and returns back to the targeted range. However, our self-tuning data replication algorithm is automatic, requiring no parameter tuning, and it achieves a lower balance ratio in a shorter time.

Table 1 gives the average job response time and the bandwidth cost. Our algorithm achieves the shortest response time with the smallest bandwidth cost.

Static Workload with Computing Capacity Change.

The second simulation evaluates the algorithm’s performance with a site computing capacity change. During the simulation, a site shuts down half of its computing nodes. All configurations are the same as the previous simulation except that we simulate a *static workload* here. Again, the three data replication algorithms are compared.

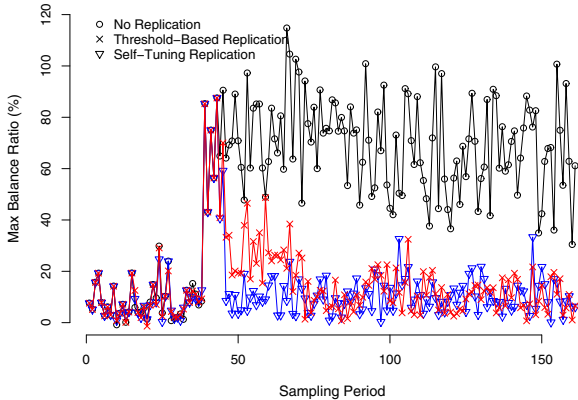


Figure 5: Capacity Change: Max Balance Ratio

From Figure 5, we can see that due to the capacity change, there is a big increase of br_{max} around the 40th sampling period. For the site, the computing capacity reduction results in a decrease in its target load, thus an in-

Algorithm	AverageResponseTime	BandwidthCost
No Replication	2170.23 sec	-
Threshold-Based	1098.13 sec	18614 GB
Self-Tuning	1049.18 sec	12972 GB

Table 2: Capacity Change: Response Time and Bandwidth Cost

crease in its balance ratio. When there is no replication, the load of that site cannot decrease and the balance ratio stays high. Data replications enable the site to share its load with other sites and returns to a balanced state. Table 2 shows that our self-tuning algorithm improves the job response time by about 100%. Although the threshold-based algorithm achieves similar response time, it consumes much higher bandwidth.

We have also carried out simulations with both *dynamic workload* and computing capacity change, where similar results are observed [9].

6 Conclusion

In this paper, we investigate automatic data management in grids. First, we propose a novel data placement algorithm, which automatically and intelligently stores new data objects in grids. Since the algorithm considers many factors such as data access pattern, site storage and computing capacity, it works well with various data grid configurations. Through simulations, we show that this new algorithm achieves much better performance than a random data placement algorithm. We then develop a self-tuning data replication algorithm, which quickly adapts to system changes and maintains good performance through efficient and effective data replications. Unlike threshold-based algorithms, which blindly replicate popular data, our new algorithm only makes necessary replications to achieve the performance goal. Simulation results show that the self-tuning algorithm outperforms existing approaches.

References

- [1] Globus toolkit 4. <http://www.globus.org/toolkit/>.
- [2] J. H. Abawajy. Placement of file replicas in data grid environments. pages 66–73, 2004.
- [3] W. Bell, D. Cameron, R. Carvajal-Schiaffino, A. Millar, K. Stockinger, and F. Zini. Evaluation of an economy-based file replication strategy for a data grid. *International Symposium on Cluster Computing and the Grid*, pages 661– 668, 2003.
- [4] R. Buyya, D. Abramson, and J. Giddy. An economy driven resource management architecture for global computational

- power grids. *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 26–29, 2000.
- [5] R. Buyya, S. J. Chapin, and D. C. DiNucci. Architectural models for resource management in the grid. In *GRID*, pages 18–35, 2000.
- [6] L. Capozza, A. P. Millar, K. Stockinger, W. H. Bell, D. G. Cameron, and F. Zini. Simulation of dynamic grid replication strategies in optorsim. *International Workshop on Grid Computing*, pages 46 – 57, 2002.
- [7] A. Chervenak and R. Schuler. A data placement service for petascale applications. *SuperComputing*, pages 63–68, 2007.
- [8] W. Chu. Optimal file allocation in a multiple computer system. *IEEE Transaction of Computers*, 18:No.10, 1969.
- [9] Y. Ding. Intelligent data management in data grids. In *M.S. Thesis, University of Nebraska-Lincoln*, July 2008.
- [10] L. Dowdy and D. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14, 1982.
- [11] K. Eswaran. Placement of records in a file and file allocation in a computer network. *Information Processing*, pages 304–307, 1974.
- [12] W. L. C. Grid. <http://lcg.web.cern.ch/lcg/>.
- [13] A. Iamnitchi, S. Doraimani, and G. Garzoglio. Filecules in high-energy physics: Characteristics and impact on resource management. *IEEE International Symposium on High Performance Distributed Computing*, pages 69–80, 2006.
- [14] H. Lamahamedi, B. Szymanski, Z. Shentu, and E. Deelman. Data replication strategies in grid environments. *International Conference on Algorithms and Architectures for Parallel Processing*, pages 378–383, 2002.
- [15] H. Lamahamedi and B. K. Szymanski. Decentralized data management framework for data grids. *Future Generation Computer Systems*, 23:109–115, 2007.
- [16] P. Liu and J. Wu. Optimal replica placement strategy for hierarchical data grid systems. in *Proceedings of CC-Grid2006, The 6th ACM/IEEE International Symposium on Cluster Computing and the Grid*, IEEE CS Press, Singapore, pages 417–420, 2006.
- [17] T. Loukopoulos and I. Ahmad. Static and adaptive data replication algorithms for fast information access in large distributed systems,. *IEEE International Conference on Distributed Computing Systems*., 2000.
- [18] L. Project. Ligo - laser interferometer gravitational wave observatory. <http://www.ligo.caltech.edu/>.
- [19] R. M. Rahman, K. Barker, and R. Alhaji. Study of different replica placement and maintenance strategies in data grid. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 171–178, Washington, DC, USA, 2007. IEEE Computer Society.
- [20] K. Ranganathan and I. Foster. Identifying dynamic replication strategies for a high performance data grid. *IEEE/ACM Grid*, pages 75–86, 2001.
- [21] K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data intensive applications. *International ACM Symposium on High Performance Distributed Computing*, pages 352–358, 2002.
- [22] K. Ranganathan and I. Foster. Simulation studies of computation and data scheduling algorithms for data grids. *Journal of Grid Computing*, pages 53–62, 2003.
- [23] J. Rehn. Phedex high-throughput data transfer management system. *Computing in High Energy and Nuclear Physics (CHEP)*, 2006.
- [24] L. D. Replicator. <http://www.lsc-group.phys.uwm.edu/ldr/overview.html>.
- [25] H. Sato, S. Matsuoka, T. Endo, and N. Maruyama. Access-pattern and bandwidth aware file replication algorithm in a grid environment. In *IEEE/ACM International Conference on Grid Computing*, pages 250–257. IEEE, September 2008.
- [26] C. C. M. Solenoid:. <http://cmsinfo.cern.ch/welcome.html/>.
- [27] H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman, and B. Tierney. File and object replication in data grids. *Cluster Computing*, 5:305 – 314, 2002.
- [28] M. Tang and X. T. B.S. Lee, C. K. Yeo. Dynamic replication algorithms for the multi-tier data grid. *Future Generation Computer Systems*, 21:775–790, 2005.
- [29] P. L. Y.F. Lin and J. Wu. Optimal placement of replicas in data grid environments with locality assurance. *International Conference on Parallel and Distributed Systems*, pages 465–474, 2006.
- [30] G. K. Zipf. Human behavior and principle of least effort: An introduction to human ecology. *Addison Wesley, Cambridge, Massachusetts*, pages 204–205, 1949.