Group-based Coordinated Checkpointing for MPI: A Case Study on InfiniBand *

Qi Gao Wei Huang Matthew J. Koop Dhabaleswar K. Panda

Network-Based Computing Laboratory
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH, U.S.A.
{*gaoq, huanwei, koop, panda*}*@cse.ohio-state.edu*

Abstract

As more and more clusters with thousands of nodes are being deployed for high performance computing (HPC), fault tolerance in cluster environments has become a critical requirement. Checkpointing and rollback recovery is a common approach to achieve fault tolerance. Although widely adopted in practice, coordinated checkpointing has a known limitation on scalability. Severe contention for bandwidth to storage system can occur as a large number of processes take a checkpoint at the same time, resulting in an extremely long checkpointing delay for large parallel applications. In this paper, we propose a novel group-based checkpointing design to alleviate this scalability limitation. By carefully scheduling the MPI processes to take checkpoints in smaller groups, our design reduces the number of processes simultaneously taking checkpoints, while allowing those processes not taking checkpoints to proceed with computation. We implement our design and carry out a detailed evaluation with micro-benchmarks, HPL, and the parallel version of a data mining toolkit, MotifMiner. Experimental results show our group-based checkpointing design can reduce the effective delay for checkpointing significantly, up to 78% for HPL and up to 70% for MotifMiner.

1. Introduction

With the rapid development in High Performance Computing (HPC) systems, an increasing number of clusters with thousands of nodes are being deployed. The failure rates of those systems, however, also increase along with their size and complexity. As a result, the running times of many scientific applications are becoming longer than the mean-time-between-failure (MTBF) of the cluster systems. Therefore, fault tolerance in cluster environments has become a critical requirement to guarantee the completion of application execution. A commonly used approach to achieve fault tolerance is checkpointing and rollback recovery, where the intermediate states of running parallel applications can be saved and used later for restart upon failures.

There are two main categories of checkpointing protocols for parallel programs: coordinated checkpointing and uncoordinated checkpointing. They use different approaches to guarantee global consistency. Uncoordinated checkpointing allows processes to save their states independently with less concurrency in accessing the storage, but requires message logging to guarantee consistency and avoid cascade rollback. Since there is a clear trend for large clusters to be equipped with high performance networks, such as InfiniBand [17], the overhead of message logging could be prohibitive considering the very high bandwidth and messaging rate of the networks. On the other hand, instead of using message logging, coordinated checkpointing guarantees the consistency by global coordination, but typically requires all processes to save their states at relatively the same time, which unfortunately, often causes the large parallel applications suffering from extremely long checkpointing delays. Because in a real-world scenario, checkpoint files are often required to be stored on a reliable central storage system. Given the limited bandwidth the central storage system provides, the more processes accessing it concurrently, the less bandwidth each process obtains. We call this effect *Storage Bottleneck*, which significantly limits the scalability of coordinated checkpointing for large parallel applications.

In this paper, we address the scalability of checkpointing for parallel applications by a novel group-based checkpointing scheme. Our design is based on coordinated checkpointing, which avoids message logging and only incurs a

*This research is supported in part by DOE grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; NSF grants #CNS-0403342, #CNS-0509452, and #CCF-0702675; grants from Intel, Mellanox, Cisco systems, Linux Networx and Sun Microsystems; and equipment donations from Intel, Mellanox, AMD, Apple, Appro, Dell, Microway, PathScale, IBM, SilverStorm, and Sun Microsystems.

very low overhead in the failure-free case. At the same time, our design takes a good feature of less concurrency in storage accessing from uncoordinated checkpointing to avoid the storage bottleneck. We divide the processes into multiple smaller groups, and take the checkpoint group by group through careful scheduling so that all processes do not access the central storage at the same time. Thus, each process in the checkpointing group obtains a larger share of the bandwidth to storage and observes a much shorter checkpoint delay. While a group of processes are taking a checkpoint, our design allows processes in other groups to proceed with their execution as much as possible, and only defers the communications that can cause inconsistency. In this way, our design can significantly reduce the checkpoint delay for large scale applications.

In order to evaluate the benefits of group-based checkpointing, we have implemented our design in MVAPICH2 [19, 3], a high performance MPI implementation over InfiniBand based on MPICH2 [2]. We also propose multiple performance metrics, which accurately characterize the time overhead of checkpointing parallel applications. We evaluate our design using both micro-benchmarks and applications. Experimental results show that our new group-based checkpointing design reduces the effective delay for checkpointing significantly, up to 78% for HPL [20] and up to 70% for MotifMiner [24]. Although our current implementation is based on InfiniBand and MVAPICH2, the design can be readily applicable to other coordination checkpointing protocols for other MPI implementations and networks.

The rest of the paper is organized as follows: In Section 2 we briefly discuss the background. In Sections 3 and 4, we present our group-based checkpointing design and discuss some key design issues. In Sections 5 and 6, we describe the metrics of checkpoint delay and the evaluation results. In Section 7, we discuss related work. Finally, we conclude and briefly mention future directions in Section 8.

2. Background

2.1. Checkpointing Technologies

Checkpointing and rollback recovery is a commonly used approach for failure recovery. By periodically saving the running state of applications, the checkpointing system allows applications to restart from a time point in the middle of their executions upon failure, instead of from the beginning. A survey of rollback recovery can be found in [12].

As discussed in Section 1, uncoordinated checkpointing has a larger overhead in failure-free cases because logging all the messages in HPC applications would create a huge volume of message logs, and storing them even temporarily in an efficient manner could be a problem. On the other hand, coordinated checkpointing has potentially limited scalability largely due to the storage bottleneck. It is possible to make use of local disk or spare memory on peer nodes as temporary buffers to hold checkpoint files. However, there are three facts which make these approaches less practical: a) Scientific applications tend to use as much memory as possible for computation, which makes the node memory to be a scarce resource and not able to be used to store checkpoints; b) The local disk may not be available on the computing nodes in new large clusters, e.g. the compute nodes in the Peloton clusters at LLNL, are diskless [18]; and c) Buffering the checkpoint files on local disk may undermine the ability to recover from failure, because if the node crashes before the checkpoint files are completely transferred to reliable storage, the checkpoint files may be lost.

Note that in this paper we refer the blocking design of coordinated checkpointing protocol as 'coordinated checkpointing', which does not require any message logging. Although the non-blocking design of Chandy-Lamport coordinated checkpointing protocol [9] allows processes to make progress when other processes are taking checkpoints, it does not explicitly schedule them globally. Therefore, in a parallel application, all the processes are still very likely to take checkpoints at the same time and hit the storage bottleneck. In addition, the non-blocking protocol also requires logging some messages in the channels during checkpointing. The high message logging overhead for this protocol on high speed interconnects is reported in [10].

2.2. Checkpointing on OS-bypass Networks

Recently, high speed networks have been deployed on clusters to achieve optimal performance. These networks usually use intelligent network interface cards (NIC) and provide user-level protocols to allow OS-bypass communications for low latency and high bandwidth. However, as compared to Ethernet, they introduce additional challenges for system-level checkpointing. Essentially, since communication contexts are cached on NIC and also memory mapped to process address space for user-level access, network connections are usually required to be explicitly torn down before the checkpoint and reconnected afterwards. These challenges are discussed in detail in our previous work [14].

InfiniBand [17] is a typical and widely used high performance interconnect, which uses a connection-oriented model for best performance and advanced features, such as Remote Direct Memory Access (RDMA). Most MPI implementations on InfiniBand use the connection-oriented model as default. To establish connections on InfiniBand, two peers have to exchange connection parameters using an out-of-band channel. As a result, the cost for connection management is much higher as compared to using the TCP/IP protocol. And, even for a local checkpoint on one node to take place, all the remote processes it connects to need to participate in the costly procedure for connection tear-down and rebuild. This makes the checkpointing pro-

tocols like uncoordinated checkpointing and non-blocking coordinated checkpointing even more expensive.

In our previous work [14], we have designed and implemented a checkpoint/restart (C/R) framework in MVAPICH2 [19, 3] using a coordinated checkpointing protocol to take global checkpoints of all MPI processes at the same time. The framework consists of a global C/R coordinator, and a local C/R controller in each MPI process. Upon a checkpoint request, the global C/R coordinator orchestrates all MPI processes through a checkpointing cycle including four phases: Initial Synchronization, Pre-checkpoint Coordination, Local Checkpointing, and Post-checkpoint Coordination. In the Local Checkpointing phase, the Berkeley Lab Checkpoint/Restart (BLCR) [11] toolkit is used to take the snapshot of a single MPI process. The consistency of InfiniBand communication channels is guaranteed by a regular coordination protocol with all processes. Executed in both Pre-checkpoint Coordination phase and Post-checkpoint Coordination phase, the protocol stops the execution of all the processes, flushes all the in-transit network messages, globally tears-down all connections before the checkpoint, and rebuilds them afterwards. Although ensuring the correctness, this regular coordinated checkpointing protocol leads to suboptimal performance by hitting the storage bottleneck when the job is large. In this paper, we redesign the system using group-based checkpointing to improve performance by avoiding the storage bottleneck.

3. Group-based Coordinated Checkpointing

3.1. Motivation

In coordinated checkpointing, the overall checkpointing delay consists of two parts, coordination time and storage access time. Coordination time is the time spent passing control messages between processes for synchronizing checkpointing actions, while the storage access time is the time for saving the program running states to the stable storage. With the order-of-magnitude improvement in network speed but limited improvement in disk speed in recent years, the storage accessing time is the dominating part for checkpointing [12]. And in our previous work [14], we found the storage accessing time to be over 95% of the overall checkpointing delay.

Storage access time can be even longer when a central storage is shared by a large number of processes to store the checkpoint. Figure 1 demonstrates the bandwidth to storage with various number of clients writing checkpoint files concurrently. It shows that using four PVFS2 [4] servers for storage with a total of about 140MB/s aggregated throughput, the bandwidth each client obtains decreases significantly as the number of clients increases. The case where 140MB/s storage throughput shared by 32 clients (4.38 MB/client) is also an optimistic ratio considering the storage bandwidth in current cluster deployments. For instance, the Thunderbird cluster at Sandia National Lab has 4,480 nodes with 8,960 CPUs and its storage throughput is 6.0 GB/s [18]. (1.37 MB/node)

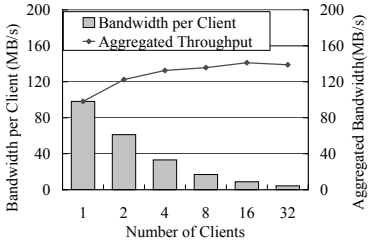

Figure 1. Bandwidth per Client to Storage with Different Number of Clients

Especially for large-scale parallel applications, the central storage system is likely to be a bottleneck of checkpointing. It is because that the throughput of a storage system is limited, and thus the more processes accessing the storage concurrently, the less bandwidth each process obtains. With the number of processes (N), checkpoint image size per process (S), and the total throughput to storage system (B), the ideal storage accessing time for checkpointing (T) can be estimated by $T = \frac{N \times S}{B}$, assuming no congestion effects and a fair bandwidth sharing among the processes to the storage server. Taking the Thunderbird cluster as an example, suppose the memory footprint for each process is 1 GB, and all storage bandwidth is effectively used for checkpointing, it still needs 1493 seconds (about 25 minutes) to save the checkpoint to the storage server. This estimation is still optimistic. In practice, with a large number of processes taking checkpoints concurrently, the system noise, network congestion, and unbalanced share of throughput to the storage server can significantly increase the delay. Therefore, it is desirable to limit the number of processes taking checkpoints at the same time.

Fortunately, previous studies [22] indicate that in many parallel applications each process only communicates to a limited number of peer processes. Therefore the MPI processes can potentially make progress while other MPI processes in the MPI job take their checkpoints.

3.2. Proposed Design

The main goal of group-based coordinated checkpointing is that when checkpointing a large MPI job with many processes, MPI processes should be carefully scheduled to take checkpoints at slightly different times to avoid the storage bottleneck. While the design allows the processes which are not currently taking the checkpoints to proceed with their execution, the global consistency is maintained by a coordination protocol without message logging to avoid the overhead associated with logging.

As illustrated in Figure 2, when taking a consistent global checkpoint in our group-based checkpointing design, instead of all processes taking checkpoints at the same time

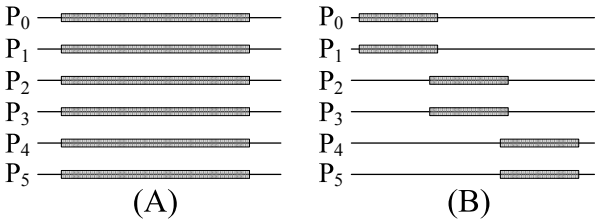

Figure 2. (a) Regular Coordinated Checkpointing and (b) Group-based Checkpointing

and suffering a large checkpointing delay, processes are divided into smaller groups and scheduled to take their own checkpoints group by group through careful coordination. Therefore, each process in the checkpointing group can obtain a much larger bandwidth to the storage system and thus, reduce their checkpointing delay greatly.

Group-based checkpointing has three steps. First, upon a checkpoint request, we divide all processes into a set of groups, and decide the order in which they take a checkpoint. Second, the groups of processes take checkpoints in turn. In this step, two levels of coordination protocols are used. The intra-group coordination is only performed by the processes in the group that is currently taking a checkpoint. They are synchronized with each other to ensure the consistency in the group. The inter-group coordination is performed by all processes, but processes in other groups only coordinate in a passive manner as needed without stopping computation. To guarantee the consistency, the groups which have already taken the checkpoint should not communicate with the groups which have not yet taken the checkpoint. The messages to be passed between two such groups are deferred until both of then have taken their checkpoint. Finally, after all processes have taken their checkpoint, the global checkpoint is marked complete.

Another alternative to handle messages between two groups where only one of them has taken checkpoint is to log the messages and resolve the potential inconsistency at restart time. However, there are performance trade-offs involved for that approach especially on high speed networks. Besides the logging overhead itself, to enable message logging, zero-copy rendezvous protocols, which are commonly used in high speed networks to improve performance, can not be used, which will affect the performance in failure-free cases negatively. Therefore, in our current design, we choose to defer the message passing between two such groups using message or request buffering to resolve inconsistency. The differences between buffering and message logging are discussed in detail in Section 4.3.

4. Design Issues

In this section, we highlight several key design issues of group-based checkpointing, such as group formation, connection management, message and request buffering, and asynchronous progress.

4.1. Group Formation

To achieve maximum benefit of group-based checkpointing, checkpoint groups should be formed in a way that the most frequent communication happens within groups. In addition, the size of the group also affects the checkpointing delay since it is related not only to the communication pattern but also the efficiency for accessing the storage.

The group formation can be done either statically or dynamically. In static group formation, the checkpoint groups are formed based on a user-defined group size and the global rank of each process. In dynamic group formation, a protocol is executed at runtime. First, information related to communication patterns such as connection status, user-defined communicators, etc., is collected to provide heuristic. Then, a lightweight algorithm is executed to find the transitive closure of frequently communicated processes. If the algorithm finds that the application mainly does global communication, it will fall back to the static group formation to limit the analysis cost.

4.2. Connection Management

As mentioned in Section 2, connection management for InfiniBand is more complex and more costly than TCP/IP networks. In the regular coordinated checkpointing where all processes participate in checkpointing at the same time [14], all connections can be torn-down and rebuilt collectively using a global protocol. However, the global tear-down/rebuild model does not work in group-based checkpointing. The main difficulty here is to allow other processes to continue computation when a group of processes are taking a checkpoint. Thus, the connections must be controlled dynamically, so that: a) only a small subset of connections are disconnected when checkpointing a group; and b) a process in the checkpointing group can disconnect/reconnect to a peer process which is not in the current checkpointing group (and thus will not actively participate in connection management).

We have designed a connection manager for group-based checkpointing. Our design maintains the connection status for checkpointing on a per connection basis, so that it allows each MPI process to disconnect/reconnect to only a specific set of peer processes. In addition, it uses a client/server (active/passive) protocol to provide the flexibility for any side to initiate the disconnect/reconnect operation.

4.3. Message and Request Buffering

To defer passing messages for consistency, two buffering techniques, namely message buffering and request buffering, are used under different circumstances. Message buffering is to temporarily hold the messages that are to be

sent in message queues until communication to the destination is allowed. It is used only for small messages that have already been copied into communication buffers but have not yet been posted to the network. Request buffering is to avoid the expensive content storage in message buffering case. In most MPI implementations, there are internal data structures to manage the communication requests from user applications to the bottom layer (network device layer). Request buffering is to keep the communication requests in an 'incomplete' state and buffers them in queues. It is used in all possible cases, i.e. for all large messages and the small messages which have not yet been copied to a communication buffer.

There are two main differences between message buffering and message logging. First, instead of sending the messages and recording their content as in the message logging case, in the message buffering case the message is only buffered temporarily but not sent. It allows a consistent recovery line to be formed without relying on any message log and thus the message buffer can be freed as soon as the messages are sent. Second, in message logging, the content of messages must always be fully logged, while in the message buffering case, the request buffering can be used to reduce the cost of storing large messages. Both of these differences lead to less storage overhead for message buffering.

4.4. Asynchronous Progress

In regular coordinated checkpointing where all processes take a checkpoint at the same time, the coordination protocol is automatically guaranteed to make timely progress. In group-based checkpointing, however, the coordination may require the participation of the processes in other groups which are potentially busy with computation. Therefore, asynchronous progress is critical for inter-group coordination to avoid a large coordination delay.

Therefore, we introduce another state, called *passive coordination*, to the state machine. A process enters the passive coordination state when another group is taking a checkpoint. In this state, the process will temporarily activate a helper thread and a lightweight time recorder in progress engine. Whenever it has been a long time since the last progress, e.g. 100 milliseconds, the helper thread will request a progress check. That way, even if the MPI application is busy in computation for a long time, the inter-group coordination will progress within a bounded time. Since this helper thread is only activated in the passive coordination state and does not wake up very often, it has negligible impact on the overall performance.

5. Performance Metrics

In this section, we introduce three metrics to characterize the time overhead of checkpointing parallel applications and explain their correlations. These metrics are:

Effective Checkpoint Delay: The increase in the application running time caused by taking one checkpoint during application execution. Since it reflects the effective performance impact of taking a checkpoint of the application, our end goal is to minimize the Effective Checkpoint Delay.

Individual Checkpoint Time: The delay observed by each individual MPI process when taking a checkpoint, which reflects the down time of each individual process in the middle of execution.

Total Checkpoint Time: The total time from the point when a checkpoint request is issued to the point when all processes have finished taking their checkpoints.

Since the checkpoint image size per process is same as the memory footprint for each process, and Individual Checkpoint Time is dominated by the storage accessing time (more than 95% in our experiments), we have:

$$\begin{aligned} & Individual\ Checkpoint\ Time \\ \approx\ & \frac{Process\ Memory\ Footprint \times No.\ of\ Processes}{Aggregated\ Throughput\ to\ Storage} \end{aligned} \tag{1}$$

For regular coordinated checkpointing where all the processes take their checkpoints at the same time, we have:

$$\begin{aligned} & Individual\ Checkpoint\ Time \\ \approx\ & \frac{Process\ Memory\ Footprint \times Total\ No.\ of\ Processes}{Aggregated\ Throughput\ to\ Storage} \end{aligned} \tag{2a}$$

$$\begin{aligned} & Individual\ Checkpoint\ Time \\ =\ & Effective\ Checkpoint\ Delay \\ =\ & Total\ Checkpoint\ Time \end{aligned} \tag{2b}$$

However for group-based checkpointing, where processes take checkpoints group by group, we have:

$$\begin{aligned} & Individual\ Checkpoint\ Time \\ \approx\ & \frac{Process\ Memory\ Footprint \times No.\ of\ Processes\ in\ Group}{Aggregated\ Throughput\ to\ Storage} \end{aligned} \tag{3a}$$

$$\begin{aligned} & Total\ Checkpoint\ Time \\ =\ & No.\ of\ Groups \times Individual\ Checkpoint\ Time \end{aligned} \tag{3b}$$

$$\begin{aligned} & Individual\ Checkpoint\ Time \\ \leq\ & Effective\ Checkpoint\ Delay \\ \leq\ & Total\ Checkpoint\ Time \end{aligned} \tag{3c}$$

Ideally, the Effective Checkpoint Delay should be the Individual Checkpoint Time. Due to the synchronization and dependencies, however, MPI processes can not always make full progress when another process group is taking a checkpoint. Therefore in practice, the Effective Checkpoint Delay lies between the Individual Checkpoint Time and Total Checkpoint Time. Note that in group-based checkpointing, the Total Checkpoint Time only reflects the *worst case* impact to applications.

There are several parameters affecting the Effective Checkpoint Delay. In the next section, we focus on two most important parameters: checkpoint group size and issuance time of the checkpoint request.

6. Performance Evaluation

In this section, we analyze the benefits of group-based checkpointing in terms of reduction in Effective Checkpoint Delay with both micro-benchmarks and applications.

The experiments were conducted on an InfiniBand cluster of 36 nodes, with 32 compute nodes and 4 storage nodes. Each compute node is equipped with dual Intel 64-bit Xeon 3.6GHz CPUs, 2GB memory, and a Mellanox MT25208 InfiniBand HCA. Each storage node is equipped with dual AMD Opteron 2.8 GHz CPUs, 4GB memory, and same InfiniBand DDR HCA. The operating system used is RedHat AS4 with kernel 2.6.17.7. The file system used is PVFS2 on top of local SATA disks. The network protocol used by PVFS2 is IP over IB (IPoIB). The base performance of this file system configuration is shown in Figure 1.

6.1. Evaluation with Micro-benchmarks

To evaluate the benefits of group-based checkpointing, we designed a micro-benchmark to emulate communication in MPI applications. Since the performance benefits mainly depend on how processes are synchronized, we use abstracted models as explained later. The memory footprint in the micro-benchmarks is configured to be 180MB per process, which is sufficiently large to show the trend.

Figure 3 shows the impact of different checkpoint group sizes on the Effective Checkpoint Delay. In this experiment, MPI processes communicate only within a communication group using blocking MPI calls continuously, effectively synchronizing themselves in groups. The group size (Comm. Group Size) is set to be 16, 8, 4, 2, and 1 (embarrassingly parallel). The results show that when a checkpoint group covers one or more communication groups, the delay will be reduced approximately by half when the checkpoint group size is reduced by half. When a checkpoint group is smaller than a communication group, the delay remains on the same level, or even increases when the checkpoint group size is very small (2 or 1). That is mainly due to the under-utilization of the storage throughput.

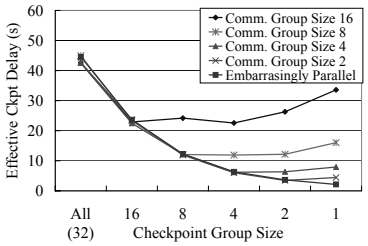

Figure 3. Checkpoint Group Size

The issuance time of checkpoint request, i.e. the checkpoint placement time, is also an important parameter affecting the benefits of group-based checkpointing. In this experiment, we set both the checkpoint group size and the communication group size to be eight, and enforce a global synchronization using `MPI_Barrier` every minute. As shown in Figure 4, the Effective Checkpoint Delay lies in between the Individual Checkpoint Time and Total Checkpoint Time. When the checkpoint is placed closer to the synchronization line (indicated by the vertical line), the delay is larger, closer to the Total Checkpoint Time. It is because that in this case, the process groups which finish their checkpoint earlier can not progress across the global barrier to the next phase of execution without violating the semantics of the barrier.

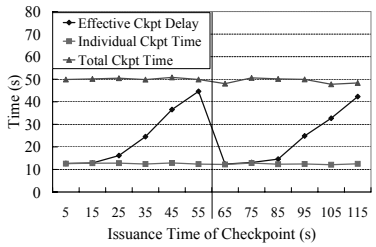

Figure 4. Checkpoint Placement

From the micro-benchmark results, we can see that both checkpoint group size and issuance time of the checkpoint are important parameters. In practice, checkpoint group size should be chosen according to application's communication group size and storage characteristics. And checkpoint request should be placed long before synchronization to achieve better overlap of storage access and computation.

6.2. Evaluation using HPL

In this section, we describe the experimental results for High Performance Linpack (HPL) [20] benchmark. HPL is to solve a dense linear system on distributed-memory computers. The matrix data are distributed to a two-dimensional grid of processes, and processes mostly communicate in the same row or column. In our experiments, we choose a 8×4 configuration with a larger block size. Thus the communication group size is effectively four.

In the experiment, we choose eight time points evenly distributed across the execution time, and measure the Effective Checkpoint Delay for different checkpoint group sizes. Figure 5 shows the detailed experimental results, from which we observe that in general the delay in the cases with group size 2, 4, 8, or 16 is noticeably less than the regular checkpointing case, where the group size is 32. Up to a 78% reduction in delay for group size 4 at the point of 50 seconds was observed. Average reductions for all time points with group sizes 2, 4, 8, and 16 are 37%, 46%, 46%, and 35%, respectively. However, with group size 1, the delay is not reduced much, and in some cases becomes even worse. That is expected because the processes are not able to make progress individually and the full bandwidth provided by parallel file system is not fully utilized. Note that checkpointing delays are different for different time points even in regular checkpointing case, because the memory footprint is not constant during the execution time.

Figure 6 shows the average checkpoint delay with respect to different checkpoint group sizes, with a vertical line indicating the maximum and minimum delay for each group size. We can clearly see that the checkpoint group sizes of 4 and 8 give the best performance. These results match the configuration, 8×4 processes, as used in HPL experiments.

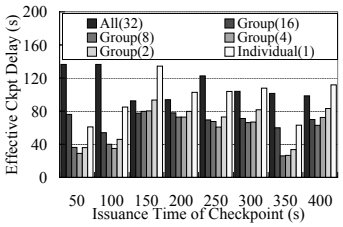

Figure 5. Effective Checkpoint Delay at 8 Time Points for HPL

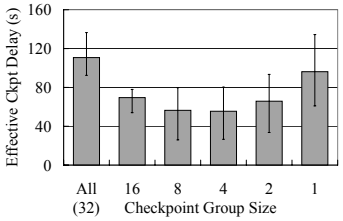

Figure 6. Effective Checkpoint Delay with Different Checkpoint Group Sizes for HPL

6.3. Evaluation using MotifMiner

In this section, we demonstrate that even for the applications which do not follow a group-based communication pattern, the group-based checkpointing can still benefit to a certain degree. To evaluate the performance, we use the parallel version of MotifMiner [24], a data mining toolkit that can mine for structural motifs in a wide area of biomolecular datasets. The algorithm follows an iterative pattern, and the `MPI_Allgather` is used to exchange data after each iteration. We use an intra-molecule frequent substructures analysis query on protein lysozyme as our dataset, and run the program using 32 processes. We choose four points distributed across the execution time.

Figure 7 shows the Effective Checkpoint Delay with respect to different checkpoint group sizes. We observe that the delay is noticeably reduced, up to 70% for group size 4 at the point of 30 seconds. The average reductions of checkpointing delay are 28%, 32%, 27%, and 14% with the checkpoint group sizes of 16, 8, 4, and 2, respectively. The benefits are because MotifMiner is very computation intensive, therefore although it only does global communication, each process still has a relative large chunk of computation before they synchronize. In this case, the groups which finish their checkpoints earlier can continue their computation to some extent before waiting for other groups.

7. Related Work

Many efforts have been carried out to provide fault tolerance to MPI programs. In recent years, the MPICH-V team [1] has developed and evaluated several roll-back recovery protocols, including both uncoordinated checkpointing with message-logging protocols, such as V1 [5], V2 [6],

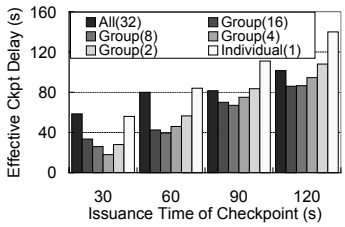

Figure 7. Effective Checkpoint Delay with Different Checkpoint Group Sizes for MotifMiner

Vcausal [7], and coordinated checkpointing protocols, such as Vcl [8] based on Chandy-Lamport Algorithm [9], and Pcl [10] based on the blocking coordinated checkpointing protocol. FT-MPI [13] has extended the MPI specification to provide support to applications to achieve fault tolerance on application level. LAM/MPI [21] has incorporated checkpoint/restart capabilities based on Berkeley Lab's Checkpoint/Restart (BLCR) [11] to checkpoint MPI program, which also uses the blocking coordinated checkpointing protocol. Their efforts have been recently incorporated in OpenMPI project with an extended framework [16]. Recently, a job pause service [23] has been proposed based on LAM/MPI's checkpointing framework to utilize process migration to achieve very efficient failure recovery. In our earlier work [14], we have proposed a framework to checkpoint MPI programs over InfiniBand using a blocking coordinated checkpointing protocol and BLCR.

In this paper, we extended the blocking coordinated checkpointing design to take group-based checkpoints, which reduces the effective checkpoint delay and improves the scalability of checkpointing for MPI. The group-based checkpointing differs from uncoordinated checkpointing and non-blocking Chandy-Lamport coordinated checkpointing in the sense that a consistent global checkpoint is formed by the individual checkpoints without message-logging at any time. This is a critical feature for clusters that use high performance interconnects such as InfiniBand, where message logging can potentially impose a large overhead. Observations for the noticeable message logging overhead in checkpointing MPI on high speed network has also been reported recently in [10].

Another approach to deal with the storage bottleneck is incremental checkpointing. Recently, a kernel-level checkpointer called TICK [15] has been designed for parallel computers with incremental checkpointing support. We believe that our group-based checkpointing mechanism can be combined with incremental checkpointing techniques to further reduce the checkpointing overhead.

8. Conclusions and Future Work

In this paper, we have presented a design of group-based checkpointing as an extension to the coordinated checkpointing protocol to address the scalability limitations. By

carefully scheduling the processes in an MPI job to take checkpoints group by group, the group-based checkpointing alleviates the storage bottleneck and reduces the checkpointing delay observed by every process. A prototype implementation has been developed based on MVAPICH2, and a detailed performance evaluation has been carried out using micro-benchmarks and applications on an InfiniBand cluster. The experimental results demonstrate that group-based checkpointing can reduce the effective delay for checkpointing significantly, up to 78% for HPL and up to 70% for MotifMiner. For future research, we plan to study in more depth on two important factors affecting the checkpoint delay, checkpoint group formation and checkpoint placement, on larger platforms. We also plan to study how to combine our design with incremental checkpointing techniques to further reduce the checkpoint delay.

Acknowledgements

We would like to thank Dr. Paul Hargrove from LBNL for helpful advice and discussions about BLCR. We would also like to thank Chao Wang and Dr. Srinivasan Parthasarathy for providing the MotifMiner code and helpful discussions.

References

[1] MPICH-V Project. http://mpich-v.lri.fr.

[2] MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich2/.

[3] MVAPICH: MPI for InfiniBand and iWARP. http://nowlab.cse.ohio-state.edu/projects/mpi-iba/.

[4] Parallel Virtual File System, Version 2. http://www.pvfs.org.

[5] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Magniette, V. Néri, and A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *IEEE/ACM SuperComputing 2002*, Baltimore, MD, November 2002.

[6] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *IEEE/ACM SuperComputing 2003*, Phoenix, AZ, November 2003.

[7] A. Bouteiller, B. Collin, T. Hérault, P. Lemarinier, and F. Cappello. Impact of event logger on causal message logging protocols for fault tolerant MPI. In *Proceedings of Int'l Parallel and Distributed Processing Symposium (IPDPS)*, Denver, CO, April 2005.

[8] A. Bouteiller, P. Lemarinier, T. Hérault, G. Krawezik, and F. Cappello. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In *Proceedings of Cluster 2004*, San Diego, CA, September 2004.

[9] M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. In *ACM Trans. Comput. Syst. 31*, 1985.

[10] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In *ACM/IEEE SuperComputing (SC)*, 2006.

[11] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941, Berkeley Lab, 2002.

[12] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3), 2002.

[13] G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. J. Dongarra. Extending the MPI Specification for Process Fault Tolerance on High Performance Computing Systems. In *International Supercomputer Conference (ICS)*, 2003.

[14] Q. Gao, W. Yu, W. Huang, and D. K. Panda. Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand. In *Int'l Conference on Parallel Processing (ICPP '06)*, Columbus, OH, August 2006.

[15] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini. Transparent incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers. In *ACM/IEEE SuperComputing 2005*, Seattle, WA, November 2005.

[16] J. Hursey, T. Mattox, A. Lumsdaine, and J. M. Squyres. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *Workshop on Dependable Parallel, Distributed and Network-Centric Systems(DPDNS), in conjunction with IPDPS*, 2007.

[17] InfiniBand Trade Association. http://www.infinibandta.org.

[18] M. Leininger. InfiniBand and OpenFabrics Successes and Future Requirements. http://www.infinibandta.org/events/DevCon2006_presentations/0930_Past_successes.pdf, 2006.

[19] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '04)*, April 2004.

[20] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. High performance linpack. http://www.netlib.org/benchmark/hpl/.

[21] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. *International Journal of High Performance Computing Applications*, pages 479–493, 2005.

[22] J. S. Vetter and F. Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS) 2002*, Fort Lauderdale, FL, April 2002.

[23] C. Wang, F. Mueller, C. Engelmann, and S. Scott. A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '07)*, 2007.

[24] C. Wang and S. Parthasarathy. Parallel Algorithms for Mining Frequent Structural Motifs in Scientific Data. In *ACM International Conference on Supercomputing (ICS) 2004*, Malo, France, June 2004.