Efficient Checkpoint/Restart for Multi-Channel MPI over Multi-core Clusters

Karthik Gopalakrishnan, Lei Chai, Wei Huang, Amith Mamidala,
Dhabaleswar K. Panda

May 25, 2009

Technical Report
OSU-CISRC-5/09-TR22

Efficient Checkpoint/Restart for Multi-Channel MPI over Multi-core Clusters

Karthik Gopalakrishnan, Lei Chai, Wei Huang, Amith Mamidala, Dhabaleswar K. Panda

Department of Computer Science & Engineering, The Ohio State University
2015 Neil Avenue, Columbus, OH 43210
{gopalakk, chail, huanwei, mamidala, panda}@cse.ohio-state.edu

***Abstract*—Ultra-scale computer clusters with high speed interconnects, such as InfiniBand, are being widely deployed for their excellent performance and cost effectiveness. However, the failure rate on these clusters also increases along with their augmented number of components. Thus, it becomes critical for such systems to be equipped with fault tolerance capability. Previous work has been done to enable MPI libraries with checkpoint/restart support. However, they all focus on the network channel only, and will not be the best solutions on multi-core clusters. In multi-core clusters, there exist several levels of communication, including intra-CMP, inter-CMP, and inter-node communication, depending on the positions of the processes. Many MPI libraries implement multiple channels to achieve best performance for the different levels of communication. Since the previous work all focus on the network channel only, they cannot take advantage of other channels. More specifically, they cannot utilize the efficient shared memory channel for intra-node communication. Therefore, application performance may get hurt with checkpoint/restart support. In this paper, we design a framework that is able to checkpoint multiple channels in MPI and provides high performance. In addition, our design allows flexible process re-distribution after restart. It can also be easily extended when new channels are added. We have integrated the design into MVAPICH2, and evaluated the performance on an Intel Clovertown multi-core cluster connected by InfiniBand. From the experimental results, we see that our design has little overhead compared with the native MVAPICH2 performance. Compared with the previous work, our design reduces intra-node latency from 4.55us to 0.76us, and increases peak bandwidth from 1510MB/s to 1954MB/s. It improves the performance of MPI collective operations by up to 90%, and application performance by up to 6%. Further, our design is efficient in taking the checkpoint. To the best of our knowledge, this is the first paper which focuses on designing a multi-channel enabled MPI checkpoint/restart protocol for multi-core clusters with Infiniband.**

I. Introduction

High End Computing systems are quickly gaining in their speed and size. In particular, adoption of clusters with tens of thousands of cores has been steadily increasing in the recent years due to their low price/performance ratio. While the failure rate of the entire system grows rapidly with the number of components, few large-scale systems are equipped with built-in support for fault tolerance. Applications running over these systems tend to be more prone to error since the failure of any single component can easily cascade to other components due to the interaction and dependence among them.

The Message Passing Interface (MPI) [1] is the de facto programming model on which parallel applications are typically written. Most clusters are installed with MPI libraries as the communication middleware. On large-scale clusters, it is highly desirable that MPI libraries have the capability of fault tolerance so that faults occurring during the execution of the applications can be recovered and do not result in program abort.

Checkpointing and rollback recovery is one of the commonly used techniques for fault recovery. To save valuable computing resources, the state of the parallel application is periodically saved so that in the event of a failure, the application can be restarted from a previous state and continue execution.

There exist MPI libraries that support checkpoint/restart (CR) over TCP/IP networks, such as LAM/MPI [2] and MPICH-V [3]. More recently, InfiniBand [4] has emerged as the high speed network to design the next generation high-end clusters. It is being widely deployed for both data centers and high performance computing. In fact, 121 systems in the Top500 supercomputer list released in November 2007 use InfiniBand as the interconnect. Realizing the increased popularity of InfiniBand in large-scale clusters, we have proposed a checkpoint/restart framework in MVAPICH2 (High Performance MPI over InfiniBand) [5]. The design challenges in this initial CR design for InfiniBand were mentioned in [6].

On the other hand, as multi-core technologies are becoming mainstream, more and more clusters are deploying multi-core processors as the build unit. Multi-core is also referred to as Chip-level Multi-Processing (CMP). In the latest Top500 [7] supercomputer list, 77% of the sites use multi-core processors. Figure 1 shows a typical multi-core cluster setup. In such a cluster, there exist several levels of communication, namely intra-CMP, inter-CMP, and inter-node communication, depending on the positions of the processes. Both intra-CMP and inter-CMP belong to intra-node communication. Many MPI libraries implement the different levels of communication in a multi-channel way so that they can take advantage of features of each channel and get optimal performance. For example, MVAPICH2 uses a network channel and a shared memory channel for inter- and intra-node communication respectively. The shared memory channel optimizes intra-node communication performance on multi-core clusters [8]. Further, MVAPICH2 employs multi-core aware collective algorithms which utilizes the shared memory channel to optimize the performance of MPI collective operations [9].

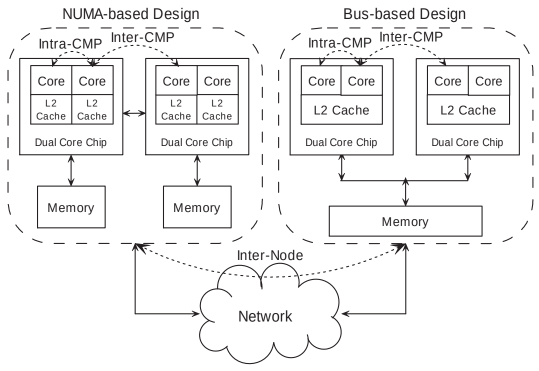


Fig. 1. Communication in Multi-core clusters [10]

The MPI libraries with CR support mentioned earlier, while address the fault tolerance issues, will not be the best solutions on multi-core clusters. The main concern is regarding the performance. The aforementioned work focuses on the network channel only, and uses network loopback for intra-node communication. While the speed of network is advancing rapidly, it is still much slower than the memory speed. The consequence is that applications performance will get hurt with CR support enabled. Therefore, the challenge is:

- **Can we design an efficient framework that is able to checkpoint multiple channels in MPI and achieve best performance for applications?**

In this paper, we take on the challenge and design such a framework that is able to checkpoint multiple channels in MPI and provides high performance. In addition, our design allows flexible process re-distribution after restart, i.e. processes previously on the same node can be restarted on different nodes or the other way around, and utilizes different channels dynamically. It can also be easily extended when new channels are added. We have integrated the design into MVAPICH2, and evaluated the performance on an Intel Clovertown multi-core cluster connected by InfiniBand. From the experimental results, we see that our design has little overhead compared with the native MVAPICH2 performance. Compared with the previous work, our design reduces intra-node latency from 4.55us to 0.76us, and increases peak bandwidth from 1510MB/s to 1954MB/s. It improves the performance of MPI collective operations by up to 90%, and application performance by up to 6%. Further, our design is efficient in taking the checkpoint.

The rest of the paper is organized as follows: We provide an overview of MPI checkpoint/restart in Section II. The overall and detailed design issues are illustrated in Section III and IV respectively. Section V presents the performance of our design and Section VI discusses related work. Finally, we conclude this paper and point out future work directions in Section VII.

II. Background

Checkpointing and rollback recovery are the most commonly used techniques for system-level failure recovery in distributed systems. While various other techniques using application-level checkpointing [11], [12], [13] exist, system-level solutions are still popular because of application transparency. A detailed comparison of various rollback recovery schemes can be found in [14]. As the *de facto* standard for parallel programming, MPI is the ideal place to integrate many system-level failure recovery mechanisms and hide the complexity from end user applications. Earlier studies have targeted both various checkpointing and rollback recovery schemes including both coordinated [6], [15] and uncoordinated checkpointing [16], [14]. In the context of modern cluster with high speed interconnects, coordinated checkpointing has its advantage. Uncoordinated checkpointing requires message logging, which could impose considerable overhead when network traffic can be produced in an extremely high rate. Furthermore, uncoordinated checkpointing is susceptible to the domino effect [17] where dependencies between processes may cause all processes to rollback to the initial state.

As part of our earlier work, we proposed coordinated checkpoint/restart for MVAPICH2, a MPI-2 library over InfiniBand. Figure 2 shows the architecture that we proposed. It consists of several components. A global Checkpoint/Restart (CR) manager, local CR managers, InfiniBand communication channel managers and CR library.

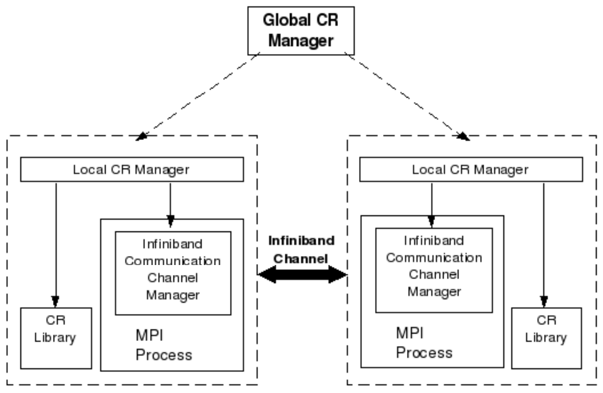


Fig. 2. MVAPICH2 C/R architecture over InfiniBand

When the user wants to take a checkpoint, a "checkpoint request" is generated by the global CR manager and is sent to the local CR managers which control the local MPI processes. The local CR managers inform the InfiniBand communication channel managers which is part of the MPI process. The InfiniBand communication channel manager then locks down the all the communication, including drain out and buffer all in-fly network messages and release the network resources. After that it gives a callback to the local CR manager to indicate that it is safe to take individual checkpoint. The local CR manager then invokes the CR library to checkpoint the processes. Our implementation uses Berkley Lab's Checkpoint Restart package (BLCR [18]) as the the CR library. Once the process state has been locally checkpointed, the local CR manager informs the InfiniBand communication channel manager to reactivate the network communication. Similarly, to restart an MPI job, a "restart request" is sent from the global CR manager to all local CR managers. Upon receiving the request, the local CR managers launch the local MPI process through BLCR and reactivates the network by notifying the InfiniBand communication manager. As it can be seen, apart of saving local process state, a very important step of coordinated checkpoint is to handle the communication channels to make sure all individual checkpoints are taken at a consistent point.

Similar to many other work on coordinated checkpoint/restart, including [6], our earlier solution picks a specific communication channel (InfiniBand in our case) and assumes all processes communicate only through this channel. This is not the case, however, for most state-of-the-art MPI solutions including both MVAPICH [5] and OpenMPI [19]. On modern multi-core clusters, an efficient MPI implementation will have at least two communication channels, an intra-node communication channel, which typically take advantage of shared memory communication, and an inter-node communication channel over high speed interconnects. Efficient intra-node communication is not only important for point to point communication, but can be exploited to design collective operations in a much more efficient manner [9]. Multiple communication channels, however, pose more challenges to handle communication during checkpoint/restart. We describe these additional design challenges and propose our solution in the next section.

III. A Framework for Checkpointing Multi-channel MPI

In this section we present a framework for checkpointing multi-channel MPI library. Our approach is based on the coordinated checkpoint protocols. In the following, We first describe several design challenges in addition to checkpoint a single communication channel MPI, and then we will present the overall framework.

A. Design Challenges

Although many open-source MPI library including [6], [15] provide checkpoint/restart support through coordinated protocols, hardly any of these of have addressed checkpointing MPI libraries which have multiple communication channels. To design an efficient checkpoint framework for multi-channel MPI, we need to achieve the following objectives:

- *Abstraction:* as we have described in Section II, suspending and resuming communication is one of the key aspects for coordinated checkpointing. In the case of multiple communication channels, every communication channel needs to be suspended and resumed separately. As a result, not only the code becomes extremely complex due to handling different channels, it also becomes hard to extend the checkpoint functionality if new communication channels are designed. Our aim is to design a clear abstraction between the general suspend/resume functionality, which takes care of the coordination among peer processes, and the channel specific plug-ins, which provides the necessary functionality to checkpoint/restart individual communication channel.
- *Process re-distribution:* checkpoint/restart protocols achieve fault tolerance by checkpointing the computing processes and restarting them on new computing nodes if the old ones are malfunctioned. In this case, it is possible that the process topology will be changed after restart. For example, processes located on the same nodes can be restarted on different ones. Processes communicate through shared memory, in this case, may have to use network after restart. Such process re-allocation poses additional challenges to checkpointing multi-channel MPI, whereas these design complexities may not be there if only one communication channel is used.
- *Collective Operations:* Collective operations optimized for multiple communication channel may also complicate the designs. For example, many collective operations can benefit from the more efficient intra-node shared memory communication [9] to minimize the inter-node traffic through network, achieving better performance. With these designs, however, collective operations are not simply built on top of point to point communication, but requires special coordination among peer processes. Thus, while many existing solutions can ignore additional complexities on collective operations, a multi-channel enabled design must carefully address such co-ordinations.

B. Design Overview

Our framework is illustrated in Figure 3.

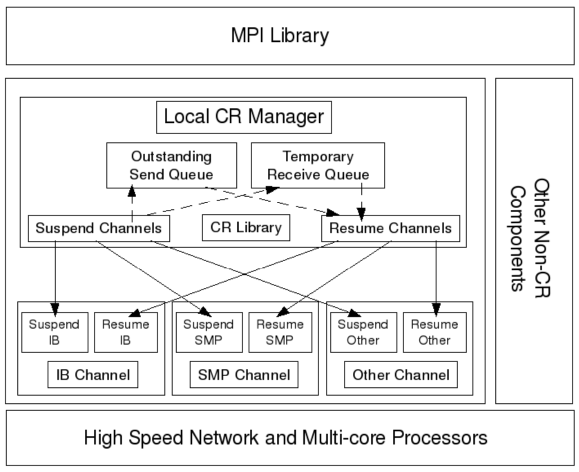


Fig. 3. Checkpoint/restart framework for multi-channel MPI

As it can be seen in the figure, we design a CR layer to perform all the general checkpoint/restart functionality. This

CR layer will take care of the coordination with other parts of the system. For example, the CR layer is responsible to receive the checkpoint request from the job manager, notify each communication layer to suspend/resume the traffic, decide on a consistent time to issue local checkpoint operations using CR library (BLCR in our case). Many of these functionalities have been discussed in the Section II. Additionally, the CR layer will keep track of the process location. For example, when two processes which are originally on the same node are restarted on different nodes, it is able to detect such topology changes and switch the communication channel from intra-node shared memory to network, and vice versa. To ensure in-order message delivery while switching the communication channels, the CR layer keeps two important queues, the outstanding send queue and temporarily receive queue. Once the checkpoint stage starts, all later send operations will be queued in the outstanding send queue. Meanwhile, all the channels drain in-fly messages and deliver them to the temporarily receive queue. In this way, no message is kept in any communication channel during the checkpoint. Thus, it is safe to switch communication channel after restart.

Every channel that needs to be suspended during the checkpoint stage will implement such functionality and provide suspend/resume hooks to the CR layer through a standard interface. The CR layer will simply use this interface when necessary without having to worry about design details specific to the channel. In the next section, we will address some detailed design issues to suspend and resume communication traffic.

IV. Detailed Design Issues

In this section, we will look at the detailed design of the checkpoint/restart solution for point-to-point and collective communication using the checkpoint/restart framework.

A. Point-to-point Communication

The new checkpoint/restart framework provides a method that enables every communication channel to register callback functions with the framework. The callback functions of each of the communication channels are invoked by the framework during the checkpointing process. The framework allows each channel to register two callback functions for the purpose of providing the ability to checkpoint point-to-point communication. The two functions, namely the "Suspend callback" function and the "Resume callback" function are discussed in greater detail in the following sections.

1) Suspend Callback function: The suspend callback function is invoked by the checkpoint/restart framework during checkpointing before the CR Library actually takes the checkpoint. It is the responsibility of this function to prepare the channel before a checkpoint. Since coordinated checkpointing is used, the suspend routine should ensure that no further sends are issued to this channel and that there are no messages in flight. This will ensure that all the processes are in a consistent state.

For the shared memory channel, the suspend callback function can be implemented to work in two phases.

Initial Synchronization Phase: In the initial synchronization phase, the callback function acquires a mutex that the main thread tries to acquire before every send operation. Once the callback function acquires the mutex, the main thread will not be able to issue any sends. Hence, the main thread is forced to wait on the mutex before proceeding with any communication.

Pre-checkpoint Coordination Phase: In the pre-checkpoint coordination phase, a "Suspend" control message is sent on the shared memory communication channel. Since the main thread can no longer perform sends and since the shared memory channel guarantees in order delivery of packets, reception of the Suspend message indicates that there are no pending messages on the communication channel. Hence, the channel can be marked as Suspended. Once all the processes have marked all their channels as suspended, the shared memory region that was allocated for the send/receive buffers are released.

At this point, the CR library can save the state of the process to a file on the disk in the Local checkpointing Phase.

2) Resume Callback function: The resume callback function is invoked by the checkpoint/restart framework during checkpointing after the CR library has taken the checkpoint, as well as during restart after the CR library has restored the previously checkpointed process from disk.

For the shared memory channel, the resume callback function re-initializes the shared memory region and other necessary data structures that were destroyed by the suspend callback function. Once the data structures are restored, a "Resume" control message is then sent on the Shared Memory Channels to mark them as Active. The callback function then releases the mutex that the suspend callback function had acquired. Once the mutex is released, the main thread that was waiting on it now acquires the mutex.

At this point, the state of the system is identical to what it was before the Suspend callback was invoked. Hence, communication on all the channels proceed normally.

B. Challenges While Checkpointing Collective Operations

In the previous sections, we have seen how the checkpoint/restart framework handles point-to-point communication. The same designs are not sufficient to gaurantee correctness of collective communication for the following reasons:

- Collective operations are invoked by multiple processes. Hence, group synchronization methods are required for coordination.
- Process skews can easily result in deadlocks as the checkpoint request may arrive at different phases across multiple processes making locking/unlocking complex to handle.

The rest of the section deals with the solution to these issues.

Synchronization and Consistency Mechanisms: It is imperative to guarantee consistency across all the processes taking part in the collective operation. This may not always be straighforward for collectives. For example, a typical shared memory collective operation consists of the following three phases:

- Intra-node communication via shared memory
- Inter-node communication via network point-to-point channels, and
- Intra-node communication for the final part of the algorithm

The point-to-point scheme described in the previous section is sufficient to ensure consistency of the processes when a checkpoint request arrives during the inter-node communication phase of all the processes. However, a different protocol is necessary when the request arrives during the intra-node communication phase of one of the processes. This is described in the following section.

Avoiding Deadlocks Unlike the point-to-point scheme, where the checkpoint/restart framework just invokes the callbacks registered by each channel, the collectives explicitly have to notify the framework when it is all right to proceed with the checkpointing, after the processes have been synchronized. Due to this two way communication between the collectives and the framework, there are posibilities of deadlock. Hence, special care has to be taken while designing the locking mechanism to avoid such scenarios.

The following section discusses the framework for checkpointing collective operations and the specifics of the implementation for the shared memory collectives.

C. Checkpointing Collective Operations

Figure 4 shows the overall operation of the collectives checkpointing. Due to the nature of the collective operations, the semantics of the callback functions necessary to implement checkpoint/restart are significantly different from those required for point-to-point communication. To be able to checkpoint collective operations, two callback functions, namely the "Request to Checkpoint callback" and the "Checkpoint Complete callback" are introduced. These functions are discussed in detail in the following sections.

1) Request to Checkpoint Callback function: The Request to Checkpoint callback function (RTC) is invoked by the local CR manager when a checkpoint is requested to be taken. This call notifies the collectives about the checkpoint request and returns immediately. Once the call returns, the local CR manager waits for a notification from the collectives indicating that they are ready to be checkpointed.

As discussed in the previous section, a different synchronization protocol is needed during the intra-node communication phase. We use the following protocol in our implementation.

The processes involved in the intra-node operation stop their communication and atomically increment a special field in the shared memory region. Each process continues to examine this field after incrementing it. Once the count becomes equal to the number of processes on local node, it indicates that all processes have stopped collective communication. At this point, all the local processes are in a consistent state. A leader is now chosen to copy the contents of the shared memory region to a local buffer and tear down the collectives' shared memory region. Once this is done, all the processes on the node call CTC and wait for checkpoint completion.

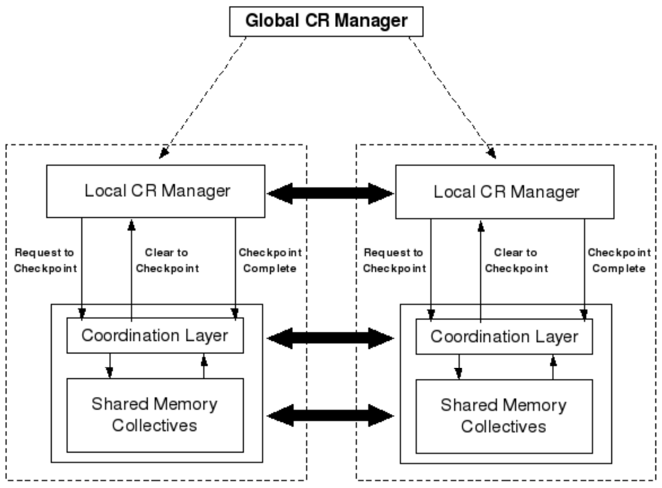


Fig. 4. Checkpointing protocol for Shared Memory Collectives

The notification is acheived thorough the "Clear to Checkpoint" function (CTC) whose pointer is passed down to the collectives during the RTC Call. The collective calls CTC when it is ready to be checkpointed. The checkpoint/restart framework proceeds with the checkpointing after receiving the CTC.

The checkpoint/restart framework now invokes the CR library to take a checkpoint.

2) Checkpoint Complete Callback function: Once the checkpoint has been taken, or when the processes have been restarted from a previously taken checkpoint, the framework invokes the Checkpoint Complete callback function (CC) to indicate that the checkpoint/restart operation has completed.

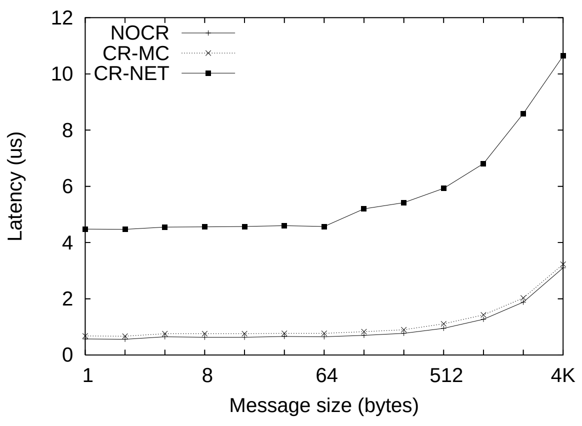
When CC is invoked, processes in a collective operation that were waiting for the checkpoint to complete are activated. The leader process initializes the collectives' shared memory region and restores the data structures that it had saved in its local buffer. Once the leader completes creating the shared memory region, all processes return from the callback.

At this point, the state of the system is identical to what it was before RTC was invoked. Hence, communication on all the channels proceed normally.

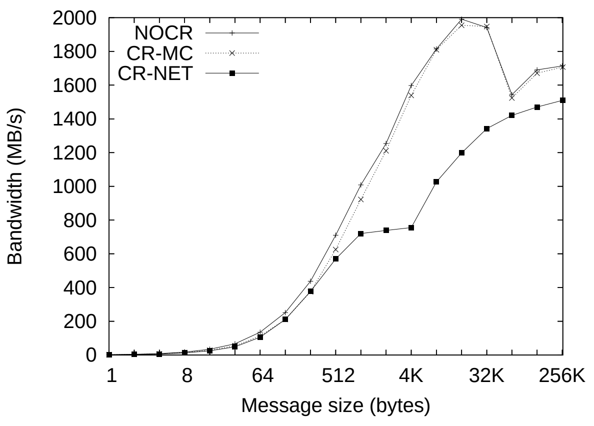
V. Performance Evaluation

In this section, we evaluate and analyze the performance of the proposed design using point-to-point, collective, and application level benchmarks. In Sections V-A, V-B, and V-C, the experiments were conducted without taking the checkpoints to examine the basic performance of the proposed design. Then we show checkpointing overhead and time breakdown in Section V-D, and process re-distribution effect in Section V-E.

Testbed: We use an Intel Clovertown cluster. Each node is equipped with dual quad-core Xeon processor, i.e. 8 cores per



(a) Latency



(b) Bandwidth

Fig. 5. Latency and Bandwidth

node, running at 2.0GHz. Each node has 4GB main memory. The nodes are connected by Mellanox InfiniBand DDR cards. The operating system is Linux 2.6.18 and the BLCR library used is version 0.6.5. The file system we use is ext3 on top of a local SATA disk.

In this section we will use the following acronyms for different settings:

- NOCR - No Checkpoint/Restart support
- CR-MC - Checkpoint/Restart support with Multi-Channel enabled (design proposed in this paper)
- CR-NET - Checkpoint/Restart support with only the Network channel enabled (design proposed in [6])

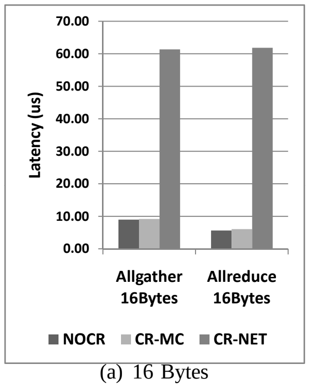
It is to be noted that the shared memory channel is available in both NOCR and CR-MC cases (both point-to-point and collectives), but not available in CR-NET.

A. Impact on Latency and Bandwidth

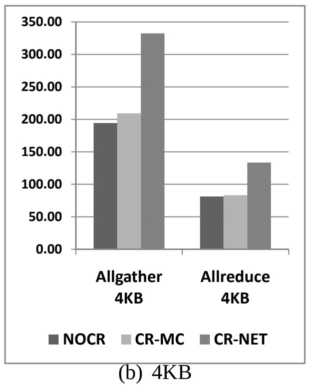
In this section, we examine the impact of the proposed CR-MC design on MPI intra-node latency and bandwidth. The results are shown in Figure 5.

From Figure 5(a) we can see that compared with CR-NET, CR-MC reduces latency significantly. The 4-byte message latency is reduced from 4.55us to 0.76us. This is because memory copy is much faster than network loopback. Similarly, Figure 5(b) shows that CR-MC increases bandwidth from 1510MB/s to 1954MB/s compared with CR-NET which is 30% improvement.

Comparing with NOCR, we observe that CR-MC adds little overhead, only around 0.1us on latency and almost no overhead on bandwidth. The overhead comes from acquiring and releasing CR locks. This indicates that with CR-MC, users can always run applications with CR support on and decide whether to take checkpoints at run time. If the applications do not take checkpoints, then the performance is not affected. On the other hand, if using CR-NET, the users need to make a decision whether to use CR support at compile time, because with CR-NET the performance may degrade even if no checkpoints are actually taken.



(a) 16 Bytes



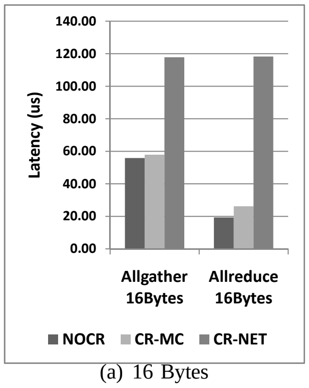
(b) 4KB

Fig. 6. Performance of MPI_Allgather and MPI_Allreduce on 8 Cores (1x8)

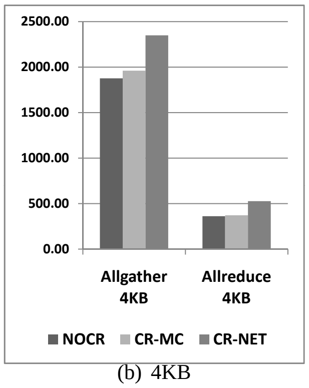
B. Impact on Collective Operations

In this section, we use IMB [20] to evaluate the performance impact of CR-MC on collective operations. The results of MPI_Allgather and MPI_Allreduce on 8 cores and 64 cores are shown in Figures 6 and 7, respectively. It is to be noted that MPI_Allgather in MVAPICH2 is implemented on top of point-to-point communication. NOCR and CR-MC can exploit the faster intra-node point-to-point communication for MPI_Allgather while CR-NET cannot. MPI_Allreduce in MVAPICH2 uses the special shared memory aware algorithm which has optimized performance and is available in NOCR and CR-MC, but not in CR-NET.

From Figure 6 we see that on a single node, CR-MC improves MPI_Allgather latency over CR-NET by 85% and 37% for 16 byte and 4KB messages, respectively. The corresponding improvements for MPI_Allreduce are 90% and 37%. On 64 cores, CR-MC improves MPI_Allgather latency by 50% and 16% for 16 byte and 4KB messages, respectively, and improves MPI_Allreduce latency by 52% and 29% (Figure 7). In all cases, CR-MC and NOCR perform comparably.



(a) 16 Bytes



(b) 4KB

Fig. 7. Performance of MPI_Allgather and MPI_Allreduce on 64 Cores (8x8)

C. Impact on Application Performance

In this section, we evaluate the performance of CR-MC using application level benchmarks, NAS [21], and compare with NOCR and CR-NET. The normalized execution time on 8 cores and 64 cores are shown in Figures 8 and 9, respectively. From the figures we can see the improvements in latency, bandwidth, and collective operations have been translated into applications. With CR-MC the execution time is reduced by up to 6% compared with CR-NET, which indicates that users can have both CR support and high performance at the same time by using the CR-MC design.

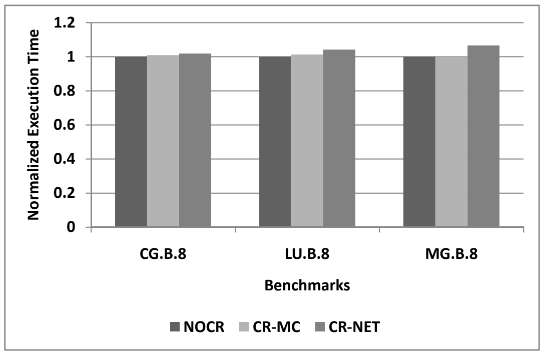


Fig. 8. Performance Impact of Checkpointing NAS on 8 Cores (1x8)

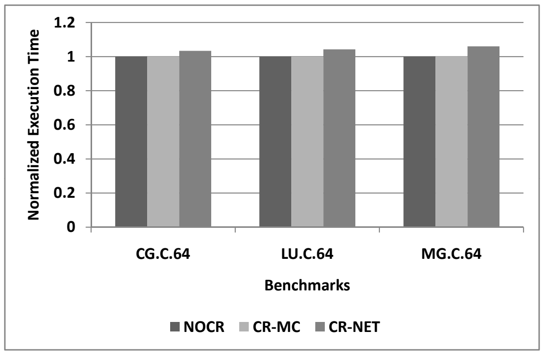


Fig. 9. Performance Impact of Checkpointing NAS on 64 Cores (8x8)

D. Checkpointing Overhead

In this section, we use HPL [22] to measure the checkpointing overhead. We run HPL on a single node with 8 processes, and take 1, 2, 4 checkpoints, respectively in 3 runs. The results are shown in Figure 10. From the results we see that the overhead of taking one checkpoint is around 2 seconds, which is less than 1% compared with the total execution time. The checkpoint file size is shown in Table I.

TABLE I
HPL CHECKPOINT FILE SIZE

Number of Checkpoints	File Size (MB)
1	1435
2	2897
4	5823

The checkpointing time breakdown is shown in Figure 11. There are two parts involved, coordination time and file writing time. We can see that our implementation of coordination is very efficient that it only takes a small percentage in the total checkpointing time. The file writing time is the dominant factor. As the number of checkpoints increases, the checkpointing time increases linearly.

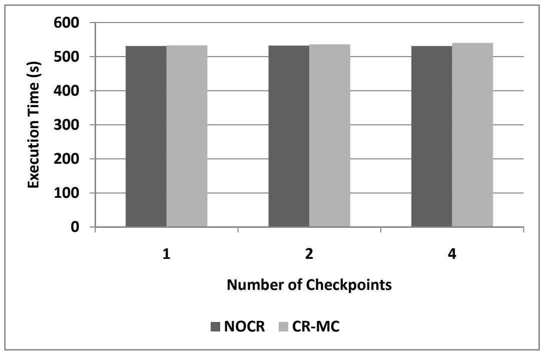


Fig. 10. Performance Impact of Checkpointing HPL

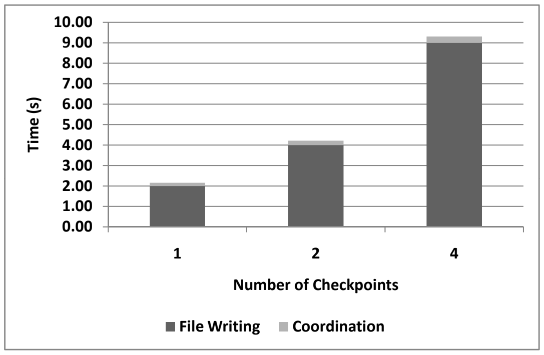


Fig. 11. Checkpointing Overhead Breakdown (HPL)

The file writing time can be reduced by using good parallel file systems (such as Lustre, PVFS2, etc) and high performance storage nodes. Since the focus of this paper is not to optimize the file writing time with parallel file systems, these results are not included here.

E. Process Re-distribution

In this section, we present the results of process re-distribution. When faults occur and applications need to restart from the previous checkpoint, the processes may need to be re-distributed, e.g. processes previously on different nodes may be re-distributed on the same node or the other way around. This is because the original distribution may not be available after restart. In this set of experiments, we run MPI latency and bandwidth tests with two processes on two different nodes initially, and after 5 iterations we take a checkpoint, and restart the processes on the same node. So starting from the 6th iteration, the processes get re-distributed. The message size is 2KB. The results are shown in Figure 12. From the results we see that our design allows process re-distribution and utilizes different channels dynamically.

VI. Related Work

Fault tolerance issues in MPI programs become a very popular topic recently. There are many research efforts in this area. Some of these efforts are aimed at tolerate network faults, including LA-MPI [2], which enables data reliability checking and uses multiple network devices. Specific to MPI applications, researchers have proposed multiple schemes to achieve fault tolerance at application level, including the work from FT-MPI [23] and from Schulz et al. [11]. Besides these effort, a lot of work has been proposed for library based application transparent checkpoint. For example, the MPICH-V team [3] has developed and/or evaluated several roll-back recovery protocols. Their work include both uncoordinated checkpointing with message-logging protocols [16], [24], [25], and coordinated checkpointing protocols, such as Vcl [26], which is based on Chandy-Lamport Algorithm [27], and Pcl [28] based on the blocking coordinated checkpointing protocol. BLCR is a library that provides single process-level checkpoint/restart [29]. Many MPI libraries build their checkpoint functionality on BLCR, including LAM/MPI [15] and the more recent job pause service [30], which is based on LAM/MPI's checkpointing framework and achieve very efficient failure recovery through process migration. In our earlier work [6], we have proposed a framework to checkpoint MPI programs over InfiniBand using a blocking coordinated checkpointing protocol and BLCR.

Most of the work described above assume single communication channel. They almost exclusively work on TCP/IP based MPI except our earlier work is on InfiniBand. In reality, however, modern clusters are deployed with multi-core computing nodes connected through high speed interconnects like InfiniBand. Thus, it is important to design a checkpoint framework that is able to handle multiple communication channels including both intra- and inter-node communication. Our work proposed in this paper exactly addresses this needs by proposing the multi-channel enabled checkpoint/restart. To the best of our knowledge, this is the first MPI that provides the checkpoint/restart functionality on multi-core InfiniBand clusters.

VII. Conclusions and Future Work

In this paper, we have designed an efficient framework that is able to checkpoint multiple channels in MPI and provides high performance. In addition, our design allows flexible process re-distribution after restart, i.e. processes previously on the same node can be restarted on different nodes or the other way around, and utilizes different channels dynamically. It can also be easily extended when new channels are added. We have integrated the design into MVAPICH2, and evaluated the performance on an Intel Clovertown multi-core cluster connected by InfiniBand. From the experimental results, we see that our design has little overhead compared with the native MVAPICH2 performance. Compared with the previous work, our design reduces intra-node latency from 4.55us to 0.76us, and increases peak bandwidth from 1510MB/s to 1954MB/s. It improves the performance of MPI collective operations by up to 90%, and application performance by up to 6%. Further, our design is efficient in taking the checkpoint.

In the future, we plan to study on efficient checkpoint/restart for MPI one-sided operations. We also would like to carry out a comprehensive study on the performance and scalability of our design on large-scale clusters with real world applications and parallel file systems such as Lustre.

Software Distribution: *The design proposed in this paper will be available for downloading in upcoming MVAPICH2 releases.*

References

[1] Message Passing Interface Forum, "MPI: A Message-Passing Interface standard," *The International Journal of Supercomputer Applications and High Performance Computing*, 1994.

[2] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, and M. W. S. M. A. Taylor, T. S. Woodall, "Architecture of la-mpi, a network-fault-tolerant mpi," in *Proceedings of Int'l Parallel and Distributed Processing Symposium*, Santa Fe, NM, April 2004.

[3] "MPICH-V Project," http://mpich-v.lri.fr.

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

[5] Network-Based Computing Laboratory, "MVAPICH: MPI for InfiniBand," http://mvapich.cse.ohio-state.edu/projects/mpi-iba/.

[6] Q. Gao, W. Yu, W. Huang, and D. K. Panda, "Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand," in *Proceedings of Int'l Conference on Parallel Processing (ICPP)*, August 2006.

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

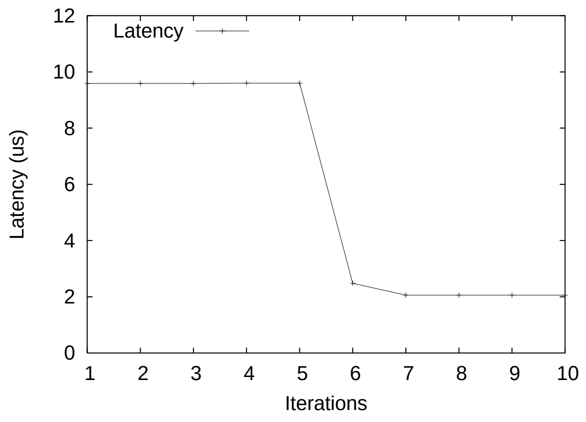
[8] L. Chai, A. Hartono, and D. K. Panda, "Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters," in *The IEEE International Conference on Cluster Computing*, 2006.

[9] A. Mamidala, R. Kumar, D. Ded, and D. K. Panda, "MPI Collectives on modern Multicore clusters: Performance Optimizations and Communication Characteristics," in *International Symposium on Cluster Computing and the Grid (CCGrid)*, 2008.

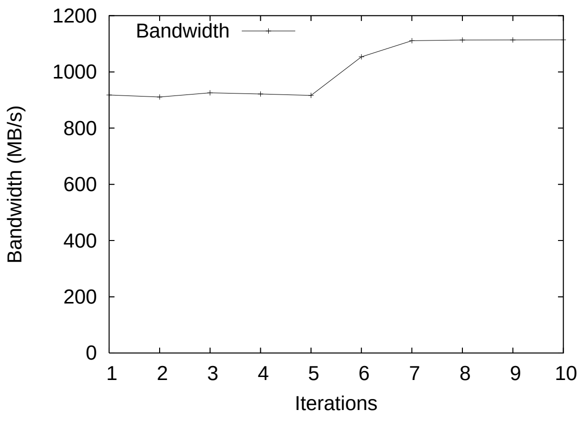
[10] L. Chai, Q. Gao, and D. K. Panda, "Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System," in *Proceedings of IEEE International Sympsoium on Cluster Computing and the Grid*, 2007.

[11] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pengali, and P. Stodghill, "Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004.

[12] K. P. Greg Bronevetsky, Daniel Marques and P. Stodghill, "Automated application-level checkpointing of mpi programs," in *Proceedings of the 2003 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, July 2003.



(a) Latency



(b) Bandwidth

Fig. 12. Process Re-distribution

[13] K. P. Greg Bronevetsky and P. Stodghill, "Application-level checkpointing for openmp programs," in *International Conference on Supercomputing (ICS)*, 2006.

[14] 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.*, vol. 34, no. 3, 2002.

[15] 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*, pp. 479–493, 2005.

[16] 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.

[17] B. Randell, "Systems structure for software fault tolerance," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 220–232, 1975.

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

[19] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[20] "Intel Cluster Toolkit 3.1," http://www.intel.com/cd/software/products/asmo-na/eng/cluster/clustertoolkit/219848.htm.

[21] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler, "Architectural Requirements and Scalability of the NAS Parallel Benchmarks," in *ACM/IEEE Supercomputing*, 1999.

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

[23] 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 *Proceeding of International Supercomputer Conference (ICS)*, Heidelberg, Germany, 2003.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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 *Proceedings of ACM/IEEE SC'2006*, Tampa, FL, 11 2006.

[29] J. Duell, P. Hargrove, and E. Roman, "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart," Berkeley Lab, Tech. Rep. LBNL-54941, 2002.

[30] 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.