Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand

QI GAO, WEIKUAN YU, WEI HUANG, AND DHABALESWAR K. PANDA

Technical Report
OSU-CISRC-2/06-TR25

Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand *

Qi Gao Weikuan Yu Wei Huang Dhabaleswar K. Panda

Network-Based Computing Lab
Dept. of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{*gaoq, yuw, huanwei, panda*}*@cse.ohio-state.edu*

Abstract

Ultra-scale computer clusters with high speed interconnects, such as InfiniBand, are widely deployed for their excellent performance and cost effectiveness. However, the failure rate in turn exponentially increases along with their augmented number of components. It becomes imperative for such systems to be equipped with fault tolerance support. In this paper, we present our design and implementation of checkpoint/restart framework for MPI programs running over InfiniBand clusters. Our design enables low-overhead, application-transparent checkpointing. It uses coordinated protocol to save the current state of the whole MPI job to reliable storage, which allows users to perform rollback recovery if the system runs into some faulty state later. Our solution has been incorporated into MVAPICH2, an open-source high performance MPI-2 implementation over InfiniBand. Performance evaluation of this implementation has been carried out using NAS benchmarks, HPL benchmark, and a real-world application called Gromacs. Experimental results indicate that in our design, the overhead to take checkpoints is low, and the performance impact for checkpointing applications periodically is insignificant. For example, time for checkpointing Gromacs is less than 0.3% of the execution time, and its performance only decreases 4% with checkpoints taken every minute. To the best of our knowledge, this work is the first report of checkpoint/restart support for MPI over InfiniBand clusters in the literature.

1 Introduction

High End Computing (HEC) systems are quickly gaining in their speed and size. In particular, more and more computer clusters with multi-thousand nodes are getting deployed during recent years because of their low price/performance ratio. While the simple statistics calculation tells us that the failure rate of an entire system grows exponentially with the number of the components, few of such large-scale systems are equipped with built-in fault tolerance support. The applications running over these systems also tend to be more error-prone as the failure of any single component cascades widely to other components because of the interaction and dependence between them.

The Message Passing Interface (MPI) [19] is the *de facto* industrial parallel programming standard on which parallel applications are typically written. However, it has no specification about the fault tolerance support that a particular implementation must achieve. As a result, most MPI implementations are designed without the fault tolerant support, providing only two modes of the working state, either RUNNING or FAILED. Faults occurred during the execution time often abort the program and the program has to start from the beginning. For long running programs, this can waste a large amount of computing resources because all the computation that has already been accomplished is lost. To save the valuable computation resources, it is desirable that a parallel application can restart from some previous state before a failure occurs and continue the execution. Thus checkpointing and rollback recovery is the most commonly used technique in fault recovery.

The InfiniBand Architecture (IBA) [16] has been recently standardized in industry to design next generation high-end clusters for both data-center and high performance computing. Large cluster systems with InfiniBand are being deployed. For example, in the Top500 list recently released on November 2005 [28], the 5th, 20th, and 51st most powerful supercomputers use InfiniBand as their parallel application communication interconnect. These systems can have as many as 8000 processors. It becomes imperative for such large-scale systems to be deployed with checkpoint/restart support so that the long-running MPI parallel programs are able to recover from failures. However, it is still an open challenge to provide checkpoint/restart support for MPI programs over such InfiniBand clusters.

*This research is supported in part by a DOE grant #DE-FC02-01ER25506 and NSF Grants #CNS-0403342 and #CNS-0509452; grants from Intel, Mellanox, Cisco Systems, Linux Networx, and Sun MicroSystems; and equipment donations from Intel, Mellanox, AMD, Apple, Appro, Microway, PathScale, IBM, SilverStorm and Sun MicroSystems.

In this paper, we take on this challenge to provide checkpoint/restart for MPI programs over InfiniBand clusters. Based on the capability of Berkeley Lab's Checkpoint/Restart(BLCR) [11] to take snapshots of processes on a single node, we design a checkpoint/restart framework to take global checkpoints of the whole MPI program while ensuring their consistency. We have implemented our design of checkpoint/restart in MVAPICH2 [22], which is an open-source high performance MPI-2 implementation over InfiniBand, and is being widely used by the high performance computing community. Checkpoing/restart-capable MVAPICH2 (MVAPICH2-CR) enables low-overhead, application-transparent checkpointing for MPI applications with only insignificant performance impact on MPI applications. For example, time for checkpointing Gromacs [1] is less than 0.3% of the execution time, and its performance only decreases 4% with checkpoints taken every minute. To the best of our knowledge, this work is the first report of checkpoint/restart support for MPI over InfiniBand clusters in the literature.

The rest of the paper is organized as follows: In section 2 and section 3, we describe the background of our work, and identify the challenges involved in checkpointing InfiniBand parallel applications. In section 4, we present our design in detail with discussions on some key design issues. In section 5, we describe the experimental results of our current implementation. In section 6, we discuss related works. Finally, we provide our conclusions and describe future works in section 7.

2 Background

2.1 InfiniBand and MVAPICH2

InfiniBand[16] is emerging as an open standard of next generation high speed interconnect. In addition to send/receive semantics, it provides memory-based semantics, Remote Direct Memory Access (RDMA), for high performance interprocess communication. By directly accessing and/or modifying the contents of remote memory, RDMA operations are one sided and do not incur CPU overhead on the remote side. Because of its high performance, InfiniBand is gaining wider deployment as high end computing platforms [28].

MVAPICH2 [22] is designed and implemented based on its predecessor MVAPICH [18] and MPICH2 [2], developed by Argonne National Laboratory. MVAPICH2 is an open-source high performance implementation of MPI-2 standard. MVAPICH2, along with MVAPICH, is currently being used by more than 320 organizations across the world. Currently enabling several large-scale InfiniBand clusters, MVAPICH2 includes a high performance transport device over InfiniBand, which takes advantage of RDMA capabilities.

2.2 Fault Tolerance and Checkpoint/Restart

With the increasing number of components integrated into one system, the failure rate of the system grows exponentially. Therefore, among several approaches to achieve fault tolerance in parallel computing, failure recovery is a fundamental one to control the damage to applications caused by a system failure. Checkpointing and rollback recovery is the most commonly used technique for failure recovery. Periodic checkpointing can bound the loss in the failure to the amount of computation between two consecutive checkpoints.

With respect to the transparency to user application, checkpoint/restart techniques can be divided into two categories: user-level checkpointing and system-level checkpointing. The former usually involves user application in the checkpointing procedure. While gaining some advantage of efficiency with assistance from the user application, this approach has a major drawback: user applications need to be tailored to the system, which often involves a significant amount of work for each application. The latter is also called application-transparent checkpointing, because it can perform checkpoint/restart in a totally transparent manner to user applications. Although it may involve more overhead, it does not need any code modification of applications. Thus we follow this approach.

In distributed checkpointing, there are two main categories of protocols: coordinated checkpointing and uncoordinated checkpointing with message logging. The former involves coordination between processes, and thus has more overhead of checkpointing. The latter often leads to general performance degradation even in fault-free running. We opt to choose the former because: (a) it is free from *domino effect* [24], and (b) network message logging can potentially impose considerable overhead for high-bandwidth interconnect such as InfiniBand.

In most MPI implementations, a communication channel is an abstraction of a network connection. Despite the different transportation services provided by different networks, communication channels must provide the same set of interfaces to upper layers. Therefore, one channel not only includes the network connection information, but also maintains communication buffers and progress information. Because network connection information is involved, communication channels are usually difficult to be totally checkpointed. In coordinated checkpointing protocol, there are two different alternatives to handle the communication channels when checkpointing. In the first alternative, processes communicate with each other to ensure that they all reach a point where all communication channels are totally empty. Then they can safely discard communication channel state and take checkpoint individually. These checkpoints are guaranteed to be globally consistent. However, depending on the different communication patterns of user application, the coordination step is potentially very long. In the second alternative, processes store the states for all communication channels into checkpoints, including progress information and communication buffers. Although the reduced coordination time leads to higher responsiveness, preserving the consistency among communication channels is nontrivial, especially for the communication channel based on high performance interconnects, such as InfiniBand. Challenges and issues involved in checkpointing parallel applications on InfiniBand cluster are elaborated in

next section.

3 Challenges

Checkpointing parallel applications running over InfiniBand cluster is different from checkpointing those over other networks, such as Myrinet [20], regular TCP/IP networks, etc. Although InfiniBand also provides TCP/IP support using IP over IB (IPoIB), it does not deliver as good performance as native InfiniBand verbs. In this section, we explore several challenging issues in checkpointing the parallel programs that are built over native InfiniBand protocols as follows.

First, parallel processes over InfiniBand communicate via an OS-bypass user-level protocol. In regular TCP/IP networks, operating system (OS) kernels intercept all network activities, so these network activities can be temporarily stopped in an application-transparent manner. However, InfiniBand provides its high performance communication via OS-bypass capabilities in its user-level protocol [6]. The use of these user-level protocols has the following side effect: the operating system is skipped in the actual communication and does not maintain the complete information of ongoing network activities. Because of this gap of information regarding the communication activities between the OS kernel and the user-land of application process, it becomes difficult for the operating system to directly stop network activities and take checkpoints without loosing consistency.

Second, the context of network connection is available only in network adapter. In regular TCP/IP networks, the network communication context is stored in kernel memory, which can be saved to checkpoints. Different from TCP/IP networks, InfiniBand network adapter stores the network connection context in the adapter memory. This part of information is designed to be volatile, and thus very difficult to be reused by restarted process. Therefore, network connection context has to be released before checkpoint, and rebuilt afterwards. As InfiniBand uses user-level protocol, some network context information, such as Queue Pairs (QPs), is also cached in user memory, which must be reconstructed according to new network connection context before a process continues communication. And the releasing/rebuilding of network connections should be totally transparent to applications.

Third, some network connection context is even copied in remote node. Because of their high performance, many applications take advantage of the RDMA operations provided by InfiniBand. Different from some other RDMA capable networks such as Myrinet, InfiniBand requires authentication for accessing remote memory. Before process A accesses remote memory in process B, process B must register the memory to network adapter, and then inform process A about the virtual address of the registered memory and the remote key to access that part of memory. Then process A must cache that key and include it in the RDMA requests so that the network adapter for process A can match the keys and authorize the memory access. Since these keys will become invalid when network connection context is rebuilt, potential inconsistency may be introduced by the invalid keys.

4 Checkpoint/Restart Framework and Design Issues

In this section, we present the detailed checkpoint/restart framework for MPI over InfiniBand and some important design issues. As we characterize the issues, we focus on these issues in particular: (a) how to stop an MPI program into a state which can be consistently saved to a checkpoint, and (b) how to resume an MPI program based on a checkpoint. There are three design objectives for this framework:

- Consistency: the global consistency of the MPI program must be preserved.
- Transparency: the checkpoints must be taken transparently to MPI application.
- Responsiveness: upon a request, checkpoint must be taken as soon as possible.

We design a protocol to coordinate all MPI processes in the job to consistently and transparently suspend all InfiniBand communication channels between them, and preserve the communication channel states to checkpoint, which provides high responsiveness.

In the remaining part of this section, we start with the Checkpoint/Restart (C/R) framework, describing components in the framework and their functionalities. Then we provide a global view of C/R procedure by describing the overall state diagram and state transition. Finally, we discuss some design issues in a local view to show how to suspend/reactivate InfiniBand communication channels.

4.1 Checkpoint/Restart Framework

In a cluster environment, a typical MPI program consists of: a front-end **MPI job console**, a **process manager** crossing multiple nodes, and individual **MPI processes** running on these nodes. Multi Purpose Daemon (MPD) [9] is the default process manager for MVAPICH2. All the MPD daemons are connected as a ring. As depicted in Figure 1, the proposed C/R framework is built upon the MPI job structure, and there are five key components in this framework described as follows:

- **Global C/R Coordinator**: As a part of MPI job console, global C/R coordinator is responsible for global management of checkpoint/restart the whole MPI job. It handles checkpoint/restart requests from users or administrators, and it also can automatically initiate checkpoints periodically. It maintains a state machine for C/R, and organizes the global synchronization when needed.
- **Control Message Manager**: Control message manager provides an interface between global C/R coordinator and local C/R controller. It utilizes the process manager already deployed in the cluster to provide out-of-band messaging between MPI processes and the job console.

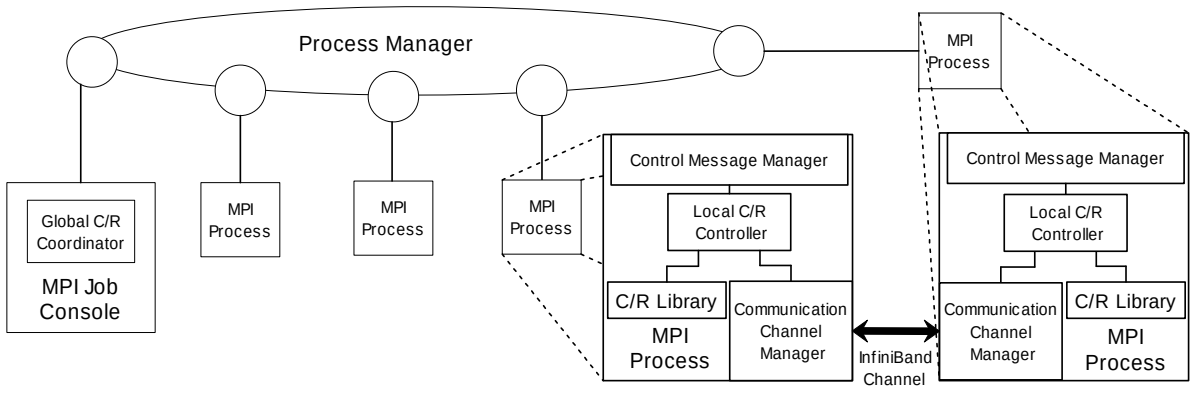

Figure 1. Checkpoint/Restart Framework

With this support, control messages for C/R can avoid interfering or being interfered by the in-band communication. In our current implementation, we extend the functionality of MPD to support C/R control messages.

- **Local C/R Controller**: Local C/R controller takes the responsibility of local management of the C/R operations for each MPI process. Its functionality can be described as follows: (a) to take C/R requests from and report the results to global C/R coordinator; (b) to cooperate with communication channel managers and other C/R controllers in peer MPI processes to converge the whole MPI job to a state which can be consistently checkpointed; and (c) to invoke C/R library to take checkpoints locally. In our current implementation, we use a separate pthread dedicated for C/R controller so that it can accept requests at any time. Although it involves some additional synchronization with the main thread, this design improves the responsiveness of checkpointing and eliminates some deadlock possibilities.
- **C/R Library**: C/R library is responsible for checkpointing/restarting the local process. Checkpointing a single process on a single node has been studied extensively and there are several packages available to the community. In our current implementation, we use Berkeley Lab's Checkpoint/Restart (BLCR) [13] toolkit.
- **Communication Channel Manager**: Communication channel manager controls the in-band message passing. In C/R framework, it has extended functionalities of suspending/reactivating the communication channel, and the temporary suspension does not impair the channel consistency, and is transparent to upper layers. Currently, we implement the C/R functionality on the InfiniBand channel based on OpenIB [23] Gen2 stack.

4.2 Overall Checkpoint/Restart Procedure

Figure 2 shows the state diagram of the the overall design of our checkpoint/restart framework. During a normal run, the job periodically goes over the checkpointing

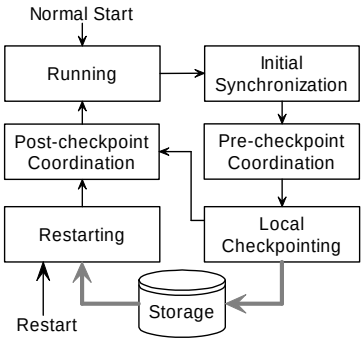

Figure 2. State Diagram for Checkpoint/Restart

cycle, which consists of four phases: initial synchronization, pre-checkpoint coordination, local checkpointing, post-checkpoint coordination, and then comes back to running state.

- In **Initial Synchronization Phase**, all processes in the job synchronize with each other and prepare for pre-checkpoint coordination. First, the global C/R coordinator from the MPI job console propagates a checkpoint request to all local C/R controllers running in individual MPI processes. Then, upon the arrival of the request, the local C/R controller takes the control of communication channels from the main thread to avoid possible inconsistency of communication channels that might be caused by interleaved accesses from both C/R controller and main thread.
- In **Pre-checkpoint Coordination Phase**, C/R controllers coordinate with each other to make all MPI processes individually checkpointable while preserving global consistency. To do so, C/R controllers cooperate with communication channel managers to suspend all communication channels temporarily and release the network connections in these channels.
- In **Local Checkpointing Phase**, C/R controllers invoke C/R library to save the current state of the local MPI process, including the state of suspended communication channel to a checkpoint file.

- In **Post-checkpoint Coordination Phase**, C/R controllers cooperate with communication channel managers to reactivate communication channels. This step involves rebuilding the low level network connections and resolving the possible inconsistency introduced by the potentially different network connection information.

The issues involved in how to suspend/reactivate communication channels consistently and transparently will be discussed in section 4.3.

To restart from a checkpoint, a restarting procedure is performed, which consists of restarting phase and post-checkpoint coordination phase.

In **Restarting Phase**, the global C/R coordinator first propagates the restart request to each node, where the C/R library is responsible for restarting the local MPI process from the checkpoint file. Then, local C/R controller reestablishes the connection between a MPI process and its process manager and performs necessary coordination between them.

At this point of time, the MPI job is restarted from a state same as a previous state in local checkpointing phase. Therefore, to continue running, it first goes to post-checkpoint coordination phase, and when all communication channels are reactivated, it comes back to running state.

4.3 Suspension/Reactivation InfiniBand Channel

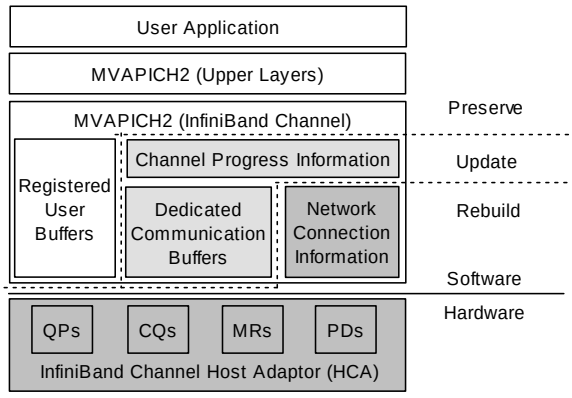

Figure 3. InfiniBand Channel for MVAPICH2

During the checkpoint/restart procedure described in previous section, the consistency and transparency are two key requirements. In this section, we explain how we transparently suspend/reactivate the InfiniBand communication channel while preserving the channel consistency.

The structure of InfiniBand communication channel in MVAPICH2 can be described by Figure 3. Below the MVAPICH2 InfiniBand channel is the InfiniBand Host Channel Adapter (HCA), which maintains the network connection context, such as Queue Pairs (QPs), Completion Queues (CQs), Memory Regions (MRs), and Protection Domains (PDs). MVAPICH2 InfiniBand channel state consists of four parts:

- **Network connection information** is the user-level data structures corresponding to the network connection context.
- **Dedicated communication buffers** are the registered buffers which can be directly accessed by HCA for sending/receiving small messages.
- **Channel progress information** is the data structures for book-keeping and flow control, such as pending requests, credits, etc.
- **Registered user buffers** are the memory allocated by user applications. These buffers are registered by communication channel to HCA for zero-copy transmission of large messages.

These four parts need to be handled differently according to their different natures when performing checkpointing. Network connection information needs to be cleaned before checkpointing and rebuilt afterwards, as the network connection context in HCA is released and rebuilt. Dedicated communication buffers, and channel progress information need to be mostly kept same but also updated partially because they are closely coupled with network connection information. Registered user buffers need to be re-registered but the content of them need to be totally preserved.

Now we explain the protocol for suspending and reactivating communication channel, including the discussion on some design issues.

In pre-checkpoint coordination phase, to suspend communication channels, channel managers first drain all the in-transit messages, which means that to a certain synchronization point, all the messages before that point must have been delivered and all the messages after that point must have not been posted to network.

Two things need to be noted here: (a) the word ‘messages’ refer to the network level messages rather than MPI level messages, and one MPI level message may involve several network level messages, and (b) the synchronization points for different channels do not need to correspond to the same time point, and each channel can have its own synchronization point.

Due to the First-In-First-Out (FIFO) nature of InfiniBand Reliable Connection-based (RC) channel, draining in-transit messages can be achieved by exchanging flag messages between each pair of channel managers. These flag messages represent the synchronization points for the channels. Once the channel manager detects the local completion for each flag message sent to each channel, and receives a flag message from each channel, the channel is successfully suspended. Then, the channel manager releases the underlying network connection.

One key issue involved is when the channel manager should process the messages received before the flag message, which are the drained in-transit messages. Because the communication channel is designed to execute the transmission protocol chosen by upper layers in the MPI library, processing an incoming message may cause sending a response message, which may lead to an infinite ‘ping-pong’ livelock condition. To avoid that, the channel manager has to either buffer the drained messages for future processing, or process these mes-

Benchmark:	lu.C.8	bt.C.9	sp.C.9
Checkpoint File Size (MBs):	126	213	193

Table 1. Checkpoint File Size per Process

sages but buffer the response messages instead of sending them immediately. We choose the latter approach because: (a) some control messages need to be processed immediately, and these control messages will not lead to any response message; (b) the overhead for buffering is lower as the number of response messages is generally smaller than the number of incoming messages.

In post-checkpoint coordination phase, after rebuilding underlying network connections, the channel manager first updates the local communication channel as we described before, and then sends control messages to update the other side of the channel. The remote updating is to resolve the potential inconsistency introduced by invalid remote keys for RDMA operation. This issue has been discussed in Section 3. For example, for performance reason, the rendezvous protocol for transmitting large messages is implemented by RDMA write operation. To achieve high responsiveness and transparency, our design allows rendezvous protocol being interrupted by checkpointing. Therefore the remote keys cached in sender side for RDMA write will become invalid because of the re-registration on receiver side. Hence, the channel manager on receiver side needs to capture the refreshed remote keys and send them to the sender side.

5 Performance Evaluation

In this section, we describe experimental results and analyze the performance of our current implementation based on MVAPICH2-0.9.2. The experiments are conducted on an InfiniBand cluster of 12 nodes. Each node is equipped with dual Intel Xeon 3.4GHz CPUs, 2GB memory and a Mellanox MT25208 PCI-Express InfiniBand HCA. The operating system used is Linux Redhat AS4 with kernel 2.6.11. The filesystem we use is ext3 on top of local SATA disk.

We evaluate the performance of our implementation using NAS parallel Benchmarks [29], High-Performance Linpack (HPL) [3] Benchmark, and a real-world application called Gromacs [1]. First, we analyze the overhead for taking checkpoints and restarting from checkpoints, and then we show the performance impact to applications for taking checkpoints periodically.

5.1 Overhead Analysis for Checkpoint/Restart

In this section, we analyze the overhead for C/R in terms of checkpoint file size, checkpointing time, and restarting time. We choose BT, LU, and SP from NAS Parallel Benchmarks and HPL Benchmarks, because they reflect the computation kernel commonly used in scientific applications.

Because checkpointing involves saving the current state of running processes into reliable storage, taking a system-level full checkpoint involves writing all used memory pages within process address space to the checkpoint file, therefore, checkpoint file size is determined by the memory footprint of the process, in this case, MPI process. Table 1 shows the checkpoint file sizes per process for BT, LU, and SP, class C, using 8 or 9 processes.

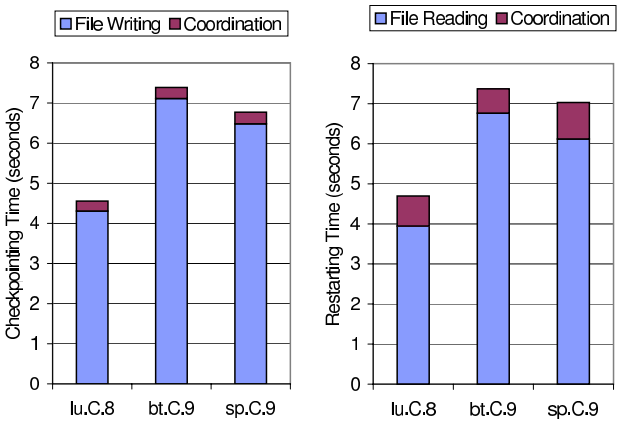

Figure 4. Overall Time for Checkpointing/Restarting NAS

Time for checkpointing/restarting is determined mainly by three factors: the time for synchronization, which increases with the system size; the time for writing/reading the checkpoint file to/from file systems, which depends on both the checkpoint file size and the performance of the underlying file system; and the time for suspending and reactivating communication channels, which mostly depends on the number of InfiniBand connections one process has.

Figure 4 shows the time for checkpointing/restarting NAS benchmarks. It also provides the file accessing time for the checkpoint file for comparison. With the limited performance of underlying ext3 file systems over SATA disks on our system, we have observed that the file accessing time is the dominating factor. In the real-world deployment, high performance parallel file system can be used to store the checkpoint files for better performance. We plan to further investigate the issues on how to speed up the commitment of checkpoint files.

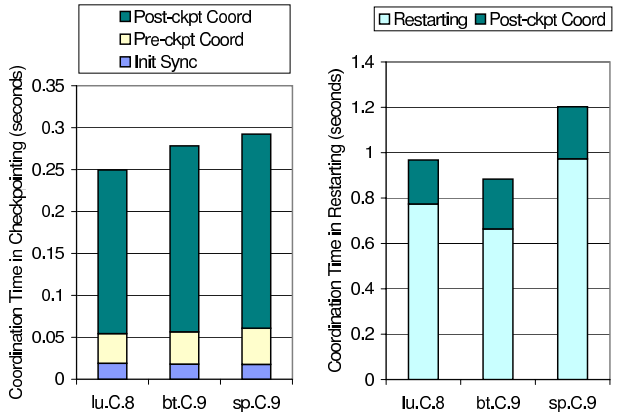

Figure 5. Coordination Time for Checkpointing/Restarting NAS

To further analyze the coordination overhead, we excluded the file accessing time and broke the coordination time down to individual phases. As shown in Figure 5, for checkpointing, post-checkpoint coordination consumes most of the time. The reason is that this step involves a relatively time-consuming component, the establishment of InfiniBand connections, which has been explored in our previous study [31]. The response time, which is the sum of initial synchronization time and pre-checkpoint coordination time, represents the delay from the issuance of checkpoint request to the time point where every MPI process is ready to be saved to storage. Our system has a very small response time, around 0.06 second. For restarting, the post-checkpoint coordination consumes almost the same amount of time as for checkpointing, but the major part of time is in restarting phase, mainly spent by MPD and BLCR for spawning processes on multiple nodes.

To evaluate the scalability of our design, we measure the average checkpointing time for HPL benchmark using 2, 4, 6, 8, 10, and 12 processes. In the experiment we choose the problem size to let HPL benchmark consume around 800MB memory for each process.

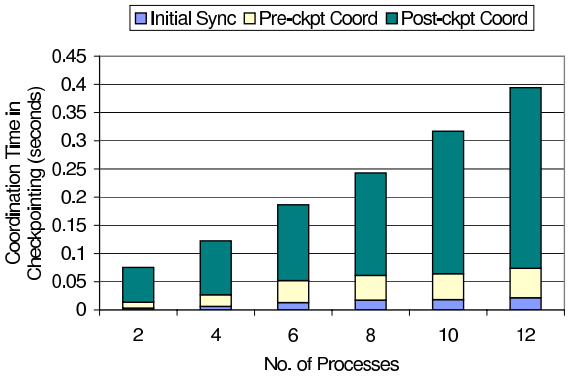

Figure 6. Coordination Time for Checkpointing HPL

To improve the scalability, we adopt the technique of bootstrap channel described in [31] to reduce the InfiniBand connection establishment time from the order of $O(N^2)$ to $O(N)$, where N is the number of connections. As shown in Figure 6, because the dominating factor, post-checkpoint coordination time, is $O(N)$, the overall coordination time is also in the order of $O(N)$. To further improve the scalability of checkpoint/restart, we plan to utilize adaptive connection management model [30] to reduce the number of active InfiniBand connections.

Nonetheless, with current performance, our design is sufficient for checkpointing most applications because the time for checkpointing/restarting is insignificant when comparing to the execution time of applications.

5.2 Performance Impact for Checkpointing

In this section, we evaluate the performance of our system in a working scenario. In real world, periodically checkpointing MPI application is a commonly used method to achieve fault tolerance. We conduct experiments to analyze the performance impact for taking checkpoints at different frequencies during the execution time of applications. We used LU, BT, and SP from NAS benchmarks and HPL benchmark to simulate user application. And we also include a real-world application called Gromacs [1], which is a package to perform molecular dynamics for biochemical analysis.

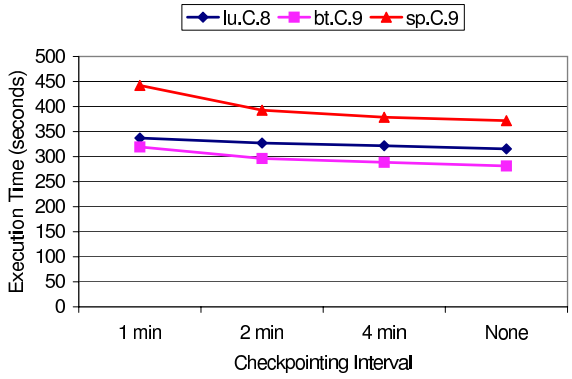

Figure 7. Performance Impact for Checkpointing NAS

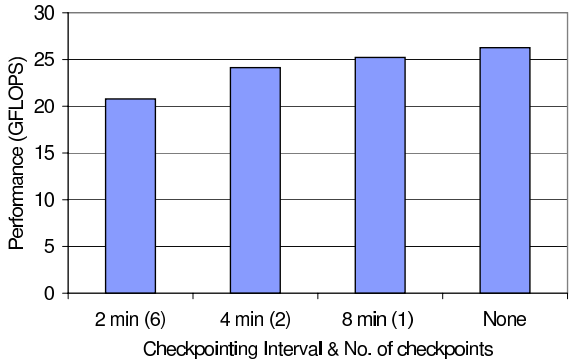

Figure 8. Performance Impact for Checkpointing HPL

As shown in Figure 7, the total running time of LU, BT, and SP decreases as the checkpointing interval increases. The additional execution time caused by checkpointing matches the theoretical value: *checkpointing time* $\times$ *number of checkpoints*. Figure 8 shows the impact on calculated performance in GFLOPS of HPL benchmarks for 8 processes.

Because these benchmarks load all data to memory at the beginning of execution, the checkpoint file size is relatively large. Therefore, in our experiments, the dominating part of the overhead for checkpointing, the file writing time, is relatively large. But even with this overhead, the performance does not decrease much with a reasonable long checkpointing interval, 4 minutes for example.

On the other hand, many real-world applications may spend hours even days to process many thousands of datasets for a run. Normally only a small portion of datasets are loaded into memory at any point time, so the memory footprints for these applications are relatively small. Therefore the overhead for checkpointing is lower with respect to their running time.

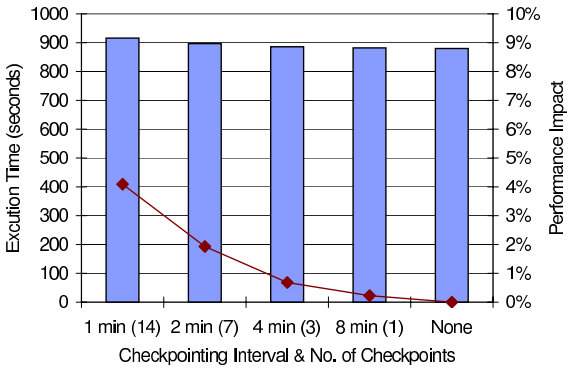

Figure 9. Performance Impact for Checkpointing Gromacs

To evaluate this case, we run Gromacs on DPPC benchmark dataset [1], which is to simulate a phospholipid membrane, consisting of 1024 dipalmitoylphosphatidylcholine (DPPC) lipids in a bilayer configuration As shown in Figure 9, the time for checkpointing Gromacs is less than 0.3% of its execution time, and even if Gromacs is checkpointed every minute, the performance degradation is still around 4%. From these experiments we can conclude that for long running applications, the performance impact of checkpointing is negligible, and even for memory intensive applications, with a reasonable checkpointing frequency, the performance impact is insignificant.

6 Related Works

Many efforts have been carried out to provide fault tolerance to message-passing based parallel programs. LAM/MPI [26] has incorporated checkpoint/restart capabilities based on Berkeley Lab's Checkpoint/Restart(BLCR) [11]. A framework to checkpoint MPI program running over TCP/IP network is developed. Another approach to achieve fault tolerance using uncoordinated checkpointing and message logging is studied in MPICH-V project [7, 8]. They have used the *Condor* standalone checkpoint library [17] to checkpoint MPI processes, and designed and evaluated a variety of message logging protocols for uncoordinated checkpointing for MPI programs over TCP/IP network. In [15], the design of a fault-tolerant MPI over Myrinet based on MPICH-GM [21] is described. Other researches toward fault tolerant message passing systems include: Starfish [4], CoCheck [27], FT-MPI [12], LA-MPI [5], Egida [25], CLIP [10], etc. Recently, a low-overhead, kernel-level checkpointer called TICK [14] has been designed for parallel computers with incremental checkpointing support.

Our work differs from the previous related works in the way that we addressed the challenges to checkpoint MPI programs over InfiniBand. To the best of our knowledge, this work is the first report in the literature on checkpoint/restart support for MPI programs over InfiniBand clusters.

7 Conclusions and Future Work

In this paper, we have presented our design of checkpoint/restart framework for MPI over InfiniBand. Our design enables application-transparent, coordinated checkpointing to save the state of the whole MPI program into checkpoints stored in reliable storage for future restart. We have evaluated our design using NAS benchmarks, HPL benchmark and a real-world application called Gromacs. Experimental results indicate that our design incurs a low overhead for checkpointing, and the performance impact of checkpointing to long running applications is negligible.

In future, we plan to incorporate the adaptive connection management to reduce the checkpoint time and checkpoint file size. We also plan to take advantage of high-bandwidth parallel file system to store checkpoint files. In a longer term, we intend to study the issues related to uncoordinated checkpointing for MPI over InfiniBand.

References

[1] http://www.gromacs.org/.

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

[3] A. Petitet and R. C. Whaley and J. Dongarra and A. Cleary. http://www.netlib.org/benchmark/hpl/.

[4] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *Proceedings of IEEE Symposium on High Performance Distributed Computing (HPDC) 1999*, pages 167–176, August 1999.

[5] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, and M. W. Sukalski M. A. Taylor, T. S. Woodall. Architecture of lampi, a network-fault-tolerant MPI. In *Proceedings of Int'l Parallel and Distributed Processing Symposium (IPDPS) 2004*, Santa Fe, NM, April 2004.

[6] R. A. F Bhoedjang, T. Rubl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, 1998.

[7] 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 *Proceedings of IEEE/ACM SC'2002*, Baltimore, MD, November 2002.

[8] 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 *Proceedings of IEEE/ACM SC'2003*, Phoenix, AZ, November 2003.

[9] R. Butler, W. Gropp, and E. Lusk. Components and interfaces of a process management system for parallel programs. *Parallel Computing*, 27(11):1417–1429, 2001.

[10] Y. Chen, K. Li, and J. S. Plank. CLIP: A Checkpointing Tool for Message-passing Parallel Programs. In *Proceedings of IEEE/ACM SC'97*, NOV 1997.

[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] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, A. Bukovski, and J. J. Dongarra. Fault Tolerant Communication Library and Applications for High Perofrmance. In *Los Alamos Computer Science Institute Symposium*, Santa Fee, NM, October 27-29 2003.

[13] Future Technologies Group (FTG). http://ftg.lbl.gov/CheckpointRestart/CheckpointRestart.shtml.

[14] 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 *Proceedings of ACM/IEEE SC'2005*, Seattle, WA, November 2005.

[15] H. Jung and D. Shin and H. Han and J. W. Kim and H. Y. Yeom and J. Lee. Design and Implementation of Multiple Fault-Tolerant MPI over Myrinet (M^3). In *Proceedings of ACM/IEEE SC'2005*, Seattle, WA, November 2005.

[16] Infiniband Trade Association. http://www.infinibandta.org.

[17] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[18] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *17th Annual ACM International Conference on Supercomputing (ICS '03)*, June 2003.

[19] Message Passing Interface Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3–4), 1994.

[20] Myricom. http://www.myri.com.

[21] Myricom. Myrinet Software and Customer Support. http://www.myri.com/scs/GM/doc/, 2003.

[22] Network-Based Computing Laboratory. MVAPICH: MPI for InfiniBand. http://nowlab.cse.ohio-state.edu/projects/mpi-iba/.

[23] Open Infiniband Alliance. http://www.openib.org.

[24] B. Randell. Systems structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, 1975.

[25] S. Rao, L. Alvisi, and H. M. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. In *Symposium on Fault-Tolerant Computing*, pages 48–55, 1999.

[26] 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*, 19(4):479–493, Winter 2005.

[27] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *Proceedings of the International Parallel Processing Symposium*, pages 526–531, April 1996.

[28] TOP 500 Supercomputers. http://www.top500.org/.

[29] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural Requirements and Scalability of the NAS Parallel Benchmarks. In *Proceedings of Supercomputing*, 1999.

[30] W. Yu, Q. Gao, and D. K. Panda. Adaptive connection management for scalable mpi over infiniband. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS) 2006*, Rhodes Island, Greece, April 2006. To appear.

[31] W. Yu, J. Wu, and D. K. Panda. Fast and scalable startup of mpi programs in infiniband clusters. In *Proceedings of International Conference on High Performance Computing 2004*, Banglore, India, December 2004.