FREERIDE-G: Supporting Applications that Mine Remote Data Repositories*

Leonid Glimcher* Ruoming Jin† Gagan Agrawal*
*Department of Computer Science and Engineering
Ohio State University, Columbus OH 43210
{glimcher,agrawal}@cse.ohio-state.edu
† Department of Computer Science
Kent State University, Kent OH 44242
{jin}@cs.kent.edu

ABSTRACT

Analysis of large geographically distributed scientific datasets, also referred to as *distributed data-intensive science*, has emerged as an important area in recent years. An application that processes data from a remote repository needs to be broken into several stages, including a data retrieval task at the data repository, a data movement task, and a data processing task at a computing site. Because of the volume of data that is involved and the amount of processing, it is desirable that both the data repository and computing site may be clusters. This can further complicate the development of such data processing applications.

In this paper, we present a middleware, FREERIDE-G (FRamework for Rapid Implementation of Datamining Engines in Grid), which support a high-level interface for developing data mining and scientific data processing applications that involve data stored in remote repositories. Particularly, we had the following goals behind designing the FREERIDE-G middleware: 1) Support high-end processing, i.e., use parallel configurations for both hosting the data and processing the data, 2) Ease use of parallel configurations, i.e., support a high-level API for specifying the processing, and 3) Hide details of data movement and caching.

We have evaluated our system using three popular data mining algorithms and two scientific data analysis applications. The main observations from our experiments are as follows. First, FREERIDE-G is able to scale the processing extremely well when the number of data server and compute nodes are scaled evenly. Second, when only the number of compute nodes are scaled, our target class of applications achieve modest additional speedups. Finally, for applications that involve multiple passes on the dataset, caching remote data provides significant improvement.

1. INTRODUCTION

Traditionally, the focus in computational sciences has been on developing algorithms, implementations, and enabling tools to facilitate simulation of physical processes and phenomena. However, as our ability to collect, store, and distribute huge amounts of data increases with advancing technology, analysis of large datasets can also provide useful insights into physical processes.

With increasing computational power, it is now feasible to simulate processes at a greater scale. This, however, creates a challenge in analysis of data that is being generated. As scientific simulations can generate very large volumes of data, analyzing this data is becoming increasingly hard.

Analysis of large geographically distributed scientific datasets, also referred to as *distributed data-intensive science* [5], has emerged as an important area in recent years. Scientific discoveries are increasingly being facilitated by analysis of very large datasets distributed in wide area environments. Careful coordination of storage, computing, and networking resources is required for efficiently analyzing these datasets. Even if all data is available at a single repository, it is not possible to perform all analysis at the site hosting such a shared repository. Networking and storage limitations make it impossible to down-load all data at a single site before processing.

Thus, an application that processes data from a remote repository needs to be broken into several stages, including a data retrieval task at the data repository, a data movement task, and a data processing task at a computing site. Because of the volume of data that is involved and the amount of processing, it is desirable that both the data repository and computing site may be clusters. This can further complicate the development of such data processing applications.

In this paper, we present a middleware, FREERIDE-G (FRamework for Rapid Implementation of Datamining Engines in Grid), which support a high-level interface for developing data mining and scientific data processing applications that involve data stored in remote repositories. Particularly, we had the following goals behind designing the FREERIDE-G middleware:

Support High-End Processing: Parallel configurations, including clusters, are being used to support large scale data repositories. Many data mining applications involve very large datasets. At the same time, data mining tasks are often compute-intensive, and parallel computing can be effectively used to speed them up [25]. Thus, an important goal of the FREERIDE-G system is to enable efficient processing of large scale data mining computations. It supports use of parallel configurations for both hosting the data and processing it.

Ease Use of Parallel Configurations: Developing parallel data mining applications can be a challenging task. In a distributed environment, resources may be discovered dynamically, which means that a parallel application should be able to execute on a variety of parallel systems. Thus, one of the goals of the FREERIDE-G system is to support execution on distributed memory and shared memory systems, as well as on cluster of SMPs, starting from a common high-level interface.

Hide Details of Data Movement and Caching: A major difficulty in developing applications that involve remote data is appropriate staging of remote data, and possibly caching when feasible and appropriate. FREERIDE-G is designed to make data movement and caching transparent to application developers.

FREERIDE-G has been developed as a set of services for data server nodes and compute nodes. The services at the data server node includes those for data retrieval, data distribution, and data communication. The services at compute nodes include those for data communication, data retrieval, data caching, and parallel execution for generalized reductions, starting from a high-level API.

We have evaluated our system using three popular data mining algorithms and two scientific data analysis applications. The main observations from our experiments are as follows. First, FREERIDE-G is able to scale the processing extremely well when the number of data server and compute nodes are scaled evenly. Second, when only the number of compute nodes are scaled, our target class of applications achieve modest additional speedups. Finally, for applications that involve multiple passes on the dataset, caching remote data provides significant improvement.

As part of our ongoing work on FREERIDE-G, we are currently developing a resource selection framework. This will involve the use of performance models for choosing computational resources, and also integration of FREERIDE-G with grid resource frameworks. However, this paper only focuses on the existing services at data and compute servers.

2. RELATED WORK

Several groups have been developing support for grid-based data mining. One effort in this area is from Cannataro *et al*. [20, 21]. They present a structured Knowledge Grid toolset for developing distributed data mining applications through workflow composition. Brezanny *et al.* [16, 2, 18] have also developed a GridMiner toolkit for creating, registering and composing datamining services into complex distributed and parallel workflows. Ghanem *et al.* [6, 8] have developed Discovery Net, an application layer for providing grid-based services allowing creation, deployment and management of complex data mining workflows. The goal of DataMiningGrid, carried out by Stankovski *et al.* [23], is to serve as a framework for distributed knowledge discovery on the grid.

There are significant differences between these efforts and our work. These systems do not offer a high-level interface for easing parallelization and abstracting remote data extraction and transfer. We believe that FREERIDE-G is able to reduce the time required for developing applications that perform remote data analysis. On the other hand, our system is not yet integrated with Grid standards and services.

Jacob *et al.* have created GRIST [14], a grid middleware for astronomy related mining. This effort, however, is very domain specific, unlike FREERIDE-G, which has been used for a variety of data mining and scientific analysis algorithms.

The work presented here builds directly on top of the previously published work on the FREERIDE system and parallelization of scientific and data mining applications [17, 11]. This work, however, focused entirely on processing data stored locally.

3. SYSTEM OVERVIEW AND FUNCTIONALITY

This section presents the overall design of the middleware. In subsection 3.1 we outline the middleware design, as well as a brief description of its components. In subsection 3.2 the middleware API is presented.

3.1 System Design

This subsection describes the overall design of our middleware. The basic functionality of the system is to automate retrieval of data from remote repositories and coordinate parallel analysis of such data using end-user's computing resources, provided an inter-connection exists between the repository disk and the end-user's computing nodes. This system expects data to be stored in chunks, whose size is manageable for the repository nodes.

This middleware is modeled as a *client-server* system. Figure 1 shows the three major components, including the *data server*, the *compute node client*, and a *resource selection framework*. As we stated earlier, the resource selection framework is part of our ongoing work on FREERIDE-G, and is beyond the scope of this paper.

The data server runs on every on-line data repository node in order to automate data delivery to the end-users processing node(s). More specifically, it has 3 roles:

1. *Data retrieval:* data chunks are read in from repository disk.
2. *Data distribution:* each data chunk is assigned a destination – a specific processing node in the end-user's system.
3. *Data communication:* after destination assignment is made in the previous step, each data chunk is sent to the appropriate processing node.

A compute server runs on every end-user processing node in order to receive the data from the on-line repository and perform application specific analysis of it. This component has 4 roles:

1. *Data Communication:* data chunks are delivered from a corresponding data server node.
2. *Data Retrieval:* if caching was performed on the initial iteration, each subsequent pass retrieves data chunks from local disk, instead of receiving it via network.
3. *Computation:* Application specific data processing is performed on each chunk.
4. *Data Caching:* if multiple passes over the data chunks will be required, the chunks are saved to a local disk.

The current implementation of the system is configurable to accommodate N data server nodes and M user processing nodes between which the data has to be divided, as long as $M \geq N$. The reason for not considering cases where $M < N$ is that our target applications involve significant amount of computing, and cannot effectively process data that is retrieved from a larger number of nodes.

The configuration illustrated in Figure 1 presents a setup with $N = 2$ data servers and $M = 4$ compute nodes. Active Data Repository (ADR) [3, 4] was used to automate the data retrieval parts of both components.

3.2 Middleware Interface

FREERIDE-G API is based on the observation that a number of popular data mining and scientific data processing algorithms share a relatively similar structure. Their common processing structure is essentially that of *generalized reductions*. The popular algorithms where this observation applies include apriori association mining [1], k-means clustering [15], k-nearest neighbor classifier [12] and artificial neural networks [12]. During each *phase* of these algorithms, the computation involves reading the data instances in an arbitrary order, processing each data instance, and updating elements of a *reduction object* using associative and commutative operators.

In a distributed memory setting, such algorithms can be parallelized by dividing the data items among the processors and replicating the

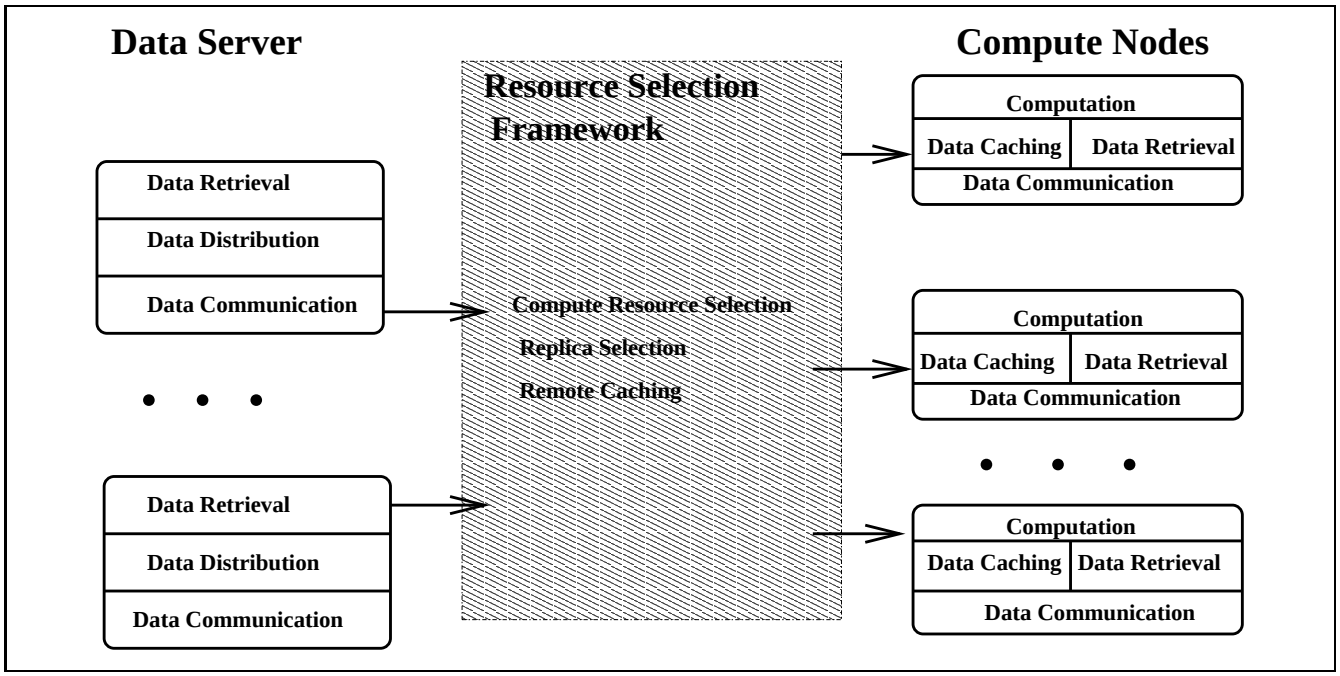

Figure 1: FREERIDE-G System Architecture

reduction object. Each node can process the data items it owns to perform a local reduction. After local reduction on all processors, a global reduction can be performed. In a shared memory setting, parallelization can be done by assigning different data items to different threads. The main challenge in maintaining the correctness is avoiding race conditions when different threads may be trying to update the same element of the reduction object. We have developed a number of techniques for avoiding such race conditions, particularly focusing on the memory hierarchy impact of the use of locking. However, if the size of the reduction object is relatively small, race conditions can be avoided by simply replicating the reduction object.

The middleware API for specifying parallel processing of a data mining algorithm is simplified since we only need to support generalized reductions. The following functions need to be written by the application developer using our middleware.

The Subset of Data to be Processed: In many cases, only a subset of the available data needs to be analyzed for a given data mining task. These can be specified as part of this function.

Local Reductions: The data instances or chunks owned by a processor and belonging to the subset specified are read. A local reduction function specifies how, after processing one chunk, a *reduction object* (declared by the programmer), is updated. The result of this processing must be independent of the order in which the chunks are processed on each processor.

Global Reductions: The reduction objects on all processors are combined using a global reduction function.

Iterator: A parallel data mining application often comprises of one or more distinct pairs of local and global reduction functions, which may be invoked in an iterative fashion. An iterator function specifies a loop which is initiated after the initial processing and invokes local and global reduction functions.

4. SYSTEM IMPLEMENTATION ISSUES

This section describes a number of implementation issues in our middleware system. The main issues are: managing and communicating remote data, load distribution, parallel processing on compute nodes, and caching of remote data.

4.1 Managing and Communicating Remote Data

As we stated in the previous section, data is organized as chunks on remote repositories, using an existing ADR middleware. The processing of data is organized in *phases*. In each phase, a generalized reduction is performed on the computing nodes. Because of the property of reductions, the order of retrieving, communicating, and processing data elements does not impact the correctness.

At the beginning of each phase, the compute nodes forward the information on the subset of the data to be processed to data server. The data server determines the chunks of the data that need to be retrieved, as well as a schedule for retrieving these on each data server node.

Initially, let us suppose that the number of data server nodes equals the number of compute nodes. In such a scenario, each data server node forwards all the chunks it retrieves to a single compute node. The support for declustering of chunks in ADR helps maintain a good balance, even with such a simple scheme. The corresponding data server and compute nodes coordinate when the next chunk should be communicated, and also the size of the buffer that needs to be allocated on the compute node. In our current implementation, stream socket mechanism was used for all such communication.

4.2 Load Distribution

Data mining and scientific processing applications are often compute-intensive. In such cases, they can benefit from a configuration where the number of compute nodes is larger than the number of data server nodes. However, in such cases, careful load distribution must be performed.

We again use a simple mechanism. Each data server node now com-

municates its chunks to M compute nodes. The value M is the smallest value which will still enable load balance on each compute node. A hash function (mod) based on a unique chunk id is used to distribute the retrieved chunks among the M compute nodes a data server node is communicating with.

4.3 Parallel Processing on Compute Nodes

One of the major advantages of FREERIDE-G is its ability to support parallel processing on compute nodes. As we discussed in the previous section, the processing is specified using a high-level API, which particularly targets generalized reduction. Parallel execution is also simplified because we focus only on reduction computations.

After data has been distributed between different computing nodes, each node can execute initial processing and local reduction functions on the data items it owns. After each invocation of local reduction function, local copies of reduction objects on each node are broadcasted to one node, which performs the global reduction. If the size of the reduction object is large, the global reduction phase can also be parallelized, i.e. portions of reduction objects are received by each node, and each node performs a part of the global reduction. After global reduction, the reduction object is broadcasted to all nodes, which then continue with the next pass of the mining algorithm. The communication required for gathering and broadcasting the reduction object is facilitated by the middleware.

4.4 Caching

If an iterative mining application needs to take more than a single pass over the data, reading the data from the remote location on every iteration is redundant. For such applications, data chunks belonging to a certain compute node can be saved onto the local disk, provided sufficient space. Such *caching* is performed during the initial iteration, after each data chunk is communicated to its compute node by the data server and the first pass of application specific processing has been completed.

Each chunk is written out to the compute node's disk in a separate file, whose name is uniquely defined by the chunk id. These filenames are also indexed by the chunk ids, speeding up retrieval for the subsequent iterations. The benefit of such caching scheme is evident: for an application requiring P passes over the data, the last $P-1$ iterations will have the data available locally on the compute node. Since each round out data communication from the server would have to perform retrieval in order to send the data, the total number of retrievals does not change. Instead, for iterations subsequent to the initial one, data retrieval is performed on the compute node.

5. APPLICATIONS

In this section we describe the applications that we have used to carry out the experimental evaluation of our middleware. We have focused on three traditional datamining techniques: k-means clustering [13], EM clustering [7], k-nearest neighbor search [12], as well as two scientific feature mining algorithms: vortex analysis [19] and molecular defect detection [22].

5.1 k-means Clustering

The first data mining algorithm we describe is the k-means clustering technique [13], which is one of the most popular and widely studied data mining algorithm. This method considers data instances represented by points in a high-dimensional space. Proximity within this space is used as criterion for classifying the points into clusters.

Three steps in the sequential version of this algorithm are as follows:

1. Start with k given centers for clusters,
2. Scan the data instances. For each data instance (point), find the center closest to it, assign this point to a corresponding cluster, and then move the center of the cluster closer to this point, and
3. Repeat this process until the assignment of the points to cluster does not change.

This method can be parallelized as follows. The data instances are partitioned among the nodes. Each node processes the data instances it owns. Instead of moving the center of the cluster immediately after the data instance is assigned to the cluster, the *local sum* of movements of each center due to all points owned on that node is computed. A *global reduction* is performed on these local sums to determine the centers of clusters for the next iteration.

5.2 Expectation Maximization Clustering

The second data mining algorithm we have used is the Expectation Maximization (EM) clustering algorithm [7], which is one of the most popular clustering algorithms. EM is a distance-based based algorithm that assumes the data set can be modeled as a linear combination of multivariate normal distributions. The goal of the EM algorithm is to use a sequence of Expectation and Maximization steps to estimate the means C, the covariances R, and the mixture weights W of a Gaussian probability function. The algorithm works by successively improving the solution found so far. The algorithm stops when the quality of the current solution becomes stable, which is measured by a monotonically increasing statistical quantity called *loglikelihood*.

This algorithm can be parallelized in the following manner. The input data instances (the array Y) are distributed between the nodes. The arrays C, R, and W, whose initial values are provided by the user, are replicated on all nodes. The E step is carried out on each node, using data instances local to it. *Global combination* involved in the E step consists of the information necessary to compute the means and mixture weights arrays being aggregated by the master node, and then being re-broadcasted. Next, the M step is performed locally on each node's data instances. Information necessary to compute covariance is then updated during the M step, through an aggregation step followed by a re-broadcast.

At the end of any iteration, each node has an updated value for C, R, W and llh, and the decision to execute or abort another iteration is made locally.

These parallelization steps can be expressed easily using the FREERIDE-G API described earlier in this paper [9].

5.3 k-Nearest Neighbor Search

k-Nearest neighbor classifier is based on learning by analogy [12]. The training samples are described by an n-dimensional numeric space. Given an unknown sample, the k-nearest neighbor classifier searches the pattern space for k training samples that are closest, using the euclidean distance as measure of proximity, to the unknown sample.

Again, this technique can be parallelized as follows. The training samples are distributed among the nodes. Given an unknown sample, each node processes the training samples it owns to calculate the k-nearest neighbors *locally*. After this local phase, a global reduction computes the overall k-nearest neighbors from the k-nearest neighbor on each node.

5.4 Vortex Detection Algorithm

Vortex detection is the first of the two scientific data processing applications we have used. Particularly, we have parallelized a fea-

ture mining based algorithm developed by Machiraju *et al.*. A more detailed overview of the algorithm is available in a recent publication [24]. The key to the approach is extracting and using *volumetric regions* to represent features in a CFD simulation output.

This approach identifies individual points (*detection step*) as belonging to a feature (*classification step*). It then aggregates them into regions. The points are obtained from a tour of the discrete domain and can be in many cases the vertices of a physical grid. The sensor used in the detection phase and the criteria used in the classification phase are physically based point-wise characteristics of the feature. For vortices, the detection step consists of computing the eigenvalues of the velocity gradient tensor at each field point. The classification step consists of checking for complex eigenvalues and assigning a swirl value if they exist. The aggregation step then defines the region of interest (ROI) containing the vortex. Regions insignificant in size are then eliminated, and the remaining regions are sorted based on a certain parameter (like size or swirl).

Parallelizing this application requires the following steps [11]. First, when data is partitioned between nodes, an overlap area between data from neighboring partitions is created, in order to avoid communication in the detection phase. Detection, classification and aggregation are first performed locally on each node, followed by *global combination* that joins parts of a vortex belonging to different nodes. Denoising and sorting of vortices is performed after the final aggregation has been completed.

5.5 Molecular Defect Detection Algorithm

The second of the two scientific data processing applications we have used performs molecular defect detection [22]. More specifically, its goal is to uncover fundamental defect nucleation and growth processes in Silicon (*Si*) lattices, either in the presence of thermal sources or extra atoms (e.g., additional *Si* atoms or dopants such as Boron). A detection and categorization framework has been developed to address the above need. Both phases (each consisting of multiple sub-steps) are briefly described below.

Phase 1-Defect Detection: In this phase the atoms are marked as defect atoms based on statistical rules and are then clustered to form one or more defect structures.

Phase 2–Defect Categorization: This phase consist of two sub-steps. The first step, which is computationally inexpensive, uses a kNN classifier to provide the system with a set of candidate defect classes. The second step tries to match the candidate classes from first step using a relatively expensive exact shape matching algorithm. If no class matches a particular defect, the class database is updated to include the *newly discovered* class.

The first phase of this algorithm can be parallelized [10] in a manner very similar to vortex detection algorithm. Contiguous chunks of *Si* grid are partitioned between nodes, and defects are first detected and aggregated locally, and followed by defects spanning multiple nodes being joined in the *global combination* stage. After all aggregation has been completed, defects are re-distributed to their original nodes, in order to improve load balancing for the second phase.

Next, all defects that have a match in the class database are categorized locally. Non-matching defects are given temporary class assignments and then used to update local catalog copy. Local catalogs are then merged in the *global combination* step, and the merged copy is then re-broadcasted to the processing nodes in order to finalize temporary class assignments.

6. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of our middleware. We use the five data analysis applications described in Section 5. Several different datasets, of varying sizes, were used for each of these. We had the following goals in our experiments:

1. Studying parallel scalability of applications developed using FREERIDE-G. Here, we focused on configurations where the numbers of compute and data repository nodes are always equal.

2. Investigating how the computing can be scaled, i.e., performance improvements from increasing the number of compute nodes independent of the number of data server nodes.

3. Evaluating the benefits of performing caching in applications that require multiple passes over data.

For efficient and distributed processing of datasets available in a remote data repository, we need high bandwidth networks and a certain level of quality of service support. Recent trends are clearly pointing in this direction. However, for our study, we did not have access to a wide-area network that gave high bandwidth and allowed repeatable experiments. Therefore, all our experiments were conducted within a single cluster. The cluster used for our experiment comprised 700 MHz Pentium machines connected through Myrinet LANai 7.0. In experiments involving caching, the communication bandwidth was simulated to be 500 KB/sec and 1 MB/ sec. In our future, we will conduct experiments with geographically distributed clusters.

6.1 Evaluating Overall System Scalability

The number of compute nodes used for these experiments was always equal to the number of data repository nodes. In this situation pair-wise correspondence between data and compute nodes can be established, and no distribution of data to multiple compute nodes is required from the data server. All scalability experiments were conducted on up to 16 nodes (8 data and compute node pairs).

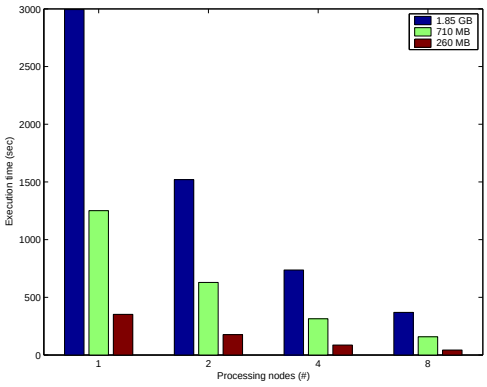

Figure 2: Vortex detection application parallel performance on 1.85 GB, 710 MB, and 260 MB datasets.

Vortex detection was evaluated with three datasets, with size of 260 MB, 710 MB, and 1.85 GB, respectively. Figure 2 presents the execution times from these three data sets on 1, 2, 4, and 8 pairs of nodes. On 2 pairs of nodes, the speedups are 1.99 for the 260 MB dataset, 1.98 for 710 MB dataset, and 1.97 for the 1.8 GB dataset. This demonstrates that distributed memory parallelization is working very well, resulting in nearly perfect speedups. Speedups are good even for the smallest dataset, where execution time is expected to be mostly dominated by

the parallelization overhead. Also, since data communication overhead is kept relatively low, communication time scales as well with data size as data retrieval and analysis times.

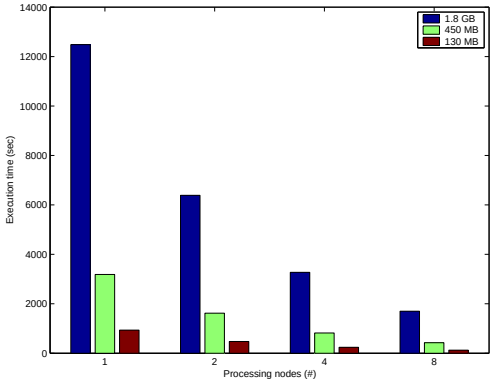

Figure 3: Defect detection application parallel performance on 1.8 GB, 450 MB, and 130 MB datasets.

On 4 pairs of nodes, the speedups are 3.99 for the 260 MB dataset, 3.98 for 710 MB dataset, and 3.96 for the 1.8 GB dataset. On 8 pairs of nodes, the speedups are 7.95 for the 260 MB dataset, 7.92 for 710 MB dataset, and 7.90 for the 1.8 GB dataset.

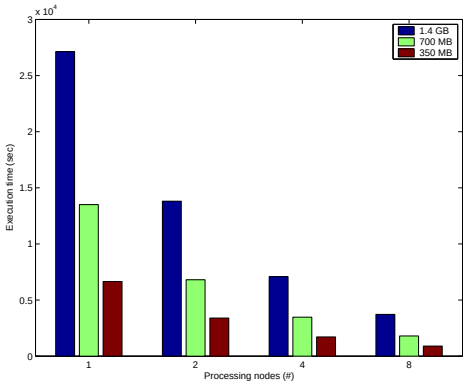

Figure 4: Expectation Maximization clustering parallel performance on 1.4 GB, 700 MB, and 350 MB datasets.

Figure 3 presents execution parallel execution times for the molecular defect detection algorithm. This application was evaluated on three datasets of sizes 130 MB, 450 MB, and 1.8 GB. On 2 pairs of nodes, the speedups in execution time were 1.97 for the 130 MB dataset, 1.97 for the 450 MB dataset and 1.96 for the 1.8 GB dataset. Again, near perfect speedups demonstrate good parallelization efficiency.

On 4 pairs of nodes, the speedups were 3.92 for the 130 MB dataset, 3.89 for the 450 MB dataset and 3.82 for the 1.8 GB dataset. The drop-off in speedups here demonstrates that the overhead associated with communication between compute nodes that is required for defect detection is not as small as that for vortex detection. But, with parallel efficiency somewhat limited by the application itself, the speedups are still very good. On 8 pairs of nodes, the speedups are 7.52 for the 130 MB dataset, 7.50 for the 450 MB dataset and 7.34 for the 1.8 GB dataset.

Figures 4, 5, and 6, present execution times from the additional scalability experiments that were conducted. EM clustering, k-means clustering, and k-nearest neighbor search were evaluated on three datasets of size 350 MB, 700 MB, and 1.4 GB.

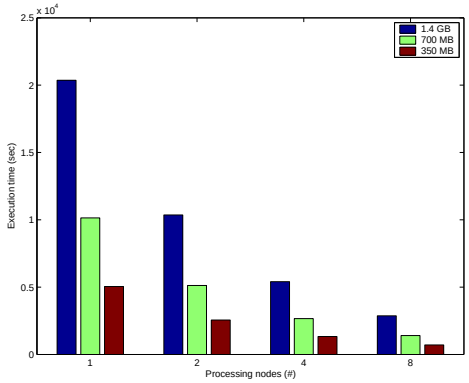

Figure 5: K-means clustering parallel performance on 1.4 GB, 700 MB, and 350 MB datasets.

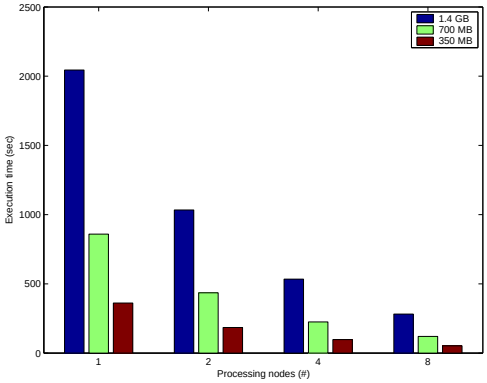

Figure 6: k-nearest neighbor search parallel performance on 1.4 GB, 700 MB, and 350 MB datasets.

On 8 pairs of nodes, parallel EM achieved speedups of 7.56 for 350 MB dataset, 7.49 for 700 MB dataset and 7.30 for 1.4 GB dataset. In the same configuration, parallel k-means achieved speedups of 7.25 for 350 MB dataset, 7.21 for 700 MB dataset and 7.10 for 1.4 GB dataset. Parallel k-nearest neighbor search, executed on 8 pairs of nodes, achieved speedups of 7.26 for 350 MB dataset, 7.15 for 700 MB dataset and 6.98 for 1.4 GB dataset.

Results were once again consistent with those of the previous two experiments. Parallel efficiency observed was high, although in some cases limited by the application. Data retrieval, communication and processing all demonstrated good scalability with respect to increasing both the problem size and the number of compute nodes.

6.2 Evaluating Scalability of Compute Nodes

In processing data from remote repositories, the number of available nodes for processing may be larger than the number of nodes on which data is hosted. As we described earlier, our middleware can support processing in such configurations. In this subsection, we evaluate the performance of applications in such cases.

We used three of the five applications, i.e., defect detection, vortex detection and k-nearest neighbor search, for these experiments. Unlike the other two applications (k-means and EM clustering), each of these three applications only take a single pass (of retrieval and com-

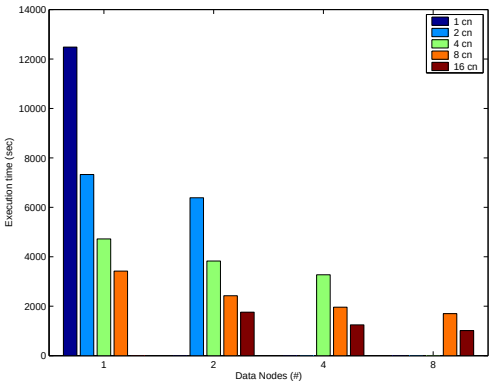

Figure 7: Defect detection parallel performance as the number of compute nodes is scaled (1.8 GB dataset).

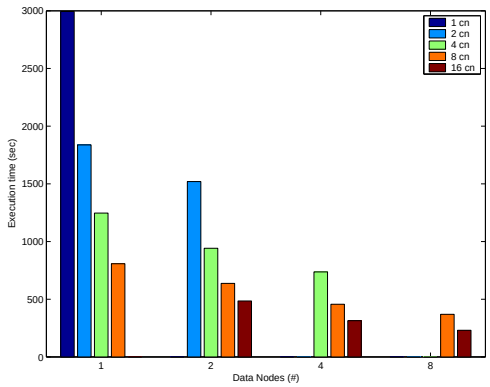

Figure 8: Vortex detection parallel performance as the number of compute nodes is scaled (1.85 GB dataset).

munication) over the data. So, any change in performance achieved by the middleware would be due to each data node distributing processing work to multiple compute nodes, and not due to caching.

Among the datasets used in the experiments in the previous subsection, we report results from only the largest ones. The number of data nodes was varied up to 8 and the number of compute nodes was varied up to 16 for each experiment. While both numbers were restricted to be powers of two to achieve perfect load balance, nothing in the middleware implementation requires such restriction.

Figure 7 presents parallel defect detection execution times on a 1.8 GB dataset, as the number of both data nodes and compute nodes was varied. Using a single compute node, the speedups achieved were 1.70 for two compute nodes, 2.64 for four, and 3.65 for eight. The speedups are sub-linear because only the data processing work is being parallelized, with data retrieval and communication tasks remaining sequential. However, these experiments do show that in cases where additional compute nodes are available, our middleware can use them to obtain further speedups, even if these speedups are sub-linear.

Using two data nodes, the additional speedups achieved were 1.67 for four compute nodes, 2.63 for eight, and 3.63 for sixteen. With four data nodes, the speedups were 1.67 for eight compute nodes, and 2.62 for sixteen. And, finally, using 8 data and 16 compute nodes, the speedup was 1.67. These results demonstrate that a very decent speedup can be achieved by using twice as many compute nodes as data nodes, but as the number of compute nodes keeps increasing, a drop off in parallel efficiency is to be expected.

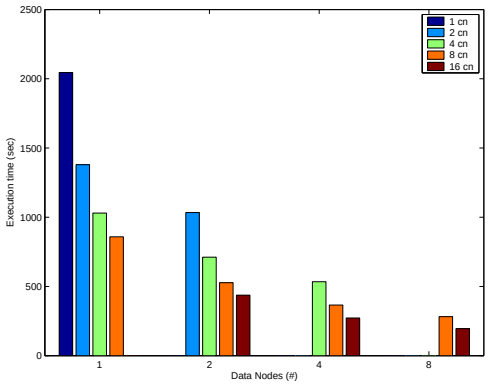

Figure 9: k-nearest neighbor search parallel performance as the number of compute nodes is scaled (1.4 GB dataset).

Figure 8 presents parallel vortex detection execution times on a 1.85 GB dataset. Again, number of both data and compute nodes is varied. Using a single data node, the speedups achieved were 1.63 for two compute nodes, 2.40 for four, and 3.61 for eight. Again, speedups are sub-linear because only a fraction of execution time has been parallelized. In fact, a larger fraction of time is spent on data retrieval in the vortex detection application, resulting in slightly lower speedups. Using two data nodes, the additional speedups are 1.61 for four compute nodes, 2.39 for eight, and 3.14 for sixteen. The lower speedup of the last configuration is attributed to parallelization overhead starting to dominate over execution time. With four data nodes, the speedups achieved were 1.61 for eight data nodes, and 2.35 for sixteen. And, finally, using 8 data and 16 compute nodes, the speedup was 1.60. These results are consistent with the defect detection experiment, only indicating a slightly higher tendency for vortex detection to be "I/O bound."

Figure 9 presents parallel execution times for k-nearest neighbor search evaluated on the 1.4 GB dataset. Once again, number of data and compute nodes is varied. Using a single data node, the speedups achieved were 1.48 on two compute nodes, 1.98 on four, and 2.38 on eight. This indicates that the fraction of time spent on data retrieval is even higher for this application. Again, as a larger fraction of execution time remains sequentialized, the speedup decreases. With two data nodes, the additional speedups achieved are 1.45 on four compute nodes, 1.96 on eight, and 2.36 on sixteen. These results are consistent with previous experiments with both this application and other applications. Using four data nodes, the speedups achieved are 1.46 on eight compute nodes, and 1.96 for sixteen. Finally, using 8 data and 16 compute nodes, the speedup was 1.44.

Overall, the results indicate that scaling up the number of compute nodes beyond the number of data nodes results in a more modest speedup than scaling both compute and data nodes. However, these results do show that additional computing nodes can be used to decrease processing rates.

6.3 Evaluating Effects of Caching

When a data processing application involves multiple passes over data, FREERIDE-G supports the ability to cache remote data. This subsection describes experiments evaluating the benefits of such caching. We use the two multi-pass applications from our set of applications, which are k-means and EM clustering. As the results from these two applications were very similar, we are only presenting results from EM

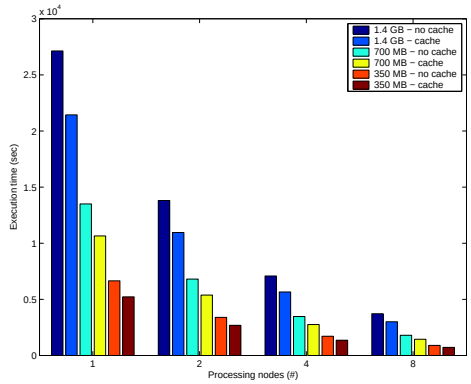

Figure 10: Comparing EM performance with and without caching on 350 MB, 700 MB, and 1.4 GB datasets (1 MB/sec bandwidth).

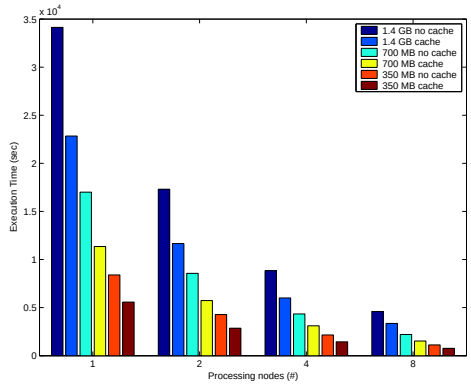

Figure 11: Comparing EM performance with and without caching on 350 MB, 700 MB, and 1.4 GB datasets (500 KB/sec bandwidth).

in this subsection. We executed this application for five iterations, and used simulated cluster interconnection bandwidth of 500 KB/sec and 1 MB/sec.

As in subsection 6.1 three datasets of size 350 MB, 700 MB, and 1.4 GB, respectively, were used. Two versions were created: `Cache` version utilizes a caching framework, as described in Section 4.4, and the `No cache` version, which does not save the data locally during the initial iteration, and, therefore, requires that the server node communicates it again to the compute node during each subsequent iteration.

Figure 10 demonstrates a comparison of parallel execution times of the `cache` and `no cache` versions of the EM clustering application, with 1 MB/sec bandwidth. In all 1-to-1 parallel configurations across all three datasets, the decrease in execution time due to caching is around 1.27. This demonstrates that there is a significant benefit to caching the data locally. In fact, when the breakdown of the execution times were considered, data communication time for the `cache` version was about 20% of the same time for the `no cache` version. Such results were to be expected, since `cache` communicates data only once, whereas `no cache` communicates it five times, once per iteration.

Finally, Figure 11 illustrates the caching benefits for the EM application, but with communication bandwidth of 500 KB/sec. Parallel EM in this setup demonstrates a speedup of around 1.51 in all 1-to-1 parallel configurations, across three datasets.

Overall, caching experiments presented demonstrate that the relative benefit achieved from our caching framework is relatively independent of the size of the problem or the parallel configuration. Instead, communication bandwidth available and the ratio of communication time to compute time determine the factor of improvement in execution times.

7. CONCLUSIONS

In this paper, we have presented a middleware, FREERIDE-G (FRamework for Rapid Implementation of Datamining Engines in Grid), which support a high-level interface for developing data mining and scientific data processing applications that involve data stored in remote repositories. Particularly, we had the following goals behind designing the FREERIDE-G middleware: 1) Support high-end processing, i.e., use parallel configurations for both hosting the data and processing the data, 2) Ease use of parallel configurations, i.e., support a high-level API for specifying the processing, and 3) Hide details of data movement and caching.

8. REFERENCES

[1] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962 – 969, June 1996.

[2] P. Brezany, J. Hofer, A. Tjoa, and A. Wohrer. Gridminer: An infrastructure for data mining on computational grids. In *Proceedings of Australian Partnership for Advanced Computing Conference (APAC)*, Gold Coast, Australia, October 2003.

[3] C. Chang, A. Acharya, A. Sussman, and J. Saltz. T2: A customizable parallel database for multi-dimensional data. *ACM SIGMOD Record*, 27(1):58–66, March 1998.

[4] C. Chang, R. Ferreira, A. Acharya, A. Sussman, and J. Saltz. Infrastructure for building parallel database systems for multidimensional data. In *Proceedings of the Second Merged IPPS/SPDP (13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing)*. IEEE Computer Society Press, April 1999.

[5] A. Chervenak, I. Foster, C. Kesselman, C. Salisbusy, and S. Tuecke. The Data Grid: Towards An Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*, 2001.

[6] V. Curcin, M. Ghanem, Y. Guo, M. Kohler, A. Rowe, J. Syed, and P. Wendel. Grid knowledge discovery processes and an architecture for their composition. In *The Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Edmonton, Alberta, Canada, July 2002.

[7] Arthur Dempster, Nan Laird, and Donald Rubin. Maximum Likelihood Estimation from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977.

[8] M. Ghanem, Y. Guo, A. Rowe, and P. Wendel. Grid-based knowledge discovery services for high throghput informatics. In *The Eleventh IEEE International Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002.

[9] Leo Glimcher and Gagan Agrawal. Parallelizing EM Clustering Algorithm on a Cluster of SMPs. In *Proceedings of Europar (to appear)*, 2004.

[10] Leo Glimcher, Gagan Agrawal, Sameep Mehta, Ruoming Jin, and Raghu Machiraju. Parallelizing a Defect Detection and Categorization Application. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[11] Leo Glimcher, Xuan Zhang, and Gagan Agrawal. Scaling and Parallelizing a Scientific Feature Mining Application Using a Cluster Middleware. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[12] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[13] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. *Applied Statistics*, (28):100–108, 1979.

[14] J. C. Jacob, R. Williams, J. Babu, S. G. Djorgovski, M. J. Graham, D. S. Katz, A. Mahabal, C. D. Miller, R. Nichol, D. E. Vanden Berk, and H. Walia. Grist: Grid data mining for astronomy. In *Astronomical Data Analysis Software and Systems (ADASS) XIV*, October 2004.

[15] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[16] Ivan Janciak, Peter Brezany, and A. Min Tjoa. Towards the Wisdom Grid: Goals and Architecture. In *Proceedings of 4th International Conference on Parallel Processing and Applied Mathematics (PPAM)*, pages 796–803, 2003.

[17] Ruoming Jin and Gagan Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2005. to appear.

[18] G. Kickinger, P.Brezany, A. Tjoa, and J. Hofer. Grid knowledge discovery processes and an architecture for their composition. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN 2004)*, Innsbruck, Austria, February 2004.

[19] R. Machiraju, J. Fowler, D. Thompson, B. Soni, and W. Schroeder. EVITA - Efficient Visualization and Interrogation of Terascale Datasets. In et al R. L. Grossman, editor, *Data Mining for Scientific and Engineering Applications*, pages 257–279. Kluwer Academic Publishers, 2001.

[20] M.Cannataro, A. Congiusta, A. Pugliese, D. Talia, and P. Trunfio. Distributed Data Mining on Grids: Services, Tools, and Applications. *IEEE Transactions on Systems, Man and Cybernetics, Part B*, 34(6):2451–2465, December 2004.

[21] M.Cannataro and D. Talia. KNOWLEDGE GRID: An Architecture for Distributed Knowledge Discovery. *Communications of the ACM*, 46(1):89–93, January 2003.

[22] Sameep Mehta and Kaden Hazzard and Raghu Machiraju and Srinivasan Parthasarathy and John Willkins. Detection and Visualization of Anomalous Structures in Molecular Dynamics Simulation Data. In *IEEE Conference on Visualization*, 2004.

[23] Vlado Stankovski, Michael May, Jürgen Franke, Assaf Schuster, Damian McCourt, and Werner Dubitzky. A service-centric perspective for data mining in complex problem solving environments. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 780–787, 2004.

[24] D.S. Thompson, R. Machiraju, M. Jiang, V. S. Dusi, J. Nair, and G. Craciun. Physics-Based Mining of Computational Fluid Dynamics Datasets. *IEEE Computational Science & Engineering*, 4(3), 2002.

[25] Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14 – 25, 1999.