Hierarchical MapReduce Programming Model and Scheduling Algorithms

Yuan Luo and Beth Plale
School of Informatics and Computing, Indiana University Bloomington
Bloomington, IN, USA
$\{$*yuanluo, plale*$\}$*@indiana.edu*

***Abstract*—We present a Hierarchical MapReduce framework that gathers computation resources from different clusters and runs MapReduce jobs across them. The applications implemented in this framework adopt the *Map-Reduce-GlobalReduce* model where computations are expressed as three functions: Map, Reduce, and GlobalReduce. Two scheduling algorithms are introduced: Compute Capacity Aware Scheduling for compute-intensive jobs and Data Location Aware Scheduling for data-intensive jobs. Experimental evaluations using a molecule binding prediction tool, AutoDock, and grep demonstrate promising results for our framework.**

***Keywords*-MapReduce; Cross Domain; Multi-Cluster; Data Intensive;**

I. Introduction

MapReduce [1] is a programming model well suited to processing large datasets using high-throughput parallelism running on a large number of compute resources. While it has proven useful on data-intensive high throughput applications, conventional MapReduce model limits itself to scheduling jobs within a single cluster. As job sizes become larger, single-cluster solutions grow increasingly inadequate. A researcher at a research institution typically has access to several research clusters, each of which consists of a small number of nodes. The nodes in one cluster may be very different from those in another cluster in terms of CPU frequency, number of cores, cache size, memory size, network bandwidth and storage capacity. For example, researchers at Indiana University have access to IU Quarry, FutureGrid [2], and Teragrid [3] clusters but each cluster imposes a limit on the maximum number of nodes a user can allocate at any one time. If these isolated clusters can work together, they collectively become more powerful.

Additionally, the input dataset could be very large and widely distributed across multiple clusters. There are large differences in I/O speeds from local disk storage to wide area networks. Feeding a large dataset repeatedly to remote computing resources becomes the bottleneck. When mapping such data-intensive tasks to compute resources, scheduling mechanisms need not only take into account the execution time of the tasks, but also the overheads of staging the dataset. To scale up such tasks, there are tradeoffs to be made, such as determining whether to bring data to computation or bring computation to data, or even regenerate data on-the-fly.

Users cannot directly deploy a MapReduce framework such as Hadoop on top of multiple clusters to form a single larger MapReduce cluster because the internal nodes of a cluster are not directly reachable from outside. However, MapReduce requires the master node to directly communicate with any slave node, which is also one of the reasons why MapReduce frameworks are usually deployed within a single cluster. Therefore, in our research we strive to make multiple clusters act collaboratively as though they were a single cluster to improve the efficiency of running MapReduce.

Another contribution of research is to schedule and coordinate MapReduce jobs efficiently among local clusters, and this is closely tied to how the datasets are partitioned. This is particularly true for data-intensive tasks requiring access to large datasets stored across multiple locations and geographies. The input dataset could be distributed among clusters, or even across data centers where the data load is equivalent to or larger than the computational load.

The remainder of the paper is organized as follows. Section II presents the hierarchical map reduce. Section III demonstrates the feasibility of our solution by applying compute-intensive and data-intensive applications. Section IV lists related work. The conclusion and future work are given in Section V.

II. Hierarchical MapReduce

There are two possible approaches to addressing the challenge of internal nodes not being reachable from outside. The first is to unify the underlying physical clusters as a single virtual cluster by adding a special infrastructure layer, and run MapReduce on top of this virtual cluster. However, data shuffling across wide area network is too expensive under certain common conditions. The other is to make the MapReduce framework work directly with multiple clusters without additional special infrastructure layers. We proposed in [4] a solution which takes the second approach, and gathers isolated cluster resources into a more capable one for running MapReduce jobs.

A. Hierarchical MapReduce (HMR) Architecture

The HMR framework consists of two layers. As is shown in Figure 1, at the top layer is a Global Controller, which consists of a Job Scheduler, a Data Manager, and a Workload

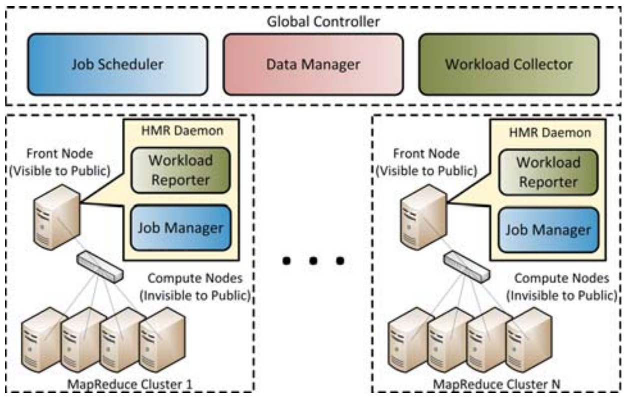

Figure 1. Hierarchical MapReduce Architecture

Collector; the bottom layer consists of multiple clusters where the distributed local MapReduce jobs are run. Each cluster has a MapReduce master node with a HMR daemon, which is responsible for reporting workloads and managing local MapReduce jobs. The compute nodes inside each of the clusters are not accessible from the outside.

When a MapReduce job is submitted to a HMR framework, it is split into a number of sub-jobs and assigned each to a local cluster by the Job Scheduler at the Global Controller level. A scheduling decision will be made based on several factors, including but not limit to the characteristics of the input dataset, the current workload reported by the workload reporter from each local cluster, as well as the capability of individual nodes making up each cluster. The Global Controller also partitions the input data in proportion to the sub-job sizes if the input data has not been deployed before-hand. The Data Manager transfers the user supplied MapReduce jar and job configuration files with the input data partitions to the local clusters. Once the data and job executable are ready, the job manager of the local cluster's HMR daemon starts the local MapReduce sub-job. When the local sub-jobs are finished on a local cluster, the clusters will transfer the output back to one of the local clusters for global reduction, if is required by the application. Upon finishing all the local MapReduce sub-jobs, a global $Reduce$ function will be invoked to perform the final reduction task, unless the original job is map-only.

Since data transfer is very expensive, we recommend that users use the global controller to transfer data only when the size of the input data is small and the time spent for transferring the data is insignificant compared to the computation time. For large datasets, it is more efficient to deploy the datasets before-hand, so that the jobs get the full benefit of parallelization and the overall time does not become dominated by data transfer.

Table I
INPUT AND OUTPUT TYPES OF MAP, REDUCE, AND GLOBAL REDUCE FUNCTIONS.

Function Name	Input	Output
Map	(k^i, v^i)	(k^m, v^m)
Reduce	$(k^m, [v_1^m, ..., v_n^m])$	(k^r, v^r)
GlobalReduce	$(k^r, [v_1^r, ..., v_n^r])$	(k^o, v^o)

B. Programming Model

The programming model of the HMR is the *Map-Reduce-GlobalReduce* model where computations are expressed as three functions: Map, $Reduce$, and $GlobalReduce$. We use the term "$GlobalReduce$" to distinguish it from the "local" $Reduce$ which happens right after Map execution, but conceptually as well as syntactically, a $GlobalReduce$ is just another conventional $Reduce$ function. The Map, just as with a conventional Map, takes an input data split and produces an intermediate key/value pair; likewise, the $Reduce$, just as with a conventional $Reduce$, takes an intermediate input key and a set of corresponding values produced by the Map task, and outputs a different set of key/value pairs. Both the Map and the $Reduce$ functions are executed on local clusters first. The $GlobalReduce$ is executed on one of the local clusters using the output from all the local clusters. Table I lists these functions and the corresponding input and output data types.

Figure 2 uses a tree-like structure to show the data flow across Map, $Reduce$, and $GlobalReduce$ functions. The leaf rectangle nodes with dotted line represent MapReduce clusters that perform the Map and $Reduce$ functions, and the root rectangle node with dotted line is a MapReduce cluster on which the $GlobalReduce$ takes place. The arrows indicate the direction in which the data sets (key/value pairs) flow. When a job is submitted into the system, the input dataset is partitioned into splits. The splits then are passed to the leaf nodes where Map tasks are launched. Each Map task consumes an input key/value pair and produces a set of intermediate key/value pairs. The set of intermediate pairs then are passed to the $Reduce$ tasks, which are also launched at the same cluster as the Map tasks. Each $Reduce$ task consumes an intermediate key with a set of corresponding values, and produces a different set of key/value pairs as output. All the local $Reduce$ results are sent to one local cluster to perform $GlobalReduce$ task. Each $GlobalReduce$ takes in a key and a set of corresponding values that were originally produced from the local $Reduce$ tasks, and computes and produces the final output.

Theoretically, the model can be extended to additional layers, i.e., the tree structure in Figure 2 can have more depth by adding more intermediate $Reduce$ steps that partially reduce the output from previous layer of $Reduce$ functions. Following the classification of Elteir et al. [5] MapReduce jobs can be classified into recursively reducible jobs and

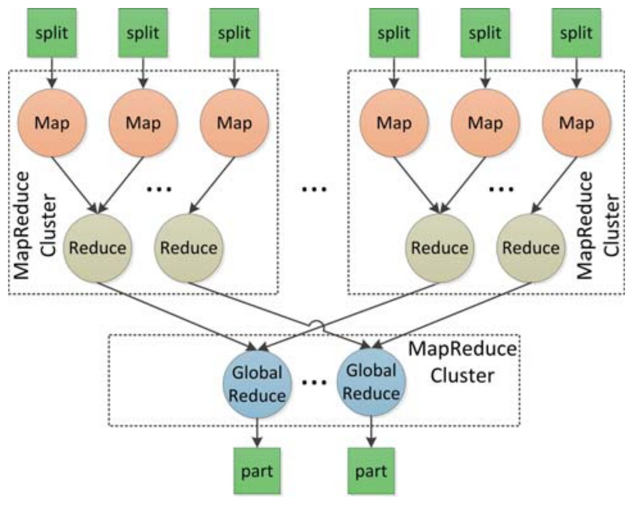

Figure 2. HMR Data Flow

non-recursively jobs. Recursively reducible jobs have no inherent synchronization requirement between the Map and $Reduce$ phases. Such jobs can be processed in a hierarchical reduction or incremental reduction. This is a simplified case in our HMR system where the $Reduce$ function and $GlobalReduce$ function are one-in-the-same. Such jobs can be further optimized using a combiner function between local Map and $Reduce$ functions.

C. Job Scheduling

We propose two scheduling algorithms. The first is the Compute Capacity Aware Scheduling (CCAS) algorithm, which aims to optimize data-intensive jobs. The second is a data location aware scheduling algorithm, Data Location Aware Scheduling (DLAS), where a candidate cluster for processing a data partition requires the physical residence of the data partition. Each are introduced here.

1) Compute Capacity Aware Scheduling(CCAS):

The CCAS aims to optimize compute-intensive jobs. A dataset consists of multiple data partitions that are distributed among multiple clusters. We make the assumption that the input data of each map tasks are equal in size and the map tasks take approximately the same amount of time to run.

Consider running a MapReduce job on n clusters. Let $MapSlots_i$ be the maximum number of Map tasks that can be run concurrently on $Cluster_i$; $MapOccupied_i$ be the number of Map tasks currently running on $Cluster_i$; $FreeMap_i$ be the number of available slot to run Map tasks on $Cluster_i$; $NumCore_i$ be the total number of CPU Cores on $Cluster_i$, where $i \in \{1, \ldots, n\}$. And ρ_i defines how many Map tasks a user assigns to each core, that is,

$$MapSlots_i = \rho_i \times NumCore_i \tag{1}$$

To simplify the algorithm, we set $\rho_i = 1$ in the local MapReduce clusters for compute-intensive jobs, so that

$$FreeMap_i = MapSlots_i - MapOccupied_i \tag{2}$$

For simplicity, let

$$\gamma_i = FreeMap_i \tag{3}$$

The weight of each sub-job can be calculated from (4) where the factor θ_i is the computing power of each cluster, e.g., the CPU speed, memory size, storage capacity, etc. The actual θ_i varies depending on the characteristics of the jobs, i.e., whether they are computation intensive or I/O intensive

$$W_i = \frac{\gamma_i \times \theta_i}{\sum_{i=1}^{n} \gamma_i \times \theta_i} \tag{4}$$

Let $JobMap_x$ be the total number of Map tasks for a particular job x, which can be calculated from the number of keys in the input to the Map tasks, and $JobMap_x^i$ be the number of Map tasks to be scheduled to $Cluster_i$ for job x, so that

$$JobMap_x^i = W_i \times JobMap_x \tag{5}$$

After partitioning the MapReduce job to Sub-MapReduce jobs using equation (5), we number the data items of the datasets and move the data items accordingly.

2) Data Location Aware Scheduling(DLAS):

The data location aware scheduling algorithm, Data Location Aware Scheduling (DLAS), requires of a candidate cluster the physical residence of the data partition. Typically, with the assistance of a global file system, data partitions can be replicated among clusters. If more than one candidate cluster is found, the scheduling algorithm maps that data partition to one of the candidate clusters in a way that all selected clusters have similar ratio of data over cluster compute capacity, or called, balanced processing time.

Consider a dataset $DS = \{D_1, \ldots, D_m\}$ which has been partitioned to m partitions, residing on n clusters. Each partition has been replicated among N_j clusters, with the data size defined as SZ_{D_j}, where $1 \le N_j \le n$ and $j \in \{1, \ldots, m\}$. A scheduling plan k contains a list of subset of dataset DS. Let SDS_i^k be a subset of DS for $Cluster_i$, and

$$\exists k((\cup_{i=1}^{n} SDS_i^k = DS) \wedge (\cap_{i=1}^{n} SDS_i^k = \emptyset)) \tag{6}$$

The compute capacity of $Cluster_i$ is defined as W_i , where $i \in \{1, \ldots, n\}$, so that in $Cluster_i$, the ratio of data partitions collection over compute capacity in scheduling plan k is defined as R_i^k, and

$$R_i^k = \frac{\sum_{x \in SDS_i^k} SZ_x}{W_i} \tag{7}$$

And the total data processing time under a scheduling plan k is defined as,

$$T_k = \omega(max_{i \in \{1, \ldots, n\}} R_i^k) \tag{8}$$

```
1. Sort the data partitions $D_j$ in decreasing order of size
2. For $i = 1$ to $n$ do
    2.1. $UASS_i = Sum$ the size of all unassigned data partitions which exists in $Cluster_i$
    2.2. $Exp_i$ = Expected load of data partitions collection size in $Cluster_i$ (The distribution of $Exp_i$ among all clusters is based on $W_i$, derived from equation 7)
3. For $j = 1$ to $m$ do
    3.1. Sort candidate clusters (which contains replica of $D_j$) in increasing order of $UASS$ values.
    3.2. Assign data partition $D_j$ to the cluster with smallest $UASS$ value, if current load on that cluster plus $SZ_{D_j}$ is less than $Exp$ value; otherwise, assign data partition $D_j$ to the cluster with least current load.
    3.3. Update load of the cluster that receives data partition $D_j$
    3.4. Update $UASS_i$, if $Cluster_i$ Contains $D_j$
```

Figure 3. Data Location Aware Scheduling Algorithm

in which ω is a constant value. The following equation finds minimized total processing time in all K scheduling plans.

$$T_{min} = min_{k \in K} T_k \tag{9}$$

If $T_{min} = T_k$, then plan k is the optimal plan. To make the workload balanced among these clusters, R_i^k values in plan k should be as close as possible. Therefore, smaller granularity of data partition leads to better chance of load balance.

The DLAS algorithm in Figure 3 is an approximation algorithm towards finding an optimal plan. In the current version of DLAS, a few assumptions are made, 1) the data partitions are relatively small in comparison to the whole dataset so that no further partitions to be made; 2) data replicas are randomly replaced among clusters; 3) no data transfer activities are allowed.

III. Use Cases and Preliminary Experiments

A. Compute-intensive applications

We apply the Hierarchical MapReduce programming model to the AutoDock [6] application. An AutoDock virtual screening consists of multiple AutoDock processes which are data independent. The paradigm of map-reduce and extensions to hierarchical map reduce is a good fit for AutoDock virtual screening, where Map, $Reduce$, and $GlobalReduce$ functions are implemented as follows:

1) The Map function takes a ligand to run the AutoDock binary executable against a shared receptor, and summarize binary executable output to record lowest energy value with a global intermediate key.
2) The $Reduce$ function takes all the energy values corresponding to the global intermediate key, sorts the values by the energy from low to high, and records the sorted results to a file with the global intermediate key.
3) The $GlobalReduce$ function takes all the values of the global intermediate key from the output of local $Reduce$ functions, sorts and combines them into a single file sort by increasing order of energy values.

An evaluation of AutoDock HMR appears in Luo et al. 2011 [4]. It was carried out on three clusters — Hotel and Alamo from the FutureGrid testbed and Quarry at Indiana University. 21 nodes were allocated from each cluster, within which one node is a dedicated master node (HDFS [7] namenode and MapReduce jobtracker) and other nodes which are data nodes and tasktrackers. The version of AutoDock used, 4.2, is the latest stable version at the time of writing. During the experiments, 6,000 ligands and 1 receptor are used. One of the configuration parameters that dominate the docking execution time and result accuracy is ga_num_evals — number of evaluations. Based on prior experiences, the ga_num_evals is typically set in a range of $2,500,000$ to $5,000,000$. We configure it to $2,500,000$. Considering that AutoDock is a CPU-intensive application, we set $\rho_i = 1$ per Section C.1) so that the maximum number of map tasks on each node is equal to the number of cores on the node. The CPU frequency was considered as major factor to set θ_i . So we set $\theta_1 = 2.93$ for Hotel, $\theta_2 = 2.67$ for Alamo, and $\theta_3 = 2$ for Quarry. We have calculated $\gamma_1 = \gamma_2 = \gamma_3 = 160$, given no MapReduce jobs are running beforehand. Thus, the weights are $W_1 = 0.3860$, $W_2 = 0.3505$, and $W_3 = 0.2635$ for Hotel, Alamo, and Quarry respectively. The dataset is also partitioned according to the new weight, which is $2,316$ Map tasks on Hotel, $2,103$ on Alamo, and $1,581$ on Quarry.

The result shows that the workloads are well balanced among these clusters (around 6000 seconds execution time in each cluster) and the total execution time is kept in minimum.

B. Data-intensive applications

For data-intensive applications, instead of transferring data explicitly from site to site, we explore using a shared file system called Gfarm [8] to share data sets among the Global Controller and local Hadoop clusters. Gfarm is a file-based distributed file system that federates local file systems on data nodes to maximize distributed file I/O bandwidth. A file in the Gfarm has multiple replicas stored in different data nodes. We wrote a MapReduce version of *grep* as a test case. The Map function captures the matching lines of input regular expressions. The $Reduce$ function collects all the matching lines from local Map output. The $GlobalReduce$ function collects all the output from local $Reduce$ and combines them into a single output file. The total execution time varies with different file distribution and different regular expression input.

Two virtual clusters (pragma-f0 and pragma-f1) were provisioned on the PRAGMA testbed [9]. Each cluster contains a virtual front node and 3 virtual compute nodes. All the virtual nodes are configured with 1 core of CPU, 1GB memory, and 80GB storage. Both of the virtual clusters are deployed as local MapReduce clusters and one of the virtual clusters (pragma-f0) is deployed as Global Controller. A Gfarm Metadata server is deployed also on pragma-f0 cluster. Both pragma-f0 and pragma-f1 are also configured as Gfarm data nodes), and Gfarm client. In our preliminary test, the input dataset contains 10 files. We generate each file with the size of 200MB, 400MB, 600MB, 800MB, 1GB respectively. These 10 files were evenly distributed between the two clusters with no replicas. The total execution time increases linearly from 311 seconds to 608 seconds when the size of input increases.

IV. Related Work

Elteir et al. [5] classifies MapReduce jobs into recursively reducible jobs and non-recursively jobs. Recursively reducible MapReduce has no inherent synchronization requirement between the map and reduce phases. Such jobs can be processed in hierarchical reduction or incremental reduction. However, their solution only support recursively reducible jobs and is restricted within a single cluster domain. The Map-Reduce-Merge [10] extends the MapReduce model with an additional Merge phase that merges output data from reduce modules. It aims to accomplish relational algebra operations over distributed heterogeneous datasets. However, it does not address scheduling issues and is not for multi-cluster environment.

Cardosa et al. [11], published the same day as our HMR paper, discusses problems running Hadoop jobs when data and computing plarform are widely distributed. They introduce distributed MapReduce which is similar to our solution. But the solution lacks a programming model and scheduling algorithms to support it. Sky Computing [12] provides end users a virtual cluster interconnected with ViNe [13] across different domains. It brings convenience by hiding the underlying physical clusters details. However, this transparency may cause an imbalanced workload where a job is dispatched over heterogeneous resources among different physical domains.

V. Conclusion and Future Directions

In this paper we present a hierarchical MapReduce framework that utilizes computation resources from multiple clusters simultaneously to run MapReduce job across them. Two scheduling algorithms are proposed and preliminarily evaluated using a high throughput application, AutoDock, and a simple Grep. There are several areas for improvement and extension that we will address in the remainder of the PhD work.

Scheduling Algorithms and Programming Model

The next step is to evaluate scheduling algorithms thoroughly under different conditions. The current version of DLAS scheduling algorithm does not consider data transferring among clusters, so does not perform well under conditions of an imbalanced replica distribution scenario. Moreover, if data partitions are relatively large in proportion of the whole dataset, DLAS fails to balance the workload without further partitioning the existing data partitions. Furthermore, CCAS and DLAS are static algorithms limited to prior knowledge without considering runtime change. To this end, possible improvements of scheduling algorithms should be made to address these issues.

The iterative MapReduce, which can be considered as a sub-class of MapReduce model, has yet to be investigated.

Applications

The AutoDock application and grep application provide useful initial experience to evaluate the HMR model. These two applications also need a more complete evaluation under the improvements of CCAS and DLAS algorithms. We are also investigating the suitability of HMR applied to ensemble runs. SLOSH is a NOAA developed storm surge model [14] that estimate storm surge heights resulting from historical, hypothetical, or predicted hurricanes by taking into account the atmospheric pressure, size, forward speed, and track data. The input of each SLOSH runs are separated from each other and the processes are embarrassingly parallel. A SLOSH job takes as many as 14,000 input data files, which requires 14,000 SLOSH runs to process. Researchers in our lab secured 120 worker roles from Windows Azure to process this job but so far only tested for 1,000 SLOSH runs. It is unlikely for us to allocate 10 times more worker roles due to financial and other considerations. HMR could potentially distribute these 14,000 instances to other resources such as PRAGMA Cloud, to meet the execution time constrains.

Hathi Trust Research Center (HTRC) [15] enables computational access for nonprofit and educational users to published works in the public domain. Ideally, HTRC collects all books from Hathi Trust Digital Library [16], which at the time of this writing, contains over 10 million volumes and 27% of them are in the public domain. Indexing these volumes requires huge amount of storage and large computation power that it could not being done within a single domain. Hierarchical Indexing can address these issues of time consuming index builds and can reduce search time as well.

Acknowledgment

Our thanks to Professors Geoffrey Fox and Judy Qiu at Indiana University, and Dr. Wilfred Li at University of California, San Diego, for their valuable feedback on our work. We also thank Dr. Peter Arzberger, Dr. Philip Papadopoulos and Cindy Zheng of the PRAGMA community for access to PRAGMA Cloud and help bringing up our IU PRAGMA node. We finally thank Yiming Sun, Zhenhua Guo, Peng Chen, Zong Peng and You-Wei Cheah, PhD students in School of Informatics and Computing at Indiana University for insightful discussions.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, January 2008.

[2] "FutureGrid," http://www.futuregrid.org.

[3] "Teragrid," http://www.teragrid.org.

[4] Y. Luo, Z. Guo, Y. Sun, B. Plale, J. Qiu, and W. W. Li, "A hierarchical framework for cross-domain mapreduce execution," in *Proceedings of the second international workshop on Emerging computational methods for the life sciences*, ser. ECMLS '11. New York, NY, USA: ACM, 2011, pp. 15–22.

[5] M. Elteir, H. Lin, and W.-c. Feng, "Enhancing mapreduce via asynchronous data processing," in *Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems*, ser. ICPADS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 397–405.

[6] G. M. Morris, R. Huey, W. Lindstrom, M. F. Sanner, R. K. Belew, D. S. Goodsell, and A. J. Olson, "Autodock4 and autodocktools4: Automated docking with selective receptor flexibility," *Journal of Computational Chemistry*, vol. 30, no. 16, pp. 2785–2791, 2009.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[8] O. Tatebe, K. Hiraga, and N. Soda, "Gfarm grid file system," *New Generation Computing*, vol. 28, pp. 257–275, 2010, 10.1007/s00354-009-0089-5.

[9] C. Zheng, D. Abramson, P. Arzberger, S. Ayyub, C. Enticott, S. Garic, M. J. Katz, J.-H. Kwak, B. S. Lee, P. M. Papadopoulos, S. Phatanapherom, S. Sriprayoonsakul, Y. Tanaka, Y. Tanimura, O. Tatebe, and P. Uthayopas, "The pragma testbed - building a multi-application international grid," in *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, ser. CCGRID '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 57–.

[10] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '07. New York, NY, USA: ACM, 2007, pp. 1029–1040.

[11] M. Cardosa, C. Wang, A. Nangia, A. Chandra, and J. Weissman, "Exploring mapreduce efficiency with highly-distributed data," in *Proceedings of the second international workshop on MapReduce and its applications*, ser. MapReduce '11. New York, NY, USA: ACM, 2011, pp. 27–34.

[12] K. Keahey, M. Tsugawa, A. Matsunaga, and J. Fortes, "Sky computing," *IEEE Internet Computing*, vol. 13, pp. 43–51, September 2009.

[13] M. Tsugawa and J. A. B. Fortes, "A virtual network (vine) architecture for grid computing," in *Proceedings of the 20th international conference on Parallel and distributed processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 148–148.

[14] B. Glahn, A. Taylor, N. Kurkowski, and W. A. Shaffer, "The role of the slosh model in national weather service storm surge forecasting," *National Weather Digest*, vol. 33, pp. 3–14, 2009.

[15] "Hathi Trust Research Center," http://www.hathitrust.org/htrc.

[16] "Hathi Trust Digital Library," http://www.hathitrust.org.