Supporting Fault Tolerance in a Data-Intensive Computing Middleware

Tekin Bicer   Wei Jiang   Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{*bicer,jiangwei,agrawal*}*@cse.ohio-state.edu*

Abstract—**Over the last 2-3 years, the importance of data-intensive computing has increasingly been recognized, closely coupled with the emergence and popularity of map-reduce for developing this class of applications. Besides programmability and ease of parallelization, fault tolerance is clearly important for data-intensive applications, because of their long running nature, and because of the potential for using a large number of nodes for processing massive amounts of data. Fault-tolerance has been an important attribute of map-reduce as well in its Hadoop implementation, where it is based on replication of data in the file system.**

Two important goals in supporting fault-tolerance are low overheads and efficient recovery. With these goals, this paper describes a different approach for enabling data-intensive computing with fault-tolerance. Our approach is based on an API for developing data-intensive computations that is a variation of map-reduce, and it involves an explicit programmer-declared reduction object. We show how more efficient fault-tolerance support can be developed using this API. Particularly, as the reduction object represents the state of the computation on a node, we can periodically cache the reduction object from every node at another location and use it to support failure-recovery.

We have extensively evaluated our approach using two data-intensive applications. Our results show that the overheads of our scheme are extremely low, and our system outperforms Hadoop both in absence and presence of failures.

Keywords-**Fault tolerance; Map-Reduce; Data-intensive computing; Cloud computing;**

I. INTRODUCTION

The availability of large datasets and increasing importance of data analysis in commercial and scientific domains is creating a new class of high-end applications. Recently, the term *Data-Intensive SuperComputing* (DISC) has been gaining popularity [1], reflecting the growing importance of applications that perform large-scale computations over massive datasets.

The increasing interest in data-intensive computing has been closely coupled with the emergence and popularity of the map-reduce approach for developing this class of applications [2]. Implementations of the map-reduce model provide high-level APIs and runtime support for developing and executing applications that process large-scale datasets. Map-reduce has been a topic of growing interest in the last 2-3 years. On one hand, multiple projects have focused on trying to improve the API or implementations [3], [4], [5], [6]. On the other hand, many projects are underway focusing

on the use of map-reduce implementations for data-intensive computations in a variety of domains. For example, early in 2009, NSF funded several projects for using the Google-IBM cluster and the Hadoop implementation of map-reduce for a variety of applications, including graph mining, genome sequencing, machine translation, analysis of mesh data, text mining, image analysis, and astronomy[1].

Even prior to the popularity of data-intensive computing, an important trend in high performance computing has been towards the use of commodity or off-the-shelf components for building large-scale clusters. While they enable better cost-effectiveness, a critical challenge in such environments is that one needs to be able to deal with failure of individual processing nodes. As a result, *fault-tolerance* has become an important topic in high-end computing [7], [8]. Fault tolerance is clearly important for data-intensive applications because of their long running nature and because of the potential for using a large number of nodes for processing massive amounts of data.

Fault-tolerance has been an important attribute of map-reduce as well in its Hadoop implementation [2]. Traditionally, two common approaches for supporting fault-tolerance have been based on *checkpointing* [9] and *replication* [10]. Among these, replication has been used in map-reduce implementations, i.e., the datasets are replicated by the file systems. The simple and high-level processing structure in map-reduce further helps ease maintaining consistency and correctness while recovering from failures.

In supporting fault-tolerance, *low overhead* and *efficient recovery* are two of the important challenges. By low overhead, we imply that adding failure recovery capabilities should not slow down the system significantly in the absence of failures. By efficient recovery, we imply that, in case of a failure, the system should recovery and complete application execution with reasonable time delays.

With this motivation, this paper presents an alternative approach for supporting data-intensive computing with fault tolerance. Our approach is based on an alternative API for developing and parallelizing data-intensive applications. While this API provides a similar level of abstraction to the map-reduce API, it differs in having a programmer-declared

---

[1]http://www.networkworld.com/community/node/27219

reduction object. This API, particularly the reduction object, forms the basis for low-cost and efficient support for fault-tolerance. The key observation is that the reduction object is a very compact summary of the state of the computation on any node. Our approach for supporting fault-tolerance involves frequently copying the reduction object from each node to another location. If a node fails, the data not yet processed by this node can be divided among other nodes. Then, the reduction object last copied from the failed node can be combined together with the reduction object from other nodes to produce final (and correct) results. Note that our work only targets the failure of processing nodes, and it assumes that all data is still available.

Our approach has been implemented in the context of a data-intensive computing middleware, FREERIDE-G [11], [12]. This middleware system uses the alternative API to develope scalable data-intensive applications. It supports *remote data analysis*, which implies that data is processed on a different set of nodes than the ones in which it is hosted.

We have evaluated our approach using two data-intensive applications. Our results show that the overheads of our approach are negligible. Furthermore, when a failure occurs, the failure recovery is very efficient, and the only significant reason for any slowdown is because of added work at other nodes. Our system outperforms Hadoop both in absence and presence of failures.

The rest of the paper is organized as follows. In Section II, we explain our alternative API and then describe the FREERIDE-G system in which our approach is implemented. Details of our fault tolerance approach and implementation are described in Section III. The results from our experiments are presented in Section IV. We compare our work with related research efforts in Section V and conclude in Section VI.

II. BACKGROUND: PROCESSING API, REMOTE DATA ANALYSIS, AND THE FREERIDE-G SYSTEM

This section gives an overview of the alternative API on which our work is based. We then describe the remote data analysis paradigm and the FREERIDE-G system, which uses this alternative API and supports remote data analysis.

A. API for Parallel Data-Intensive Computing

Before describing our alternative API, we initially review the map-reduce API which is now being widely used for data-intensive computing.

The map-reduce programming model can be summarized as follows [2]. The computation takes a set of input points and produces a set of output $\{key, value\}$ pairs. The user of the map-reduce library expresses the computation as two functions: *Map* and *Reduce*. *Map*, written by the user, takes an input point and produces a set of intermediate $\{key, value\}$ pairs. The map-reduce library groups together all intermediate values associated with the same key and

passes them to the *Reduce* function. The *Reduce* function, also written by the user, accepts a key and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically, only zero or one output value is produced per *Reduce* invocation.

Now, we describe the alternative API this work is based on. This API has been used in a data-intensive computing middleware, FREERIDE, developed at Ohio State [13], [14]. This middleware system for cluster-based data-intensive processing shares many similarities with the map-reduce framework. However, there are some subtle but important differences in the API offered by these two systems. First, FREERIDE allows developers to explicitly declare a reduction object and perform updates to its elements directly, while in Hadoop/map-reduce, the reduction object is implicit and not exposed to the application programmer. Another important distinction is that, in Hadoop/map-reduce, all data elements are processed in the map step and the intermediate results are then combined in the reduce step, whereas in FREERIDE, both map and reduce steps are combined into a single step where each data element is processed and reduced before the next data element is processed. This choice of design avoids the overhead due to sorting, grouping, and shuffling, which can be significant costs in a map-reduce implementation.

The following functions must be written by an application developer as part of the API:
Local Reductions: The data instances owned by a processor and belonging to the subset specified are read. A local reduction function specifies how, after processing one data instance, a *reduction object* (declared by the programmer), is updated. The result of this process must be independent of the order in which data instances are processed on each processor. The order in which data instances are read from the disks is determined by the runtime system.
Global Reductions: The reduction objects on all processors are combined using a global reduction function.
Iterator: A parallel data-intensive application 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.

Throughout the execution of the application, the reduction object is maintained in main memory. After every iteration of processing all data instances, the results from multiple threads in a single node are combined locally depending on the shared memory technique chosen by the application developer. After local combination, the results produced by all nodes in a cluster are combined again to form the final result, which is the global combination phase. The global combination phase can be achieved by a simple all-to-one reduce algorithm. If the size of the reduction object is large, both local and global combination phases perform a parallel

merge to speed up the process. The local combination and the communication involved in the global combination phase are handled internally by the middleware and is transparent to the application programmer.

```
FREERIDE
{* Outer Sequential Loop *}
While() {
    {* Reduction Loop *}
    Foreach(element e) {
        (i, val)    =    Process(e) ;
        RObj(i)     =    Reduce(RObj(i),val) ;
    }
    Global Reduction to Combine RObj
}

Map-Reduce
{* Outer Sequential Loop *}
While() {
    {* Reduction Loop *}
    Foreach(element e) {
        (i, val)    =    Process(e) ;
    }
    Sort (i,val) pairs using i
    Reduce to compute each RObj(i)
}
```

Figure 1. Processing Structure: FREERIDE(top) and Map-Reduce(bottom)

Fig. 1 further illustrates the distinction in the processing structure enabled by FREERIDE and map-reduce. The function *Reduce* is an associative and commutative function. Thus, the iterations of the for-each loop can be performed in any order. The data-structure *RObj* is referred to as the reduction object.

B. Remote Data Analysis and FREERIDE-G

Our support for fault-tolerance is in the context of supporting *transparent remote data analysis*. In this model, the resources hosting the data, the resources processing the data, and the user may all be at distinct locations. Furthermore, the user may not even be aware of the specific locations of data hosting and data processing resources. As we will show, separating the resource for hosting the data and those for processing the data helps support fault-tolerance, since failure of a processing node does not imply unavailability of data.

If we separate the concern for supporting failure-recovery, co-locating data and computation, if feasible, achieves the best performance. However, there are several scenarios co-locating data and computation may not be possible. For example, in using a networked set of clusters within an organizational grid for a data processing task, the processing of data may not always be possible where the data is resident. There could be several reasons for this. First, a data repository may be a shared resource, and cannot allow a large number of cycles to be used for processing of data. Second, certain types of processing may only be possible, or preferable, at a different cluster. Furthermore, grid technologies have enabled the development of *virtual organizations* [15], where data hosting and data processing resources may be geographically distributed.

The same can also apply in cloud or utility computing. A system like Amazon's Elastic Compute Cloud has a separate cost for the data that is hosted, and for the computing cycles that are used. A research group sharing a dataset may prefer to use their own resources for hosting the data. The research group which is processing this data may use a different set of resources, possibly from a utility provider, and may want to just pay for the data movement and processing it performs. In another scenario, a group sharing data may use a service provider, but is likely to be unwilling to pay for the processing that another group wants to perform on this data. As a specific example, the San Diego Supercomputing Center (SDSC) currently hosts more than 6 Petabytes of data, but most potential users of this data are only allowed to download, and not process this data at SDSC resources. The group using this data may have its own local resources, and may not be willing to pay for the processing at the same service provider, thus forcing the need for processing data away from where it is hosted.

When co-locating data and computation is not possible, remote data analysis offers many advantages over another feasible model, which could be referred to as *data staging*. Data staging implies that data is transferred, stored, and then analyzed. Remote data analysis requires fewer resources at the data analysis site, avoids caching of unnecessary or *process once* data, and may abstract away details of data movement from application developers and users.

We now give a brief overview of the design and implementation of the FREERIDE-G middleware. More details are available from our earlier publications [16], [12]. The FREERIDE-G middleware is modeled as a *client-server* system, where the *compute node* clients interact with both *data host* servers and a *code repository* server. The overall system architecture is presented in Figure 2.

Following is a brief description of four major components of an implementation of our middleware that specifically processes data resident in Storage Resource Broker (SRB) servers.

SRB Data Host Server: A data host runs on every on-line data repository node in order to automate data retrieval and its delivery to the end-users' processing node(s). Because of its popularity, for this purpose we used Storage Resource Broker, a middleware that provides distributed clients with uniform access to diverse storage resources in a heterogeneous computing environment. This implementation of FREERIDE-G uses SRB server version 3.4.2&G (Master and Agent) as its data host component. PostgreSQL database software (version 8.1.5) and a separate ODBC
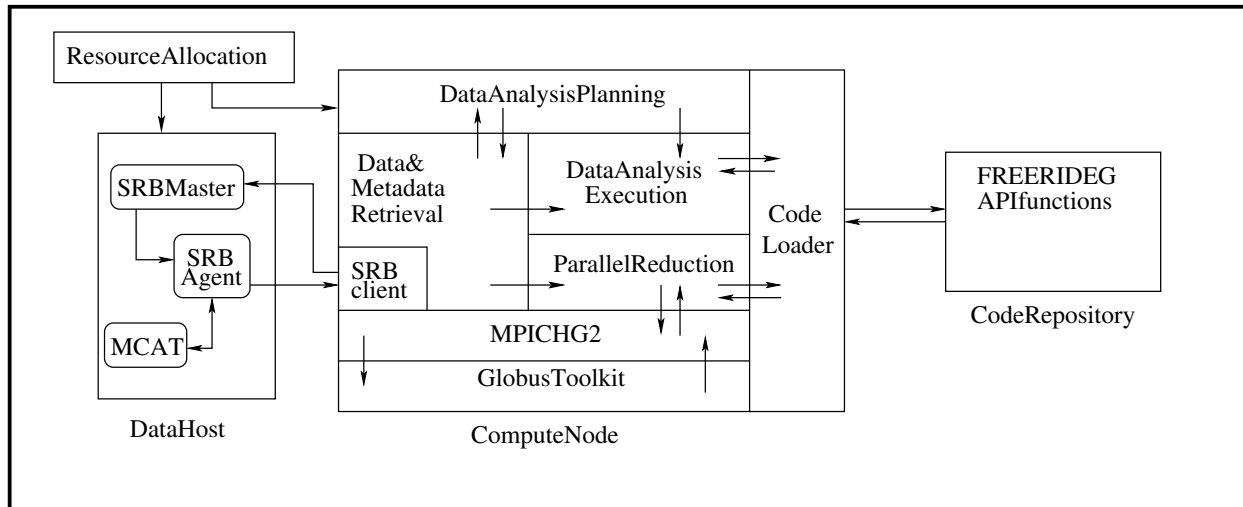
Figure 2.   FREERIDE-G System Architecture

driver (version 07.03.0200) were used to support the MCAT, the catalog used to store metadata associated with different datasets, users and resources managed by the SRB.

Code Repository Web Server: The code repository is used to store the implementations of the FREERIDE-G-based applications, as specified through the API. The only requirement for this component is a web server capable of supporting a user application repository. For our current implementation and experimental evaluation we used Apache Tomcat Web Server (version 5.5.25) to support the code repository, XML descriptors for end-point addressing, and Java Virtual Machine mechanisms for code retrieval and execution.

Compute Node Client: A compute node client runs on every end-user processing node in order to initiate retrieval of data from a remote on-line repository, and perform application specific analysis of the data, as specified through the API implementation. This component is the most complex one, performing data and metadata retrieval, data analysis planning, API code retrieval, and data analysis execution. The processing is based on the generic loop we described earlier, and uses application specific iterator and local and global reduction functions. The communication of reduction objects during global reductions is implemented by the middleware using MPI. Particularly, MPICH-G2 and Globus Toolkit 4.0 are used by this component to support MPI-based computation in grid by hiding potential heterogeneity during grid service startup and management.

Resource Allocation Framework: An important challenge in processing remote data is to allocate computing resources for such processing. Additionally, if a dataset is replicated, we also need to choose a replica for data retrieval. We have developed performance models for performing such selection [11] and will be integrating it with the middleware in the future.

Figure 2 demonstrates the interaction of system components. Once data processing on the compute node has been initiated, data index information is retrieved by the client and a plan of data retrieval and analysis is created. In order to create this plan, a list of all data chunks is extracted from the index. From the worklist a schedule of remote read requests is generated to each data repository node. After the creation of the retrieval plan, the SRB-related information is used by the compute node to initiate a connection to the appropriate node of the data repository and to authenticate such connection. The connection is initiated through an SRB Master, which acts as a main connection daemon. To service each connection, an SRB Agent is forked to perform authentication and other services, with MCAT metadata catalog providing necessary information to the data server. Once the data repository connection has been authenticated, data retrieval through an appropriate SRB Agent can commence. To perform data analysis, the code loader is used to retrieve application specific API functions from the code repository and to apply them to the data.

III. Fault Tolerance Approach

This section describes our approach and implementation for supporting fault-tolerance in a remote data analysis system.

A. Alternatives

Replicating jobs in the system is one way of tolerating failures. In this approach, the job that processes the data can be replicated, with all replicas working on the same data. Therefore, all replicas should produce the same output. In case of a failure, the system can continue its execution by switching to one of the other replicas. This approach,

however, can be very expensive in terms of resource utilization. Instead of using more processing nodes for speeding up a single copy of the task, we use the same resources for executing multiple replicas, limiting the speedup of any of the replicas.

Another possibility could be to only replicate the input data. In this case, if a node fails, the jobs based on this data can be restarted on another node, which also hosts a replica of the same data. This is the approach taken by current map-reduce implementations. However, as we will show experimentally, this approach results in a large slowdown if a failure occurs close to finishing a task.

Another popular fault tolerance approach is *checkpointing*, where snapshots of the system are taken periodically. The state information of an application is stored in persistent storage unit(s). In case of a failure, the most recent state information can be retrieved, and the system can be recovered. The main drawback of this approach is that a system's state information can be very large in size. This can result in high overheads of taking and storing the checkpoints.

B. Our Approach

Our approach exploits the properties of the processing structure we target. Let us consider the processing structure supported by our middleware, shown earlier in Figure 1, top. Let us suppose the set of data elements to be processed is E. Suppose a subset E_i of these elements is processed by the processor i, resulting in $RObj(E_i)$. Let G be the *global reduction function*, which combines the reduction objects from all nodes, and generates the final results.

The key observation in our approach is as follows. Consider any possible disjoint partition, $E_1, E_2, \ldots E_n$, of the processing elements between n nodes. The result of the global reduction function,

$$(RObj(E_1), RObj(E_2), \ldots, RObj(E_n))$$

will be same for any such disjoint partition of the element set E. In other words, if $E_1, E_2, \ldots, E_n$ and $E_1', E_2', \ldots, E_n'$ are two disjoint partitions of the element set E, then,

$$G(RObj(E_1), RObj(E_2), \ldots, RObj(E_n)) = \\ G(RObj(E_1'), RObj(E_2'), \ldots, RObj(E_n'))$$

This observation can be exploited to support fault-tolerance with very low overheads. From each node, the reduction object after processing of a certain number of elements is copied to another location. We mark the set of elements that have already been processed before copying the reduction object. Now, suppose a node i fails. Let E_i be the set of elements that were supposed to be processed by the node i, and let E_i' be the set of elements processed before the reduction object was last copied. Now, the set of elements $E_i - E_i'$ must be distributed and processed by the remaining set of nodes. Thus, a node j will process a subset of the

elements $E_i - E_i'$, along with the set of elements E_j that it was originally supposed to process. The reduction objects resulting after such processing on the remaining $n-1$ nodes can be combined together with the cached reduction object from the failed node to produce the final results. By the argument above, we know that this will create the same final results as the original processing.

Although our system design can be viewed as a checkpoint-based approach, it has significant differences. First, unlike snapshots of the checkpoint-based systems, the size of the reduction object is generally small for most of the data mining applications. Therefore, the overhead of storing the reduction object to another location is smaller than storing the snapshots of the entire system. Furthermore, we do not try to restart the failed process. Instead, by using the properties of the processing structure, the work from the failed node is distributed to other nodes, and final results are computed. Our approach can be viewed as an adaptation of the *application-level checkpointing* approach [7], [17]. Again, the key difference is that we exploit the properties of reductions to redistribute the work, and do not need to restart the failed process.

C. Fault Tolerance Implementation

We now discuss how our approach is implemented in FREERIDE-G. Figure 3 shows the execution that takes place in compute nodes and the *logical data host* in FREERIDE-G. The logical data host is a virtual entity representing a set of physical data hosts. The dataset is distributed among the physical data hosts in *chunks*, which could be considered as equivalent of disk blocks on the file system. Each data chunk has a unique id.

When an application starts, the data chunks are evenly distributed among the compute nodes using the chunk ids. When a compute node starts its execution, it requests the chunk ids, which are assigned to the compute node, and the corresponding data chunks are retrieved from the data hosts. After receiving the requested data chunk, the compute node processes the data and accumulates the results onto the reduction object. The key observation in our approach is that the reduction object and the last processed data chunk id can be correctly used as a snapshot of the system. If this information is stored at another location, it can enable a restart of the failed process at another node. The new process will simply have to process all chunks that have an id higher than the stored chunk id, and were originally given to the failed node.

Furthermore, we can use the property of the reduction we described not just to restart a failed process, but also to redistribute the remaining chunks evenly across other remaining/alive processes. The other processes can accumulate the results of the reduction from the chunks originally assigned to them and the redistributed chunks. The resulting

Execution on the Logical Data Host:
INPUT:
 CNs: Set of compute nodes in the system
 CIDs: Ordered set of chunk ids
OUTPUT:
 Compute nodes with assigned chunk ids
  {* *Find the number of chunks per compute node* *}
  `ChPN = GetSize(CIDs) / GetSize(CNs);`
  {* *Assign chunk ids to compute nodes* *}
  `Foreach compute node cn in CNs {`
    {* *Get compute node id* *}
    `CNID = GetID(cn);`
    {* *Find starting and ending chunk ids* *}
    `StartID = CNID x ChPN;`
    `EndID = StartID + ChPN;`
    {* *Assign chunks to the compute node* *}
    `Foreach chunkID cid from StartID to EndID {`
        `Assign cid to cn;`
    `}`
  `}`

Execution on a Compute Node:
INPUT:
 CIDs: Ordered set of assigned chunkIDs
OUTPUT:
 Reduc: Reduction object
  {* *Execute outer sequential loop* *}
  `While () {`
    {* *Execute reduction loop* *}
    `Foreach chunk id cid in CIDs {`
      `Retrieve data chunk c with cid;`
      `Foreach element e in chunk c {`
        `(i,val) = process(e);`
        `Reduc(i) = Reduc(i)` *op* `val;`
      `}`
    `}`
    `Perform Global Reduction`
  `}`

Figure 3.   Application Execution in FREERIDE-G

reduction objects can be combined together with the cached reduction object to obtain the final results.

Our implementation allows a system manager to configure how the reduction objects can be cached. For the executions used for obtaining results we will present, we grouped our compute nodes together and set the storage location of the snapshots as the other members of the group. Therefore, after a compute node processes each data chunk, it replicates its snapshot into the other compute nodes in the group. In case of a failure, the group members can determine the failure and recover from the stored replica. The size of the group determines how many failures can be tolerated. If the size of the group is 3, a reduction object is cached at 2 other locations. Our system can tolerate failure of up to 2 nodes from the group. If all members of a group fail, the system cannot recover from failures. In our implementation, the size of the group was 2.

Overall, the fault-tolerance implementation involves three components:

Configuration: Before compute nodes start their data chunk retrieval and processing, some information should be provided by the programmer to the fault tolerance system. This includes the *exchange frequency* of reduction objects, destination compute nodes where reduction object will be saved, the number of connection trials before the node is assigned as failed in the system, and exchange method for reduction objects. The frequency of the reduction object implies how frequently the reduction object is replicated in the system. For instance, if the frequency number is 4, the fault tolerance system waits until 4 data blocks have been retrieved and processed, before communicating the reduction object. A larger value reduces the overheads of copying, communicating, and storing the reduction object. However, it can also lead to more repeated work in case of a failure.

Fault Detection: FREERIDE-G invokes an initialization function, which starts peer-to-peer communication among the nodes in which the replicas will be stored. These connections are kept alive during the execution. Whenever a node fails, connections to the failed nodes become corrupt. If the node is still not reachable by any of its peers after a specific number of connection trials, it is considered as a failed node in the system.

Fault Recovery: Computing nodes query if there is a failed node in the system before they enter the global combination phase. If a failed node is determined, then the recovery process begins.

We further explain how our approach is integrated with the processing in FREERIDE-G (Figure 4). First, the system initializes the application specific constant values, such as the node's address in the network. Second, some of the FREERIDE-G components are registered into our fault tolerance system. This is necessary because in case of a failure, the fault tolerance system needs to interact with the FREERIDE-G components, thus reorganizing the work-flow in order to process the failed node's remaining data blocks. The third step starts the fault tolerance communication protocol among the nodes that will exchange the reduction objects. Since this step involves initial data transfer among the nodes, it also exchanges some valuable information such as the data blocks that belong to the nodes. The fourth step shows how we modified the processing structure of FREERIDE-G and port our system into global and local reduction. Specifically, each retrieved data chunk is processed by the local reduction function, and then it is accumulated into the reduction object. The function *SetNewStateInformation* resets the state values in the fault tolerance system. These values include last processed data block number and current iteration number. The *ReplicateSnapshot* function invokes the programmer's API and starts transferring the reduction object.

Currently, our system supports two types of exchange method. The first is *synchronous data exchange*, which

```
Algorithm: Fault Tolerance System Implementation
INPUT:
  ChunkIDs: Ordered set of assigned chunkIDs
OUTPUT:
  Reduc: Reduction object
  {* Initialize node specific information *}
  InitNodeSpecInfo();
  {*Register recovery related components *}
  RegisterComponents();
  {* Start fault tolerance system *}
  StartFTS();
  {* Execute outer sequential loop *}
  While () {
      Foreach chunk id cid in ChunkIDs {
          Retrieve data chunk c with cid;
          {* Process retrieved data chunk *}
          {* Accumulate result into reduction obj *}
          {* Set current state info. in FTS *}
          SetNewStateInformation();
          {* Store snapshot to another location *}
          ReplicateSnapshot(Reduc);
      }
      If (CheckFailure()) {
          RecoverFailure();
      }
      Perform Global Reduction
  }
```

Figure 4.   Compute Node Processing with Fault Tolerance Support

blocks the application until the fault tolerance system finishes the reduction object exchange. The second method is *asynchronous data exchange*. If the programmer selects this method in the configuration file, the communication protocol of our fault tolerance system does not block the system's execution. More specifically, whenever FREERIDE-G invokes the *ReplicateSnapshot* function, the fault tolerance system copies the reduction object into its own context and creates a thread. This thread starts the reduction object exchange. However, if a thread is in progress when the *ReplicateSnapshot* function is called, then system is blocked until it finishes its data exchange. Therefore, if the sum of data chunk retrieval time and data processing time are longer than the reduction object exchange time, the overhead of the fault tolerance system becomes minimal. Our preliminary experiments demonstrated that the asychronous method clearly outperform the synchronous method. Thus, the results reported in the next section are only from the asychronous implementation.

If a node finishes processing all of its data chunks, then it is ready to enter the global reduction phase. Before it enters this phase, it calls *CheckFailure* function which is responsible for detecting the failed nodes. If there is no failure, all the nodes enter the global reduction phase without any interruption. Otherwise, if a node is assigned as failed in the system, *RecoverFailure* function is invoked. This function initiates the recovery process, and the remaining data

blocks are distributed to the alive nodes in the system. The next step is reorganizing the work flow of the FREERIDE-G and restarting the local reduction loop with failed node's remaining data blocks. This process continues until all the data blocks are processed. Finally, a global combination is performed using the reduction objects from the remaining nodes, and the cached reduction object(s) from failed nodes.

IV. APPLICATIONS AND EXPERIMENTAL RESULTS

In this section, we report results from a number of experiments that evaluate our approach for supporting fault-tolerance. The goals in our experiments include the following. First, we want to evaluate the overheads our approach involves if there are no failures. Second, we want to study the slowdown in completion times when one of the nodes fails. We particularly study this factor with varying *failure points*, i.e., the fraction of the work that is completed by a node before failing. Finally, we compare our approach to the Hadoop implementation of map-reduce using both of the above metrics.

The configuration used for our experiments is as follows. Our compute nodes have dual processor Opteron 254 (single core) with 4GB of RAM and are connected through Mellanox Infiniband (1 Gb). As our focus is on tolerating failures of processing cores, the data is stored on a separate set of nodes. While we can tolerate failures of data hosting nodes by replicating the data, it is not considered in our current work.

For our experiments, we used two data-intensive applications, which are k-means clustering and Principal Component Analysis (PCA). Clustering is one of the key data mining problems and k-means [18] is one of the most popular clustering algorithms. Similarly, Principal Components Analysis (PCA) is a popular dimensionality reduction method that was developed by Pearson in 1901. For k-means clustering experiments, the number of cluster centers was 50 and this resulted in a reduction object with a size of 2 KB. We used two different dataset sizes: 6.4 GB and 25.6 GB. Each dataset was divided into 4096 data blocks and evenly distributed into 4 data hosting nodes. The reduction object size for PCA is 128 KB. For PCA experiments, two different datasets were used: 4 GB and 17 GB, which were again divided into 4096 data blocks.

A. Overheads of Supporting Fault-Tolerance

In this set of experiments, we examined the execution of k-means clustering and PCA using three different versions: Without fault tolerance support (`Without FTS`), with fault tolerance support (`With FTS`), and the case when a failure actually occurs (`With Failure`). The first version, `Without FTS`, is the earlier version of FREERIDE-G, and cannot recover from any failure. The second version, `With FTS`, provides failure recovery using the reduction object exchange. For the experiments reported in this subsection,

this exchange is performed asynchronously after processing of each data block or chunk. However, the results from this version do not involve any failures or failure recovery. The third version, `With Failure`, shows the execution time of the system in case of a failure. For this subsection, this failure is introduced after exactly 50% of the processing has been completed by the failing nodes.
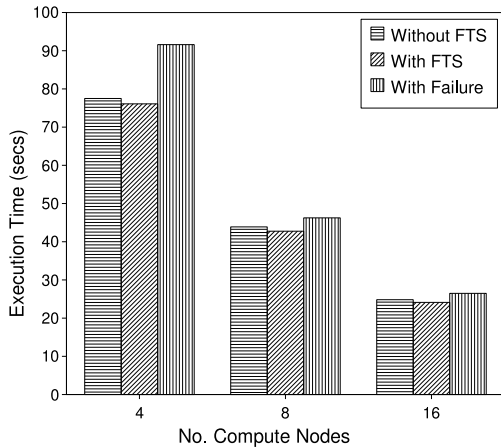
In our experiments, we used 4, 8 and 16 compute nodes.

Figure 5.   Evaluating Overheads using k-means clustering (6.4 GB dataset)

In Figure 5, we show results from k-means using the 6.4 GB dataset. The overheads of the fault tolerance system when no failure actually occurs, i.e., the `With FTS` version, is very close to 0%. Even though this version is exchanging the reduction objects, the small size of the k-means reduction object and the use of asynchronous communication avoids any overheads. Note that in some cases, this version is actually slightly faster than the `Without FTS` version. This is simply due to random variations in execution times, as they include data retrieval and communication of data from other nodes.

Now, we focus on the performance of the `With Failure` versions. As one of the nodes is failing after processing 50% of the data, and the remaining work is being done by other nodes, we can clearly expect some overheads. The *relative slowdowns* of the system are close to 18% for 4, 5.4% for 8, and 6.7% for 16 compute nodes. Note that in these three cases, the remaining work is being distributed among 3, 7, and 15 nodes, respectively. Thus, the higher overhead for the 4 nodes case is expected. We further analyzed this relative slowdown. If we ignore the time spent on the remaining nodes in processing the redistributed data, the resulting slowdown can be considered as an *absolute overhead*. This absolute overhead ranges between 0% and 3.2% for the three cases. The overhead is the highest for the 16 node case, and is likely because of the cost of detecting failure, and the time spent determining how the data should
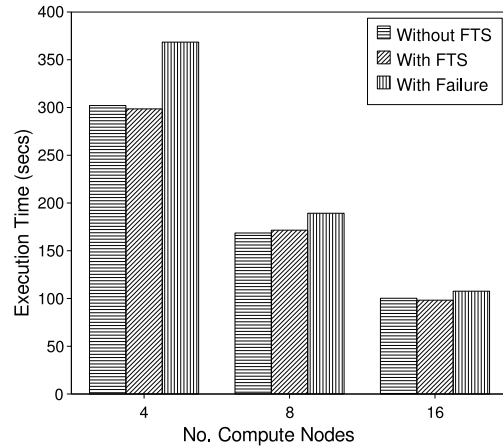
be redistributed.

Figure 6.   Evaluating Overheads using k-means clustering (25.6 GB dataset)

We repeated the same experiment with a larger dataset, and the results are shown in Figure 6. The overheads of the `With FTS` version are close to 0% for 4, 1.7% for 8 and 0% for 16 compute node cases. The relative slowdowns of the FTS in case of `With Failure` version are 22% for 4, 12.3% for 8 and 7.4% for 16 compute nodes. Moreover, the absolute overheads of the FTS are 4.6% for 4, 4.8% for 8, and 3.9% for 16 compute node cases.
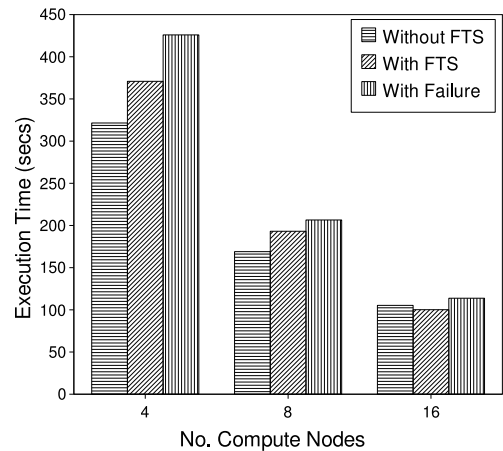

Figure 7.   Evaluating Overheads using PCA clustering (4 GB dataset)

In Figure 7, the same experiments are reported with PCA as the application and 4 GB of dataset. Considering the `With FTS` version, the overheads of supporting fault tolerance in our system are 15.4% for 4, 14.3% for 8, and 0% for 16 compute nodes. The total volume of communication, which depends on how many blocks are processed by each node, is higher as we have a smaller total number of

nodes to process all data. Thus, for 4 and 8 node cases, asynchronous data exchange does not seem to be able to hide the communication latencies. As we stated earlier, the size of the reduction object is large for PCA, which also seems to contribute to higher overheads.

For the `With Failure` version, where one node fails after processing 50% of the data, the relative slowdowns are 32.5%, 22.2%, and 8.1% for 4, 8, and 16 compute node cases, respectively. After removing the times for processing the redistributed data, the absolute overheads are only 8.5%, 14.1%, and 0.9% for 4, 8, and 16 nodes cases.
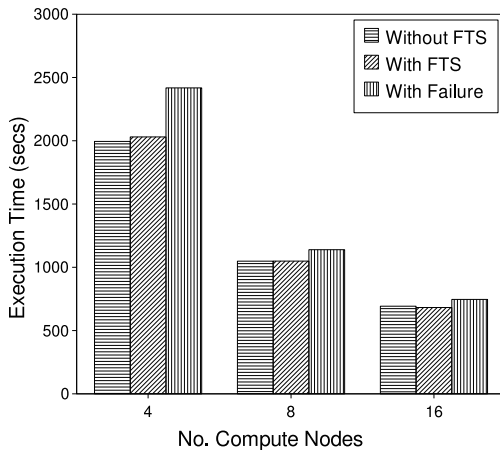
Figure 8. Evaluating Overheads using PCA clustering (17 GB dataset)

In Figure 8, we repeated the same experiment with 17 GB of data. Note that the larger dataset is still divided into the same number of data blocks (4096), so the data block size is now larger. As a result, more processing is now performed between exchanging reduction objects. For the `With FTS` versions, the overheads are 1.8% for 4 and 0% for both 8 and 16 compute nodes. Clearly, as the relative frequency of exchanging the reduction object is lowered, the communication latencies can be hidden by asynchronous exchanges.

Considering the `With Failure` versions, the relative slowdowns are 21.2%, 8.6%, and 7.8% with 4, 8, and 16 compute node cases, respectively. The absolute overheads are 3.9%, 1.4%, and 4.3%, for the 4, 8, and 16 node cases, respectively.

B. Overheads of Recovery: Different Failure Points

In all experiments reported in the previous subsection, all failures occurred after the failing node had processed 50% of the data. In this subsection, we further evaluate relative slowdowns and absolute overheads varying the *failure points*. These failure points are now set to 25% (close to beginning), 50% (middle) and 75% (close to end) of the execution. We used the same datasets and set the number of compute nodes to be 8 for each of the experiments.
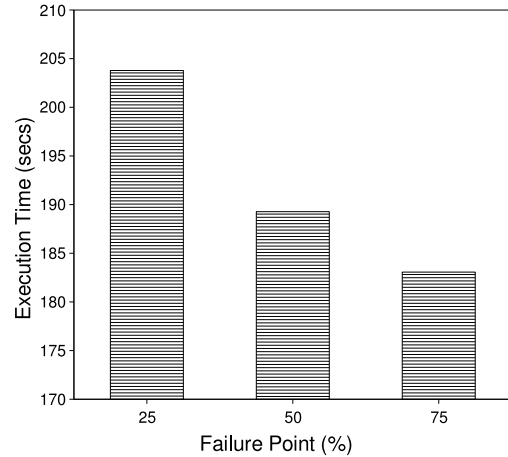

Figure 9. K-means Clustering with Different Failure Points (25.6 GB dataset, 8 compute nodes)

In Figure 9, the relative slowdowns are 16.6%, 9%, and 7.2%, with failures close to beginning, middle, and close to end, respectively. The sooner the failure occurs, more work needs to be redistributed among the remaining nodes. This explains why the relative slowdowns decrease with a later failure point. After removing the time spent on processing redistributed data blocks, the absolute overheads are 5.3%, 1.7%, and 3.5%, for the three cases. In other words, the absolute overheads of failure-recovery are very small in all cases, and most of the other slowdowns are because of the additional work being done by the remaining nodes.
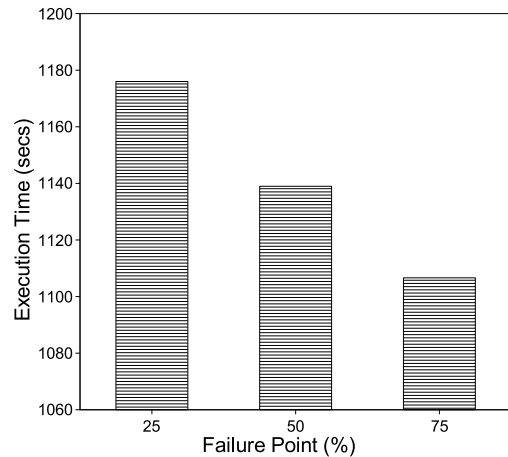

Figure 10. PCA with Different Failure Points (17 GB dataset, 8 compute nodes)

The same experiments were repeated with PCA and the results are shown in Figure 10. The relative slowdowns are 12.2%, 8.6% and 5.6%, with failure occurring close to the beginning, middle, and close to end, respectively. The absolute overheads are 1.3%, 1.4% and 1.9%, respectively,

for the same three cases.

C. Comparison of FREERIDE-G and Hadoop

As we had originally stated, our goal in this paper was to present a method for fault-tolerant data-intensive computing, which has lower overheads and more efficient failure-recovery as compared to map-reduce. So far in this section, we have shown that our method has very low overheads. Excluding the time the remaining nodes spend processing additional data blocks, the recovery is also very efficient. Now, we compare our approach and implementation, both in absence and presence of failures, with a map-reduce implementation. Hadoop is an open source version of map-reduce and is being widely used. Thus, we have compared our system with Hadoop. Because we did not have PCA implemented using Hadoop, this comparison is performed only with the k-means clustering application.

Hadoop supports fault-tolerance by replication of data. There is no version of Hadoop available without support for fault-tolerance. Thus, we compared the `With FTS` version of FREERIDE-G with Hadoop, and then further compared the two systems with failures at different points in execution.
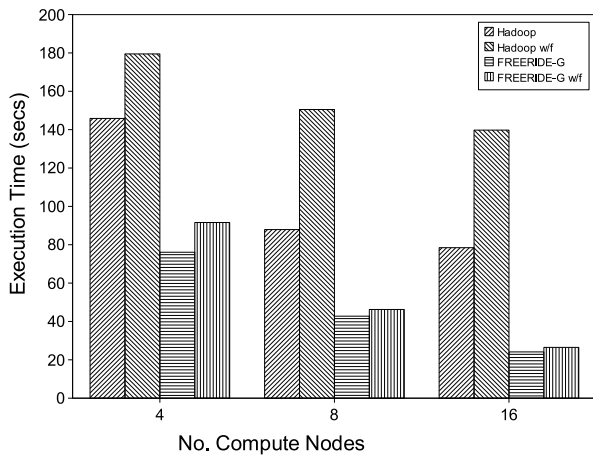

Figure 11. Comparison of Hadoop and FREERIDE-G: Different No. of Nodes - k-means clustering (6.4 GB Dataset)

In Figure 11, we compared two versions each from the two systems. The first version is the normal execution times of the systems and the second version is the execution times of the systems in case of a failure. For both the systems, the failure occurs on one of the compute nodes, after processing 50% of its data chunks. The versions in which failures happen are referred to as `w/f` in the figure.

We can see that FREERIDE-G is faster than Hadoop and scales better with increasing number of nodes. Furthermore, when the failure occurs, Hadoop has higher overheads. The overheads are 23.06%, 71.21% and 78.11% with 4, 8, and 16 compute nodes. On the other hand, the overheads of the FREERIDE-G are 20.37%, 8.18% and 9.18% for 4, 8 and 16

compute nodes. As we had explained earlier, the slowdown for FREERIDE-G becomes smaller as we go from 4 to 8 nodes, because the remaining data chunks from the failed nodes can now be distributed among 7 other nodes, instead of only 3. Note that the same trend should be true in going from 8 to 16 nodes, but the resulting advantage is smaller, and seems to be offset by other overheads in failure recovery. In the case of Hadoop, we instead see a higher overhead as the number of nodes is increased. The reason is that in case of a failure, Hadoop needs to restart the failed node's tasks among the remaining compute nodes, and the overhead of restarting the tasks increases while the number of the compute nodes in the system increases.
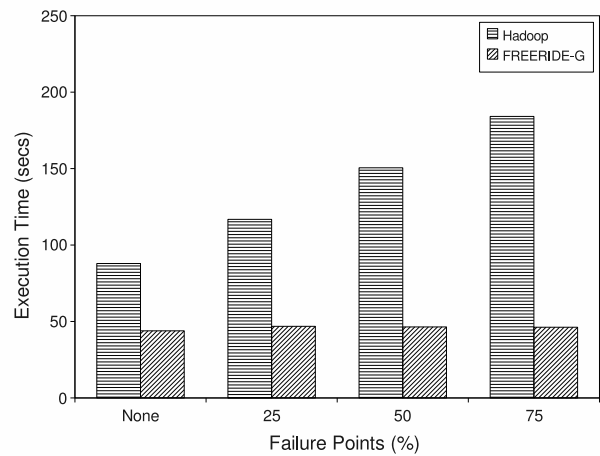

Figure 12. Performance With Different Failure Points (k-means, 6.4 GB dataset, 8 compute nodes)

The overheads of both Hadoop and FREERIDE-G at different failure points are compared in Figure 12. The Hadoop's overheads are 32.85%, 71.21% and 109.45% for the failure points 25%, 50%, and 75%. The FREERIDE-G's overheads are 6.69%, 5.77% and 5.34% for the same failure points. We can see that in the case of FREERIDE-G, the overheads are much smaller, and seem to decrease as the failure occurs later during the execution. This is because with a later failure point, fewer data chunks needs to be redistributed and executed among other nodes. But, in the case of Hadoop, the later the failure occurs, the total execution time slowdown is higher. This is because the state of computation is not cached in Hadoop. Instead, data replication across nodes is its mechanism for fault-tolerance. After a failure, the work already completed by the failed node needs to be redone at one or more other nodes, depending upon how the replication has been done by the file system. Thus, the later the time the failure occurs, the total time for the execution of the application is higher.

V. RELATED WORK

We now compare our work with existing work on fault-tolerance for data-intensive computing, and on other prominent efforts on fault-tolerance for parallel and distributed computing.

The topics of data-intensive computing and map-reduce have received much attention within the last 2-3 years. Efforts underway include projects that have used and evaluated map-reduce implementations, as well as those that are trying to improve performance and programmability. However, beyond the original approach for fault-tolerance in map-reduce, there has not been much work in this direction. Projects in both academia and industry are working towards improving map-reduce. CGL-MapReduce [3] uses streaming for all the communications, and thus improves the performance to some extent. Yahoo has developed *Pig Latin* [5] and Map-Reduce-Merge [19], both of which are extensions to Hadoop, with the goal being to support more high-level primitives and improve the performance. Google has developed *Sawzall* [6] on top of map-reduce to process large document collections. Microsoft has built *Dryad* [4], which is more flexible than map-reduce, since it allows execution of computations that can be expressed as *DAGs*. Dryad supports fault-tolerance by reexecuting computations. We are not aware of any work evaluating its performance. Phoenix is a system for enabling fault-tolerance for grid-based data-intensive applications [20].

Our approach can be viewed as being somewhat similar to the work from Cornell on *application-level checkpointing* for parallel programs [7], [21], [17]. Their approach investigates the use of compiler technology to instrument codes to enable self-checkpointing and self-restarting. As we stated earlier, our work can be viewed as an optimization and adaptation of this work to a different execution environment and a specific class of applications. The key difference in our approach is that we exploit the reduction property of data-intensive computations, and can redistribute the remaining work from failed node across other processors. This turns out be more efficient than restarting the process.

In recent years, fault-tolerance support for MPI processes has been developed by several research groups. Checkpointing, coupled with *message logging* have been most often used for supporting fault-tolerance [22], [23], [24], [8]. Some approaches have also been based on replication [25].

In distributed and grid computing, much work has been done on achieving fault-tolerance by running a number of job replicas simultaneously [26], [27], [28], [10], [29], [30]. Usually, these efforts involve a primary task and other backup tasks. In the case of failure on the primary task, processing continues on one of the backups. Considering the checkpointing-based approaches, some efforts have been taken to reduce the size of checkpoints. Zheng *et al.* [31] have proposed an in-memory double checkpointing protocol for fault-tolerance. Without relying on any reliable storage, the checkpoint data, which is encapsulated by the programmer, is stored at two different processors. Also, the checkpoints are taken at a time when the application memory footprint is small. Another approach proposed by Marques *et al.* [9] dynamically partitions objects of the program into *subheaps* in memory. By specifying how the checkpoint mechanism treat objects in different *subheaps* as *always_save*, *never_save* and *once_save*, they reduce the checkpoint size at runtime. Our work has some similarities, but the key difference is our ability to use the reduction property to redistribute the remaining work across other nodes.

VI. CONCLUSION

With growing class of applications that process large datasets on commodity clusters, there is need for supporting both programmability and fault-tolerance. This paper has described a system that offers a high-level API for developing data-intensive applications. Further, the properties associated with the programs that can be expressed using this API are exploited to provide a very low-overhead support for fault-tolerance and failure-recovery.

We have evaluated our approach using two data-intensive applications. Our results show that the overheads of our approach are negligible. Furthermore, when a failure occurs, the failure recovery is very efficient, and the only significant reason for any slowdown is because of added work at other nodes. Our system outperforms Hadoop both in absence and presence of failures.

REFERENCES

[1] R. E. Bryant, "Data-intensive supercomputing: The case for disc," School of Computer Science, Carnegie Mellon University, Tech. Rep. Technical Report CMU-CS-07-128, 2007.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of OSDI*, 2004, pp. 137–150.

[3] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for data intensive scientific analyses," in *IEEE Fourth International Conference on e-Science*, Dec 2008, pp. 277–284.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2007 EuroSys Conference*. ACM, 2007, pp. 59–72.

[5] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of SIGMOD Conference*. ACM, 2008, pp. 1099–1110.

[6] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277–298, 2005.

[7] G.Bronevetsky, D.Marques, M.Schulz, P.Szwed, and K.Pingali, "Application-level checkpointing for shared memory programs," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004)*, Oct. 2004, pp. 235–247.

[8] G. Stellner, "Cocheck: Checkpointing and process migration for mpi," in *Proceedings of IPPS 1996*, 1996, pp. 526–531.

[9] D. Marques, G. Bronevetsky, R. Fernandes, K. Pingali, and P. Stodghill, "Optimizing checkpoint size in the c3 system," in *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2005.

[10] J.H.Hwang, M.Balazinska, A.Rasin, U.Cetintemel, M.Stonebraker, and S.Zdonik, "High-availability algorithms for distributed stream processing," in *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)*, April 2005, pp. 779–790.

[11] L. Glimcher and G. Agrawal, "A Performance Prediction Framework for Grid-based Data Mining Applications," in *In proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[12] L. Glimcher and G. Agrawal, "A Middleware for Developing and Deploying Scalable Remote Mining Services," in *In proceedings of Conference on Clustering Computing and Grids (CCGRID)*, 2008.

[13] R. Jin and G. Agrawal, "Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance," in *Proceedings of the second SIAM conference on Data Mining*, Apr. 2002.

[14] R. Jin and G. Agrawal, "A middleware for developing parallel data mining implementations," in *Proceedings of the first SIAM conference on Data Mining*, Apr. 2001.

[15] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of Grid: Enabling Scalable Virtual Organizations," *International Journal of Supercomputing Applications*, 2001.

[16] L. Glimcher, R. Jin, and G. Agrawal, "FREERIDE-G: Supporting Applications that Mine Data Repositories," in *In proceedings of International Conference on Parallel Processing (ICPP)*, 2006.

[17] G.Bronevetsky, D.Marques, K.Pingali, and P.Stodghill, "Automated application-level checkpointing of mpi programs," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2003)*, Oct. 2003, pp. 84–94.

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

[19] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. P. Jr., "Mapreduce-merge: simplified relational data processing on large clusters," in *Proceedings of SIGMOD Conference*. ACM, 2007, pp. 1029–1040.

[20] G. Kola, T. Kosar, and M. Livny, "Phoenix: Making data-intensive grid applications fault-tolerant," in *GRID*, R. Buyya, Ed. IEEE Computer Society, 2004, pp. 251–258.

[21] G.Bronevetsky, K.Pingali, and P.Stodghill, "Experimental evaluation of application-level checkpointing for openmp programs," in *Proceedings of the 20th International Conference on SuperComputing (SC 2006)*, Dec. 2006, pp. 2–13.

[22] A. Agbaria and R. Friedman, "Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations," *Cluster Computing*, vol. 6, no. 3, pp. 227–236, 2003.

[23] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello, "Mpich-v project: A multiprotocol automatic fault-tolerant mpi," *The International Journal of High Performance Computing Applications*, vol. 20, no. 3, pp. 319–333, 2006, iD: 363685160.

[24] J. Dongarra, P. Kacsuk, and N. Podhorszki, Eds., *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users' Group Meeting, Balatonfüred, Hungary, September 2000, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1908. Springer, 2000.

[25] R. Batchu, A. Skjellum, Z. Cui, M. Beddhu, J. P. Neelamegam, Y. S. Dandass, and M. Apte, "Mpi/fttm: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing," in *CCGRID*. IEEE Computer Society, 2001, pp. 26–33.

[26] T.Tsuchiya, Y.Kakuda, and T. Kikuno, "Fault-tolerant scheduling algorithm for distributed real-time systems," in *Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 95)*, April 1995, pp. 99–103.

[27] J.H.Abawajy, "Fault-tolerant scheduling policy for grid computing systems," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2004, pp. 238–245.

[28] X. Zhang, D. Zagorodnov, M. Hiltunen, K. Marzullo, and R. Schlichting, "Fault-tolerant grid services using primary-backup: Feasibility and performance," in *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, September 2004, pp. 105–114.

[29] M.Balazinska, H.Balakrishnan, S. Madden, and M.Stonebraker, "Fault-tolerance in the borealis distributed stream processing system," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, June 2005, pp. 13–24.

[30] R.Murty and M.Welsh, "Towards a dependable architecture for internet-scale sensing," in *Proceedings of the 2nd Workshop on Hot Topics in Dependability (HotDep06)*, Nov. 2006.

[31] G. Zheng, L. Shi, and L. V.Kale, "Ftc-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi," in *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, September 2004, pp. 93–103.