Modeling and Designing Fault-Tolerance Mechanisms for MPI-based MapReduce Data Computing Framework

Jian Lin, Fan Liang, Xiaoyi Lu, Li Zha, Zhiwei Xu
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190, China
{linjian,liangfan,luxiaoyi,char,zxu}@ict.ac.cn

***Abstract*—Fault-tolerance is a significant property for distributed and parallel computing systems. An emerging trend of Big Data computing is to combine MPI and MapReduce technologies in a single framework. The distinctive state model in this kind of frameworks brings challenges to designing an efficient and transparent fault-tolerance mechanism. In this paper, a state model analysis method is proposed for uniformly modeling independent MPI, MapReduce and MPI-based MapReduce data computing frameworks. Based on this analysis, a library-level fault-tolerance mechanism with global persistent state model is proposed; a data-staging and routine-sharing based checkpoint approach is designed within this mechanism. The proposed mechanism has been implemented in DataMPI, a communication library supporting MPI-based MapReduce data computing applications. The experiments show that it can transparently enable fault-tolerance for applications. Taking TeraSort as an example, it introduces only 6.8% time overhead and 11% space overhead. For a failure-resume execution, it has a 10%–32% performance advantage compared with the naive checkpoint solutions based on local or parallel storages. The proposed mechanism also provides superior performance and resource utilization compared with Hadoop for both fault-free and failure-resume executions.**

***Keywords*-MapReduce; MPI; data computing; fault-tolerance; checkpoint**

I. Introduction

Along with the development and application of Big Data technology, a type of research that tries to introduce mature technologies and mechanisms of high-performance computing to large-scale data computing systems is noteworthy [1], [2], [3]. The potential benefits of this kind of research include superior performance, efficient resource utilization, and reuse of existent HPC infrastructures. However, these two fields have many different hypotheses and features. High-performance computing systems usually run on reliable high-end hardware, while large-scale data computing systems often consider failures of commodity cluster and uneven quality of data as normal situations. These are important challenges for technology integration.

A. DataMPI and fault-tolerance

DataMPI [4], [5] is an open-source communication library, which is a bridging technology between HPC and Big Data. It aims at extending the Message Passing Interface (MPI) with key-value pair based primitives to provide high-performance communication in the large-scale data computing scenario. DataMPI supports building computing frameworks with multiple programming paradigms. Among them, the MPI-based MapReduce-like paradigm is a common and basic one for constructing other complicated paradigms. This paradigm divides a data computing process into a map-like operation (O) phase and a reduce-like aggregation (A) phase, while a shuffle-like transfer mechanism is proposed based on MPI communication. Experimental results and analysis show that DataMPI has a significant performance improvement compared with Apache Hadoop [6], and it is much easier to develop Big Data applications compared with using the original MPI primitives directly.

When designing a data computing framework integrating MPI and MapReduce technologies, fault-tolerance is recognized as one of the major challenges, because the traditional fault-tolerance solutions from existent ecosystems are inapplicable. For high-performance computing systems using MPI, checkpoint/restart is the most classic fault-tolerance mechanism. It does not apply to large-scale data computing because of performance and storage overhead, as well as implementation complexity: (1) Data computing systems are often memory intensive, so the periodical memory dump required by checkpoint will bring in huge time and space overhead. (2) Checkpoint/restart demands to handle the reconstruction of inter-process communication and synchronization, which significantly increases the programming complexity and code coupling degree of the distributed data computing framework. For large-scale data computing systems using MapReduce, backup task and process-level restart are the most usual inherent fault-tolerance mechanisms. It does not apply to applications with MPI communication because of the restrictions of MPI design and implementation: (1) According to the specification [7], the correctness of a communicator depends on the integrity of its corresponding process group. It is difficult to restart a process and reconstruct the communicator in a consistent manner. (2) The error handler mechanism is only used for resource cleanup and information output. Therefore, when one of the processes is failed, the remaining processes cannot perform the subsequent communication safely.

B. Objectives

Taking the above challenges into account, this paper aims at studying the fault-tolerance design in MPI-based MapReduce data computing frameworks. The detailed research objectives are as follows.

1) Feasibility: The functional objective of this research is to enable the fault-tolerance capability in the MPI-based MapReduce data computing environment. To define "feasibility" more accurate, two constraints are given: (1) Only the fail-stop failures are considered, and other types of failures (such as the Byzantine failure) are out of scope. (2) A reliable global storage system exists, so that the information exchange among fault-free processes works well. Thus, the targeted scenarios supported by the proposed fault-tolerance solution include failure of partial processes, failure of partial computing nodes, failure of the runtime management program, etc.

2) Efficiency: Efficiency is reflected on the time and space overhead of the fault-tolerance mechanism. In terms of time, it should not introduce noticeable overhead for the fault-free execution, since a failure is a small probability event for an individual execution. It should also take much less additional time for the failure-resume execution compared with the total time of re-execution from beginning, or else the fault-tolerance mechanism will be useless. In terms of space, the redundant storage overhead should be reduced as much as possible, in order to save the limited resource for the effective business logic of computing tasks.

3) Transparency: Transparency concerns the experience of application users, application developers, and computing framework developers. It determines the practicality of the fault-tolerance mechanism in large-scale data computing environments, and the efficiency of application development and management. It is necessary to provide transparency to the application developers and users, so that the existent applications will benefit from fault-tolerance without any modification. The modification or re-configuration to the computing framework is usually inevitable, so it is better to pursue a solution with minimal interference.

C. Contributions

The contributions of this research include:

(1) A state model analysis method is proposed, which can uniformly model the runtime state maintenance mechanisms of MPI and MapReduce. Furthermore, a global persistent state model for MPI-based MapReduce data computing frameworks is designed to solve the problem of mismatched state models between MPI and MapReduce. This model reveals the key factors determining the feasibility of fault-tolerance, which include the dataset coupling relationship and state synchronization relationship.

(2) A data-staging and routine-sharing based checkpoint approach is proposed. It leverages the hybrid storage stack of distributed computing environments to hide the checkpoint latency, and leverages the execution model characteristics of data computing frameworks to minimize the storage overhead. This approach can improve the runtime performance and spatial efficiency of checkpoint operations for both fault-free and failure-resume executions. It works in a transparent manner for both application developers and users.

(3) A library-level fault-tolerance mechanism with the proposed approach is designed and implemented in DataMPI, and a group of evaluations are presented on the proposed mechanism, alternative mechanisms, and Hadoop. The experiments show that DataMPI with this mechanism can adequately utilize the resource potential of the data computing framework. It introduces a small time and space overhead, and has a significant performance advantage compared with both naive checkpoint solutions and Hadoop.

II. Related work

The related work to this paper includes the fault-tolerance solutions in existent HPC and Big Data systems.

A. Fault-tolerance mechanisms in MPI

Traditional high-performance clusters emphasize high availability and reliability. In the MPI ecosystem, many types of fault-tolerance mechanisms have been proposed [8], such as checkpoint/restart, primitive semantic extension, algorithm based fault-tolerance, etc. System-level checkpoint/restart is the most widely used mechanism in production environments because of its universality and transparency [9]. The representative technology is BLCR [10]. It periodically makes the volatile data in memories of distributed processes persistent, and keeps the records of communication and synchronization relationships among them. After a failure, the new assigned computing nodes can load the checkpoints, resume the processes, reconstruct the communication, and negotiate a consistent state. Application-level checkpoint [11] is another kind of common mechanism, which requires the application developers to manage the checkpoint logic explicitly. However, it usually has a lower time and space overhead.

B. Fault-tolerance mechanisms in MapReduce

Failure is a normal situation in large-scale data computing systems because of inexpensive commodity hardware, rapidly evolving software, and uneven data. In the MapReduce ecosystem, computing frameworks represented by Hadoop provide a set of fault-tolerance designs: (1) For applications, the minimal dependence among processes and the task attempt mechanism make the process-level restart feasible. (2) For data, the probability of failures caused by unreliable hardware and uneven data can be reduced by data replication and bad record skipping mechanisms, as well as local persistence of intermediate data. (3) For the computing framework, although JobTracker and NameNode are single

points of failure, many studies and practices aim at improving their availability by hot standby [12]. Meanwhile, some researchers [1], [13] have discussed the differences in design and performance between the fault-tolerance mechanisms of MPI and MapReduce. These studies have the reference significance for our research. Different from these separate discussions, this paper focuses on the fault-tolerance design of MPI-based MapReduce data computing frameworks.

III. State modeling and analysis

To reveal the essential reason why traditional fault-tolerance solutions are inapplicable for MPI-based MapReduce data computing frameworks, we start with state modeling and analysis. The fault-tolerance problem is closely related to state model, because its responsibility is to convert a random abnormal state to a determinate controllable state.

A. Mismatched state models between MPI and MapReduce

To model the runtime state maintenance mechanisms of different computing frameworks, a state model analysis method is proposed at first. A set of formal symbols are given here: For an application instance (i.e., a job) a with a group of distributed processes (i.e., tasks) $\{e_i\}$, the dataset of e_i is denoted as d_i. For the stored-program computer architecture, d_i can be divided into two parts: code activity dataset (d_i^A) and application business dataset (d_i^B). d_i^A is the system-oriented dataset reflecting and maintaining the runtime activities of a program, such as the stack of function calls. d_i^B is the application-oriented dataset involving the specific business logic, such as the explicitly allocated space in heaps and the externally referred resources. The sets of e_i, d_i, d_i^A and d_i^B in a are denoted as E, D, D^A and D^B. The state of an entity x at time t is denoted as $state(x,t)$, while the activities on x during the interval $[t_1,t_2)$ are denoted as $activity(x,t_1,t_2)$. The state transition function of e_i is denoted as f_i. Besides, a symbol "$\hat{}$" is employed to indicate that the state of an entity has a persistent storage.

Although MPI is a universal communication library, there are still a few classic usage patterns that can be used as criteria for the analysis of state model. The representative one is the Single-Program Multiple-Data (SPMD) pattern implementing the Bulk Synchronous Parallel (BSP) model [14]. In this pattern, the correctness of world communicator depends on the integrity of all the processes, so the state transition based on collective communication requires a global synchronization, and depends on the states of all the processes. Denoting the recent global synchronization time before t as $rgs(t)$, the state transition relationship of the typical MPI usage pattern can be expressed as: $state(e_i,t) = f_i(state(E,rgs(t)),activity(e_i,rgs(t),t))$. It indicates that MPI has a process-centric state model.

The state model of MapReduce is more unified. The split mechanism distributes the input data to different map tasks, and the partition mechanism assigns the intermediate data to different reduce tasks. In this pattern, the state synchronization occurs in the shuffle phase. The state of a reduce task is constructed according to the intermediate data stored on local disks, which belong to the application business dataset. Meanwhile, a specific reduce task only depends on a part of the intermediate data determined by the partition function, which can be defined as a partitioned synchronization. Denoting the recent partitioned synchronization time before t as $rps(t)$, the state transition relationship of MapReduce can be expressed as: $state(e_i,t) = f_i(state(\hat{d_i^B},rps(t)),activity(e_i,rps(t),t))$. It indicates that MapReduce has a data-centric state model.

Comparing the two types of state maintenance mechanisms in MPI and MapReduce, it can be deduced that the differences of state models are the main obstacles to achieve fault-tolerance for the MPI-based MapReduce data computing scenario. More specifically: (1) The contradiction between the coupled relationship of application business dataset and code activity dataset in MPI, and the uncoupled relationship of those in MapReduce. It will significantly increase the complexity of designing a state maintenance mechanism when implementing a MapReduce-style fault-tolerance mechanism. It will also dramatically increase the dumped data size when implementing an MPI-style fault-tolerance mechanism. (2) The contradiction between the global synchronization relationship among processes in MPI, and the partitioned relationship of those in MapReduce. It makes the process-level restart impossible when implementing a MapReduce-style fault-tolerance mechanism. It does not comply with the requirements of MPI's global collective communication as well when implementing an MPI-style fault-tolerance mechanism.

B. Global persistent state model

Due to the differences between state models of MPI and MapReduce, a proper new state model is required for guiding the design of fault-tolerance in the hybrid scenario. We use DataMPI as an instance to explore the characteristics of MPI-based MapReduce data computing frameworks. DataMPI maintains its runtime state with the following characteristics.

(1) The patterned code activity dataset. Although DataMPI supports various applications, it generalizes the applications into a limited number of application modes according to their communication characteristics. Thus, it has a group of patterned code activity datasets bound to specific application modes, rather than specific applications.

(2) The disparity between scales of code activity dataset and application business dataset. The scale of code activity dataset is determined by the algorithm, whose upper bound is the address space of a process. The scale of application business dataset is determined by the workload, whose upper bound can be considered as infinite.

These characteristics give us the following enlightenments when designing a fault-tolerance oriented state model.

(1) The code activity dataset and application business dataset can be decoupled according to application mode. The key to resolve the complexity of application state maintenance is to simplify the maintenance of communication and synchronization relationships. Since each application mode has its patterned code activity dataset, it is possible to design a limited number of application-mode-specific state maintenance mechanisms for certain fault-tolerance patterns.

(2) The persistent storage of application business dataset can be co-designed with the inherent data spill mechanism. Because data spill to hierarchical storage devices is a common choice for large-scale data computing frameworks, it has potential that the requirement of persistent storage for fault-tolerance can leverage this existent mechanism effectively. Thus, the redundant storage overhead and duplicate data operations will be avoided.

(3) The persistent storage of application business dataset can be implemented as incremental storage. The computing framework can provide application-mode-specific strategies for persistent storage. Incremental storage is an efficient strategy compared with the complete memory dump in traditional system-level fault-tolerance solutions. It is universal for various application modes such as MapReduce, iterative processing, and stream processing.

Based on the above enlightenments, a global persistent state model for MPI-based MapReduce data computing frameworks is proposed. The state transition relationship of this model can be expressed as: $state(e_i, t) = f_i(state(\hat{D}^B, rgs(t)), activity(e_i, rgs(t), t))$. To address the contradiction between coupled and uncoupled relationships of code activity and application business datasets, this model chooses the uncoupled style in MapReduce. The key to its feasibility is the division of application modes, which transfers the fault-tolerance responsibility to underlying computing frameworks. To address the contradiction between global and partitioned relationships of inter-process synchronization, this model chooses the global style in MPI. This is a trade-off between feasibility and efficiency because of the constraints of MPI's collective communication.

IV. Resource utilization characteristics and potential for fault-tolerance

Fault-tolerance is generally considered as a source of performance overhead. We investigate the resource utilization characteristics of applications so as to design a fault-tolerance approach making full use of the resource potential.

A group of experiments is performed on DataMPI. Our testbed is an 8-node Linux cluster equipped with Intel® Xeon® CPU E5620, 16GB memory, 2TB disk, and 1Gbps Ethernet. The workload is TeraSort on HDFS, and the input data size is 48GB. This application has a minimal business logic outside computing frameworks. Without considering data replicas, its scales of input, intermediate and output data are equal, and the time of global synchronization is proportional to the data size. Hence, TeraSort is a representative application for the study of inherent resource utilization characteristics of computing frameworks.

Fig. 1 shows the resource utilization characteristics of TeraSort on DataMPI. The following phenomena can be observed: (1) The average CPU utilization rate is 50%. The peak CPU utilization rate is 78%, and the peak takes a very short time. So CPU is not a bottleneck resource. (2) Memory is a critical resource, but its footprint increases slowly, which takes about 2/3 time in the O phase to get fully utilized. (3) The disk write throughput is relatively low in the O phase. The disk read throughput is also showing potential in the O phase. (4) The network bandwidth for both sending and receiving is not fully utilized in the O phase and barely used in the later A phase because of data locality.

Our previous work [15] with other workloads over Hadoop and DataMPI show similar resource utilization characteristics. The causes of these common characteristics can be attributed to the common designs in data computing systems, such as: (1) Distributed file system in user space, like HDFS. The I/O performance of such file system will be a limiting factor in the hierarchical storage stack, which can reduce the efficiency of upper in-memory storage. (2) Imbalanced data exchange as a side-effect of data locality. The frequency and density of data access vary among processes and phases, so that the disk and network bandwidths present imbalanced utilization rates.

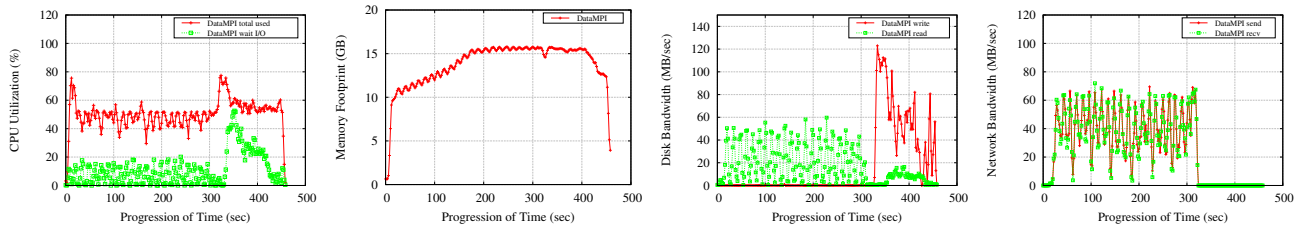
These resource utilization characteristics can derive the resource potential of data computing frameworks for designing effective fault-tolerance approach. For DataMPI, the resource potential can be summarized as: (1) Additional CPU-intensive operations are acceptable to some extent. (2) The free memory space in the O phase can be further leveraged. (3) The underutilized disk and network bandwidths are the main resources that can be adequately exploited to achieve efficiency. By understanding the underlying causes, the fault-tolerance mechanism can rebalance the resource utilization.

V. Data-staging and routine-sharing based checkpoint approach

As a performance-oriented instance of the global persistent state model, the proposed fault-tolerance approach is expected to leverage resources fully, so that the objective of efficiency can be achieved. The proposed approach should also follow the basic logic of the existent mechanisms in data computing frameworks, so that the objective of transparency can be achieved. Thus, a data-staging and routine-sharing based checkpoint approach for DataMPI is designed.

A. Data staging

Memory hierarchy is a widely used design idea in computing systems. A uniform storage system combined with



(a) CPU utilization (b) Memory footprint (c) Disk throughput (d) Network throughput

Figure 1. The resource utilization characteristics of TeraSort on DataMPI without fault-tolerance

storage devices of different speeds, latencies and capacities will leverage the locality of data access to optimize performance and cost. Data staging is a kind of storage model using memory hierarchy. It is employed to improve the performance of data access for fault-tolerance in the MPI-based MapReduce data computing framework. The key reason why data staging can benefit fault-tolerance is that it hides the latency caused by checkpoint operations. Meanwhile, it meets the requirement of global persistent storage for the application business dataset.

A storage architecture with data staging for the fault-tolerance of DataMPI is designed as shown in Fig. 2. A group of local hybrid storage systems are deployed on all the computing nodes. Each of them is combined by file systems on RAM disk and local disk. Data are written into RAM disk preferentially. When its usage reaches to a threshold, the subsequent data will be written into local disk. A global storage system is constructed by combining the local hybrid storage systems with a parallel file system (Lustre). A data synchronization service is designed based on lsyncd [16]. It synchronizes the data on local hybrid storage system to parallel file system periodically, so that the checkpoint data are made globally available. In addition, a lightweight distributed coordination service (ZooKeeper) is introduced for the consistent storage of meta-information.

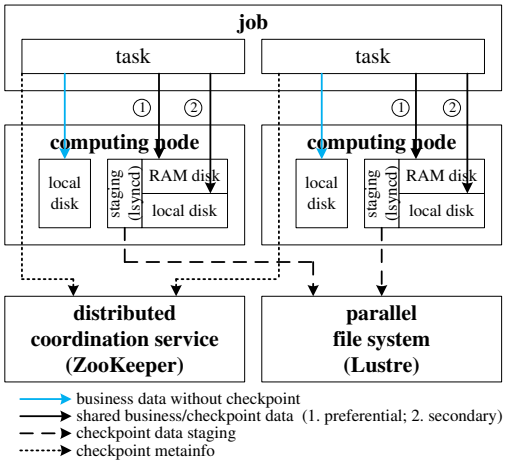


Figure 2. Storage architecture for the fault-tolerance of DataMPI

The local hybrid storage system aims at making full use of the free memory space in the O phase. Intuitively, it may compete for the critical memory resource with the computing tasks. However, configured with appropriate capacity, it will significantly relieve the I/O overhead caused by page swap, instead of degrading the overall performance. The periodical batch synchronization in global storage system is intended to aggregate the small I/O requests and reduce the interference with application business. Meanwhile, the periodical synchronization mechanism is compatible with the design of incremental checkpoint, which can avoid the redundant data transfer.

B. *Routine sharing*

In DataMPI, the co-design of fault-tolerance mechanism with other existent mechanisms is summarized as a routine-sharing design. There are two types of co-design here: sharing of data structures, and sharing of data flows.

A uniform meta-information data structure (`SpillInfo`) and a co-designed spill-and-checkpoint data structure (`CommonPartitionList`) are proposed. For the purpose of fault-tolerance, a set of fields are appended in `SpillInfo` for logging and calculating the global synchronization time. A set of polymorphic interfaces are provided by `CommonPartitionList` and its buffer sub-structure, so that data in memory, data spilled to disk, and data loaded from checkpoint file can be processed in a uniform manner.

The data flows of checkpoint with routine-sharing design is shown in Fig. 3. For a specific job, the involved execution entities include the runtime manage program (mpidrun), all the processes of the job, and the data synchronization service. The data flows corresponding to fault-tolerance are marked with dashed boxes. After a job or a task starting, its meta-information is logged. Then the three threads of each process, including the main thread, communication thread and merge thread, execute their duty cycles continuously. The merge thread logs the number of processed key-value pairs and the round of global synchronization after data spilling to the local hybrid storage system, and the data synchronization service logs the completion flag after data synchronizing to the parallel file system.

When a failure occurs, all the processes of the job will

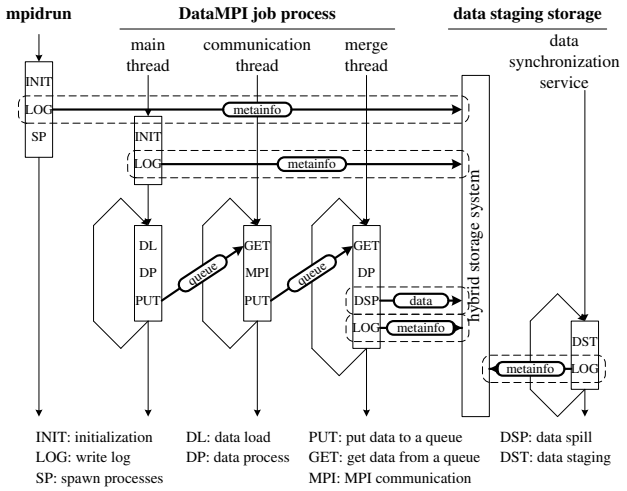


Figure 3. Checkpoint workflow in DataMPI

be terminated by MPI's runtime manage mechanism. If mpidrun is alive at this time, it will invoke the resume routine. Otherwise, the user or the external job manage program can re-execute mpidrun with a "`-restart`" parameter to resume the job. Mpidrun will try to schedule the restarted processes to the computing nodes which have kept the checkpoint in their local hybrid storage systems if possible. All the processes will load the logged meta-information, and an all-reduce communication is performed to calculate the recent global synchronization time. Then, all the processes will be synchronized to this state and continue to execute. Because of the co-designed data structures, the checkpoint data can be used as intermediate data directly.

The data-staging and routine-sharing based checkpoint approach has been implemented as a library-level mechanism in DataMPI. "Library-level" means the fault-tolerance technologies are constructed inside the computing framework. It is a trade-off between the classic system-level mechanism and application-level mechanism in order to balance efficiency and transparency.

VI. Evaluation

The performance and resource utilization of the proposed fault-tolerance solution in DataMPI have been evaluated to illustrate the effects of design. The experimental environment and workload are same as those of Section IV.

A. Performance

A group of fault-free and failure-resume executions of TeraSort on DataMPI are performed and analyzed. The comparative objects include a group of naive checkpoint solutions that use local disk or Lustre as checkpoint storage.

1) Fault-free execution: As shown in Fig. 4, six tests with different data spill types are compared. Tests a) and b) represent the performance of using the hybrid storage system. Comparing their execution time, it shows that the additional time overhead of checkpoint is 6.8%. Tests c) to f) represent the performance of using local disk or Lustre. Comparing the execution time of b), d) and f), it shows that the proposed approach [b)] reduces the execution time by 28% compared with that of local disk [d)], and 27% compared with that of Lustre [f)]. Tests c) and e) show that the local disk or Lustre based approaches are slower than the proposed approach even if no checkpoint. Additionally, test b) shows the data staging after job completion takes 4.9% extra time, which is not considered as a part of the overhead, because the synchronization at that time can be discontinued in real applications.

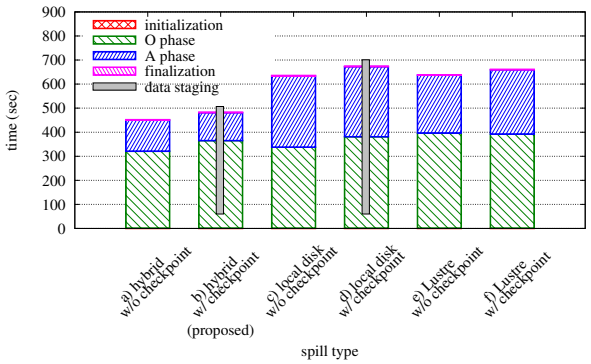


Figure 4. Execution time of different spill types in DataMPI

In these tests, the input data take up 48.4GB in HDFS. The spilled data size with a default spilling strategy is 44.2GB, while the checkpoint size with fault-tolerance is 49.2GB. The additional space overhead of fault-tolerance is 11%, and the checkpoint size is only 1% larger than the input data size. The discrepancy between them derives from the space efficiency of different data structures. Meanwhile, the size of meta-information for fault-tolerance is only 624KB.

2) Failure-resume execution: As shown in Fig. 5, four tests with different data spill types and checkpoint locations are compared. Intended failures are introduced when a certain number of records have been processed. Test g) represents the performance of failure resuming for partial processes. Comparing with test b) in Fig. 4, it shows that the additional time overhead of local resume is 3.7%. Test h) represents the performance of failure resuming for partial computing nodes. Comparing with test b) in Fig. 4, it shows that the additional time overhead of remote resume is 25%. Tests i) and j) represent the performance of failure resuming when using local disk or Lustre. Comparing the execution time of g) with i) and h) with j), it shows that the data-staging based solution will reduce the execution time of local resume and remote resume by 10%–32%.

B. Resource utilization

Fig. 6 shows the resource utilization characteristics of TeraSort on DataMPI with fault-tolerance. Comparing with

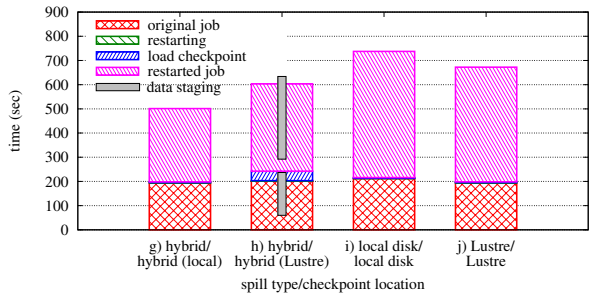


Figure 5. Execution and resume time of different spill types and checkpoint locations in DataMPI

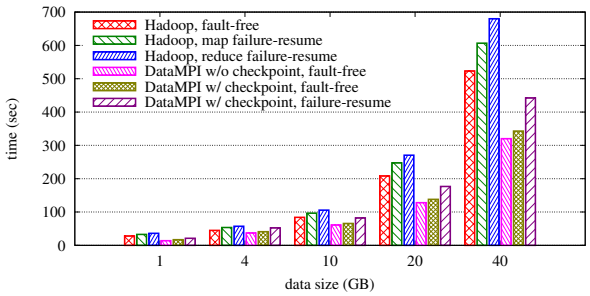


Figure 7. Performance comparison between DataMPI and Hadoop

the original characteristics described in Section IV, the following effects can be observed: (1) The average CPU utilization rate is slightly increased to 52%, and the peak value is 82%. The small overhead mainly comes from the maintenance of meta-information for checkpoints. (2) The memory footprint increases faster in the O phase. The time to reach full utilization has been reduced from around 200s to 100s. This is because the checkpoint can take full advantage of RAM disk. (3) The disk bandwidth in the O phase has been utilized because of the data-staging based checkpoint storage. The cycle of its fluctuation is same as the cycle of data synchronization to Lustre. (4) The peak network throughput in the O phase has grown from 72MB/s to 88MB/s, and the network resource in the A phase has been utilized. This is attributed to the data access of Lustre.

C. Comparison with Hadoop

We also compare our proposed fault-tolerance solution on DataMPI with Hadoop to show the advantages in performance and resource utilization. The Hadoop version used here is 1.2.1, and the cluster environment is the same.

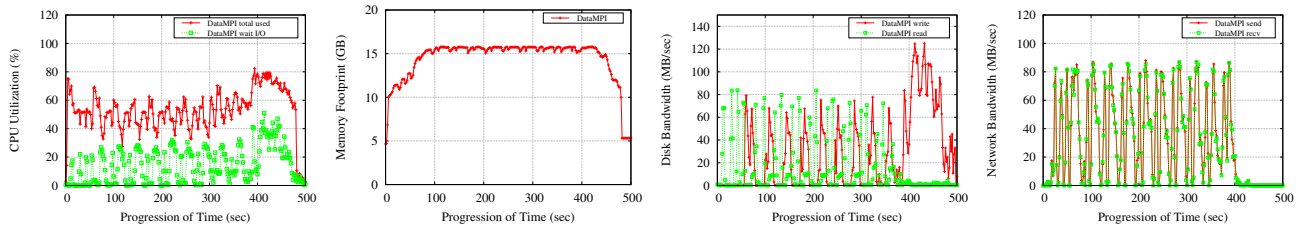
Fig. 7 shows the performance comparison. The workload is TeraSort with different data sizes. Both fault-free and failure-resume executions are compared. As we can see, DataMPI has a shorter execution time than Hadoop in different cases. Taking the 40GB case as an example, DataMPI's fault-free execution time is 66% of Hadoop's fault-free execution, and DataMPI's failure-resume execution time is 73% of Hadoop failed at map phase. Even if a failure occurs and a resume procedure invoked, DataMPI can still save 15% time compared to Hadoop without any failure. The experiment also shows that DataMPI with the proposed fault-tolerance approach keeps a good scalability.

Fig. 8 shows the resource utilization characteristics of 48GB TeraSort on Hadoop without failure. Comparing with Fig. 6: (1) The average CPU utilization rate is around 44%, which is slightly lower than that of DataMPI. The I/O wait also keeps a longer span. (2) The memory footprint has a similar trend to DataMPI, but it rises relatively slowly in the very beginning of map phase. (3) The disk and network throughputs show more irregular fluctuations, and the overall values are lower than those of DataMPI.

The main reason why DataMPI with fault-tolerance has a more superior performance and resource utilization is that the proposed approach produces a balanced overlap among computation, disk access, and network communication. The disk and network latencies caused by dataset persistence can be hidden by data staging, while the disk and network can be leveraged efficiently with a minor performance impact. The fault-tolerance mechanism also benefits from the capacities of high-performance communication, pipeline-based design, and efficient data spilling in DataMPI. In contrast, the fault-tolerance mechanism of Hadoop is based on the redundancy of input files and task-level recovery with local disk caching. When tasks are failed, they will be restarted from the beginning with split files, and retransfer the intermediate data. The operations of restauration have less consideration about the overlapping between computation and communication.

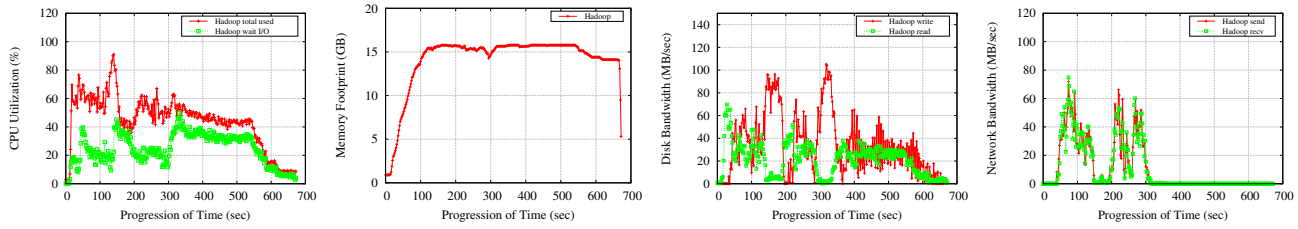
VII. Conclusion

It is a notable trend to combine high-performance computing and Big Data technologies in a single framework. Fault-tolerance is a challenge in such computing frameworks because of mismatched state models of the core mechanisms represented by MPI and MapReduce. In this paper, a global persistent state model is proposed to solve the contradiction of dataset coupling relationships and the contradiction of state synchronization relationships. A data-staging and routine-sharing based checkpoint approach is designed. It is implemented in DataMPI, a communication library supporting MPI-based MapReduce data computing frameworks. The proposed mechanism achieves the objectives of feasibility, efficiency, and transparency. For a typical 48GB TeraSort application, it introduces only 6.8% time overhead and 11% space overhead. For a failure-resume execution, the additional time overhead of local resume and remote resume are only 3.7% and 25%, which provides a 10%–32% performance advantage compared with the naive checkpoint solutions based on local or parallel storages. The proposed mechanism also provides superior performance and resource utilization compared with Hadoop even if a



(a) CPU utilization (b) Memory footprint (c) Disk throughput (d) Network throughput

Figure 6. The resource utilization characteristics of TeraSort on DataMPI with fault-tolerance



(a) CPU utilization (b) Memory footprint (c) Disk throughput (d) Network throughput

Figure 8. The resource utilization characteristics of TeraSort on Hadoop

failure occurs, while keeping good scalability.

The future work includes the customization and optimization of fault-tolerance state models for specific data computing paradigms. Meanwhile, it is necessary to study the process-level resume in the MPI-based MapReduce data computing framework to further reduce the overhead.

ACKNOWLEDGEMENT

We are grateful to Lang Wu for his help of this work. This research is supported in part by the Hi-Tech Research and Development (863) Program of China under Grant Nos. 2013AA01A209, 2013AA01A213, and the Guangdong Talents Program of China under Grant No. 201001D0104726115.

REFERENCES

[1] T. Hoefler, A. Lumsdaine, and J. Dongarra, "Towards efficient MapReduce using MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Berlin: Springer-Verlag, 2009, pp. 240–249.

[2] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for large-scale graph algorithms," *Parallel Computing*, vol. 37, no. 9, pp. 610–632, 2011.

[3] X. Lu, B. Wang, L. Zha, and Z. Xu, "Can MPI benefit hadoop and MapReduce applications?" in *Proceedings of the 40th International Conference on Parallel Processing Workshops*, ser. ICPPW '11, 2011, pp. 371–379.

[4] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "DataMPI: Extending MPI to hadoop-like big data computing," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '14, 2014.

[5] "DataMPI." [Online]. Available: http://www.datampi.org

[6] "Hadoop." [Online]. Available: http://hadoop.apache.org

[7] "MPI: A message-passing interface standard v3.0." [Online]. Available: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[8] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[9] N. Debardeleben, J. Laros, J. Daly, and S. Scott, "High-end computing resilience: analysis of issues facing the HEC community and path-forward for research and development," Los Alamos National Laboratory, Los Alamos, NM, Tech. Rep., 2009.

[10] J. Duell, "The design and implementation of berkeley lab's linux checkpoint/restart," Lawrence Berkeley National Laboratory, Berkeley, CA, Tech. Rep., 2002.

[11] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing frontiers and innovations*, vol. 1, no. 1, pp. 5–28, 2014.

[12] M. Cogorno, J. Rey, and S. Nesmachnow, "Fault tolerance in hadoop MapReduce implementation," Faculty of Engineering, University of the Republic, Montevideo, Uruguay, Tech. Rep., 2013.

[13] H. Jin and X.-H. Sun, "Performance comparison under failures of MPI and MapReduce: An analytical approach," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1808–1815, 2013.

[14] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[15] F. Liang, C. Feng, X. Lu, and Z. Xu, "Performance characterization of hadoop and DataMPI based on amdahls second law," in *Proceedings of the 9th IEEE International Conference on Networking, Architecture and Storage*, ser. NAS '14, 2014.

[16] "Lsyncd (live syncing daemon)." [Online]. Available: https://code.google.com/p/lsyncd/