RDMA-Based Job Migration Framework for MPI over InfiniBand

Xiangyong Ouyang, Sonya Marcarelli, Raghunath Rajachandrasekar,
and Dhabaleswar K. Panda
Department of Computer Science and Engineering,
The Ohio State University
{ouyangx, smarcare, rajachan, panda}@cse.ohio-state.edu

***Abstract*—Coordinated checkpoint and recovery is a common approach to achieve fault tolerance on large-scale systems. The traditional mechanism dumps the process image to a local disk or a central storage area of all the processes involved in the parallel job. When a failure occurs, the processes are restarted and restored to the latest checkpoint image. However, this kind of approach is unable to provide the scalability required by increasingly large-sized jobs, since it puts heavy I/O burden on the storage subsystem, and resubmitting a job during restart phase incurs lengthy queuing delay.**

In this paper, we enhance the fault tolerance of MVAPICH2 [1], an open-source high performance MPI-2 implementation, by using a proactive job migration scheme. Instead of checkpointing all the processes of the job and saving their process images to a stable storage, we transfer the processes running on a health-deteriorating node to a healthy spare node, and resume these processes from the spare node. RDMA-based process image transmission is designed to take advantage of high performance communication in InfiniBand. Experimental results show that the Job Migration scheme can achieve a speedup of 4.49 times over the Checkpoint/Restart scheme to handle a node failure for a 64-process application running on 8 compute nodes. To the best of our knowledge, this is the first such job migration design for InfiniBand-based clusters.

I. Introduction

In the last several years, the High Performance Computer Clusters are continuously growing in terms of scale and complexity. There has been an exponential increase in the number of components in the cluster. As a consequence, failures can be more frequent and can stop an entire execution, causing the restarting of the entire application. In this context, Fault-Tolerance becomes an extremely important requirement. MPI is the most popular programming model on a distributed memory system. The current MPI standard does not indicate any specifications to ensure fault-tolerance. However, MPI Forum is currently discussing about such fault-tolerance specifications in the upcoming MPI-3 standard [2].

Checkpoint/Restart is a common practice to guarantee Fault-Tolerance for large scale applications. A typical Checkpoint/Restart mechanism saves a snapshot of the current state of the job to a global shared file system, which is later used to recover the application from a failure by rolling back the entire application to the previous checkpoint. A lot of work has been done to understand the most convenient interval time to take checkpoint in order to minimize the checkpointing overhead [3], [4]. However, how to apply Checkpoint/Restart to large-scale systems still remains a grand challenge. Most of current implementations handle faults in a reactive manner, i.e., if one of compute nodes encounters a failure, all processes of the job must be restarted from their most recent saved state. This implies two major drawbacks:

- First, every process must be checkpointed at certain intervals, saving their memory snapshot to some stable storage to be used in a possible future restart. This results in huge data volume being dumped to a (local or global) file system, which becomes a bottleneck for the overall performance of the application.
- Second, the entire application has to be aborted even if only one node fails. This application is then re-submitted to the job scheduler to go through the lengthy queuing latency. As a consequence, the throughput of the computer cluster as a whole degrades significantly.

The key insight to the above problem is that a checkpoint is only needed on the faulty node in order to recover the processes running on it. Other processes on healthy nodes can be paused and wait while the processes on the failing node are being migrated to a spare node. The assumption here is the availability of some spare nodes, which has become a common practice in node allocation at large clusters.

In this paper, we explore a solution that handles node failures by migrating processes from a health-deteriorating node to a healthy spare node, while retaining execution of the application as a whole. A migration can be triggered by an abnormal event of system health status such as reported by IPMI [5] or other failure prediction models [6], [7]. Our design also enables direct user intervention to trigger a migration, such as for load-balancing or system maintenance purposes.

Once a migration is triggered, coordinated actions are taken by all processes to reach a consistent global state. After arriving at such a state, processes at the failing node are migrated to a spare node, while processes at healthy nodes are blocked and wait for the migration to complete. Once the processes are restarted at the spare node, all processes synchronize to resume execution. In order to reduce the overhead to move large process images during migration, we have designed a RDMA-based process migration approach that takes advantage of InfiniBand [8], a high performance communication interconnect.

Similar studies exist, such as a proactive process-level live migration mechanism proposed by Wang et. al. [9], [10] for LAM/MPI over on TCP/IP. Unlike their work, our design is optimized for InfiniBand architecture, which puts different requirements and constraints on process migration. The authors in [11], [12] explore the Virtual Machine migration as an alternative to process-level migration. However one inherent limitation of such a strategy is the in-distinguished transfer of entire virtual machine memory. On the contrary, process migration strategies, presented in this paper and in [9], [10], only transfer the memory content actually used by the processes.

Including the aforementioned strategies, we have implemented a transparent and automatic Job Migration Framework for MVAPICH2, a high performance implementation of MPI-2. Experiments show that our RDMA-based Job Migration strategy causes only marginal (3.9% to 6.7%) overhead on the total execution duration for 64-process applications running on 8 compute nodes. Our approach significantly reduces the amount of data that need to be dumped by a Checkpoint/Restart strategy, so as to relieve the I/O bottleneck. Compared to the traditional Checkpoint/Restart strategy, our approach reduces the time to handle a node failure from 28.3 seconds to 6.3 seconds, a speed up of 4.49 times.

Provided the capability to detect/predict a subset of imminent failures by current techniques [5]–[7], our approach can cope with them by proactively migrating processes away from those affected nodes. As a result, our approach has the potential to benefit the existing Checkpoint/Restart strategy by prolonging the interval between full job-wide checkpoints. We plan to explore along this direction in our future studies.

In a brief summary, our contribution in this paper is as follows:

- We have designed and implemented a Job-Migration based Fault Tolerance Framework for MVAPICH2 as a complement to the existing Checkpoint/Restart Framework [13]–[16]. With this framework, the MPI job remains alive while the processes on the involved nodes are being migrated, and a complete dump of the entire job is avoided.
- We have designed a RDMA-based process migration strategy, which significantly reduces the overhead to migrate process images from the failing node to the spare node. This strategy checkpoints processes on a failing node using BLCR [17], aggregates the memory snapshots from multiple processes, and transfers them on-the-fly to a spare node using RDMA, where the processes are restarted.
- We have adopted FTB (Fault Tolerance Backplane) [18], [19] into this Job Migration Framework as a communication infrastructure for all the components to exchange fault-related messages during a migration. As can be seen in later sections, FTB proves to be very useful in such a scenario.
- We have conducted comprehensive performance evaluation of the proposed design with a state-of-the-art InfiniBand cluster.

The paper is organized as follows. In section II, we give a background about the key components involved in our design. Then in section III, we introduce our Design and Implementation. In section IV, we present our experiments and evaluation. Related work is discussed in Section V, and in section VI, we discuss the conclusion and future work.

II. Background

A. InfiniBand and MVAPICH2

InfiniBand is an open standard of high speed interconnect, which provides send/receive semantics, and memory-based semantics called Remote Direct Memory Access (RDMA) [8]. RDMA operations allow a node to directly access a remote node's memory contents without using the CPU at the remote side. These operations are transparent at the remote end since they do not involve the remote CPU in the communication. InfiniBand empowers many of today's Top500 Super Computers [20].

MVAPICH2 [1] is an open source MPI-2 implementation using InfiniBand, iWARP and other RDMA-enabled interconnect networking technologies. MVAPICH2 is being used by more than 1,185 organizations world-wide. MVAPICH2 supports application initiated and system initiated coordinated Checkpoint and Restart (CR) [13], [14] using the BLCR Library for Checkpoint/Restart [17], [21]. During a CR cycle, MVAPICH2 drains the communication channels of all pending messages, uses the BLCR Library to independently request the checkpoint of every process that is part of the MPI job, and re-establishes the communication endpoints on every process during restart. After the checkpoint is taken the application continues its execution.

B. *CIFTS and FTB*

The Coordinated Infrastructure for Fault Tolerant Systems (CIFTS) [18], [19] is an asynchronous messaging backplane that provides an environment and infrastructure for sharing fault-related information. The FTB (Fault Tolerance Backplane) is the backbone of this CIFTS environment. Its physical infrastructure is based on a distributed architecture with a set of distributed daemons, named FTB agents. These agents connect to each other to form a fault-tolerant and self-healing tree-based topology. If an agent loses connectivity during its lifetime, it can reconnect itself to a new parent in the topology tree.

The FTB Software Stack consists of three layers: the Client Layer, the Manager Layer, and the Network Layer. The Client Layer comprises a set of APIs that the clients use to interact with each other and to connect to the FTB framework. Once connected, a client can publish/receive fault-related messages to be communicated throughout the FTB system. The Manager Layer handles the bookkeeping and decision making logic. It handles the client subscriptions, subscription mechanisms and event notification criteria. It's also responsible for event matching and routing events across to other FTB Agents. The Network Layer deals with the sending and receiving of data. The Network Layer is transparent to the upper layers and is designed to support multiple communication protocols such as TCP/IP and shared-memory communication.

C. *Berkeley Lab Checkpoint/Restart (BLCR)*

Berkeley Lab Checkpoint/Restart (BLCR) [21] allows programs running on Linux systems to be checkpointed by writing the process image to a file and then later to be restarted from the saved process image file. BLCR by itself can only checkpoint/restart processes on a single node. But it provides callbacks to be extended by applications, so that a parallel application can also be checkpointed.

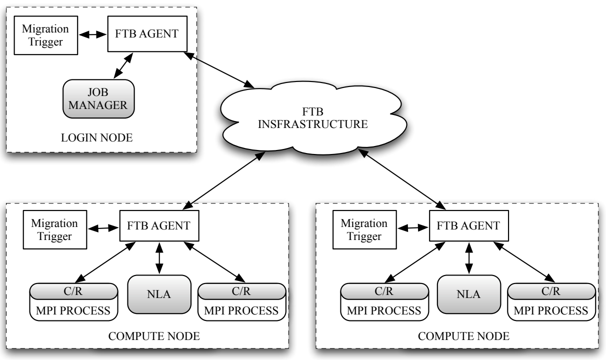


Fig. 1. Job Migration Framework for MVAPICH2

III. Design and Implementation

The architecture of the proposed Job Migration framework for MVAPICH2 is illustrated in Figure 1. An FTB agent is deployed in each node to connect with other agents to form the communication framework (FTB backplane). Aided with this backplane, fault-tolerance related messages are delivered throughout the framework. A Migration Trigger fires events that initiate a migration, either upon a user request, or at the detection of system abnormal status by some health monitoring component, such as IPMI [5]. In our design, MVAPICH2 has three principal components (dark boxes in Figure 1) that subscribe to the FTB-agent to publish/receive messages.

- *Job Manager.* During the startup, the Job Manager launches the Node Launch Agents (NLA) on the primary compute nodes as well as on the spare nodes. During a migration, the Job Manager acts as a coordinator to orchestrate the actions of other components in the framework.
- *Node Launch Agent (NLA).* NLA combines with Job Manager to form a hierarchical scalable job launch architecture [22]. It is responsible for starting/terminating the application processes at local node. We have extended NLA to support restarting migrated processes on a spare node as necessary.
- *Checkpoint/Restart (C/R) Thread within each MPI Process.* MVAPICH2 library provides a Checkpoint/Restart (C/R) thread for each MPI process. By calling BLCR [17], the C/R thread takes the responsibility to checkpoint a MPI process at migration source node, and restart the process at the target node.

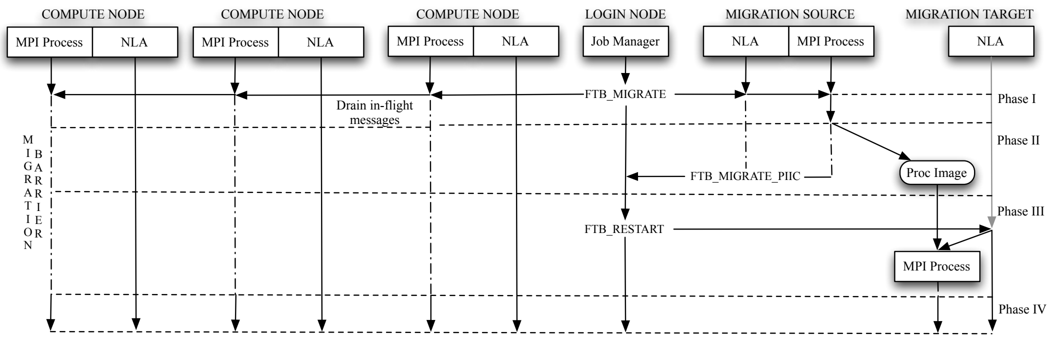


Fig. 2. Sequence Diagram of Process Migration

A. *Proposed Job Migration Procedure in MVAPICH2*

A migration can be triggered either by a user request, or by a health-deteriorating event reported by a health-monitoring component. Figure 2 depicts the interactions between different components during a migration. The migration cycle can be divided into four phases.

Phase 1: Job Stall. At the startup time, the Job Manager is supplied with a list of spare nodes in addition to the primary compute nodes required to run a parallel job. During the startup phase, the Job Manager launches the Node Launch Agents (NLA) on the spare nodes in addition to the primary host nodes, and it subscribes to FTB backplane for all fault tolerance related messages. The NLAs on all the primary nodes are in the “MIGRATION_READY” state, while those on the Hot Spare Nodes are in the “MIGRATION_SPARE” state.

Once a migration is triggered, the Job Manager determines the migration source and the target nodes and initiates process migration by publishing an “FTB_MIGRATE” message including the names of these nodes. This message is received by all the NLAs and all MPI Processes. On receiving this message, all MPI processes suspend their MPI communication activities, drain all inflight MPI messages, and tear down their communication end-points. This step is necessary for all MPI processes to achieve a consistent global state in order to checkpoint individual MPI processes, due to the characteristics native to InfiniBand network.

- First, InfiniBand provides its high performance communication via an OS-bypass user-level protocol. The OS is skipped in the actual communication and does not own complete knowledge of ongoing network activities. Therefore it becomes very difficult for the OS to directly stop network activities without losing global consistency.
- Second, unlike regular TCP/IP which stores its communication context within kernel memory, the network connection context for many InfiniBand implementations is only available in network adaptor cache. Therefore network connection context has to be released before checkpoint, and rebuilt afterwards.
- Third, some InfiniBand network connection context is even cached on remote nodes, such as RDMA remote key, in order to achieve high performance. These cached resources must be released before checkpoint. Otherwise potential inconsistency will be introduced since these keys become invalid when network connection context is rebuilt at restart.

Phase 2: Job Migration. By the end of Phase 1, all processes have suspended their MPI communication activities and released their communication channels. At this instant, they have reached a consistent global state, and Phase 2 begins. The MPI processes not on the migration source node enter a migration barrier and remain stalled. Meanwhile, the processes running on the migration source node are checkpointed using BLCR [21], and their process images are migrated to the target node. We have implemented a RDMA-based process migration strategy with extensions to BLCR library. Our strategy takes the advantages of InfiniBand network to achieve a low-overhead and low-latency process migration. This strategy is elaborated in Section III-B. Once all the process states have been migrated to the target node, the NLA on the source node propagates a “FTB_MIGRATE_PIIC” message marking the end of Phase 2. After that, the NLA on source node transits to the “MIGRATION_INACTIVE” state, which indicates that the NLA’s node is no longer active.

Phase 3: Restart on Spare Node. On receiving the “FTB_MIGRATE_PIIC” message, the Job Manager adjusts the mpispawn tree structure to accommodate the topology changes caused by migrating some MPI_processes to the new node. It then broadcasts a "FTB_RESTART" message, with the payload containing the target hostname and list of process ranks that have been migrated to the target node.

When the "FTB_RESTART" message arrives at the target node, its Node Launch Agent extracts the MPI process rank information from the payload, and restarts these MPI processes from the checkpoint images which have been migrated to the target node in Phase 2. After that, the NLA on the Migration Target node changes its status from "MIGRATION_SPARE" to "MIGRATION_READY", to indicate that it is now active.

Phase 4: Resume. Once the MPI processes have been restarted on the target node, they enter the migration barrier. At this point, all MPI processes are in the migration barrier. As a result, all the processes are now synchronized and are free to exit the migration barrier. Once out of the migration barrier, all the processes reestablish their communication end-points and resume their MPI communication activity. The Job Migration cycle is now complete and is ready for the next cycle.

B. *RDMA-based Process Migration*

At Phase 2, we migrate process images from the source node to the target node. The process images are acquired by using BLCR library to checkpoint individual process on the source node. In a naive strategy, process migration can be implemented by using BLCR to checkpoint a process to a local file, copying the file to the target node, and restarting the process from the checkpoint file. However, this approach incurs significant overhead to store the memory snapshot of a process image to a file and move this file to target node. Although the checkpoint files can be stored to a global file system, the file system related data copy overhead is not mitigated. When multiple processes on the same node are checkpointed, the conflicts between concurrent write streams can cause severe performance degradation on the global file system, as is demonstrated in [23]. Additionally, storing checkpoint data to a global file system has the drawback to interfere with other applications sharing the same file system.

The Process Live Migration mechanism [9] is able to copy memory snapshot of a process to another node with no involvement of file system overhead. This mechanism is implemented in TCP/IP network where BLCR treats a TCP socket as a file descriptor for output/input at the source/target node, therefore it is subject to the heavy memory-copy overhead through the TCP/IP protocol stack. Although InfiniBand provides a socket abstraction through IPoIB, it can only achieve an suboptimal performance because it still follows the memory-copy based socket protocol and cannot take full advantage of InfiniBand's zero-copy RDMA mechanism.

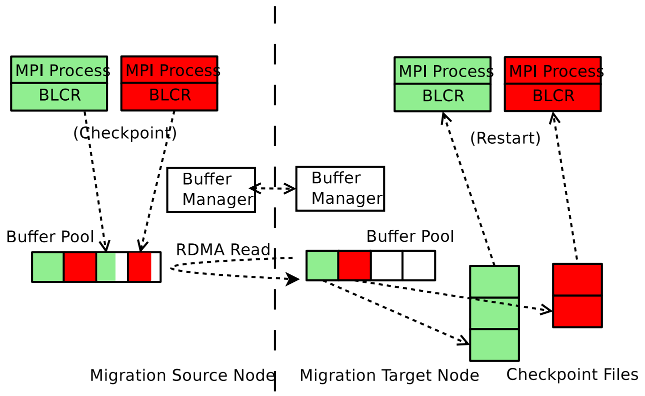


Fig. 3. RDMA-based Process Migration

In this section we propose a RDMA-based Process Migration strategy which exploits InfiniBand features to achieve high-bandwidth checkpoint data transfer. At migration source node, our strategy alters the BLCR library to aggregate checkpoint writes from multiple MPI processes into a local buffer pool, with each chunk containing data from one process. The target node performs RDMA Read to pull over big chunks of data, and rebuild checkpoint files for different processes to restart from. The basic design strategy is illustrated in Figure 3.

On the source node, a user-level buffer manager prepares a buffer pool which is mapped into kernel space by BLCR when a checkpoint is initiated in Phase 2. The buffer manager runs in user space because it is more flexible to allocate and manage big buffers from user space. During checkpointing, when a process running BLCR code in kernel space has some checkpoint data to be saved, it requires for a buffer chunk from the buffer manager. If a buffer is available in the pool, it is assigned to the process. All its checkpoint data is then saved in the chunk until it is filled up. When the chunk is full, it is returned to the buffer pool and the next free chunk is fetched. Whenever a filled chunk is returned to the buffer pool, the buffer manager on the source node sends a RDMA-Read request to the buffer manager on the target node. This request contains two types of information: (1) RDMA information for the target buffer manager to perform a RDMA Read to pull over the data, and (2) the information (such as process rank, data size, offset of the data) based on which the chunks belonging to the same process can be concatenated into a complete checkpoint file. If a free chunk is available in the target node buffer pool, a RDMA Read is performed to pull the chunk to the target node, and the chunk is concatenated to the proper position in a checkpoint file. After that, the target buffer manager sends a RDMA-Read reply telling the source buffer manager to release a buffer chunk. By the time all processes on the source node have been checkpointed, their process images are ready in the target node to be used in Phase 3 to restart the migrated processes.

IV. Performance Evaluation

We have implemented the Job Migration framework into MVAPICH2 with extensions to BLCR. In this section, we conduct various experiments to evaluate the performance of our design, with respect to: (a) Overhead caused by process migration; (b) Scalability of the Job Migration framework at different problem sizes; and (c) Comparison between Job Migration strategy and Checkpoint/Restart strategy. In our experiment, we simulate the migration trigger by firing a user signal to the Job Manager. Other mechanism, such as node health monitoring events, can also be used to kick off a migration.

An InfiniBand Linux cluster is used in the evaluation. In this cluster each node has eight processor cores on two Intel Xeon 2.33 GHz Quad-core CPUs. The nodes are connected with Mellanox MT25208 DDR InfiniBand HCAs for high performance MPI communication and process migration. In addition to InfiniBand, they are also connected with a GigE network for maintenance purposes, over which the Fault Tolerance Backplane runs. Each node runs RedHat Enterprise 5. All experiments use MVAPICH2 1.4 as the MPI library with extended BLCR 0.8.0. NAS parallel benchmark (NPB) [24] suite version 3.2 is run to obtain the results. We choose LU/BT/SP applications of class C out of the NPB benchmark suite, because these applications run for a sufficiently long duration and generate significant amount of data in one migration. For each application, 64 processes are run on eight compute nodes, with eight processes per node. Another compute node is set aside to be used as the migration target.

A. Migration Overhead

In the first set of experiments, we measured the overhead incurred by a migration. Figure 4 shows the time cost for a complete migration cycle, from the instant when the migration is triggered, till all application processes resume execution. The time cost is decomposed into four phases as described in section III-A: Phase 1 (Job Stall), Phase 2 (Job Migration), Phase 3 (Restart) and Phase 4 (Resume).

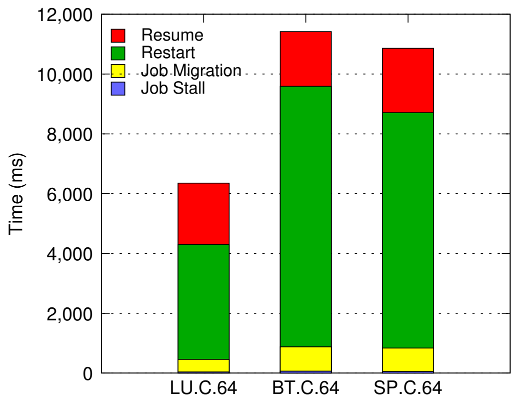


Fig. 4. Process Migration Overhead

It can be observed that Phase 1 (Job Stall) is very swift which completes in tens of milliseconds. With the RDMA-based process migration design, Phase 2 (Job Migration) finishes in 0.4-0.8 seconds, depending on the process image size to be migrated (which is shown in Table I). The cost is dominated by Phase 3 when the migrated processes are restarted on the spare node. In our current design, the process images are stored into temporary checkpoint files on the target node, and BLCR restarts the processes by loading their images from these files. Longer delay is incurred due to the file I/O latency. As a next step, we plan to revise the restart strategy on the target node by restarting the processes on-the-fly as the process image data arrives at the buffer pool (as shown in Figure 3). In our test, we fix the buffer pool to be 10 MB with chunk size of 1 MB. We find that the process-migration overhead does not vary significantly as buffer pool size changes, because it is dominated by Phase 3 where the processes are rebuilt by reading temporary checkpoint files. Therefore we stick to 10 MB buffer pool and 1 MB chunk size in all the experiments. In Phase 4, all the MPI processes reestablish their communication end-points and resume execution. For a given task scale, the cost in phase 4 is relatively constant.

We have also assessed the application execution time without any migration, and with one migration. As shown in Figure 5, the execution time with 1 migration exceeds the base run by 3.9% for LU.C.64. The execution time is prolonged by 6.7% and 4.6% for BT.C.64 and SP.C.64, respectively. With even longer-running applications, the overhead caused by migration is going to be less noticeable. It is worth noting that, much of this migration overhead is caused by the file-based restart scheme at the spare node. Our future work of memory-based restart scheme will drive down this cost.

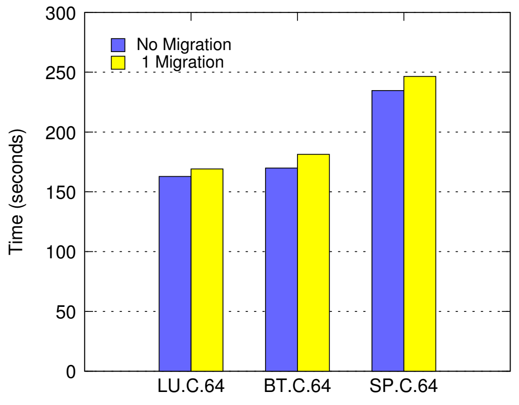


Fig. 5. Application Execution Time with/without Migration

B. Migration Scalability

We have also evaluated the scalability of the Job Migration framework with varied task scale. Here we take the application LU.C for example. By running 8/16/32/64 application processes on eight compute nodes (hence 1/2/4/8 processes per node), we evaluated the duration time to perform one migration. The results are shown in Figure 6. Our RDMA-based process migration scheme is very efficient in migrating processes images, and the time spent in Phase 2 remains at low level. The time cost in Phase 3 becomes prominent due to the file-based restart scheme on the migration target node. At Phase 3, the target node restarts the processes by restoring their memory snapshot from checkpoint files, therefore this cost is in proportion to the task scale.

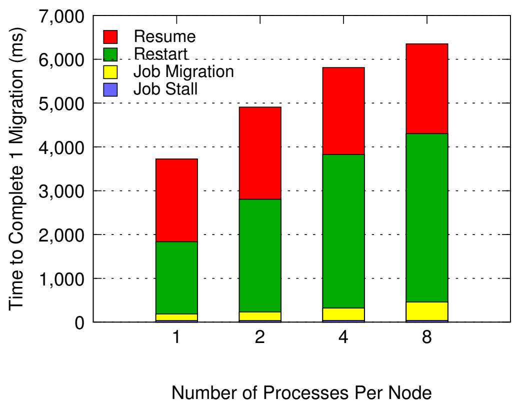


Fig. 6. Scalability of Job Migration Framework (LU.C, 8 Compute Nodes)

C. Comparison with Checkpoint/Restart

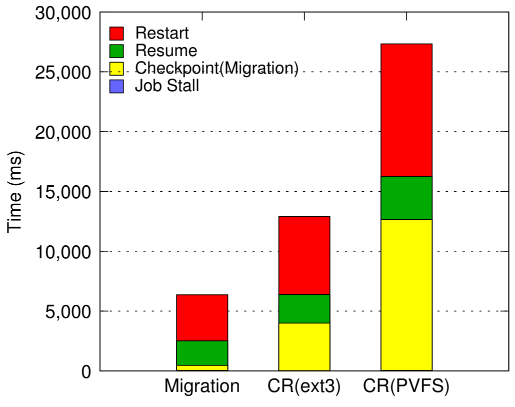
Checkpoint/Restart (CR) is a widely deployed strategy to achieve fault tolerance. MVAPICH2 includes a Checkpoint/Restart framework aided by BLCR [14]. In this section, we compare the performance of the proposed Job Migration framework with the existing Checkpoint/Restart strategy used in MVAPICH2. Similar to Job Migration, MVAPICH2 C/R goes through four phases, but with some differences as explained below.

- **Job Stall**. In this phase, all processes coordinate to reach a consistent global state, tear down their communication end-points, and freeze their execution. This phase is the same for both strategies.
- **Checkpoint**: In this phase, each process is checkpointed individually using BLCR library, and the process image is saved either to local disk, or to a global file system. This phase corresponds to the **"Migration"** phase for Job Migration, when only processes running on the health-deteriorating node are checkpointed, and their process images are migrated to the target node.
- **Resume**: In this phase, all processes rebuild their communication end-points, and resume execution. This phase is the same for both strategies.
- **Restart**: With C/R, this phase is optional and happens only if a job fails. In this phase, all processes load their process images from the checkpoint files, and restore themselves to the last saved state. On the contrary, this phase is mandatory by Job Migration to recover the processes on the target node (as is detailed in section III-A).

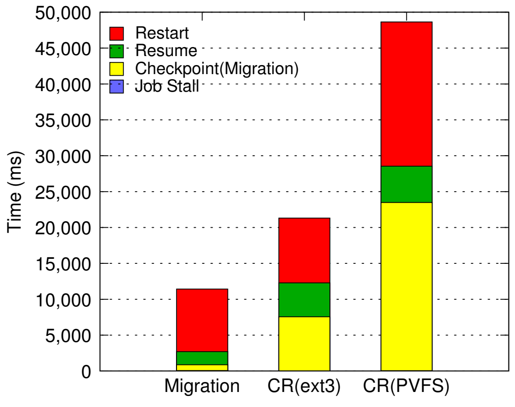
In this section, we evaluated the time to checkpoint NPB LU/BT/SP of class C with 64 processes running on eight compute nodes. The checkpoint files are saved to local ext3 or PVFS [25] at different runs. PVFS 2.8.1 with InfiniBand transport is used in the experiment, with four separate nodes serve as both data servers and metadata servers. The stripe size is set to 1 MB for PVFS.

Figure 7 gives the time cost to perform a checkpoint with checkpoint files saved to local ext3 or PVFS (the stacks labeled with "CR"). The duration is decomposed into different phases as mentioned above. The time to restart the application is also included in the figure to complement the results. As a comparison, the corresponding results with our proposed Job Migration framework are also depicted in the figure (the stacks labeled with "Migration").

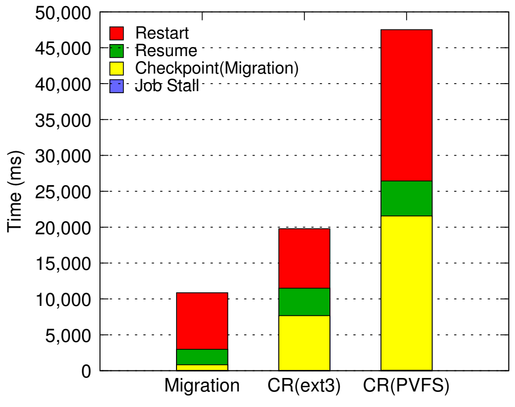
It is observed that Job Migration can greatly reduce the cost in Checkpoint (Migration) phase, since only the processes on the migration source node are



(a) LU.C.64



(b) BT.C.64



(c) SP.C.64

Fig. 7. Comparing Job Migration with Checkpoint/Restart (CR)

migrated. On the contrary, CR scheme dumps process images for all processes of the application. This fact is reflected in Table I. In the example of LU.C.64, only 170.4 MB data is migrated during a migration. However, 1363.2 MB checkpoint data must be dumped with Checkpoint/Restart (CR) scheme. Figure 7 also indicates that the storage subsystem has a huge impact on CR performance. In the example of BT.C.64, all checkpoint data is dumped to local ext3 file system in 7.5 seconds. This time is prolonged to 23.4 seconds if the checkpoint is saved to PVFS. A total of 9.1 seconds is needed to restart this application from checkpoint files at local ext3 file system, while this time becomes 20.1 seconds if the checkpoint files are stored in PVFS. The low I/O bandwidth achieved by PVFS can be ascribed to the contentions caused by the concurrent I/O streams to write/read checkpoint files to/from the shared storage.

TABLE I
AMOUNT OF DATA MOVEMENT (MB)

	Job Migration	CR
LU.C.64	170.4	1363.2
BT.C.64	308.8	2470.4
SP.C.64	303.2	2425.6

As shown in Figure 7, the proposed Job Migration scheme can greatly drive down the cost to handle a node failure. For example, in LU.C.64, a Job Migration can be completed in 6.3 seconds. As a contrast, a complete CR cycle (including all four phases) needs 12.9 seconds to checkpoint the application to local ext3, and then restart from it. Job Migration accelerates this procedure by 2.03 times. With PVFS, a complete CR cycle costs 28.3 seconds: Job Migration achieves a speedup of 4.49 times. Even if the restart cost is not counted into CR strategy, our Job Migration still outperforms CR strategy. For LU.C.64, 6.4 seconds are required to checkpoint the application to local ext3, and 16.3 seconds are needed if checkpointing to PVFS. Job Migration is comparable to CR with local ext3, and outperforms CR with PVFS2 by 2.58 times.

In a typical cluster, however, the checkpoint is usually stored to a global file system. Although a lot of work has been done to improve the checkpoint performance to a shared storage [23], [26], dumping huge amount of data to the shared file system still poses a significant performance bottleneck, since it competes with other application for the I/O bandwidth, thus adversely affecting the performance of all applications. This problem is eradicated by Job Migration, which only migrates process images to a spare node, without any impact on the shared storage subsystem.

V. RELATED WORK

In the field of High Performance Computing systems, many efforts have been carried out to provide fault tolerance to MPI applications. Many message passing libraries such as LAM/MPI [27], MVAPICH2 [14] and

OpenMPI [28] have Checkpoint/Restart techniques. Generally, the application image is periodically saved and used to restart the application when a failure occurs and the checkpoint is coordinated among the processors. CoCheck [29], Starfish [30] and Clip [31] are representatives of this class of checkpointing. These coordinated checkpoint approach shares a downside in that all processes must save their process images in a coordinated manner, which imposes a heavy burden on the I/O subsystem. Un-coordinated checkpoint(Message-Logging based) approach, on the other side, has the central idea of retransmitting one or more messages when a system failure is detected. They can be optimistic, pessimistic or casual. MPICH-V [32] implements an uncoordinated checkpoint/restart protocol with message logging to account for process state. However, message-logging requires some book-keeping upon every send/receipt of a message, which implies significant overhead in high-bandwidth network environment such as InfiniBand. Additionally it is susceptible to the well-known *domino effect* [33], [34] that leads to cascading abort of interdependent processes.

Recent research focuses on proactive Fault Tolerance. The authors in [9], [10] propose a proactive process-level live migration mechanism for LAM/MPI over TCP/IP. The authors in [35] propose a mechanism that combines processor virtualization and dynamic task migration to migrate tasks away from processors which are expected to fail. Unlike their work, our design is optimized for InfiniBand to take the advantage of RDMA data transfer. The authors in [11], [12] exploit an approach based on operating system virtualization techniques to migrate an MPI task. OS Virtualization strategy, however, comes with the inextricable cost to migrate the complete memory content used by the guest OS. Process migration strategy, on the other hand, migrates only the memory content used by the processes.

VI. Conclusions

In this paper we present our design and implementation of a Job Migration framework for MVAPICH2, a high performance implementation of MPI-2. This new scheme complements the existing Checkpoint/Restart based fault tolerance design in MVAPICH2. With this framework, an application can react to an imminent unavailability of a node reported by some failure prediction/detection module, and proactively migrate application processes from the health-deteriorating node to a spare node. By using the Migration Trigger, moreover, a migration can also be triggered by user request or a job scheduler under different circumstances, i.e., a system-maintenance task or a load imbalance. An RDMA-based migration strategy with aggregation is implemented to improve efficiency of process migration. Experimental results show that the Job Migration design is able to drastically reduce the time spent to handle a node failure.

As part of our future work, we plan to improve the process-restart component on the spare node by using a memory-based restart strategy, so as to further drive down the cost of process migration. We also want to investigate the potentials of our process-migration approach to benefit the existing Checkpoint/Restart strategy by prolonging the interval between full job-wide checkpoints.

VII. Funding Acknowledge

This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CNS-0403342, #CCF-0621484, #CCF-0702675, #CCF-0833169, #CCF-0916302, #OCI-0926691 and #CCF-0937842; grant from Wright Center for Innovation #WCI04-010-OSU-0; grants from Intel, Mellanox, Cisco, QLogic, and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Advanced Clustering, Appro, QLogic, and Sun Microsystems.

VIII. Acknowledgement

We thank Karthik Gopalakrishnan and Manikantan Kalaiya for their initial work in the implementation of this Job Migration framework.

IX. Software Distribution

The proposed design will be included in an upcoming MVAPICH2 release.

References

[1] "MPI over InfiniBand, 10GigE/iWARP and RDMAoE," in *http://mvapich.cse.ohio-state.edu/*.

[2] "MPI 3.0 Standardization Effort," `http://meetings.mpi-forum.org/MPI_3.0_main_page.php`.

[3] H. Song, C. b. Leangsuksun, and R. Nassar, "Availability Modeling and Analysis on High Performance Cluster Computing Systems," in *ARES '06: Proceedings of the First International Conference on Availability, Reliability and Security*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 305–313.

[4] S. Fu and C.-Z. Xu, "Exploring event correlation for failure prediction in coalitions of clusters," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–12.

[5] "Intelligent Platform Management Interface (IPMI)," http://www.intel.com/design/servers/ipmi/.

[6] S. Chakravorty and L. V. Kale, "A fault tolerance protocol with fast fault recovery," in *IPDPS 2003*, 2003.

[7] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam, "Critical event prediction for proactive management in large-scale computer clusters," in *KDD '03*, 2003, pp. 426–435.

[8] InfiniBand Trade Association, "The InfiniBand Architecture," http://www.infinibandta.org.

[9] Chao Wang and Frank Mueller and Christian Engelmann and Stephen L. Scott, "Proactive process-level live migration in HPC environments," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[10] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance," in *IPDPS*, 2007, pp. 1–10.

[11] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive fault tolerance for HPC with Xen virtualization," in *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, 2007, pp. 23–32.

[12] W. Huang, Q. Gao, J. Liu, and D. K. Panda, "High performance virtual machine migration with rdma over modern interconnects," in *CLUSTER '07: Proceedings of the 2007 IEEE International Conference on Cluster Computing*, 2007.

[13] Q. Gao, W. Huang, M. J. Koop, and D. K. Panda, "Group-based Coordinated Checkpointing for MPI: A Case Study on InfiniBand," in *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2007, p. 47.

[14] Q. Gao, W. Yu, W. Huang, and D. K. Panda, "Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand," in *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 471–478.

[15] X. Ouyang, K. Gopalakrishnan, and D. K. Panda, "Accelerating Checkpoint Operation by Node-Level Write Aggregation on Multicore Systems," *ICPP 2009*, September 2009.

[16] X. Ouyang, K. Gopalakrishnan, T. Gangadharappa, and D. K. Panda, "Fast Checkpointing by Write Aggregation with Dynamic Buffer and Interleaving on Multicore Architecture," *HiPC 2009*, December 2009.

[17] Duell, J., Hargrove, P., and Roman, E., "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart," Lawrence Berkeley National Laboratory, Berkeley, CA 94720, Tech. Rep. LBNL-54941, 2002. [Online]. Available: {https://ftg.lbl.gov/CheckpointRestart/Pubs/LBNL-54941.pdf}

[18] "CIFTS Web Page," http://www.mcs.anl.gov/research/cifts.

[19] R. Gupta, P. Beckman, B. Park, E. Lusk, P.Hargrove, A. Geist, D. Panda, A.Lumsdaine, and J. Dongarra, "CIFTS: A Coordinated Infrastucture for Fault-Tolerant Systems." in *In Intĺ Conference on Parallel Processing (ICPP 0́9)*, 2009.

[20] "Top 500 Supercomputers," http://www.top500.org.

[21] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters," in *SciDAC*, 6 2006.

[22] J. K. Sridhar, M. J. Koop, J. L. Perkins, and D. K. Panda, "ScELA: Scalable and Extensible Launching Architecture for Clusters," in *HiPC*, 2008, pp. 323–335.

[23] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: a checkpoint filesystem for parallel applications," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[24] F. C. Wong and R. P. M. etc., "Architectural requirements and scalability of the NAS parallel benchmarks," in *Supercomputing '99*, 1999, p. 41.

[25] "PVFS2," http://www.pvfs.org/.

[26] S. Al-Kiswany, M. Ripeanu, S. Vazhkudai, and A. Gharaibeh, "stdchk: A Checkpoint Storage System for Desktop Grid Computing," June 2008.

[27] 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*, vol. 19, no. 4, pp. 479–493, Winter 2005.

[28] J. Hursey, J. Squyres, T. Mattox, and A. Lumsdaine, "The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI," in *12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems*, March 2007.

[29] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," in *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, 1996.

[30] A. Agbaria and R. Friedman, "Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations," *High-Performance Distributed Computing, International Symposium on*, vol. 0, p. 31, 1999.

[31] Y. Chen, J. S. Plank, and K. Li, "CLIP: A Checkpointing Tool for Message-Passing Parallel Programs," in *In SC97: High Performance Networking and Computing*, 1997, pp. 1–11.

[32] G. Bosilca, A. Bouteiller, S. Djilali, G. Fedak, C. Germain, T. Herault, V. Neri, and A. Selikhov, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," in *In Supercomputing*, 2002, pp. 1–18.

[33] B. Randell, "System structure for software fault tolerance," in *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM, 1975, pp. 437–449.

[34] E. N. M. 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, pp. 375–408, 2002.

[35] S. Chakravorty, C. Mendes, and L. Kale, " Proactive fault tolerance in MPI applications via task migration ," in *HiPC*, 2006.