Fault Tolerant MapReduce-MPI for HPC Clusters

Yanfei Guo[*], Wesley Bland[**], Pavan Balaji[**], and Xiaobo Zhou[*]

[*]Department of Computer Science, University of Colorado, Colorado Springs

[**]Mathematics and Computer Science Division, Argonne National Laboratory

ABSTRACT

Building MapReduce applications using the Message-Passing Interface (MPI) enables us to exploit the performance of large HPC clusters for big data analytics. However, due to the lacking of native fault tolerance support in MPI and the incompatibility between the MapReduce fault tolerance model and HPC schedulers, it is very hard to provide a fault tolerant MapReduce runtime for HPC clusters. We propose and develop FT-MRMPI, the first fault tolerant MapReduce framework on MPI for HPC clusters. We discover a unique way to perform failure detection and recovery by exploiting the current MPI semantics and the new proposal of user-level failure mitigation. We design and develop the *checkpoint/restart* model for fault tolerant MapReduce in MPI. We further tailor the *detect/resume* model to conserve work for more efficient fault tolerance. The experimental results on a 256-node HPC cluster show that FT-MRMPI effectively masks failures and reduces the job completion time by 39%.

1. INTRODUCTION

MapReduce is a programming paradigm widely adopted for reliable large-scale data-intensive processing. As data explodes in velocity, variety, and volume, it is getting increasingly difficult to scale computing performance using enterprise class servers and networks [23]. High Performance Computing (HPC) clusters, characterized by high performance compute and storage servers and high-speed interconnections, offer immense potential for high performance data analytics. However, popular MapReduce implementations like Google MapReduce [16], Hadoop [1], and Dryad [26] are designed for the clusters that are dedicated to MapReduce. Their custom-designed task runtimes and job schedulers are not suitable for HPC clusters. More importantly, HPC clusters usually use heavily simplified kernels that may not support a Java runtime. We cannot simply deploy popular MapReduce implementations on HPC clusters. Thus, a new MapReduce framework that is designed for HPC clusters and implemented using a portable runtime, such as Message-Passing Interface (MPI) is urgently needed.

However, as the cluster size increases, failures have become a frequent event. For example, the Blue Waters supercomputer at the National Center of Supercomputing Applications has a mean time to failure (MTTF) of 4.2 hours [17]. Projections also found that the MTTF of future systems can be as low as one hour [13, 36]. What is lacking in previous efforts to implement MapReduce for large HPC clusters is to design a fault tolerant job execution framework. Building a fault tolerant MapReduce for HPC clusters is a very important but challenging task for two reasons.

Firstly, MapReduce in an HPC cluster relies on MPI to perform failure detection, notification, and recovery through MPI. But, the current MPI does not have direct support for fault tolerance. Indeed, it treats failures as local errors [3]. In the case of failure, MPI provides no guarantee of the global state consistency to the MapReduce job nor ways to recover, which violates both the safety and liveness requirements for fault tolerance [32].

Secondly, the *detect/restart* fault tolerance model of conventional MapReduce is not compatible with the scheduler in most HPC clusters. MapReduce needs the flexibility to schedule and to restart individual processes for failure recovery. HPC schedulers usually use a gang scheduling policy. Under this policy, one application will either run with all processes or wait in a queue until the resources are available. This is favorable for MPI applications because it reduces the communication delay [20, 27]. However, for MapReduce applications, it makes recovery very expensive because increasing the size of a running application incurs hours of additional delay in a busy HPC cluster. Some popular HPC systems like IBM BlueGene/Q even do not support increasing the job size, which makes failure recovery impossible.

In this paper, we design and implement FT-MRMPI, the first fault tolerant MapReduce library built on MPI. We develop a task runner with user-customizable interfaces that provide fine-grained progress tracking. It enables the library to establish locally consistent states for failure recovery. We design distributed masters to manage the job execution and maintain the globally consistent state for all processes. We exploit the semantics of the current MPI standard and extend the error handler for the *checkpoint/restart* model. It allows a failed MapReduce job to be recovered when restarted by the user. We use User Level Failure Mitigation (ULFM) [10], a new fault tolerance interface being considered for future inclusion in the MPI standard, to support a work-conserving *detect/resume* model for efficient fault tolerance. With this model, FT-MRMPI provides au-

SC '15, November 15-20, 2015, Austin, TX, USA

© 2015 ACM. ISBN 978-1-4503-3723-6/15/11...$15.00

DOI: http://dx.doi.org/10.1145/2807591.2807617

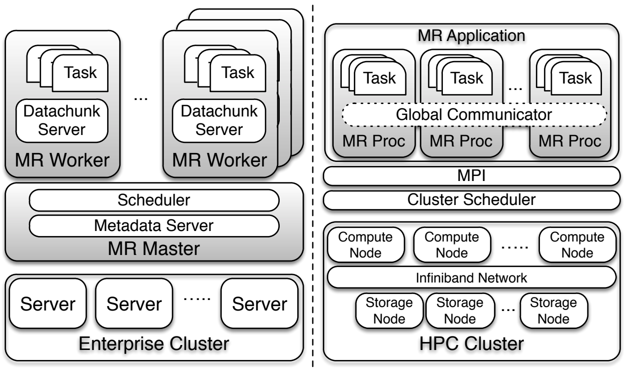


Figure 1: An overview of MapReduce Framework in an enterprise cluster and an HPC cluster.

tomated in-place failure recovery by redistributing the workload of failed processes to the surviving ones. It also makes FT-MRMPI tolerant to continuous failures. We further develop two performance refinements to improve the performance of FT-MRMPI in both normal execution and recovery. We evaluate FT-MRMPI using representative MapReduce benchmarks on a 256-node HPC cluster. The experimental results show that FT-MRMPI effectively masks failures during job execution. FT-MRMPI also reduces the total completion time of a failed job by as much as 39% compared to a MapReduce implementation without fault tolerance.

The rest of this paper is organized as follows. Section 2 introduces the background of MapReduce and discusses existing issues implementing a fault tolerant MapReduce-MPI. Section 3 presents the architectural design and key components of FT-MRMPI. Section 4 discusses the design of the fault tolerance models. Section 5 presents the implementation and performance refinements for FT-MRMPI. Section 6 shows the experimental results and analysis. Related work is presented in Section 7. We conclude this paper in Section 8.

2. BACKGROUND AND MOTIVATION

2.1 MapReduce Fault Tolerance Model

MapReduce [16] is a programming paradigm for large-scale data-intensive processing in distributed clusters. As Figure 1 shows, a conventional MapReduce implementation on an enterprise cluster consists of two subsystems: a distributed file system and a job execution engine with a customized cluster scheduler. Both subsystems form a master/worker architecture. The distributed file system (DFS) in MapReduce uses a replication-based fault tolerance model to provide availability and reliability to the input and output data that resides on it. The job execution engine uses a non-work-conserving *detect/restart* model for fault tolerance. The master monitors the status of each worker node. Once a failure is detected, the master will try to recover the lost intermediate data by rescheduling the affected tasks (both finished and unfinished) on a different worker. These two subsystems offer fault tolerance to MapReduce in dedicated enterprise clusters.

In HPC clusters, MapReduce takes a different design. Unlike the shared nothing architecture in conventional MapReduce clusters, an HPC cluster usually employs a shared disk architecture. As Figure 1 shows, an HPC cluster consists of a collection of compute nodes and a shared distributed storage system connected using high-speed networks. Such a cluster usually has its own scheduler, resource manager, and parallel file system. Therefore, it is hard to use holistic MapReduce implementations like Hadoop on HPC clusters. The only thing that the MapReduce implementation needs to provide is the job execution engine. MapReduce jobs are characterized by their all-to-all communication pattern, which makes it a logical option to build MapReduce for HPC clusters using MPI and exploit portable performance for collective communication. However, this change in platform has a significant impact on the fault tolerance model of MapReduce.

Next, we discuss two specific issues that make the design of a fault tolerant MapReduce for HPC clusters a challenging task, and discuss the opportunities to bring fault tolerance to MapReduce in HPC clusters.

2.2 Missing Fault Tolerance in MPI

MapReduce applications that are built on top of MPI require the direct support of fault tolerance in MPI, which includes failure detection, notification, and recovery. Unfortunately, none of these is supported by the current MPI standard. In MPI-3, failures are reflected as local errors in the relevant communication calls. MPI itself provides no mechanisms for handling process failures [3]. A failure notification on one process does not guarantee that all other processes will receive the same notification. Applications can easily run into an inconsistent state where some processes detect a failure while others continue normal execution or hang without error. In addition, the failure of one process will make the communicator inoperable for collective communications. There is no way to repair or replace the broken communicator without requiring all processes (including the failed ones) to participate. One alternative approach is using the master/worker architecture of conventional MapReduce and let the master process to detect failures. However, the dedicated master process not only is a resource waste but also a single point of failure by itself.

2.3 Scheduling Restrictions on Recovery

Conventional MapReduce recovers failed tasks by rescheduling them to different nodes. However, the HPC schedulers like Maui and PBS Pro are optimized for HPC applications and they do not provide an efficient way for MapReduce applications to do that. Ideally, the processes in one HPC program should be run simultaneously and use the allocated resource exclusively to minimize the communication delay [20, 27]. For this reason, gang scheduling is the most favorable for its all-or-nothing scheduling strategy. All jobs wait in the pending queue until the cluster has enough resources to meet their requests. A running job that tries to get additional resources can significantly affect the scheduling fairness, hence most HPC schedulers restrict the use of resizing running jobs and force the resized job to go back to pending queue. Some systems like the IBM BlueGene/Q have even dropped support for spawning processes in MPI programs. For these reasons, the current fault tolerance model of MapReduce is simply not compatible in HPC clusters, and changing the scheduler of a HPC cluster is not a viable alternative as it can be harmful to HPC applications.

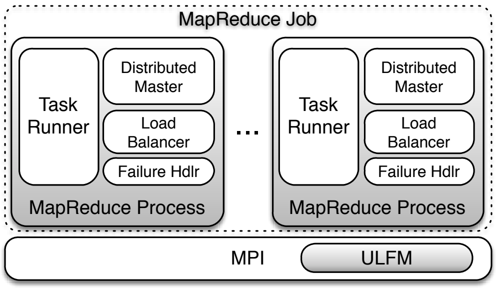


Figure 2: The architecture of FT-MRMPI.

2.4 Our Opportunities

We have found that we can force all processes of an MPI program to exit if any of them detect an error using current MPI semantics. This mimics failure detection and notification. All the processes are terminated, and the user has to restart the failed MapReduce application as a new job. For this reason, the *checkpoint/restart* fault tolerance model [8, 40] is a logically first option for MapReduce because the recovered application can continue processing from the latest checkpoint rather than starting over. Despite the additional overhead that the *checkpoint/restart* model introduces, it has distinct advantages in its compatibility with gang scheduling and it requires no changes to MPI.

Fault tolerance is one of the major focuses in the future MPI standard. One of the proposals is User Level Failure Mitigation (ULFM) proposed in our prior work. It enables application-level fault tolerance by offering interfaces to applications and libraries to mitigate failure. It allows a failed MPI program to recover without restarting the job and enables us to use the *detect/resume* fault tolerance model [15] to recover a failed job without restarting it completely. It provides an automated and efficient fault tolerant job execution for MapReduce by redistributing the workload of failed processes to the surviving ones.

To build a fault tolerant MapReduce in HPC with these models, we need a new framework that traces the job execution state and manages workload distribution so that the work of failed processes can be correctly saved and recovered. These fault tolerance models also need to be carefully tailored to adapt to MapReduce in HPC clusters. Next section, we present FT-MRMPI, a novel framework for MapReduce in MPI that supports both fault tolerance models.

3. SYSTEM DESIGN

FT-MRMPI is a fault tolerant MapReduce framework implemented on MPI. It tracks a consistent state during job execution and supports efficient fault tolerance through two models: *checkpoint/restart* and *detect/resume*. The *checkpoint/restart* model offers the basic fault tolerance using the current MPI semantics. The *detect/resume* model enables automated in-place recovery and a more efficient job execution engine.

3.1 Overview

Figure 2 shows the structure of a MapReduce application using FT-MRMPI. FT-MRMPI consists of four components: *TaskRunner*, *Master*, *FailureHandler*, and *LoadBalancer*. It provides a set of interfaces that enable progress tracking of user-defined tasks. The master is a thread dedicated to job management. It handles the data operations during checkpointing and recovery. It also monitors the job execution status in each process and maintains the global state consistency. The failure handler is a customized MPI error handler that performs the failure notification, state preservation, and recovery. The load balancer estimates the completion time of each process and redistributes the workload to mitigate load imbalance after recovery from failures. We briefly describe some major features of FT-MRMPI in the following.

3.2 Task Runner

The lifespan of a MapReduce job can be divided into three phases: map, shuffle, and reduce. The map and reduce phases are mainly user-defined logics that read input data, process each record, and writes output results. It is not trivial to trace the consistent states in all three phases at a fine granularity.

FT-MRMPI's task runner provides a set of user-customizable interfaces for the map and reduce phases. It embeds the tracing feature into the user-defined logic.

Table 1 shows the interfaces for map and reduce phases in FT-MRMPI. The main purpose of these new interfaces is to delegate the essential operations in a MapReduce job to the library. For example, instead of writing the file operations in the map function, users are expected to tell the library how the input data should be tokenized and how the output records should be serialized. This can be achieved by extending the `FileRecordReader` and the `FileRecordWriter` class templates. The library will perform the read and write operations for a MapReduce job and track the progress at fine granularity. Similarly, the user can also extend the `KVWriter` and the `KMVReader` class templates in case of special operations is needed when handling the intermediate data.

After delegating the I/O operations to the library, the implementation of the map and reduce functions can be largely simplified. The `map` and `reduce` functions only need to contain the job logic that needs to be applied to individual records. We provide the `Mapper` and the `Reducer` class templates for defining map and reduce functions.

With the interfaces, FT-MRMPI generalizes the workflow of map and reduce phases. Algorithm 1 shows an example of a map task in FT-MRMPI. The loop in the map task reads input data using the record reader that a user provides and applies the user-defined map function to each input record. Each iteration has a commit operation that tells FT-MRMPI that the processing of the current record is finished, and the task has reached a consistent state. The workflow of the reduce phase follows the same loop structure.

The state tracing in the shuffle phase is relatively simple because no user code is involved. FT-MRMPI traces the send and receive for each memory buffer in data transmission stage as well as the merging on each partition.

3.3 Distributed Masters

Although a process-local consistent state is sufficient for fault tolerance in the map and reduce phases. It is not enough for the shuffle phase. Unlike the other phases that have no inter-process coordination, the shuffle phase has collective communication between all processes. In the shuffle phase, all processes in the MapReduce job exchange interme-

Table 1: FT-MRMPI Task Runner Interfaces.

Interface	Description
`template class` **FileRecordReader**`<K, V>`	File input reader
`template class` **FileRecordWriter**`<K, V>`	File output writer
`template class` **KVWriter**`<K, V>`	Key-value buffer writer
`template class` **KMVReader**`<K, V>`	Key-multivalue buffer reader
`template class` **Mapper**`<INKEY, INVALUE, OUTKEY, OUTVALUE>`	Map task
`template class` **Reducer**`<INKEY, INVALUE, OUTKEY, OUTVALUE>`	Reduce task
`int32_t` **map**`(INKEY&, INVALUE&, OUTKEY&, OUTVALUE&, void*)`	User-defined map function
`int32_t` **reduce**`(INKEY&, List<INVALUE>&, OUTKEY&, OUTVALUE&, void*)`	User-defined reduce function

Algorithm 1 Workflow of the map task.

```
1: Variables: Record reader $RReader$; Record writer
   $RWriter$; Mapper $M$; Number of processed records $n$;
2:
3: function MAPTASK($M$, $RReader$, $RWriter$)
4:     n ← 0
5:     while $< k, v > \leftarrow RReader.next()$ do
6:         n += $M.map(< k, v >, RWriter)$
7:         commit()
8:     end while
9:     return n
10: end function
```

diate data using a series of `MPI_Alltoallv` calls. If a failure occurs in the middle of communication, there is no way to determine which part of the memory buffer is successfully transmitted. All processes have to perform a coordinated rollback to the last successful transmission. For this reason, FT-MRMPI must maintain a globaly consistent state within the shuffle phase.

FT-MRMPI uses distributed masters to manage its job execution. It maintains the globally consistent state of the job and manages the workload distribution during initialization and recovery. The master thread is responsible for creating tasks, assigning tasks to processes, and monitoring the task execution on each process. During the initialization of the MapReduce application, the master threads traverse the location of input files and divide these file among all master thread. Each master thread splits the input files into fixed-size chunks. Then, it creates one task for each input chunk. The master thread uses a hashing-based task assignment algorithm that calculates the rank of the process for each task using its task ID. As all master threads perform the same procedure in creating and assigning tasks, no coordination is needed in this stage. After this procedure, each master thread maintains two task status tables: one for local tasks and the other for global tasks. During the job execution, the master thread monitors the progress of local tasks and periodically broadcasts the status of local tasks to other master threads. In such way, the global task status table in each master thread can be updated to a consistent state. Thus, they provide the crucial information for failure recovery.

In addition to the job management, the master thread also handles the data movement in fault tolerance. We will discuss this later with the fault tolerance models.

3.4 Automated Load Balancing

Workload imbalance is a pervasive issue in MapReduce due to the inherent non-uniformity in data [7, 30, 33]. The recovery of failure can further aggravate the already skewed workload distribution of a MapReduce when redistributing the unfinished work of failed tasks. It can cause most processes idle and waiting for the ones that handle the workload of the failed process.

In FT-MRMPI, we use an online profiling-based automated load balancing approach. We add an agent thread for each process. The agent monitors the input data size and the processing time of each process. During the execution of the job, the agent makes k observations of the amount of time, that each process spends on processing its data. As suggested in [38, 39], it derives a linear model between input data size and the completion time of map and reduce phases:

$$t_{i,j} = a_i + b_i \cdot D_i + \varepsilon_j,$$

where t_i is the job completion time of the jth process in the ith observation, D_i is the corresponding input size and ε_j is the error of jth process. We use linear regression to obtain the parameters for all processes. Once a model is obtained, the final completion time of a process is calculated by replacing D_i with the actual input size. Based on this prediction, FT-MRMPI proportionally divides and distributes the workload of the failed processes to ensure that all surviving processes finish at the same pace.

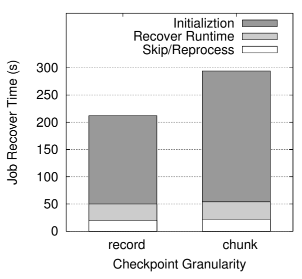
4. FAULT TOLERANCE MODELS

4.1 Checkpoint/Restart Model

With the trace of the consistent states during the job execution, we are able to create checkpoints by combining the job states and intermediate data. It offers support for the *checkpoint/restart* fault tolerance model. The failed MapReduce job is terminated, and a new job will be started to recover the execution from the last checkpoint.

As we mentioned in Section 2, the *checkpoint/restart* fault tolerance model only requires failure detection and notification, which we discovered can be achieved by exploiting the error reporting semantics in current MPI-3. Since failures are reflected as errors in the relevant communication calls, we use a customized error handler to catch these errors. In this way, some of the processes in the MapReduce program can detect the failure and propagate it by terminating themselves with the `MPI_Abort` call. The process manager in MPI will broadcast the termination of the process and trap all surviving processes into the error handler. Thus, the failure notification will be propagated to all processes, and the *checkpoint/restart* model will work as designed for.

We designed FT-MRMPI so as to periodically make checkpoints during normal execution. As checkpointing introduces overhead to the job execution, we carefully design the checkpointing mechanism in FT-MRMPI to minimize its performance impact.



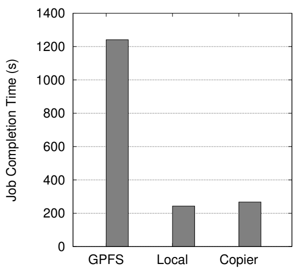


Figure 3: Recovery time of different checkpoint granularities.

Figure 4: Performance impact of different checkpoint locations.

4.1.1 Asynchronous Checkpointing

Typically, checkpointing can be performed in two ways: synchronously or asynchronously. Synchronous checkpointing requires all processes to coordinate when making the checkpoints. It usually results in all processes writing checkpoints to disks simultaneously. For MapReduce applications that involve processing large amounts of data, synchronous checkpointing can significantly slow down the job execution due to the resource contention at the storage device. More importantly, the inherent workload imbalance in MapReduce jobs is a pervasive problem that makes all processes unable to run in a synchronized pace [7, 21, 24]. Using synchronous checkpointing will force fast processes to wait for the slow ones and further reduce the job performance.

Thus, in FT-MRMPI we develop each process to perform asynchronous checkpointing. This design reduces the overhead of checkpointing and minimizes the performance impact. FT-MRMPI synchronizes all processes at the end of each phase to maintain the consistency of checkpoints. Because of the inter-process independency in the map, shuffle and reduce phases, asynchronous checkpointing does not affect the correctness of preserved job state.

4.1.2 Granularity of Checkpoints

The granularity of a checkpoint has significant impact on the amount of lost work in failure. The finer the checkpoint granularity is, the less the work is lost. Thus, the fine-grained checkpointing can significantly reduce the recovering time.

In FT-MRMPI, the input data of a MapReduce job is divided into multiple input chunks. Each process is assigned multiple input chunks. As MapReduce flushes the intermediate data to disks when one input chunk is processed, it makes a natural checkpoint. However, when a checkpoint is made at the input chunk level, all work on partially processed input chunks will be lost after the failure. The job need to reprocess these parts of the work when restarted. Depending on the complexity of the job, reprocessing can be substantially expensive. Since FT-MRMPI can trace the consistent state of individual records, we can also make checkpoints at the record level. In this case, the lost work in failure can be minimized. The restarted job will have to read the input data and skip the processed records, which is much cheaper than reprocessing.

Figure 3 shows the recovery time of pagerank benchmark with different checkpoint granularities. The results show that when checkpointing at the input chunk level, the job recovery time is 38% longer than the case of checkpointing at the record level. The decomposition of the recovery time shows that reprocessing takes significant longer time than record skipping does. Base on these observations, we use record level checkpointing in FT-MRMPI for improved recovery performance.

However, the major drawback of fine-grained checkpointing is its high overhead to the job execution. In FT-MRMPI, the number of records in one checkpoint is a user customizable parameter. It needs to be carefully tuned according to the performance and characteristics of the storage system. We will discuss the overhead of the fine-grained checkpointing in Section 6.

4.1.3 Location of Checkpoints

As the size of each record and its intermediate data is usually small, checkpointing at such a fine granularity results in many small I/O operations. In a typical HPC cluster, the persistent storage is a shared storage system, which is usually optimized for large I/O operations. The small I/O from fine-grained checkpointing can easily slow down the job execution speed.

To reduce the performance impact of checkpointing, FT-MRMPI makes checkpoints to local disks or the scratch file system and uses the master thread to move these checkpoints to the persistent storages. In this way, FT-MRMPI can aggregate the checkpoints and avoid the performance impact of small I/O. This also overlaps the I/O operations to the shared storage with the computation of the tasks, and avoids causing tasks to wait for checkpointing. Figure 4 shows the job completion time of wordcount benchmark due to different checkpointing locations. Using the background copier significantly reduces the delay due to checkpointing to persistent disks.

In summary, the *checkpoint/restart* approach provides the basic fault tolerance for MapReduce using the features of the current MPI standard. But it has major drawbacks. First, the local disk is not available in all HPC clusters. In clusters that does not have local disks, making checkpoints to shared storage systems incurs significant overhead. Second, it needs the user intervention to recover since the failed job needs to be resubmitted. The resubmitted job may have to wait for hours in the queue on a busy HPC cluster. Third, it is vulnerable to failures in the restarted job. Continuous failures can result in an endless loop of failing and restarting. Fourth, restarting a failed job makes all processes to read their checkpoints, which results in a significant amount of disk I/O and long recovery time. In order to address these issues, we design a second fault tolerance model for MapReduce on HPC clusters.

4.2 Detect/Resume Model

The *detect/resume* fault tolerance model is evolved from the *detect/restart* model in the conventional MapReduce. The major difference between these two models is that *detect/resume* model performs failure recovery by redistributing the workload of the failed processes to the surviving ones rather than restarting the new processes. The recovered MapReduce job will run with fewer processes, but it avoids the incompatibility between the *detect/restart* model and the HPC schedulers. In FT-MRMPI, the *detect/resume* model uses the User-Level Failure Mitigation (ULFM) interface to mask failures without restarting the job. It recovers the work of failed processes in either a work-conserving or a non-work-conserving way.

4.2.1 Failure Masking with ULFM

ULFM is designed to provide the minimal interface necessary to restore the complete MPI capability to transport messages after failures. It defines the set of functions that can be used by applications to repair the state of MPI. In the design of the *detect/resume* model for MapReduce, we use ULFM for uniform failure notification and rebuilding the communicator.

When processes attempt to recover, they need to propagate the detected failure and interrupt the normal execution of other processes. ULFM provides the `MPI_Comm_revoke` function for failure notification. When one process invokes this function on a communicator, MPI first declares the communicator as revoked locally, then sends out a revoke packet to other processes. Each process that receives the revoke packet revokes all ongoing communication calls on the revoked communicator and traps into the error handler. It is similar to the `MPI_Abort` used in the *checkpoint/restart* model with the exception that it does not abort the process. Instead, it interrupts the normal job execution flow and marks the communicator inoperable for the application. Note that, the tasks in map and reduce phases are independent from each other, and the failure in one task does not requires all tasks to stop normal execution to recover. However, all master threads have to to stop normal execution of tracking job progress since they need to identify the failed tasks and redistribute workload coordinately.

After all processes are notified of the failure, the application starts the recovery by replacing the broken communicator and restoring the full communication capacity. ULFM supports this by providing the `MPI_Comm_shrink` function that creates a new functional communicator based on an existing, revoked communicator. It does this as follows. First, it reaches an agreement among all surviving processes about the group of failed processes. Then it creates a new group of all processes but the failed ones and creates a new communicator based on this group. At this point, the detected failure is masked and the processes are ready for normal execution.

4.2.2 Resuming Job Execution

After masking the failure, the workload of the failed processes needs to be redistributed to the remaining processes to resume normal execution. In FT-MRMPI, we provide two ways to resume: work-conserving and non-work-conserving.

The work-conserving method is to integrate the *detect/resume* model with the checkpointing capability. Similar to the *checkpoint/restart* model, the *detect/resume* model periodically makes checkpoints to save the job state and intermediate data. During the workload redistribution, the remaining processes will recover the lost work by loading the checkpoints of the failed processes. Therefore, the work of all finished tasks is preserved. Comparing to restarting the entire job, the surviving processes in the *detect/resume* model only need to read the checkpoints of the failed processes, which significantly reduces the I/O load and boosts the recovery speed.

On the other hand, the *detect/resume* model can also use the non-work-conserving method to perform recovery. In this case, the surviving processes recover the lost work by re-running all the tasks from the failed processes. Since all the intermediate data can be recovered throughout the re-execution, it does not need to make checkpoints to preserve job state. However, this also results in longer recovery time, especially for jobs that involves intensive computation. Also, because the recovered MapReduce job will run with a shrunken size, reprocessing lost work can significantly prolong the overall completion time. We will discuss the performance impact and trade-offs of work-conserving and non-work-conserving *detect/resume* model in Section 6.

5. IMPLEMENTATION AND REFINEMENTS

We implement FT-MRMPI based on the MapReduce-MPI (MR-MPI) library [32]. We reuse the communication code and rewrite the rest of the library to support FT-MRMPI's job framework. Then, we add the *checkpoint/restart* and *detect/resume* models to the new library. We also develop two refinements that improve the performance of FT-MRMPI.

5.1 Prefetching for Recovery

FT-MRMPI saves checkpoints on local disks and uses the master thread to move data to persistent storage. This significantly reduces the overhead of checkpointing, but it also increases the recovery time as processes need to read checkpoints from the shared persistent storage. We address this issue by enabling prefetching for recovery. Since all checkpoints are made in chronological order, it is easy to predict access order of these checkpoints. Hence, we again utilize the master thread to move the checkpoints from persistent storage to node-local disks during the recovery. Processes only need to recover from local disks, reducing the I/O overhead of recovery.

5.2 Two-pass KV-KMV Conversion

The KV-KMV conversion is the process that groups the key-value pairs by their keys. It is an essential process in the shuffle stage. Because it accesses all the intermediate data on disks, it is critical to the performance of the shuffle stage. The original MR-MPI used a complex algorithm that reads and writes the intermediate data four times. It not only decreases the speed of shuffle stage, but also makes tracking shuffle progress a complex task, as these four passes are nested together.

In FT-MRMPI, we propose and develop a two-pass KV-KMV conversion that improves the shuffle performance and reduces the complexity of tracking shuffle progress. The first pass reads the key-value pairs in the input buffer and tries to put the values of the same key together. Inspired by the log-structure file system [34], we divide the output buffer of the first pass into small fixed-size segments. The values of different keys are saved in different segments. Due to the randomness of key-value pair distribution, the values of one key can have multiple non-contiguous segments. After the first pass finishes, the second pass merges these non-contiguous segments into one large segment and finishes the KV-KMV conversion. The two-pass KV-KMV conversion significantly reduces the I/O operations during shuffle and simplifies the progress tracking of the shuffle phase.

6. EVALUATION

We perform the following experiments on a 256-node Intel processor-based HPC cluster where each node is equipped with a 2-way 8-core Intel Xeon X5550 CPU, 36 GB memory, 250 GB SATA hard drive. These nodes are connected using Mellanox Infiniband QDR in a fat-tree topology. We used a branch of Open MPI 1.7 with ULFM additions [5] as our MPI environment. In the experiments, we set the processes per node (ppn) to 8. We reserve the 256 nodes from the cluster for experiment use. We evaluate FT-MRMPI with a maximum of 2048 processes.

6.1 Benchmarks and Applications

Wordcount - Wordcount is a common benchmark for MapReduce. It is a single stage MapReduce job. It counts the occurrences of each word in a large collection of documents. The map tasks parse input data and emit `<word,count>` tuples. The reduce tasks add up the counts for a given word from all map tasks and outputs the final count. It is a good example to study the performance and communication efficiency of a MapReduce library as the job involves very little computation.

Breadth First Search - BFS is a representative benchmark for graph processing. It is a single stage iterative MapReduce job. The map tasks visit and color vertices. The reduce tasks combine the visiting information of each vertex. It repeats the map and reduce tasks until the input graph is fully traversed.

PageRank - PageRank is a representative benchmark for multi-stage iterative MapReduce job. In each iteration, PageRank has two stages. Each stage is a complete map and reduce process. It parses the links on each web page and calculates the pagerank. The iteration continues until all links have been evaluated, and the algorithm converges.

MR-MPI-BLAST - MR-MRI-BLAST [37] is a parallel BLAST, which is used for comparing primary biological sequence information. MR-MPI-BLAST uses high-level methods of the NCBI C++ Toolkit. It is designed as an iterative MapReduce job. The map task searches sequences against a given DB partition. The reduce task sorts each search hit by the E-value and append hits to files.

6.2 Overhead of Checkpointing

Checkpointing in FT-MRMPI is used in both the *checkpoint/restart* model and the *detect/resume* model. Because checkpointing introduces additional operations to job execution, it is important to understand its performance impact. Here we use a wordcount benchmark job to study checkpointing overhead. We use 128 GB input data and measure the job completion time of wordcount in a run without any failure. We compare the performance of FT-MRMPI using the *checkpoint/restart* model and the *detect/resume* model in both work-conserving (WC) and non-work-conserving (NWC) mode with MR-MPI. We disabled the two refinements in our implementation for a fair comparison. We repetitively test FT-MRMPI with the number of processes ranging from 32 to 2048 to obtain the strong scalability trend.

Figure 5 shows the normalized job completion time of the wordcount job. The results show that FT-MRMPI takes 10%−13% longer time to finish the job when using the *checkpoint/restart* and the *detect/resume* (WC) model. However, FT-MRMPI using the *detect/resume* (NWC) model, which does not perform checkpointing, achieved similar job completion time as MR-MPI. We conclude that the performance difference is mainly due to the overhead of checkpointing. The strong scaling results of the test job shows poor scalability beyond 256 processes. After testing with the IOR benchmark [6], we conclude that this is a performance bottleneck on the shared storage systems. The storage bottleneck further increases the overhead of checkpointing as the checkpoints need more time to be moved to the persistent storage.

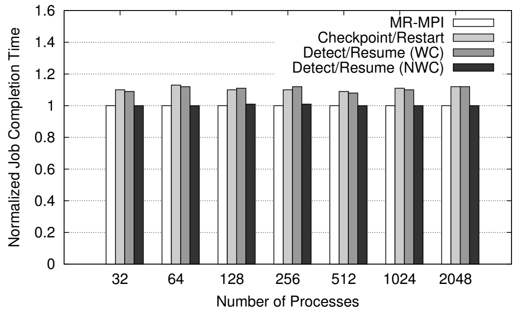


Figure 5: Normalized job completion time without failure.

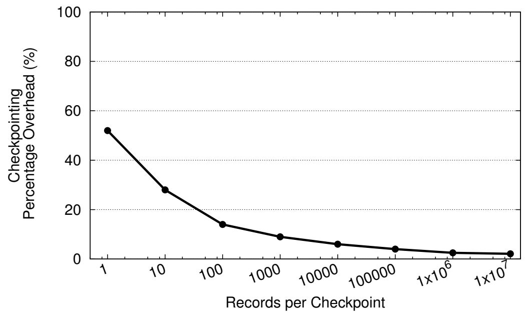


Figure 6: Percentage overhead of checkpointing granularity.

Checkpointing at fine granularity reduces the loss of work, but also increases the overhead. In the previous experiment, we make checkpoints at every 100 records. Here we run the same wordcount job with 256 processes with different checkpointing granularities. Figure 6 shows the percentage overhead of checkpointing. The overhead is huge when making checkpoints at very high frequency (one record per checkpoint) and drops significantly when the checkpoint frequency is reduced from every record to every 100 records. Note that the ideal frequency of making checkpoints depends on the application and the cluster. For our test, the average number of records per process is about 4×10^7. Checkpointing every 10^5 records provides a reasonably low overhead without losing much benefit from fine-grained checkpointing.

FT-MRMPI uses a background copier thread to move checkpoints from local disks to the persistent storage. As the copier thread uses the CPU core with the main thread that does the computation, it is important to know the performance impact of the copier thread. Figure 7 shows the decomposition of the job completion time of MR-MPI and the *checkpoint-restart* model in FT-MRMPI. The results show that CPU time of the copier thread is only 3% of the total job execution time. This is mainly due to the simple design of the copier. However, due to the increase I/O operation of checkpointing and data movement between local disks and

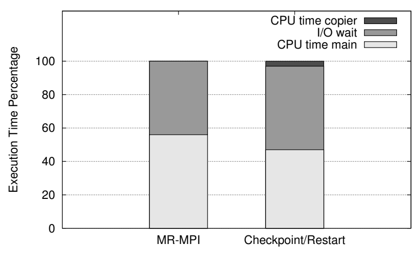


Figure 7: Overhead of copier thread.

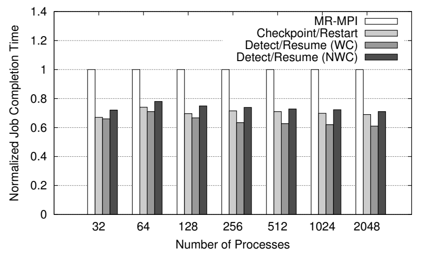


Figure 8: Normalized job completion time of failed and recovery run.

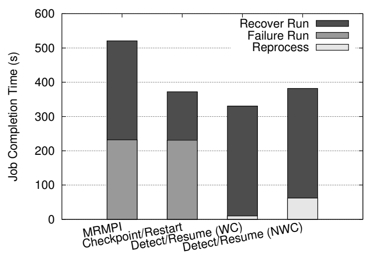
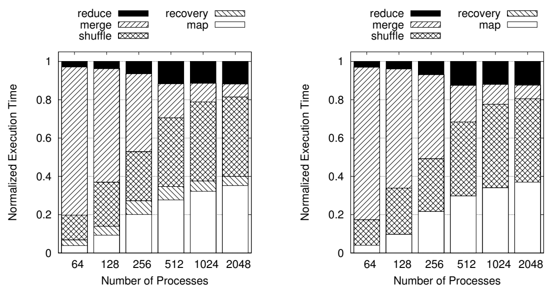


Figure 9: Completion time of failure and recovery runs.



(a) *checkpoint/restart* (b) *detect/resume* (WC)

Figure 10: Decomposition of the aggregated time for all processes.

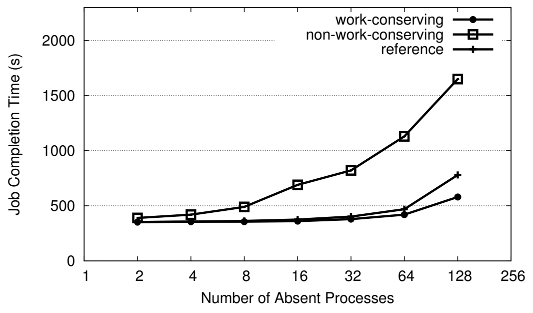


Figure 11: Completion time for PageRank with continuous failures.

the persistent storage, the I/O wait time is 11% longer than MRMPI. The main overhead for periodically checkpointing is still the increased number of I/O operations.

6.3 Performance Benefit of Fault Tolerance

Although enabling fault tolerance models in FT-MRMPI introduces overhead to the job execution, it significantly reduces the potential time needed for recovering the job after failure. Here we demonstrate the performance benefit of fault tolerance. We run a wordcount job with 128 GB input data. We measure the total time of two runs. The first run has one failed process at the reduce phase. The second run is the recovery run without any further failure. The total time of these two runs as the performance metric.

Figure 8 shows that FT-MRMPI using the *checkpoint/restart* model outperforms MR-MPI by up to 33%. Since MR-MPI is not fault tolerant, we use the total time of a failed run and a successful run for comparison. FT-MRMPI using the *detect/resume* (WC) model not only outperforms MR-MPI by up to 39%, it also achieves 10% − 12% shorter job completion time than using the *checkpoint/restart* model. FT-MRMPI using the *detect/resume* (NWC) spends 12% − 17% longer time to finished the job. The extra time is used to reprocess all the tasks of the failed process.

Figure 9 shows the completion time of the failure and recovery runs with 256 processes. Comparing the *checkpoint/restart* model with MR-MPI, it is clear that recovering from checkpoints significantly reduces the time in the recovery run. We also observe the impact of using checkpointing with the *detect/resume* model. The detect/resume (NWC) model, which has no checkpointing, takes 15% longer than the *detect/resume* (WC) model does due to the reprocessing of all tasks in the failed process. However, for simple MapReduce jobs like wordcount, the *detect/resume* (NWC) model still offers decent performance compared to MR-MPI.

The *checkpoint/restart* model and the *detect/resume* (WC) model achieved close performance in this case. The difference in the overall time of these two models is mainly because of the recovery time. As *detect/resume* only needs to read the checkpoints of the failed processes, it takes significantly less time to recover. Figure 10 shows the decomposition of job completion time of the *checkpoint/restart* and *detect/resume* (WC) models. It is clear that the recovery in the *checkpoint/restart* model takes longer than the *detect/resume* (WC) model does.

6.4 Mitigating Continuous Failures

One major reason that FT-MRMPI supports the *detect/resume* fault tolerance model is to mitigate continuous failures. The in-place recovery capability of the *detect/resume* makes it the best choice for this scenario. Here we use the BFS and the PageRank benchmarks to evaluate how FT-MRMPI handles continuous failures in complex jobs.

For each job, we prepared 250 GB of input data. We run these jobs with 256 processes to avoid the I/O performance bottleneck. Continuous failures are injected by randomly terminating one process every 5 seconds. We measure the job completion time of both the work-conserving and the non-work-conserving *detect/resume* models and compare to a reference time. The reference time is measured as the failure-free job completion time with the same number of absent processes.

Figure 11 shows the job completion time of pagerank with different number of failed nodes. The results show a signif-

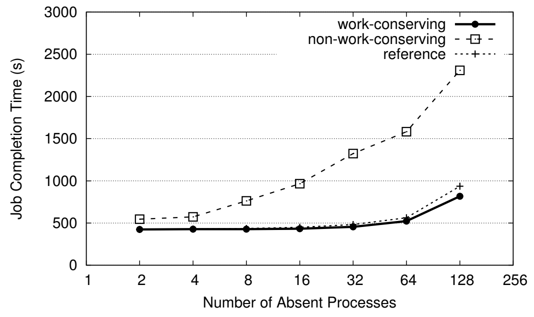


Figure 12: Completion time for BFS with continuous failures.

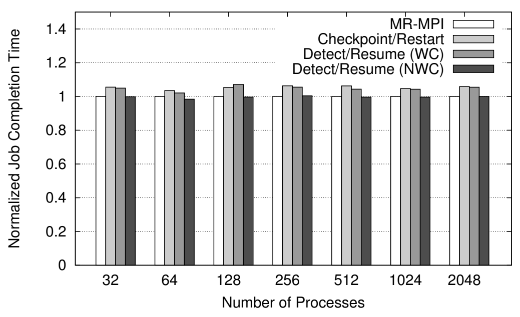


Figure 13: Normalized job completion time of `MR-MPI-BLAST`.

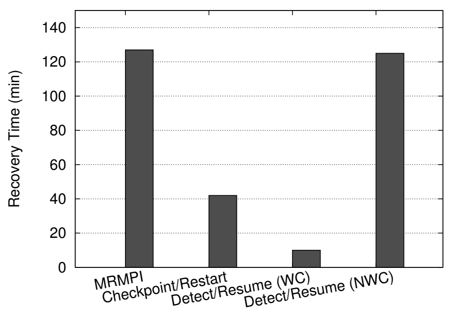


Figure 14: Recovery time of MR-MPI-BLAST.

icant performance difference between the work-conserving and the non-work-conserving *detect/resume* models. The reason is that the non-work-conserving job will lose all previously finished work. In the case of continuous failures, the MapReduce job cannot produce any useful work until no more failures occur. The work-conserving *detect/resume* model has a clear advantage in this case as no previously finished work will be lost. It also results in better performance as some of the work is done when there were more living processes. Figure 12 shows the same experiment with BFS benchmark. The results concord with the observation from the pagerank benchmark.

Note that the job completion time of reference can be higher than that of the work-conserving *detect/resume* model. In the reference, we mimics the scenario of no in-place recovery for the running job, i.e. the absence of processes forces the whole job running with less working processes. On the contrast, in the work-conserving *detect/resume* model, the job starts with the full capacity and gradually loses processes. The average number of working processes can be higher than the reference number. Thus, the work-conserving *detect/resume* model achieved shorter job completion time than the reference did.

6.5 Performance with MR-MPI-BLAST

Porting scientific applications to HPC clusters can be challenging when the applications are designed as a serial program. MapReduce offers automated data paralleling processing, which can be very useful in parallelizing serial scientific applications. One good example is MR-MPI-BLAST, which enables parallel BLAST using the serial NCBI BLAST library. Here we study how this type of scientific applications can benefit from FT-MRMPI. For this experiment, we modified the original MR-MPI-BLAST to use the new interfaces that FT-MRMPI offers. We built the query dataset which contains 12,000 queries from the NCBI RefSeq database.

Figure 13 shows the normalized job completion time with no failure. The results show that FT-MRMPI takes $5\% - 6\%$ longer time to finish the job when using the *checkpoint/restart* and *detect/resume* (WC) models. The results also show that FT-MRMPI using *detect/resume* (NWC) model achieved similar job completion time as MR-MPI did. The overhead of checkpointing with MR-MPI-BLAST is smaller than it with wordcount benchmark. The reduction of the overhead is mainly because MR-MPI-BLAST uses the NCBI C++ Toolkit library to process the queries. FT-MRMPI was not making checkpoints when the control path is in the external libraries. FT-MRMPI can only provide limited fault tolerance since the failure in the external libraries is no recoverable.

However, this limited fault tolerance still greatly reduces the time required to recover from failure. Figure 14 shows the average recovery time of MR-MPI-BLAST with 256 processes. The results show that FT-MRMPI with *checkpoint/restart* model and *detect/resume* (WC) model outperformed MR-MPI with 65% and 91% shorter average recover time, respectively. The *detect/resume* (NWC) model achieved similar recovery time with MR-MPI, which is due to the high cost of reprocessing in MR-MPI-BLAST. It is clear that the significant reduction in the recovery time worth the overhead of checkpointing for computation intensive MapReduce job.

6.6 Impact of Performance Refinements

We developed two refinements to improve the performance of FT-MRMPI. Here we study the performance impacts of them using wordcount benchmark and 128 GB input data.

We developed a prefetching mechanism for recovery. Figure 15 shows the recovery time of reading checkpoints from local disk, GPFS, and GPFS with prefetching. The results show that enabling prefetching for reading from GPFS reduces the recovery time by $52\% - 57\%$. It effectively bridges the performance gap between recovery from the local disks and recovery from GPFS. We designed and develpoed the two-pass KV-to-KMV conversion for FT-MRMPI. Figure 16 shows the performance of the KV-to-KMV conversion in FT-MRMPI and MR-MPI. By reducing the 4-pass conversion algorithm with the 2-pass conversion algorithm, FT-MRMPI significantly reduces time of KV-to-KMV by more than 50%.

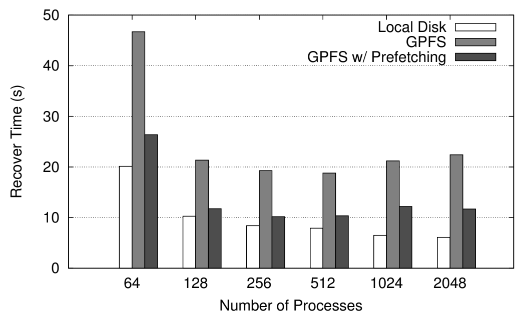


Figure 15: Recovery performance impact of prefetching.

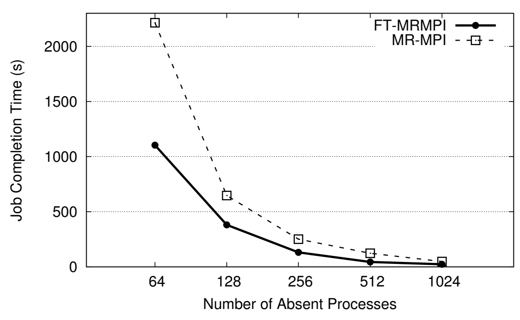


Figure 16: Performance of the KV-to-KMV conversion in FT-MRMPI and MR-MPI.

7. RELATED WORK

MapReduce is a popular programming model and a software framework to support distributed computing and reliable large data processing on clusters of commodity hardware [16]. Fault tolerance is a one of major features of popular MapReduce implementations like Hadoop [1]. It is able to tolerate failures of different types including node crashes, task process crashes, task software faults, and hanging tasks. Hadoop achieves this by using its own distributed file system and cluster scheduler that grant MapReduce full control of the cluster. There are a few studies on further improving the fault tolerance of MapReduce [14, 29, 35]. However, they only focus on the resilience of the distributed file system or intermediate data. None of them explores the fault tolerance issues of MapReduce in clusters where the full-control is not available.

Fault tolerance for HPC clusters is not a new subject. Research specifically on the *checkpoint/restart* model [8, 40] has been going on for years. Berkeley Lab Checkpoint/Restart (BLCR) [18] is the checkpoint/restart library most commonly used by applications and MPI implementations. It provides a way for saving full-system checkpoints. BLCR support was added to many MPI implementations including the most popular open source implementations MPICH [4] and Open MPI [25]. However, system-level checkpointing is too heavy-weight for MapReduce applications. The large memory footprint increases the time spent writing checkpoints, which makes them less favorable for MapReduce.

Researchers began to explore ways to improve the time necessary to write each checkpoint for large applications. Initially, the focus was on moving the checkpointing model from a synchronous model, where all processes simultaneously write a checkpoint to disk after quiescing the network, to a more asynchronous one, where checkpoints could be taken independently and messages between the checkpoints are logged to allow the system to replay them after rolling back to a previous checkpoint [12]. This work improved the checkpoint time but also increased the memory requirement of checkpointing due to the message logging. More work was then done to make checkpoints even smaller and store only the most necessary information. This led to a large body of work involving application-level checkpointing including FTI [9], SCR [31], or GVR [2].

While fault tolerance in MPI has been studied extensively [11, 19, 22, 28], our previous work User-Level Failure Mitigation (ULFM) [10] is a relative newcomer. Its goal is to form the foundational tools necessary to allow any model of resilience (roll-back recovery, roll-forward recovery, application-based fault tolerance, natural fault tolerance, transactions, etc.) to be constructible based on the provided interface in MPI. ULFM enables the application to determine the data that needs to be saved during the failure mitigation and recovery process. Our previous work provides the interfaces for us to develop the *detect/resume* fault tolerance model for FT-MRMPI.

8. CONCLUSION

Running MapReduce jobs on HPC clusters using MPI offers a useful programming model for high performance data analytics. However, the lack of support for fault tolerance in MPI and the incompatibility between MapReduce fault tolerance model with gang schedulers make reliable job execution exceptionally hard in HPC clusters. In this paper, we design FT-MRMPI, the first fault tolerant MapReduce framework for HPC clusters. It has a novel task runner design with distributed masters for tracking job execution. By exploiting the existing semantics in error handling, FT-MRMPI supports the *checkpoint/restart* fault tolerance model without changing the MPI library. We use the User-Level Failure Mitigation interfaces for MPI to implement a work-conserving *detect/resume* fault tolerance model that is tailored for MapReduce in HPC clusters. We implemented FT-MRMPI as a library on top the MPI and evaluated its effectiveness on a 256-node HPC cluster with representative benchmarks. Experimental results show that FT-MRMPI is able to reduce the overall completion time of a failed job by as much as 39%. FT-MRMPI effectively masks failures during job execution.

The future work will exploiting the multiple communicator and communication group in MPI-3 to further improves the efficiency of failure detection and recovery in FT-MRMPI.

Acknowledgement

This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC02-06CH11357, and NSF CAREER award CNS-0844983 and research grants CNS-1217979 and CNS-1422119. We gratefully acknowledge the computing resources provided on Fusion, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

9. REFERENCES

[1] Apache Hadoop Project. http://hadoop.apache.org.

[2] Global view resilience. https://sites.google.com/site/uchicagolssg/lssg/research/gvr.

[3] MPI: A message-passing interface standard. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[4] MPICH. http://www.mpich.org.

[5] Open MPI with User-Level Failure Mitigation. http://www.fault-tolerance.org.

[6] The IOR HPC Benchmark. http://sourceforge.net/projects/ior-sio.

[7] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. ShuffleWatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *Proc. of the USENIX conference on Annual Technical Conference (ATC)*, 2014.

[8] G. Barigazzi and L. Strigini. Application-transparent setting of recovery points. In *Proc. of IEEE Int'l Symposium on Fault-Tolerant Computing (FTCS)*, 1983.

[9] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *Proc. of Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[10] W. Bland, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra. A proposal for user-level failure mitigation in the mpi-3 standard. In *Tech. Rep., Department of Electrical Engineering and Computer Science, University of Tennessee*, 2012.

[11] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. A checkpoint-on-failure protocol for algorithm-based recovery in standard mpi. In *Euro-Par 2012 Parallel Processing*, 2012.

[12] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a scalable fault tolerant mpi for volatile nodes. In *Proc. of ACM/IEEE Conference on Supercomputing (SC)*, 2002.

[13] F. Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *Int'l Journal of High Performance Computing Applications*, 23(3):212–226, Aug. 2009.

[14] P. Costa, M. Pasin, A. N. Bessani, and M. Correia. Byzantine fault-tolerant mapreduce: Faults are not just crashes. In *Proc. of the IEEE Int'l Conference on Cloud Computing Technology and Science (CLOUDCOM)*, 2011.

[15] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High performance linpack benchmark: A fault tolerant implementation without checkpointing. In *Proc. of the Int'l Conference on Supercomputing (ICS)*, 2011.

[16] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[17] C. Di Martino, Z. Kalbarczyk, F. Baccanico, J. Fullop, and W. Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *Proc. of IEEE/IFIP Int'l Conference on Dependable Systems and Networks (DSN)*, 2014.

[18] J. Duell. The design and implementation of berkeley lab's linuxcheckpoint/restart. In *Technical Report LBNL-54941*, 2002.

[19] G. E. Fagg and J. Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *Proc. of the European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2000.

[20] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing (JPDC)*, 16:306–318, 1992.

[21] R. Gandhi, D. Xie, and Y. C. Hu. PIKACHU: How to rebalance load in optimizing mapreduce on heterogeneous clusters. In *Proc. of the USENIX conference on Annual Technical Conference (ATC)*, 2013.

[22] W. Gropp and E. Lusk. Fault tolerance in message passing interface programs. *Int'l Journal of High Performance Compututing Applications*, 18(3):363–372, Aug. 2004.

[23] Y. Guo, J. Rao, C. Jiang, and X. Zhou. FlexSlot: Moving hadoop into the cloud with flexible slot management. In *Proc. of ACM/IEEE Int'l Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[24] Y. Guo, J. Rao, and X. Zhou. iShuffle: Improving hadoop performance with Shuffle-on-Write. In *Proc. of USENIX/ACM Int'l Conference on Autonomic Computing (ICAC)*, 2013.

[25] J. Hursey, J. Squyres, T. Mattox, and A. Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for open mpi. In *Proc. of IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[26] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, 2007.

[27] M. A. Jette. Performance characteristics of gang scheduling in multiprogrammed environments. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC)*, 1997.

[28] A. Kanevsky, A. Skjellum, and A. Rounbehler. MPI/RT-an emerging standard for high-performance real-time systems. In *Proc. of the Int'l Conference on System Sciences*, 1998.

[29] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *Proc. of the ACM Symposium on Cloud Computing (SOCC)*, 2010.

[30] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating skew in mapreduce applications. In *Proc. of the ACM SIGMOD*, 2012.

[31] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proc. of Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[32] S. J. Plimpton and K. D. Devine. Mapreduce in mpi for large-scale graph algorithms. *Parallel Computing*,

37(9):610–632, Sept. 2011.

[33] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. In *Proc. of VLDB*, 2013.

[34] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[35] A. Sangroya, D. Serrano, and S. Bouchenak. MRBS: Towards dependability benchmarking for hadoop mapreduce. In *Proc. of the Int'l Conference on Parallel Processing Workshops (Euro-Par)*, 2012.

[36] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78, 2007.

[37] S.-J. Sul and A. Tovchigrechko. Parallelizing BLAST and SOM algorithms with MapReduce-MPI library. In *Proc. of IEEE Int'l Symposium on Parallel and Distributed Processing Workshop and PhD Forum (IPDPSW)*, 2011.

[38] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *Proc. of the ACM Int'l Conference on Autonomic Computing (ICAC)*, 2011.

[39] A. Verma, L. Cherkasova, and R. H. Campbell. Resource provisioning framework for mapreduce jobs with performance goals. In *Proc. of the ACM/IFIP/USENIX Int'l Conference on Middleware*, 2011.

[40] A. Y. H. Zomaya, editor. *Parallel and Distributed Computing Handbook*. McGraw-Hill, Inc., New York, NY, USA, 1996.