A Grid Based System for Data Mining Using MapReduce

Kelvin Cardona, Jimmy Secretan, Michael Georgiopoulos and Georgios Anagnostopoulos

***Abstract*—In this paper, we discuss a Grid data mining system based on the MapReduce paradigm of computing. The MapReduce paradigm emphasizes system automation of fault tolerance and redundancy, while keeping the programming model for the user very simple. MapReduce is built closely on top of a distributed file system, that allows efficient distributed storage of large data sets, and allows computation to be scheduled closely to this data. Many machine learning algorithms can be easily integrated into this environment.**

We explore the potential of the MapReduce paradigm for general large scale data mining. We offer several modifications to the existing MapReduce scheduling system to bring it from a cluster environment to a campus grid that includes desktop PCs, servers and clusters. We provide an example implementation of a machine learning algorithm (the Probabilistic Neural Network) in MapReduce form. We also discuss a MapReduce simulator that can be used to develop further enhancements to the MapReduce system. We provide simulation results for two new proposed scheduling algorithms, designed to improve MapReduce processing on the grid. These scheduling algorithms provide increased storage efficiency and increased job processing speed, when used in a heterogeneous grid environment. This work will be used in the future to produce a fully functioning implementation of the MapReduce runtime system for a grid environment, that will enable easy, data intensive parallel computing for machine learning, with little to no additional hardware investment.

***Index Terms*—MapReduce, Grid Computing, Distributed Storage, Probabilistic Neural Network**

I. Introduction and Motivation

There are many huge data sets in practice that would benefit from data mining. It is not abnormal for databases in the areas of health, genomics, astronomy, physics and engineering to be anywhere from gigabytes to terabytes. The need for efficient and effective models of parallel and grid computing is apparent.

While there are many systems that provide a framework for simple distributed computing in grid and P2P settings, as well as many systems that provide coordinated data storage across a grid, there are few systems that elegantly integrate both. Many data mining tasks, in comparison to many tasks in scientific computing, have a high ratio of data access to computation. This means that farming out certain data mining computations may not be worth the costs and time in bandwidth. Therefore, for data sets that are commonly processed several times, it makes sense to store them in a distributed fashion. In many data mining algorithms, the cost of accessing the data can rival or outstrip the cost of actually performing the processing. Therefore, accessing the data in parallel becomes increasingly important.

With the emphasis of grid research consistently falling on managing large disparate resources, it is easy to forget that all software interface enhancements are more often problems of human factors and economics than they are of computing. The programmer, like any other user, must have an interface that abstracts the complexity of the system while empowering the development of large scale applications. A tremendous lesson can be learned from the past 15 years of parallel computing. In a cluster setting, the choice many years ago was between MPI (Message Passing Interface) and PVM (Parallel Virtual Machine). While, in many ways, PVM had superior features for load balancing and fault tolerance, it could not match the simplicity of MPI. Therefore, MPI became the primary parallel computing toolkit for many supercomputing centers. Therefore, despite the extra features that PVM offered, MPI's simplicity won out. Now MPI is becoming less and less appropriate for computation distributed across the grid, and if a new paradigm is to be adopted, it must provide simplicity.

The MapReduce paradigm of parallel programming [1] can provide this necessary simplicity, while at the same time offering load balancing and fault tolerance. The Google File System (GFS) [2] that typically underlies a MapReduce system, can provide the efficient and reliable distributed data storage needed for applications involving large databases. The marriage of these systems, as is typical in their deployment, represents the necessary confluence of data distribution and parallel computation.

However, some challenges still remain for the MapReduce/GFS architecture. While the MapReduce/GFS combination is, by its nature, designed for dedicated cluster environments, its utility could be maximized if it were able to work in an environment where it could scavenge storage and computing resources in an organization, to allow non-trivial parallel computation. In this way, users could easily program and execute large scale machine learning tasks on huge databases, without incurring a significant investment on hardware. The assumptions under which current MapReduce/GFS systems are designed are the road blocks to fully achieving this goal. It is assumed that all nodes are about equally reliable. It is also assumed that all nodes are about equally as powerful. It is finally assumed that all nodes in the system are completely dedicated to processing. In this paper, we aim to relax some of these assumptions and begin to develop approaches that would

[1]This work was supported in part by the National Science Foundation grants IIS-REU-0647018, and IIS-REU-0647120. Kel Cardona is with the Department of Computer Engineering at the University of Puerto Rico. Jimmy Secretan and Michael Georgiopoulos are with the School of EECS at the University of Central Florida, Orlando, FL 32816. Georgios Anagnostopoulos is with the Florida Institute of Technology (FIT), Melbourne, FL 32792.

make more efficient use of a grid based system. We also aim to develop some additional algorithms in this environment to show its appropriateness for these large data mining tasks.

In section II, we discuss some necessary background that places MapReduce into the current context of parallel and distributed data mining. We also discuss some machine learning problems to which the MapReduce architecture has already been applied. In section III, we discuss the details of how a MapReduce system is implemented. We then describe the difficulties of using such a system in a heterogeneous campus grid environment. In section IV, we discuss the Probabilistic Neural Network (PNN), and provide a parallel implementation using MapReduce. In section V, we discuss some modifications that we have made to the typical MapReduce system, in order to make it more amenable to the grid. In section VI, we elaborate on a simulation that can evaluate our proposed changes. In section VII, we provide simulation results and associated explanations. In section VIII, we offer some conclusions and possible directions for future work. Finally, in the Appendix A we list the notation used throughout the paper; the notation is presented there in the order that it appears in the sections of the paper that follow.

II. Background

A significant body of research exists concerning parallel processing of machine learning applications, and more general processing and data storage in grid environments. We hope to explore each area in turn, and finally provide their confluence in the development of our own system. In the past 10 years, there has been a tremendous surge in grid computing concepts and potential applications. The systems in grids are often disparate, heterogeneous, and under different administrative control. Quite often, they are not dedicated processing resources, but are used for other tasks, and left to process tasks on the grid during idle times. These large networks of systems provide a powerful, inexpensive, already deployed resource that can be leveraged, if one has the right software. As our aim is large scale machine learning on the grid, we first discuss grid systems developed for large scale machine learning. Because these applications are typically predicated on the availability of huge databases, we first proceed by discussing developments in the area of large scale grid data storage. We then discuss MapReduce and Google File System (GFS), which represent the convergence of developments in cluster computing, fault tolerance and distributed storage. We will need to draw ideas from all of these technologies in order to develop systems that are capable of providing the ease of the MapReduce paradigm, while leveraging the computing and storage power of the grid.

A. Machine Learning and Data Mining on the Grid

The most straightforward tasks to adapt to a grid environment tend to be large grained tasks, such as processing entire data sets, or running full serial machine learning algorithms. In [3], authors describe the Grid enabled version of the popular Java based WEKA [4], machine learning algorithms. This toolkit allows training, cross-validation and testing of various data sets and WEKA algorithms distributed across the grid.

Sometimes, however, simply breaking up the data mining tasks at the coarsest level, as is done in Grid Weka and many other pieces of grid-based data mining software, may not be sufficient to provide the necessary speedup and utilize the available computing infrastructure. In [5] a system for data mining in Networks Of Workstations (NOWs) is developed. The system provides a simple primitive called distributed DOALL, which can be applied to loops, with no loop carried dependencies or conflicts. These kinds of loops are frequently encountered during in data mining. Compute servers receive and cache data from the data servers and receive tasks from the client.

B. Distributed Data Storage

Many systems are now taking fault tolerant techniques and applying them to store files on grid systems and networks of workstations (NOWs). In [6], a grid storage system is discussed. Built from hundreds to thousands of workstations in a grid storage domain, it accounts for dynamic shrinkage and growth of the storage pool by aggressively replicating files. The system is intended to accommodate large files that are written once and read many times. This is accomplished through the use of a fixed size "morsel", which is typically a 100MB segment of the data. The clients let the server know that they are available with the use of keep-alive messages, but do not need to notify the system when they leave the pool of available resources. Another system for grid storage in a network of workstations is discussed in [7]. The system, known as *Freeloader* aims to be used as a cache for large data sets, or scratch space for programs with considerable output requirements. The system uses similar requirements of heterogeneity and resilience to unreliable nodes, relying on heartbeat messages to evaluate the system status. The system is designed to minimize impact on the contributing nodes, by throttling system usage and bandwidth on the storage nodes. The authors evaluate several scheduling algorithms for this system, including one that schedules according to a past history of transfer speed.

C. GFS and MapReduce

The Google File System (GFS) [2] is used by Google's large scale cluster infrastructure. GFS aims to simplify the storage and processing of large files across large, but unreliable clusters of computers. The system can be scaled up to potentially thousands of machines. It is specialized to store multi-Gigabyte files which are mostly read and appended infrequently. This access method is common in large scale data mining applications. The design is simple and thus avoids much of the complications of other distributed storage designs [8]–[10]. The file system metadata is controlled centrally by a single *master-server*. The master-server keeps track of all of the files. Files that are large enough are split into smaller portions called *chunks*. In a typical GFS implementation, the chunks are 64MB in size. These chunks are distributed to *chunk-servers*, which may number in the hundreds, and are centrally controlled by the master-server. The chunks are replicated on different servers, a designated number of times to ensure that there is always a copy available. In a typical implementation, the chunks will be replicated three times.

MapReduce [1] is a simplified parallel program paradigm for large scale, data intensive parallel computing jobs. By constraining the parallel programming model to only the *map* function and the *reduce* function, the MapReduce infrastructure can greatly simplify the task of parallel programming. The fact that a parallel machine is involved is hidden from the programmer. The original MapReduce paper [1] gives several examples of applications easily adapted to the framework, including distributed text search and analysis of server logs. The map function first takes a list of keys and associated values, and then produces an intermediate set of keys and values. The MapReduce paradigm also includes an optional Combine step, to reduce the number of intermediate computations sent out. Finally, it reduces these intermediate values into a final result.

MapReduce and GFS together provide a system for large scale data storage and parallel processing of data intensive applications on large, unreliable, inexpensive cluster systems. The MapReduce/GFS combination represents a significant advancement in making large scale, parallel data mining easier and more accessible to programmers. It also represents an interesting new cluster paradigm. Clusters are often centered around shared, centralized storage. If one of the nodes requires data to process, it must receive it from the centralized storage, which can easily become a bottleneck for storage intensive applications. The centralized master server of the GFS is different, because it just handles data requests, and not the actual data. This allows the use of significant parallelism in the storage of data, while keeping the control of said storage simple and centralized. Because only very simple messages are being sent to and from the server, and not major data traffic, the server can easily process the requests of hundreds to thousands of computers.

MapReduce has already found its way into a few machine learning and data mining applications. In [11], the authors present versions of many different algorithms, adapted to a MapReduce form, including locally weighted linear regression, k-means, logistic regression, Naive Bayes, linear Support Vector Machines, independent component analysis, gaussian discriminant analysis, EM, and backpropagation. They discuss the performance for multi-core machines, and implement a simplified MapReduce system to support this, without the need of much of the complexity of the original MapReduce system. The authors of [12] use a large map reduce cluster to perform image clustering using a approximate nearest neighbor algorithm, aided by a data structure called a spill tree. MapReduce is actively used by Google, used in over 900 projects at Google in September of 2004 [1]. The MapReduce paradigm and implementation has already produced many practical outcomes. In [13], the authors develop algorithms for MinHash clustering and Expectation Maximization using MapReduce. They apply it to large scale, online collaborative filtering for suggesting interesting news stories to millions of users.

The original MapReduce software developed at Google is a proprietary system, and therefore, not available for public use. However, Hadoop [14], provides an open source implementation of the MapReduce paradigm. It does this on top of its own distributed file system based off of the GFS, known as the Hadoop Distributed File System (HDFS). This system is already used and researched by Yahoo, as well as at the core of the open source search engine known as Nutch.

In our work, we intend to enhance the MapReduce system itself, in order to make it fit for processing on a hetergeneous group of undedicated grid machines. We intend to show that many types of machine learning algorithms (including some that we are developing in addition to those in the literature) fit well and efficiently into this computing paradigm.

III. Basic Concepts

If our aim is to improve the applicability of a MapReduce/GFS implementation to a highly heterogeneous grid environment, we must first examine the details of how MapReduce and GFS work, in terms of how they communicate to store data and schedule tasks to be executed.

A. Details of MapReduce and GFS implementation

The GFS uses a simple client/server architecture to maintain and coordinate everything within the file system. The server is responsible for virtually all high level tasks. While this appears to represent a regression in terms of the advanced decentralized architectures that the last few years have brought, the authors cite this simplicity and centralization as advantage [1], emphasizing that the centralized architecture provides benefits for ease of implementation and simple scheduling. One of its most important functions is its interaction with any and all clients. It constantly interacts with clients providing contact to chunk-servers that would in turn provide service to each client. The master-server maintains regular communication with each of its chunk-servers, which are typically on the order of hundreds to thousands. The authors of [1] point out that, in typical applications, a single, reasonably powerful server is powerful enough to handle this many chunk servers. This is why the master-server is devoid of any sort of direct management of client data or its tasks. It handles file, chunk and chunk-server metadata, it administers file distribution and tasks status and assignment without acting as a middleman. This is due to the fact that such activities are time intensive and the file system demands the master-server to be available for the continuous service request from clients.

The chunk-servers handle client data directly, be it storage, transfer or any type of user implemented task such as map and reduce operations. They are by definition servers that keep the client data divided in pieces or chunks, which cannot exceed a specific maximum chunk size. This is innate to the GFS. The server typically stores a fixed number of copies of each chunk, which can be set on a per-file basis. To allow the master-server to know which computers are down and which are idle or busy, the master-server regularly pings each chunk-server to not only know if it is available or not but to gather other information about each of them like available space and the files and chunks that it holds.

The current MapReduce implementations model the nodes of a cluster by considering them as being organized into racks. These racks are typically on the order of fewer than 100 machines, all connected by some switch. These switches are in turn, connected to other switches that allow machines in different racks to communicate. This hierarchy is used for scheduling the chunks in such a way as to avoid failures caused by network equipment as well.

MapReduce is a means of easily implementing an algorithm to be processed in a parallel environment without needing to fully understand exactly how parallel computation proceeds. It is composed of two simple functions, the map and the reduce functions, that serve as an interface between the user and the file system. Through these two functions the user specifies what he wants the file system to process. The map function takes as input a set of a key and a value, designated as (k_1 and v_1), provided directly from the user defined input files. Within the function the user specifies, via his implementation, what to do with these keys and values. The map functions output another set of keys and values, designated as (k_2 and v_2). These map operations are distributed accordingly by the master-server to the proper chunk-servers and carried out. Meanwhile, the chunk-servers which have been assigned reduce tasks wait for any map to finish so they can go on to begin their work. After a suitable amount of maps are finished the file system begins to take the array of map outputs and sort them by key k_2. Within the reduce function the user implements the rest of his algorithm, "reducing" the array of v_2 values associated with k_2 to a final set of values v_3. These outputs will then be saved on to the file system.

The map reduce functions look as follows:

$$map(k_1, v_1) \rightarrow (k_2, v_2)[\,]$$

$$reduce(k_2, v_2[\,]) \rightarrow (k_2, v_3)[\,]$$

A diagram of a typical MapReduce/GFS architecture can be found in figure 1.

The original MapReduce paper [1] has a very simple yet effective example of a MapReduce implementation, the wordcount example. Suppose one needs to count the number of occurrences of each unique word in the file. Now it is desired that this non-parallel algorithm be run in parallel to count the word occurrences of several huge multi-gigabyte files. An example case is that we have a database that holds millions of forum posts from hundreds of sites and we want to count the number of posts that talk about a specific music band or a specific television show. With the MapReduce paradigm we can parallelize the algorithm by only implementing a Map and a Reduce function.

For the Map function, we are reading from a file or several files; in this case we would get as input a line of data from a file. We take this line and we divide it into the many words

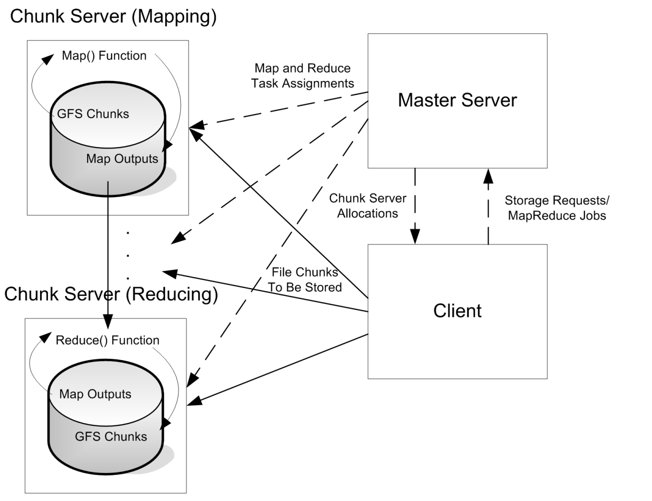

Fig. 1. The flow of data in a MapReduce/GFS architecture for file storage and MapReduce operations. Dashed lines indicate control messages, and solid lines indicate data transfer.

that it is made of and we submit each word as an output key and a number one as its value. Now the Reduce function will take as input a unique word as the key from the many output keys from the Map functions and a collection of 1's as the value. Each 1 was the value for the same output word in Map functions at different occurrences, which is now the Reduce's input key. Since each 1 is an occurrence of the same word we just need to sum them together to find the number of occurrences of that specific word. When the reduction is complete, we will have the list of words, with their associated occurrence frequency. See figure 2 for a graphic illustration of the word count example, and see algorithms 3 and 4 for the Map and Reduce pseudocode respectively. In this pseudocode, the *collect* function outputs intermediate map output to the local disk. The *emit* function outputs the final key value pairs to the GFS. These are standard functions of the MapReduce implementation.

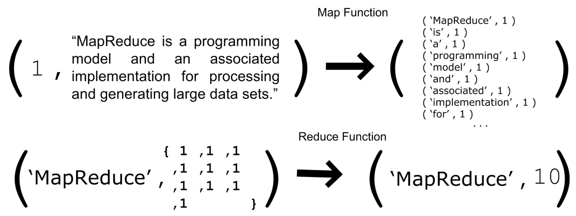

Fig. 2. An illustration of the word count map and reduce functions. The map function takes a numbered record and splits the record into a set of words with associated counts of 1. The MapReduce system then sorts by word, and counts the number of occurances, to produce the final output of word frequency.

B. Potential System Improvements

Existing MapReduce implementations are built for cluster environments. To use it on a grid that may include servers, other clusters and networks of workstations, some

```
    Input: Document portions with keys $k_1$ and text values $v_1$
    Output: Pairs with words as $k_2$ and 1's as $v_2$
1   foreach word $w \in v_1$ do
2       collect($w$, 1);
```

Fig. 3. Map function for the *WordCount* example

```
    Input: Words as $k_2$, with associated counts of $v_2 = 1$
    Output: Words as $k_2$, with associated frequency counts $v_3$
1   wordCount = 0;
2   foreach $v \in v_1$ do
3       wordCount += $v$;
4   emit(wordCount);
```

Fig. 4. Reduce function for the *WordCount* example

modifications to the architecture are needed to improve its performance. In this section, we explore which of the areas need modifications.

1) Availability: In [15], three different algorithms are presented for allocating distributed storage resources in a way that statistically provides an arbitrary amount of availability while at the same time minimizing the costs of the resources. We could adapt similar techniques to a GFS system in order to maintain an arbitrary availability while at the same time maximize our computational capability. GFS already uses the technique of replicating (in the parlance of [15], *redundant composition*) data blocks so that the data can be highly available. While this is likely very functional in a dedicated clustered environment, it is less so on an informal grid. In a grid, imagine a set of servers that stay on constantly as well as a set of workstations that are frequently turned off at night and frequently used during the day. If we wanted to set up a computational job on these machines, having duplicate copies of our data on the workstations will have less availability than having duplicate copies on the servers. If, for instance, all of the workstations were turned off at night, none of our duplicate data would be available. At the grid level, we have an obvious heterogeneity of host availability.

2) Heterogeneous CPUs: In a grid environment, there is also a great heterogeneity of host resources. Some resources will be simple workstations with small disks and slow processors, and some will be multi-core servers with terabytes of disk space. The MapReduce paradigm is still useful to simplify the implementation of grid-based machine learning algorithms; however, it must be modified to be useful in a campus grid environment. Also, whereas the cluster environment can be considered to be dedicated to the task of MapReduce processing, one can not count on such a situation in the grid. A large, powerful server, even if it has very powerful processors and a huge amount of storage space, may not be good to include in a run environment if the server is heavily loaded. Finally, balancing the distribution of chunks is not only important from the standpoint of balancing storage, but also from the standpoint of balancing processing of the chunks.

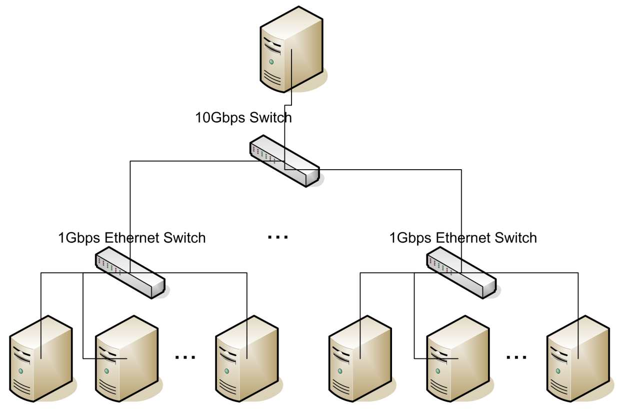

Fig. 5. A typical cluster configuration for an MapReduce/GFS System

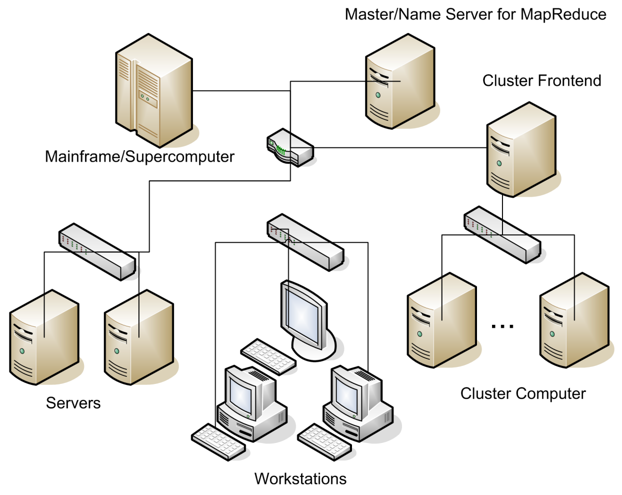

Fig. 6. A possible campus grid configuration for an MapReduce/GFS System

IV. Example Algorithm

A. PNN

The Bayes optimal classifier chooses as the most likely class that an observed datum $\mathbf{x}$ has come from, as the class that maximizes the datum's a-posteriori class probability. From Bayes' Theorem, we have that the a-posteriori probability that an observed datum $\mathbf{x}$ has come from class c_j is given by the formula:

$$p(c_j|\mathbf{x}) = \frac{p(\mathbf{x}|c_j)p(c_j)}{p(\mathbf{x})} \tag{1}$$

In order to make effective use of this formula, we must calculate the a-priori probabilities for each class, as well as the class conditional probabilities. The a-priori probability can be estimated directly from the training data, that is, $p(c_j) =$

$\frac{PT_j}{\sum_j PT_j}$ where PT_j designates the number of points in the training data set that are of class c_j . These class conditional probabilities ($p(\mathbf{x}|c_j)$) are calculated (see [16]), as follows:

$$p(\mathbf{x}|c_j) = \left(\frac{1}{(2\pi)^{D/2} \left(\prod_{i=1}^{D} \sigma_i \right) PT_j} \right) \sum_{r=1}^{PT_j} exp \left(- \sum_{i=1}^{D} \frac{(x(i) - X_r^j(i))^2}{2(\sigma_i)^2} \right) \quad (2)$$

where D is the dimensionality of the input patterns (data), PT_j represents the number of training patterns belonging to class c_j, $\mathbf{X}_r^j$ denotes the r-th such training pattern, $\mathbf{x}$ is the input pattern to be classified, and σ_i is the smoothing parameter along the ith dimension.

The Probabilistic Neural Network classifier (PNN) classifier (introduced by Specht [17]) chooses as the most likely class that an observed datum has come from the datum's maximum a-posteriori probability (as Bayes does), but in PNN's case the a-priori class probabilities and class conditional probabilities are calculated using the approximate formulas proposed above (these formulas rely on the available training data).

The PNN's training phase is virtually non-existent; it merely needs to store the training data in memory so it can be later used in the testing phase. Though the training phase time and computational cost are negligible the same can not be said for its test phase. Because the training phase is not comprised of any computational labor, the task of actually processing the data and obtaining the class conditional probabilities is placed on the testing phase. Each class conditional probability is calculated by acquiring the distance between the testing pattern and every training pattern that belongs to that specific class. The negative distances are substituted in the exponent of an exponential function and then these exponential function values are summed together. Once each class conditional probability is determined, its product with the class a-priori probability is calculated, and then the class with the largest such product value (proportional to the class's a-posteriori probability) is returned as the most likely class for that testing point. The pseudocode for the standard serial PNN version is given in 7 and a neural network architecture that implements these PNN steps is given in 8. In figure 8, we see that each dimension of the test point is presented at the first layer. This is passed to the pattern layer, where each training point, $\mathbf{X}_r^j$, is represented as its own neuron. These generate the exponential of the squared Euclidean distance between $\mathbf{x}$ and each point in $\mathbf{X}_r^j$. These pattern neurons are connected to the summation neuron that corresponds to the class of that pattern. Therefore, there is a summation neuron for each class. The output neuron chooses the class with the highest summation output, giving a class for the pattern. Figure 7 presents an easier to understand but computationally equivalent algorithm for computing the class conditional probabilities in the PNN classifier.

Adapting a parallel PNN to MapReduce form actually is quite straightforward. The wrapper around the program takes in a set of testing points and iterates through them one at a time. The MapReduce portion should then be between a single testing point and multiple training points. The map function should calculate a distance between the training and testing point and collect that result with the class as the key and the distance as the value. In the reduce, the exponentials of each of these distances should be summed over each class. Finally, outside of the map and reduce functions, the most likely class

```
1 foreach $c_j$ do
2     foreach $\mathbf{X}_r^j$ do
3         $CCP_j$ +=
          $exp\left(-\sum_{i=1}^{D}(\mathbf{x}(i) - \mathbf{X}_r^j(i))^2/(2\sigma_i^2)\right)$ ;
4 foreach $CCP_j$ do
5     $CCP_j$ /= $(2\pi)^{D/2} PT_j \prod_{i=1}^{D} \sigma_i$ ;
6 $C(\mathbf{x}) = \text{argmax}_j\{CCP_j\,(PT_j/PT)\}$;
```

Fig. 7. Pseudo-code for the PNN algorithm (serial version)

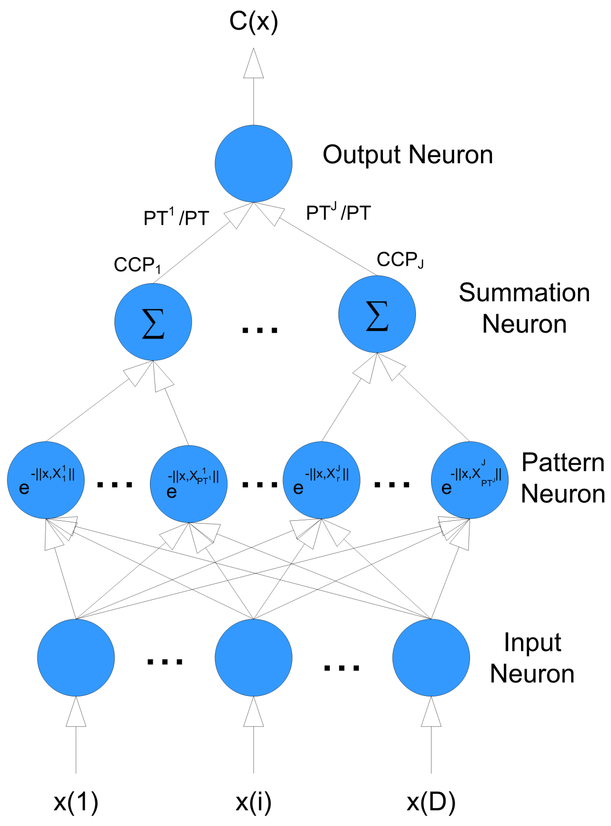

Fig. 8. The PNN Neural Network Architecture. Each dimension of the test point is presented at the first layer. Every dimension of the test point is connected to each neuron of the pattern layer. The pattern layer is where each training point, $\mathbf{X}_r^j$, is represented as its own neuron. These generate the exponential of the squared Euclidean distance between $\mathbf{x}$ and each point in $\mathbf{X}_r^j$. These pattern neurons are then connected to the summation neuron that corresponds to the class of that pattern. Therefore, there is a summation neuron for each class. The output neuron chooses the class with the highest summation output, giving a class for the pattern.

c_j is selected. The map and reduce functions for the parallel PNN are given in figures 9 and 10 respectively.

```
Input: Training points with $k_1 = r$ and $v_1 = \mathbf{X}_r^j$ (a $D$ dimensional vector)
Output: Distances between the query point $\mathbf{x}$ and each $\mathbf{X}_r^j$. $v_2 = dis(\mathbf{x}, \mathbf{X}_r^j)$ and the $k_2 = c_j$ (distances indexed by class $c_j$)
1 foreach $\mathbf{X}_r^j$ do
2     collect($c_j$, $exp\left(-\sum_{i=1}^{D}(\mathbf{x}(i) - \mathbf{X}_r^j(i))^2/(2\sigma_i^2)\right)$);
```

Fig. 9. Map function for the parallel PNN

```
Input: $k_2 = c_j$, with associated distances, $v_2 = dis(\mathbf{x}, \mathbf{X}_r^j)$
Output: $k_2 = c_j$, and $v_3$ is the sum of the distances from $v_2$.
1 $CCP_j = 0$;
2 foreach $v \in v_2$ do
3     $CCP_j$ += $v$;
4 emit($CCP_j$);
```

Fig. 10. Reduce function for the PNN

V. System Modification

We propose two different scheduling algorithms to help highly heterogeneous grid environments maintain more consistent availability and better load balancing. These are modifications of simply how the blocks are stored, when they are passed from the client to the chunk servers, or stored by the chunk servers in the GFS. To begin with, we will examine the default algorithm. In the default algorithm, each chunk C is replicated r times on a set of chunk servers $\mathbf{S}$. If there is just one replica it is stored as close as possible to the chunk server that wrote it, or, if a client wrote it, it is stored randomly. Recall that nodes in a MapReduce cluster are typically arranged in racks. This is taken into account in the scheduling to allow the system to recover from network errors as well. Therefore, the second replica is stored on a different rack from the first one, for greater reliability. The next replica is stored on a node in the same rack as the first. The pseudocode for the algorithm is listed in 11.

When storing the first replica, the standard algorithm tries to optimize execution in a series of related computations by storing the output of a file locally, if the client writing the output file is one of the chunk servers. If the client writing out the file is not a chunk server (i.e. if it is a user client, storing a large file for the first time), then a random rack is chosen for the replica. Then the scheduling algorithm makes an effort to store the replica on a different rack. By storing copies on different racks, the system can mitigate errors caused

```
1 foreach $C$ do
2     switch $r$ do
3         case 1
4             Store $C$ on a random $S_i$ or locally if a chunk server
5         case 2
6             Store $C$ on a random $S_i$, and then on a $S_j$ from another rack
7         case 3
8             Store $C$ on a random $S_i$, and then on a $S_j$ from another rack, on a random $S_k$ where $S_k$ and $S_i$ are on the same rack
9         case $r \geq 4$
10            Store $C$ on a random $S_i$, and then on a $S_j$ from another rack, on a random $S_k$ where $S_k$ and $S_i$ are on the same rack. Store all remaining $r - 3$ replicas on random servers $S_l$
```

Fig. 11. Original GFS Data Block Scheduling algorithm.

by failures of the switches in charge of the racks. If the whole rack becomes inaccessible from a network error that does not effect the other racks, then the system will continue to have access to the necessary blocks. In the third replica allocation, an effort is made to keep the third replica close to the first one. This reflects the fact that the system aims to minimize process across racks, and maximize the processing within racks. This is because, as seen in figure 5, there is less available network bandwidth between clusters than within clusters. Finally, other replicas locations are chosen randomly in an attempt to spread the load.

As we had specified before, because availability can be wildly different on a grid than on a cluster environment, this can create a situation of not having access to a particular block. For instance, consider the availability of a chunk $A(C)$. Its availability is equal to the probability that all of the servers holding it are not available. It can be expressed in the following equation:

$$A(C) = 1 - \prod (1 - A(S_i)) \tag{3}$$

That is, the probability of chunk C being available is equal to one minus the probability that all servers are simultaneously unavailable.

To compare the situation in a cluster and a grid, imagine both having three replicas of a chunk ($r = 3$). In the dedicated cluster, the systems can have an availability on the order of 99%. Therefore, $A(C) = 0.999999$ or 99.9999% availability (not including other factors like the network and the master server). However, suppose the same chunk is also replicated on three workstations. They are busy a good portion of the day, and are therefore available about half of the time $A(S) = 0.5$. A chunk replicated on these workstations would only have availability of $A(C) = 0.875$ which means that 12.5% of the time, the data is not accessible. Clearly, in order to make a grid-based MapReduce system useful, we would need to mitigate these effects.

We can do this through monitoring and intelligent scheduling. Suppose that when we received heartbeat signals from chunk servers, we logged them in the master server. We could then use this information to estimate the availability of the chunk servers by dividing the number of received heartbeats over the total number of expected heartbeats. If we assume that this is a reasonable indicator of availability, we can use this information to more intelligently schedule the data blocks for availability. The user can then specify a requested availability for his data blocks A_{req}. The first pass at this is simple. First, we sort the list in descending order, by availability. Then, for every chunk, we choose in a round robin fashion, enough servers S_i to ensure the necessary availability. We call this algorithm *Availability Sorted Round Robin* (ASRR) and we present the algorithm in 12.

```
S' = sortByAvailability(S);
$current = 0$;
foreach $C$ do
    while $A(C) \leq A_{req}$ do
        if $S_{current}$ has enough space then
            Store $C$ on $S_{current}$
        $current$++;
```

Fig. 12. Availability Sorted Round Robin algorithm.

We may also be interested in balancing chunks with respect to load in a heterogeneous grid environment. Because the scheduling of Map tasks will have an affinity for chunk servers with portions of the data to be processed, it makes a difference where we schedule the chunks. Just because a low powered desktop may have 200GB of free storage, does not mean that it is an equally good place to schedule a chunk as a high-end server with 200GB of free storage. Therefore, in addition to ensuring that each chunk is stored with the required availability, we would also like to try to make sure that the chunk can be processed as quickly as possible. Consider $P(S_i)$ to be the processing power of the chunk server S_i. We would prefer to have chunks of the file scheduled on S_i, in proportion to its speed. This will achieve a good balance of the processing load. Therefore, we take the same round robin approach to meeting availability as in ASRR. However, we introduce the ability to skip over servers that are loaded more so than their processing power would allow. We call the algorithm *Processor Aware Round Robin* and present the pseudocode in figure 13. Assume that N is the total number of chunks in the system and $N(S_i)$ is the number of chunks stored in server S_i.

The loop tests not only to see if the given server has enough space, but if its processing power divided by its load in terms of chunks, is greater than the average of this number. This

```
S' = sortByAvailability(S);
$P_{total} = \sum P(S_i)$;
foreach $C$ do
    while $A(C) \leq A_{req}$ do
        if $S_{current}$ has enough space And
        $P(S_{current})/(N(S_{current}) + 1) \geq$
        $P_{total}/(|S| + N)$ then
            Store $C$ on $S_{current}$
        $current$++;
```

Fig. 13. Processor Aware Round Robin algorithm.

will show a preference for scheduling on high performance machines, with the possible tradeoff of including more servers than necessary to ensure the specified availability.

VI. MapReduce Simulation

In order to evaluate the use of our new scheduling algorithms on a large grid of computers, we utilize a simulation. A simulation offers many benefits over actual implementations. It is quite easy to test new algorithms and execute those tests. Instead of taking the trouble to set up several computers, they can be simulated with only a couple of lines of code. The simulation also allows testing on many different types of grids, and at scales that would not be practical to seek out or setup from scratch.

To build this simulation, there are many already available libraries that can support this type of testing. GridSim [18] is a toolkit that allows modeling and simulation of entities in parallel and distributed computation. It is written in Java programming language on top of SimJava, a discrete event simulation package [19]. GridSim comes equipped with nearly everything you might need to incorporate into your simulation. It has ready to use, comprehensive implementations of any virtual component you might need for a grid simulation but it also includes abstract component modules that you can use to implement other more specific and complex units.

We have constructed a simulator of the MapReduce/GFS system that is able to store files and process MapReduce tasks. The simulator takes a list of chunk server properties, along with their processing power in MFLOPS, their available storage space and their availability. It will then simulate the process of having the master server, break up large files and store them in chunks on the GFS with the specified chunk servers. It will also then execute simulated map reduce tasks, specified in terms of their computational complexity and storage requirements.

Figure 14 shows the interaction between the simulated client, master server and chunk servers. The client passes a file handle and size to the master that determines of how to allocate the chunks. The client then sends these chunks to the appropriate chunk servers. In figure 15 the distribution of map tasks is shown. The client hands the master server a job, which is broken into tasks and sent to the appropriate chunk servers.

In figure 16 shows the reduce tasks being assigned to a few of the chunk servers while the rest of the chunk servers with map outputs being transferred to those reduce servers.

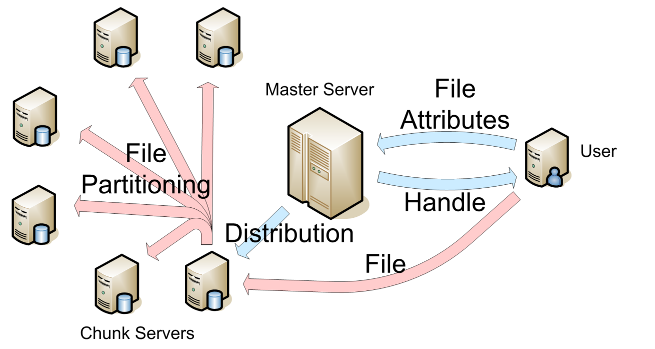

Fig. 14. Simulator actions when storing a file from the client. The red arrows represent data transfers and the blue arrows represent control messages. The user requests that the master server allocate space for the file to be stored. The master decides how the file will be partitioned. Chunks of the file are then sent from the user to the first chunk server, and then pipelined throughout the chain to distribute the replicas.

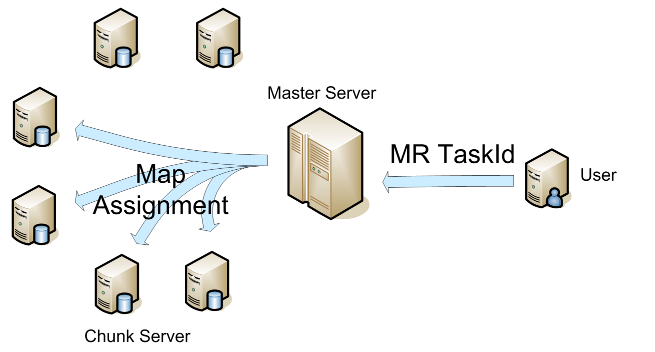

Fig. 15. Simulator actions for a Map task. The blue arrows represent control messages. The user sends the MapReduce job to the master to be completed. The master then splits the tasks accordingly, and sends the task assignments to the chunk servers, aiming to keep the computations as close to the stored data as possible.

A. SETI@Home Data

In order to build a large simulated grid, we need realistic information about the systems involved. Ideally, we would like information like processing power, storage space and memory and availability for real grid systems. Luckily, such a source of potential data exists. In [20] the authors provided some statistical summaries of the host attributes for hosts that participate in the SETI@Home project. The SETI@Home project is a large scale distributed computing project, that allows radio telescope data to be processed on idle home computers. The SETI@Home project keeps a large amount of constantly updated data on all of the computers involved. If we use parameters for systems that we find in these data, it is likely to give us a good cross section of real systems that are currently in use, therefore enabling us to more faithfully estimate how well our grid system would perform in a real environment.

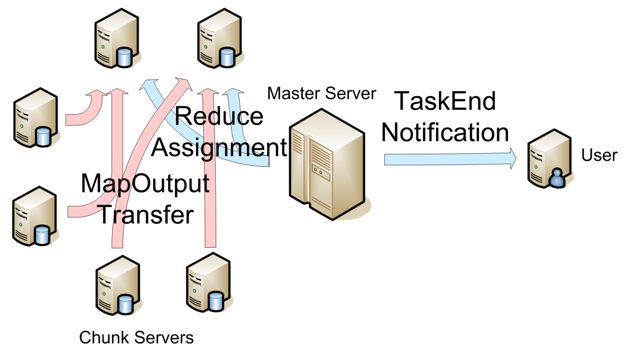

Fig. 16. Simulator actions for a Reduce task. The red arrows represent data transfers and the blue arrows represent control messages. The master assigns the reduce tasks, to be done, to the chunk servers. The chunk servers with intermediate map outputs that are to be processed by reduce functions, send their outputs to the appropriate chunk servers, which have been assigned the reduce tasks.

This is most advantageous compared to randomly generating simulated computer information.

Unfortunately, while [20] provides an excellent summary of typical computer load and availability for a wide swath of systems, there is no host availability data provided on per-host basis by the publicly available host database. Therefore, using summary statistics provided in [20], we attempted to generate some reasonable values. The paper [20] gives three important variables to calculate the overall availability of the host. The first variable is $on\_fraction$. This is the fraction of time that the SETI@Home client runs. The next variable of interest is $connected\_fraction$. This is the amount of time that the client program has access to an Internet connection. Finally, there is $active\_fraction$ which indicates how much of the time that the client is allowed to run. This is because the client is set to stay inactive when the host PC is otherwise busy. For our application we consider the average availability to be the product of these three variables:

$$\begin{aligned}\overline{A} &= (on\_fraction)(connected\_fraction)(active\_fraction)\\ &= (0.81)(0.83)(0.84) = 0.5647\end{aligned}$$

Using this average as the average of a Gaussian random distribution with a standard deviation of 0.1, availability values were generated and paired with randomly selected hosts from the SETI@Home host database.

VII. Results

In figure 17 we show the the amount of replication in the system versus the average availability of the constituent chunks. If we can provide some amount of availability with few average replicas, we can achieve greater storage efficiency. As expected, our algorithms Availability Sorted Round Robin (ASRR) and Processor Aware Round Robin (PARR) are able to beat the standard GFS allocation algorithm in terms of storage efficiency. For instance, the ASRR algorithm, for an availability of 0.9389, requires only 2.662 average replicas per chunk, compared to 3 replicas providing availability of

0.9153 in the standard algorithm. This means that the ASRR algorithm, in this instance, provides 12.7% greater storage efficiency, which results in a significant storage increase. For every availability value, the ASRR and PARR algorithms need fewer replicas, likely due to their explicit accounting for availability. Each of the algorithms will stop creating replicas of a chunk when it is calculated by the algorithm that there are enough replicas to meet the availability requirements. As also can be expected, the ASRR algorithm outperforms the PARR in terms of storage efficiency, because the ASRR algorithm only aims to minimize replicas while the PARR algorithm aims to minimize replicas and maximize processing power.

In figure 18 we show the completion time of different map tasks of various sizes executed on simulated systems where the three algorithms have been used to allocate the blocks. The PARR algorithm, because it explicitly looks to schedule with the fastest servers, provides a significant speed advantage. A map task, requiring 1000 MIPS/MB (millions of instructions per megabyte), over 2048 chunks of data (131GB) on data scheduled by the PARR algorithm will take 474.58 seconds, as opposed to the same map task run with the standard data scheduling, which will take 2728.32 seconds. However, it will take about 17% as long when data is scheduled with the PARR algorithm, compared to the standard algorithm, because it will make an effort to spread the data blocks to more powerful servers. Because the ASRR algorithm does not account for processing speed, it more closely follows the time taken by the standard scheduling algorithm. The tradeoff of slightly less storage efficiency in the PARR algorithm is more than compensated by its better scheduling of processing.

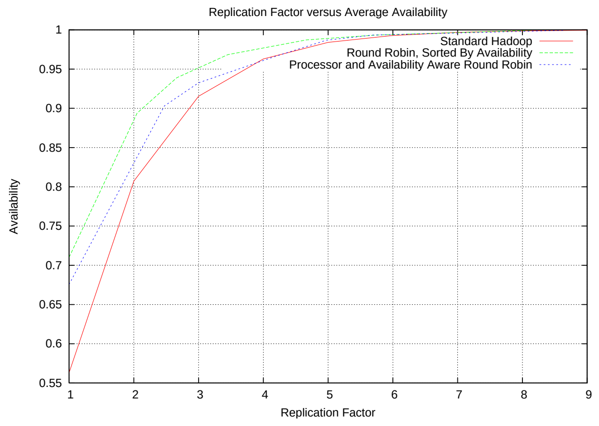

Fig. 17. Replication versus Average Availability of Chunks. For this graph, the system was asked to store 1000MB (1GB) file. For the standard algorithm, standard replication factors where used, where for the ASRR and PARR algorithms, arbitrary specified A_{req} requested availability values were used. This shows that the ASRR and PARR algorithms can provide greater availability by using fewer replicas (thereby providing storage more efficiently).

VIII. Conclusions and Future Work

In this work, we aimed to enumerate and modify some of the assumptions in the MapReduce architecture that make it appropriate for a cluster architecture, but not as appropriate for a heterogeneous grid environment. To this end, we developed two new algorithms for scheduling data blocks to increase storage and processing efficiency. We also developed a simulator for the MapReduce system that will allow us to test more sophisticated algorithms for scheduling of data replicas, and map/reduce tasks.

Our simulator has proven useful in showing that our new scheduling algorithms, ASRR and PARR, can provide better balancing and availability in a grid based MapReduce/GFS environment. The ASRR was able to meet availability requirements with greater storage efficient than the standard scheduling algorithm. PARR was able to increase the processing speed of map operations, while also remaining more efficient for a given availability than the standard algorithm. Because of this work, it will prove possible to fully adapt the MapReduce system to a less reliable, more heterogeneous grid environment. With deployed implementation of the new scheduling algorithms presented here, users will be able to perform large scale machine learning computation, with a minimum of parallel programming effort, and additional hardware. We have shown that MapReduce/GFS implementations, with some appropriate modifications, can support simple, efficient parallel programming on a complex collection of machines.

For future work, we will perform more extensive testing with the newly developed scheduling algorithms, on a greater variety of simulated grids and on a system with a variety of loads. Thereafter, we will make some small modifications to the Hadoop software, in order to evaluate our algorithms in a real system on an installed test grid.

Useful extensions to our system will include a fully interactive interface for administering data mining analysis, similar to the one used in [4]. We would also like to explore the possibility of using the MapReduce paradigm in a larger scale,

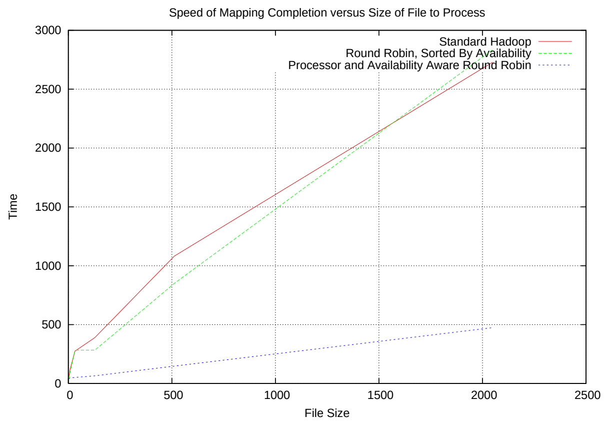

Fig. 18. File Size versus Time to Complete. For this graph, Map tasks using files of different numbers of 64MB chunks were executed, using the standard and the two new data scheduling algorithms (ASRR and PARR) to place the replicas of the data. This measures the time taken by a map task requiring 1000MIPS/MB of input data (1000 million instructions per megabyte of input data). The PARR algorithm can provide consistently faster executions by trying to schedule data to faster chunk servers first.

peer-to-peer (P2P) computing system, potentially involving hundreds of thousands or millions of chunk servers.

There are also many extensions to the original GFS and MapReduce systems that Google has presented, which may be further applicable to generalized data mining. The authors of [21], present another stage to the MapReduce computation, known as merge. This can be used to help a MapReduce system make use of relational queries, providing an invaluable resource for on-line analytical processing (OLAP). In [22], a system called BigTable is presented, which provides a row and column accessible table, that is capable of versioning the contents of cells. It has been used for tasks ranging from the batch oriented ones for which MapReduce and GFS were originally developed, to latency dependent web serving to users. This table oriented structure would most certainly be useful in OLAP and data mining applications. In [23], a system called Sawzall is discussed, which is implemented on top of the MapReduce libraries. Sawzall is an interpreted language for providing easy, parallel record based data processing. It greatly reduces the code required for many basic operations on the data by a factor of 10–20 over C++. Extensions like this may allow the development of complex data mining systems even more quickly.

Appendix A. Notation

Here we summarize relevant notation from the paper.

TABLE I
NOTATIONS FOR ANALYSIS, FROM SECTION III

Notation	Description
k_1	The keys to describe the input records to the MapReduce process. These may be as simple as line numbers in a file.
k_2	Key used by the output of the mapping functions as well as the final output in a MapReduce process.
v_1	The value of the input record for the MapReduce process. For instance, this could be a line of text data.
v_2	The values which are output by the map function, and then sorted by key k_2.
v_3	The final output, produced by the reduce function.
$map(k_1, v_1)$	A function that takes input records and keys, and outputs an array of interemediate key value pairs (k_2, v_2).
$reduce(k_2, v_2[])$	A function that takes an intermediate key k_2 with an array of associated values v_2, and produces the final map reduce output consisting of an array of (k_2, v_3) pairs.

TABLE II
NOTATIONS FOR ANALYSIS, FROM SECTION IV

Notation	Description
S	The set of training data for the classification problem.
D	The dimensionality of data points in the classification problem, agreed upon by all parties.
PT	The number of points in the training set S.
c_j	The jth class of the classification problem of interest; $j = 1...J$.
PT_j	The number of points in the training set belonging to class c_j
J	The number of classes in the classification problem.
$\mathbf{X}_r^j$	The rth training point from S that is of class c_j.
$\mathbf{x}$	A D dimensional test point in the classification problem of interest.
$\mathbf{x}(i)$	The ith component of the training point $\mathbf{x}$; $i = 1...D$
$p(c_j\|\mathbf{x})$	The a-posteriori probability that an observed point $\mathbf{x}$ comes from class c_j
$p(\mathbf{x}\|c_j) = CCP_j$	The class conditional probability that an observed point $\mathbf{x}$ is of class c_j
$p(c_j)$	The a-priori probability that a datum in the classification problem is of class c_j.
$\sigma = (\sigma_1, \sigma_2, ..., \sigma_D)$	A vector of parameter values used by the PNN algorithm to estimate the class conditional probabilities for the classification problem of interest.
PT_j^k	The number of points in training sub-set set S^k of class c_j.
$\mathbf{X}_r^{k,j}$	The rth training point from the set S^k that is part of class c_j.
$\mathbf{X}_r^j(i)$	The ith component of the training point $\mathbf{X}_r^j$; $i = 1...D$
$C(\mathbf{x})$	The class that PNN algorithm predicts for the test point $\mathbf{x}$.
CCP_j	The class conditional probability of test pattern $\mathbf{x}$, given that it has come from class c_j; $CCP_j = \sum_{k=1}^{K} CCP_j^k$.

TABLE III
NOTATIONS FOR ANALYSIS, FROM SECTION V

Notation	Description
C	A file chunk. A unique portion of a file that needs to be stored in the system.
S	The set of all chunk servers, on which chunk replicas can be stored.
r	The number of replicas specified to be stored per chunk C, in the context of the original replica scheduling algorithm.
$A(C)$	Availability of a file chunk C. This is the portion of the time that the chunk can be accessed (i.e. there is a server S_i that is able to provide it) divided by the total time. This gives the probability that the chunk can be used.
$A(S_i)$	The measured availability of chunk server S_i. This is approximated by the server by dividing the total number of heartbeats received by the total number of heartbeats expected. This gives the probability that the chunk server is available for processing.
A_{req}	Availability for a file, specified by the user (pertains to the ASRR and PARR algorithms).
N	The total number of unique chunks stored in the entire system.
$N(S_i)$	The total number of unique chunks stored in chunk server S_i

TABLE IV
NOTATIONS FOR ANALYSIS, FROM SECTION VI

Notation	Description
$on\_fraction$	The fraction of the time a system is turned on.
$connected\_fraction$	The fraction of the time a system is connected to the network.
$active\_fraction$	The fraction of the time users actively enable the volunteer computing program to run.
$\overline{A}$	Average availability, used to generate the availability values for the chunk servers. Defined to be product of $(on\_fraction)$ $(connected\_fraction)$ $(active\_fraction)$

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *In Proceedings of OSDI'04: Sixth Symposium on Operating System Design and Implementation*, December 2004.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *In Proceedings of 19th ACM Symposium on Operating Systems Principles*, October 2003.

[3] R. Khoussainov, X. Zuo, and N. Kushmerick, "Grid-enabled weka: A toolkit for machine learning on the grid," 2004.

[4] I. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*. San Francisco: Morgan Kaufmann, 2005.

[5] S. Parthasarathy and R. Subramonian, "Facilitating data mining on a network of workstations," in *Advances in Distributed and Parallel Knowledge Discovery*. AAAI Press, 2000.

[6] S. Vazhkudai, "On-demand grid storage using scavenging." in *PDPTA*, 2004, pp. 554–560.

[7] S. Vazhkudai, X. Ma, V. Freeh, J. Strickland, N. Tammineedi, T. Simon, and S. Scott, "Constructing collaborative desktop storage caches for large scientific datasets," *ACM Transactions on Storage (To Appear)*, 2006.

[8] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen, "Ivy: A read/write peer-to-peer file system," in *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, 2002.

[9] F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters," in *Proceedings of the First USENIX Conference on File and Storage Technologies*, January 2002, pp. 231–244.

[10] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: A scalable distributed file system," in *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997, pp. 224–237.

[11] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *In the Proceedings of NIPS 19*, 2006.

[12] T. Liu, C. Rosenberg, and H. A. Rowley, "Clustering billions of images with large scale nearest neighbor search," in *WACV '07: Proceedings of the Eighth IEEE Workshop on Applications of Computer Vision*. Washington, DC, USA: IEEE Computer Society, 2007, p. 28.

[13] A. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: Scalable online collaborative filtering," in *In the Proceedings of World Wide Web Conference*, 2007.

[14] "Welcome to hadoop!" http://lucene.apache.org/hadoop/, 2007.

[15] J. Secretan, M. Lawson, and L. Boloni, "Brokering algorithms for composing low cost distributed storage resources," in *In Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA)*, June 2007.

[16] E. Parzen, "On estimation of probability density function and mode," *Annals of Mathematical Statistics*, vol. 33, pp. 1065–1073, 1962.

[17] D. F. Specht, "Probabilistic neural networks," *Neural Networks*, vol. 3, pp. 109–118, 1990.

[18] R. Buyya and M. Murshed, "Gridsim: a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing," *Concurrency and Computation: Practice and Experience*, vol. 14, pp. 1175–1220, 2002.

[19] F. Howell and R. McNab, "Simjava: a discrete event simulation package for java with applications in computer systems modelling," in *Proceedings of the First International Conference on Web-based Modelling and Simulation*, January 1998.

[20] D. P. Anderson and G. Fedak, "The computational and storage potential of volunteer computing," in *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, May 16-19 2006, pp. 73–80.

[21] H. Yang, A. Dasdan, R. Hsiao, and D. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *In Proceedings of the 2007 ACM SIGMOD international Conference on Management of Data*, June 11–14 2007, pp. 1029–1040.

[22] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *In the Proceedings of OSDI'06: Seventh Symposium on Operating System Design and Implementation*, November 2006.

[23] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Scientific Programming Journal*, vol. 13, no. 4, pp. 227–298, 2005.