View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Open Repository and Bibliography - Luxembourg

High Performance Pipelined Process Migration with RDMA

Xiangyong Ouyang, Raghunath Rajachandrasekar, Xavier Besseron and Dhabaleswar K. Panda

Department of Computer Science and Engineering
The Ohio State University
{ouyangx, rajachan, besseron, panda}@cse.ohio-state.edu

***Abstract*—Coordinated Checkpoint/Restart (C/R) is a widely deployed strategy to achieve fault-tolerance. However, C/R by itself is not capable enough to meet the demands of upcoming exascale systems, due to its heavy I/O overhead. Process migration has already been proposed in literature as a pro-active fault-tolerance mechanism to complement C/R. Several popular MPI implementations have provided support for process migration, including MVAPICH2 and OpenMPI. But these existing solutions cannot yield a satisfactory performance.**

In this paper we conduct extensive profiling on several process migration mechanisms, and reveal that inefficient I/O and network transfer are the principal factors responsible for the high overhead. We then propose a new approach, Pipelined Process Migration with RDMA (PPMR), to overcome these overheads. Our new protocol fully pipelines data writing, data transfer, and data read operations during different phases of a migration cycle. PPMR aggregates data writes on the migration source node and transfers data to the target node via high throughput RDMA transport. It implements an efficient process restart mechanism at the target node to restart processes from the RDMA data streams. We have implemented this Pipelined Process Migration protocol in MVAPICH2 and studied the performance benefits. Experimental results show that PPMR achieves a 10.7X speedup to complete a process migration over the conventional approach at a moderate (8MB) memory usage. Process migration overhead on the application is significantly minimized from 38% to 5% by PPMR when three migrations are performed in succession.

I. Introduction

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 modern-day clusters, inevitably leading to more frequent failures of individual components [1, 2]. As a consequence, the possibility of an application being interrupted by a failure during its execution becomes so high that fault-tolerance has become a necessity. 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 [3].

Checkpoint/Restart (C/R) [4, 5] has been a common practice to guarantee fault-tolerance for large scale applications. A typical C/R mechanism saves a snapshot of the current state for all processes in an application 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. Upon a failure, the entire application has to be aborted and resubmitted to the job scheduler, incurring the lengthy queuing delay. Such a scheme will not be a viable solution to serve the needs of upcoming exascale systems, which will typically have a very low Mean Time Between Failures (MTBF).

Job/process migration [6–9], a pro-active fault-tolerance mechanism, has been proposed as a complement to C/R. During a migration, the processes running on a source node are checkpointed and the checkpoint data is transferred to a healthy spare node where the processes are restarted. All other processes of the application are paused when a migration is initiated and resume execution when the migrated processes are restored. Migration overcomes the two key drawbacks of C/R, namely the unnecessary dumping of all processes' snapshots and the resubmit queuing latency during restart. Job Migration can work in synergy with C/R by significantly reducing the frequency of full checkpoint [7], providing two prerequisites: the capability to predict a subset of imminent failures with health monitoring mechanisms such as IPMI [10] and varied failure prediction models [11, 12], and the availability of healthy spare nodes.

Additionally, process migration is also a desirable feature to meet many other demanding requirements such as cluster-wide load balancing, server consolidation, performance isolation and ease of management. Hence any progress to improve process migration performance will likely be perceived by a wide spectrum of demanding cluster applications.

Process Migration has been supported by several popular MPI stacks including MVAPICH2 [13] and OpenMPI [14]. Experiments show that these implementations cannot achieve an optimal performance. Detailed profiling in Section III reveals that the high cost arises from three factors involving vast amount of file IO: (1) overhead at the source node to write process images into files; (2) overhead to copy the image files from the source to the target node; (3) overhead at the target node to read process images files to restart.

In [8], we have made an initial attempt to optimize (1) and (2) by leveraging RDMA to transfer checkpoint data. However it hasn't totally solved the problem since the heavy IO overhead at (3) still dominates. Several other migration protocols [7, 15] proposed to convert a process image to a socket stream which can be read by the target node to perform restart, thus avoiding the filesystem IO. However this approach is subject to the network protocol overhead inside TCP/IP stack, and loses the opportunity to leverage the advanced network features such as RDMA.

In this paper we take the challenge to design a new approach that can significantly reduce the overhead at process migration. Specifically we want to address several questions:

- What are the main factors that dominate the heavy overhead in process migration?
- How to optimize the checkpoint IO path to reduce the IO cost at migration source/target nodes?
- How to optimize the data transfer path by leveraging advanced RDMA transport?
- What are the main factors that determine the performance of such a design?

We have designed a new approach Pipelined Process Migration with RDMA (PPMR) to optimize process migration. Our design fully pipelines the checkpoint data writing, data transfer, and data read, the three major components that dominate the cost of migration. In the migration source node, we aggregate checkpoint data writes from multiple MPI processes into a shared buffer pool which is then transmitted to a buffer pool on the migration target node via high performance RDMA transport. While the data is still in transit, restart is initiated at the migration target node and is fed with the RDMA data streams on-the-fly from the buffer pool.

We have implemented this new protocol into MVAPICH2 [13]. Experiments show that our fully-pipelined design achieves 10.7X and 2.3X speedup over the file-based approaches with a local EXT3 filesystem or a high-throughput PVFS2 [16] filesystem, respectively. Results also indicate that this new approach requires only a smaller amount of memory (8MB) to achieve an ideal performance, and causes only marginal overhead (up to 5%) on the application when three migrations are performed in succession.

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

- Through detailed profiling, we have identified the dominant factors that determine the cost of process migration.
- We have designed and implemented a new protocol that radically drives down the cost of process migration by fully pipelining the key steps in a migration and totally bypassing the IO overhead.
- We have conducted comprehensive performance evaluation of the proposed design, and revealed some key characteristics of this pipelined design.
- The new pipelined protocol significantly lowers the cost to perform process migration, making it more practical to be deployed in the future exascale systems to satisfy a wide range of requirements including fault-tolerance, load-balancing, server consolidation and so on.

The paper is organized as follows. In section II we give a background about the key components involved in our design. In section III we conduct extensive profiling to investigate the key components that determine the efficiency of process migration. Based on our observation, we propose a new pipelined design in Section IV. In section V, we present our experiments and evaluation. Related work is discussed in Section VI, and in section VII we present 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) [17]. 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 [18].

MVAPICH2 [13] is an open source MPI-2 implementation using InfiniBand, iWARP and other RDMA-enabled interconnect networking technologies. MVAPICH2 is being used by more than 1,400 organizations world-wide. MVAPICH2 supports application initiated and system initiated coordinated Checkpoint and Restart [19, 20] and process migration [8] using the BLCR library [21].

B. Berkeley Lab Checkpoint/Restart (BLCR)

Berkeley Lab Checkpoint/Restart (BLCR) [21] allows programs running on Linux systems to be checkpointed

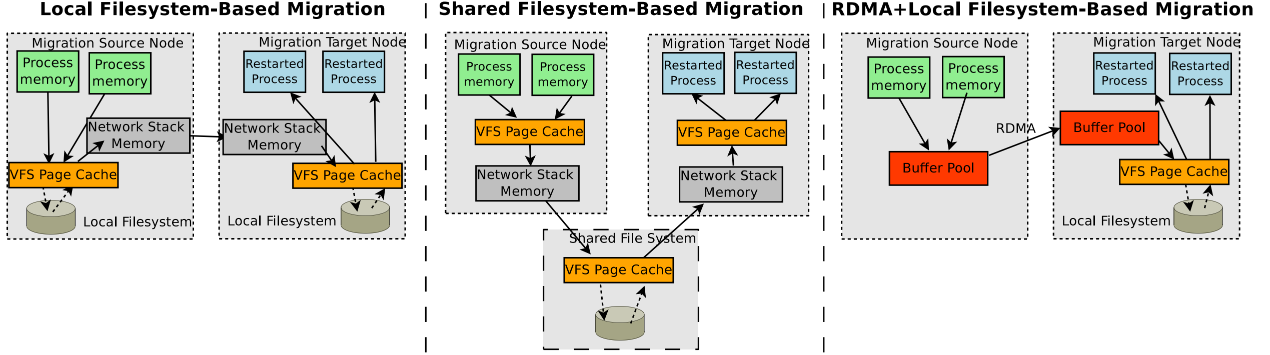


Fig. 1. Three Non-optimal Process Migration Schemes

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.

C. Filesystem in Userspace (FUSE)

Filesystem in Userspace (FUSE) [22] is a software that allows to create a virtual filesystem in the user level. It relies on a kernel module to perform privileged operations at the kernel level, and provides a userspace library that ease communication with this kernel module.

FUSE is widely used to create filesystems that do not really store the data itself but relies on other resources to effectively store the data. Then, a FUSE virtual filesystem is like a way to present and organize data to users through the classic filesystem interface.

III. Profiling Process Migration

This section studies different migration approaches in order to understand the root causes of overhead incurred during a process migration. With this knowledge we will identify the dominant factors that determine its time cost.

We consider three non-optimal process migration approaches, as illustrated in Figure 1. These schemes are evaluated with MVAPICH2 [13]

Local filesystem-based migration: The processes are checkpointed into image files stored in a local filesystem (EXT3 in this paper). Then the image files are transferred via the 1 GigE to the local filesystem on the target node. After all data are received the processes are restarted on the target node. We denoted this approach as *"Local"* for brevity.

Shared filesystem-based migration: The processes are checkpointed into image files stored in a shared filesystem (PVFS2 [16] in this paper). Then the target node restarts the processes by reading data from the shared filesystem. It's denoted as *"Shared"* in the paper.

RDMA-transfer with local filesystem: The processes on a source node are checkpointed and the data is aggregated into a staging buffer pool. Meanwhile a set of IO threads transfer the data to the target node via RDMA. On target node, the data is saved as checkpoint files in local filesystem. Later on, these files are used to restart the processes. It's named as *"RDMA+Local"* in the paper.

A. Characterizing Process Migration Protocols

A Process Migration can be characterized with a 5-step series:

Step 1: Suspend. Once a migration is initiated, all processes of the application shall suspend their communication activities and drain all in-flight messages. If the transport utilizes high-performance network with RDMA support, then their communication end-points shall be torn down. This step is necessary for all processes to reach a consistent global state in order to checkpoint individual processes, due to the characteristics native to RDMA-capable networks. The reasons are detailed in [8].

Step 2: Process Snapshot (Write). Once the application is suspended, a snapshot is taken for each processes on the source node. MVAPICH2 uses BLCR [21] to checkpoint individual process images into files. For the *"Local"* and *"Shared"* approaches, BLCR directly writes the process images on the chosen filesystem. For *"RDMA+Local"* approach, the BLCR library has been modified to aggregate all writes into a staging buffer pool. Simultaneously a set of IO threads transfer the data to target node through RDMA (Step 3).

Step 3: Process image transfer (Transfer). In this step,the process images are transfered from the source node to the target node. Depending on the considered approach, this transfer can take different forms. In *"Local"* approach the process images are transferred directly from the source node to the target node using the `scp` command. For *"Shared"* approach the data transfer is implicit: during *Write* operations in Step 2,

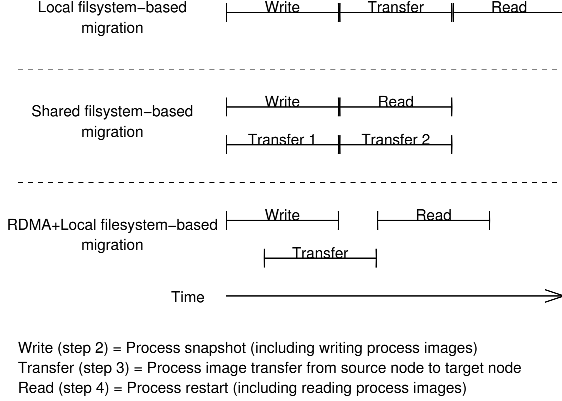


Fig. 2. Overlapping of steps in the Different Migration Approaches

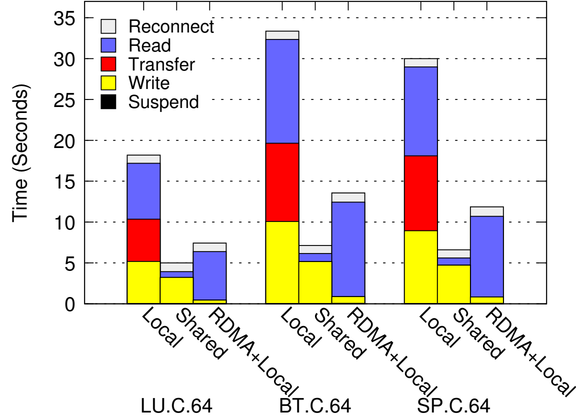


Fig. 3. Decomposed Time Cost to Complete One Migration (64 Processes on 8 Compute Nodes, 8 Processes are Migrated)

data is transferred to the shared filesystem. For the *"RDMA+Local"* approach, the process image data is transferred by chunks directly to the target node using RDMA capabilities of the InfiniBand network. On the target node, a file on the local disk is created for each process image that has been transferred.

Step 4: Process Restart (Read). This step loads the process images and restarts the application processes on target node. This task is achieved by the BLCR library by reading process image files from the local filesystem (for *"Local"* and *"RDMA+Local"* approaches) or from the network shared filesystem (for the *"Shared"* approach).

Step 5: Reconnection. Once the processes have been restarted on the target node, all processes of the application synchronize and rebuild their communication end-points and resume their communication activity. Once this is done the application has been successfully migrated.

The above five steps represent the elementary operations that need to be performed to realize process migration of an MPI application. However, depending on the considered approach, these steps can overlap. Figure 2 shows how these steps overlap in different migration approaches. In this figure, we only show steps 2, 3 and 4 for simplicity purpose (steps 1 and 5 don't differ in these approaches).

In the case of *Local* based migration, these three steps are serialized. Each step waits for the previous one to complete before starting. In the *Shared* based approach, there are two data transfer steps: one from the source node to the shared filesystem server, and one from the server to the target node, but these transfers overlap with the Write and Read steps. However, Step 2 (Write) and Step 4 (Read) are serialized because the process restart step has to wait for the process images to be fully written before restarting. Finally, the *RDMA+Local* based migration has only one transfer step which overlaps with the Write step. However, similar to the *Shared* based approach, the Step 4 (Read) waits for the process image files to be fully transferred before restarting.

B. Profiling Process Migration

We have run the three non-optimal process migration protocols and collected detailed profiling about the time cost to complete one migration which relocates 8 processes from one compute node to another spare node. Three applications - LU,BT and SP, with input class C are used in the profiling (detailed in Section V). Figure 3 breaks down the time cost into different steps.

We find data write is responsible for part of the time cost during a migration for both *Local* and *Shared* strategies. *RDMA+Local* takes advantage of RDMA transport to minimize the cost to write and transfer data. *Local* pays some price to transfer data using `scp` command via 1 Gigabit Ethernet. Both *Local* and *RDMA+Local* incur heavy overhead during restart phase. With both strategies, the processes being restarted load data from local files concurrently, which results in severe contentions in the filesystem. The *Shared* approach, on the other hand, avoids this cost by fetching data from PVFS data servers via high performance InfiniBand network. However this efficiency is obtained because the processes are reading from page cache on data servers. In this profiling, up to 320MB data is migrated and stored in the four PVFS data servers, hence they can buffer all data in their page cache. In a production deployment the shared filesystem is likely to be servicing multiple intensive data streams concurrently, and one cannot expect data for a single job to be stored in page cache for fast retrieval.

The above characterization and profiling results indicate that all the three steps (Write, Transfer and Read) should be taken care of in order to improve process migration efficiency. In the next section, we propose our new design that can fully overlap the three major components to achieve better performance.

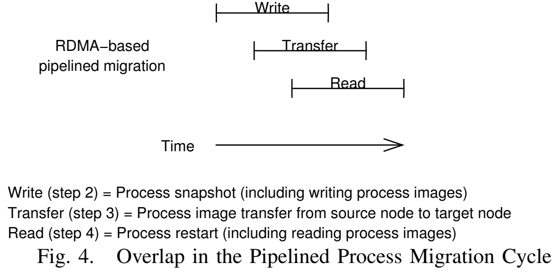


Fig. 4. Overlap in the Pipelined Process Migration Cycle

IV. Design and Implementation

In this section, we elucidate the design of our new Pipelined Process Migration with RDMA (PPMR) protocol which addresses the performance issues of the current migration approaches presented in Section III.

As mentioned earlier, the process migration framework in MVAPICH2 is based on the BLCR library [21] for checkpointing and restarting individual processes. BLCR uses files as a medium to dump the snapshot of a process. The image of a process being restarted is loaded from the same file. This constitutes an implicit barrier between Step 2 (Write) and Step 4 (Read) of the process migration model. As illustrated in Figure 2, the current migration approaches wait until the complete process snapshot is dumped to a file (Step 2) before proceeding to the process restart phase (Step 4).

This sequential file-handling is a bottleneck during process migration. It can be resolved by overlapping process snapshot (Step 2) and the process restart (Step 4). This is possible because BLCR handles checkpoint files in a sequential manner, from beginning to end, like a stream.

In this manner, the entire process can be streamlined into three fundamental steps, Write to, Transfer across and Read from a pipeline that moves the process images from the source node to the target node, as shown in Figure 4. Previous studies [7, 15] exploit TCP socket to build this streamline, but their design is subject to high protocol processing overhead of TCP/IP.

We propose the Pipelined Process Migration with RDMA (PPMR) protocol to fully streamline the data movement by leveraging high performance RDMA capabilities. The PPMR architecture is depicted in Figure 5. In the figure, the *Application processes* have BLCR library linked in which takes care of checkpointing these processes. We leverage FUSE library to intercept file IO system calls and aggregate them into data streams directed to a buffer pool. Buffer mangers on both migration source node and target node cooperate to move data chunks between the two nodes.

Conceptually PPMR maintains the 5-step migration cycle as described in Section III, but with steps 2 to 4 fully pipelined. Steps 1 and 5 remains the same in this new design. Below we will discuss Step 2,3 and 4 in detail.

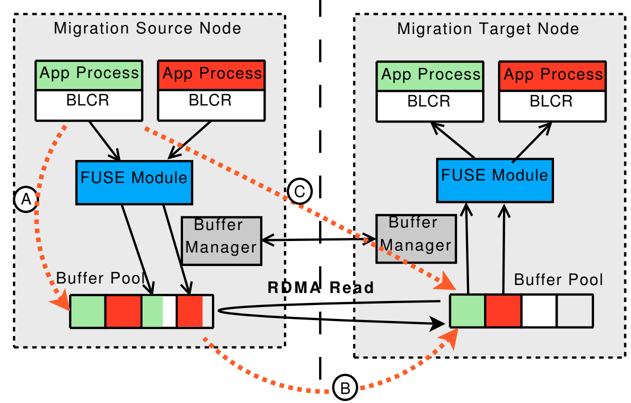


Fig. 5. Pipelined Process Migration with RDMA

Step 2: Process Snapshot (Write). In this step, the unmodified BLCR library writes the process image data by making write() system calls to a virtual filesystem which is backed by FUSE. These calls are intercepted by FUSE module, and the data is coalesced into a buffer chunk taken from the shared buffer pool. When a buffer chunk is filled up it's returned to the pool marked as full, and the next free chunk is grazed to continue the aggregation.

Step 3: Data Transfer (Transfer). Once a buffer chunk is filled on Step 2, the buffer manager on the source node sends a RDMA-Read request to its counterpart 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 the proper positions in a separate linked list. Upon such a request, the target buffer manager grabs a free chunk from the buffer pool and issues a RDMA Read request to pull over the data. Once the RDMA Read is complete, a reply is sent telling the source buffer manager to recycle a buffer chunk. The newly filled chunk gets placed to a proper position in a linked list to be used on Step 4.

It's worthwhile to note that PPMR protocol can utilize both RDMA-Read and RDMA-Write mechanisms to transfer data, with slight changes to the control message exchanges between the source/target buffer manager. Both can achieve approximately the same bandwidth given that we transfer data at big chunk sizes (128KB as indicated in later sections). Due to space constraints we only present the design and experiment results with RDMA-Read.

Step 4: Process restart (Read). In this step, the BLCR library restarts the process on the target node. For this purpose, it reads the process images from files in a

virtual filesystem built on top of FUSE in a way similar to the Step 2. These read() system calls are intercepted by FUSE, and the linked list consisting of received data chunks is scanned to locate the data being requested. If the data is found it's returned to the reading process. Otherwise the process is blocked till the demanded data chunk arrives. This is possible because the checkpoint data is generated sequentially at the source node, and BLCR reads the process image data sequentially and only once at the target node. When all data contained in a chunk has been read by a process, that chunk is recycled to receive new data coming from the source node.

PPMR enables the Write, Transfer and Read steps to be seamlessly pipelined. The other mechanisms discussed in Section III, on the other hand, require a temporary storage to keep the whole process images. Since this data is potentially too large to fit in memory, a local or shared filesystem is used to stored the process images, which create a new bottleneck in the migration.

This pipelined design allows to synchronize the throughput of the three steps Write, Transfer and Read. It has the advantage of requiring only a small temporary storage to stream the chunk, which corresponds to the buffer pool whose size is small (a few megabytes). The migration throughput for different buffer pool sizes and different chunk sizes is studied in the next section.

V. Performance Evaluation

We have implemented the Pipelined Process Migration with RDMA (PPMR) protocol into MVAPICH2 [13]. In this section we conduct extensive experiments to evaluate its performance from various perspectives including: (a) Raw performance of PPMR to pump data through the pipeline from source node to target node; (b) Time cost to perform a process migration using PPMR in comparison to other existing mechanisms; (c) Scalability of PPMR protocol with applications of varied memory footprints, and with different levels of process multiplexing.

A. *Experimental Setup*

In the evaluation, a 64-node InfiniBand Linux cluster is used. Each node has eight processor cores on two Intel Xeon 2.33 GHz Quad-core CPUs. Each node has 6GB main memory and a 250GB ST3250620NS disk drive. The nodes are connected with Mellanox MT25208 DDR InfiniBand HCAs for high performance MPI communication and process migration. The nodes are also connected with a 1 GigE network for interactive logging and maintenance purposes. Each node runs Linux 2.6.30 with FUSE library 2.8.1. We have enabled the "big_writes" mode for FUSE to perform large writes to deliver full performance. "Shared" migration protocol uses PVFS-2.8.2 [16] with InfiniBand transport and four dedicated nodes to store both data and metadata.

B. *Raw Data Bandwidth*

PPMR's performance is eventually bounded by how fast it's able to aggregate data at the source node and how quick the data can be pipelined to the target node. Multiple elements can play a role here. In this section we examine PPMR's raw performance including:

Aggregation Bandwidth: how fast the data from user processes can be aggregated via FUSE module into the shared buffer pool at the source node, assuming the buffer pool is large enough to hold all data. This corresponds to letter "A" in Figure 5.

Network Transfer Bandwidth: how fast PPMR is able to transfer data from the source node's buffer pool to target node's buffer pool. This is letter "B" in Figure 5.

Pipeline Bandwidth: how fast the data from user processes on the source node can be pumped through the whole PPMR pipeline to the buffer pool on the target node. This is represented by letter "C" in Figure 5.

Since PPMR relies on RDMA Read to transfer data, we first measure the raw bandwidth of RDMA Read using a microbenchmark "ib_read_bw" which is part of the InfiniBand Driver stack OFED-1.5.1 [23]. This microbenchmark issues 100 back-to-back RDMA Read requests to read a certain sized data chunk from another node, and measures the attained bandwidth. Figure 6 shows the results for varying chunk sizes. The network is saturated with chunk size≥16KB. This indicates that PPMR's data transfer chunk size should be ≥16KB to better utilize InfiniBand bandwidth.

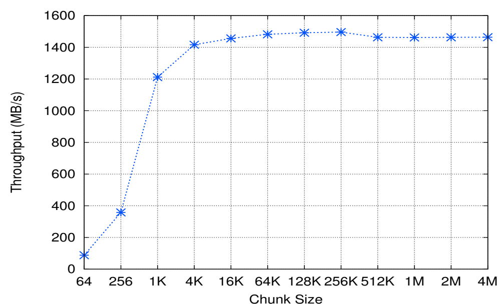
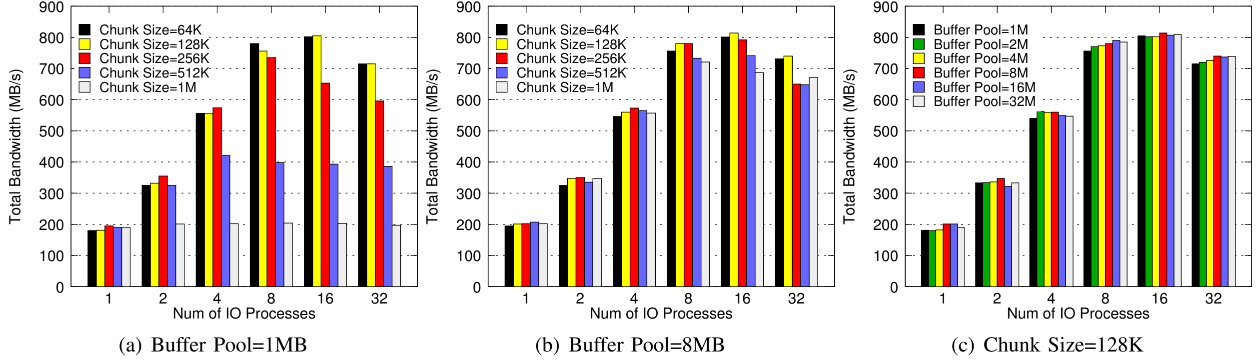


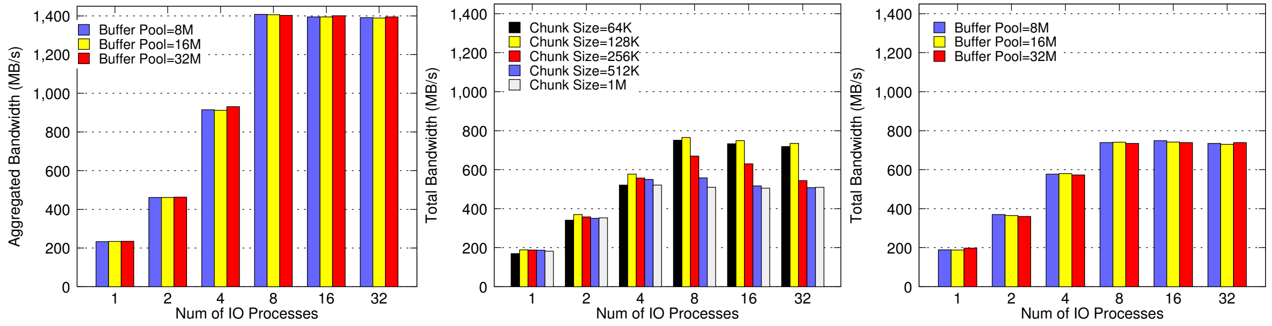
Fig. 6. InfiniBand RDMA Read Bandwidth

1) Aggregation Bandwidth: In this section we examine the *Aggregation Bandwidth* as defined before. In one compute node we start multiple IO processes. Every process makes a series of write() system calls to write 1GB data. Each write() contains 128KB data since FUSE internally coalesces data into 128KB units in "big_writes" mode. Once a buffer chunk is filled up



(a) Buffer Pool=1MB (b) Buffer Pool=8MB (c) Chunk Size=128K

Fig. 7. Aggregation Bandwidth (Higher is Better)



(a) Network Transfer Bandwidth (Chunk=128KB) (b) Pipeline Bandwidth (Buffer Pool=8MB) (c) Pipeline Bandwidth (Chunk=128KB)

Fig. 8. Network Transfer Bandwidth and Pipeline Bandwidth (Higher is Better)

with data, the data is discarded and the chunk is returned to the pool immediately to be reused.

Figure 7(a) and Figure 7(b) show the total achieved *Aggregation Bandwidth* with 1MB/8MB buffer pool respectively at varied buffer chunk sizes. We observe poor bandwidth with scarce memory (1MB) and large chunk size (>256K). In this case the IO processes are frequently blocked waiting for a free chunk from buffer pool, which detrimentally affects the performance. When 8MB buffer pool is used in Figure 7(b), the total write bandwidth reaches the peak of about 800MB/s at 16 processes, and 128KB chunk size can yield the best throughput. This is understandable since 128KB chunk size matches FUSE's internal 128KB write units. As the number of IO processes increases to 32, the FUSE internal worker threads are overloaded with more frequent lock/unlock overhead, hence the performance begins to drop.

In Figure 7(c) we vary the buffer pool from 1MB to 32MB with chunk size fixed to 128KB. We observe that *Aggregation Bandwidth* isn't very sensitive to buffer size as long as there are reasonable number of buffer chunks in the pool. All these results strongly imply that the performance of PPMR is more likely bounded by the capability of FUSE module to coalesce user processes' write streams instead of the buffer size, as long as a moderate buffer pool (8MB for example) is provisioned. FUSE intercepts write() system calls from the user processes and every such call incurs multiple memory copy overhead to move user data through FUSE internal memory to the buffer pool. Recently FUSE developers have realized such a performance hit, and zero-copy protocol has been proposed [24]. Our design can transparently benefit from such a light-weight implementation once it's available.

2) Network Transfer Bandwidth: In order to measure *Network Transfer Bandwidth* we ran multiple IO processes on a source node. Each process grabs a free memory chunk from the buffer pool and sends a request to the target node. The latter performs a RDMA Read to pull the chunk. Once RMDA Read is complete, the data is discarded and the target node sends a reply to the source node to recycle its data chunk. Figure 8(a) shows the obtained transfer bandwidth with fixed chunk size(128KB) and varied buffer pool size. With fewer IO process we see an under-utilization of the network bandwidth because of PPMR's control message overhead. With $>=$ 8 IO processes we are able to saturate the InfiniBand network with 8MB buffer pool.

3) Pipeline Bandwidth: It is bounded by the smaller one of *Aggregation Bandwidth* and *Network Transfer Bandwidth*. We ran multiple IO processes on a source node. Each process performs write() system call to write 1GB data in different chunk sizes. These write system calls are coalesced by FUSE module and redirected to the buffer pool. Then the data chunks in the buffer pool

is RDMA Read by the target node, in a way similar to how we measure *Network Transfer Bandwidth*. As shown in Figure 8(b), *Pipeline Bandwidth* reaches a peak of around 750MB/s with 8 IO processes and chunk size=128KB using 8MB buffer pool. Figure 8(c) also asserts that *Pipeline Bandwidth* isn't sensitive to buffer pool sizes.

C. Process Migration Performance

In this section we evaluate the process migration performance on a set of applications taken from NAS parallel benchmark (NPB) [25] suite version 3.2. All experiments use MVAPICH2 1.6RC1 [13] as the MPI library and BLCR 0.8.2 [21]. The buffer pool is set to be 8MB on all nodes with chunk size 128KB. Due to space constraints, we choose applications LU, BT and SP with class C input and 64 processes running on 8 compute nodes. Three spare nodes are prepared as migration targets. In one migration 8 processes are moved from a compute node to a spare node. We simulate the migration trigger by firing a user signal to the Job Manager. Other mechanisms such as node health monitoring events can also be used to kick off a migration.

Figure 9 illustrates the time cost to perform a process migration with different strategies. In the example of application BT.C.64, PPMR is able to complete one migration in 3.1 seconds. This translates into 10.7X speedups over the "Local" approach (33.3 seconds). It's also 2.3X faster than the "Shared" approach (7.1 seconds), and 4.3X faster than "RDMA+Local" mechanism (13.5 seconds).

We have also assessed the application execution time without any migration and with three migrations using different migration strategies. As shown in Figure 10, PPMR extends the base execution time by 5.1% for LU.C.64. As a contrast, "Shared" and "Local" strategies prolong LU.C.64's execution by 9.2% and 38%, respectively.

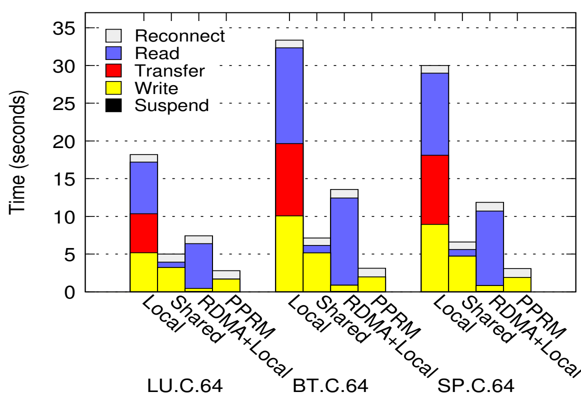


Fig. 9. Time to Complete One Migration (Lower is Better)

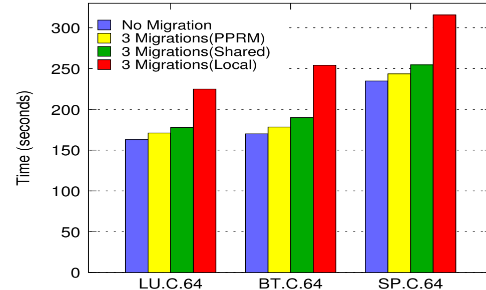


Fig. 10. Application Execution Time with and without 3 Migrations (Lower is better)

Application	Migrated Data	Time (seconds)	
		PPMR	Shared
BT.C.64	320 MB	3.5	7.4
BT.D.64	3472 MB	9.1	54.4

TABLE I
MIGRATION TIME COST WITH DIFFERENT MEMORY FOOTPRINT (BT.C/D, 64 PROCESSES ON 8 COMPUTE NODES)

D. Scalability of PPMR

In this section we evaluate PPMR's scalability from two aspects: (1) Efficiency to migrate applications with different memory footprints (Memory scalability). (2) Efficiency to migrate varying number of process from a given node (Multiplex scalability).

First we run the NAS BT class C and D benchmarks on 8 compute nodes (8 processes per node), with each node generating 320MB and 3472MB memory content respectively to be transfered in one migration (a 10.9X increase in memory footprint). As indicated in Table I, PPMR finishes a migration in 3.5 and 9.1 seconds for class C and D respectively, which represents a 2.6X increase in time cost. The "Shared" strategy spends 7.4 seconds for class C and 54.4 seconds for class D to handle a node migration (increased by a factor of 7.3).

We then run application LU.D on 8 compute nodes with 2^n processes (8 to 64). This leads to different number of processes (1 to 8) to be moved in one migration but with approximately the same amount of data (around 1500MB). As revealed in Section III, process

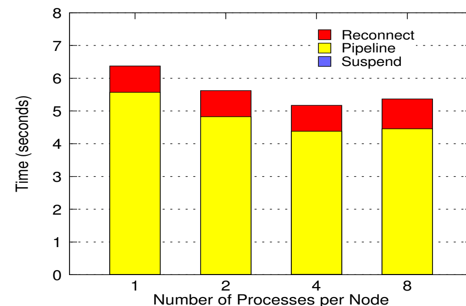


Fig. 11. Migration Time Cost at Different Number of Processes per Node (8 Compute Nodes, LU.D with 8/16/32/64 Processes)

multiplexing is a major cause of IO overhead, and this experiment is able to expose the efficiency of different migration protocols to handle this bottleneck. Figure 11 illustrates the decomposed cost to complete a migration with PPMR. As the number of processes increase from 1 to 4 the data movement cost (tagged as "Pipeline") drops slightly because better *Pipeline Bandwidth* is achieved (as seen in Figure 8(b) and 8(c)). The performance stays constant at 8 processes. As the number of processes per node keeps increasing in multicore platforms, process multiplexing is becoming more challenging. Figure 11 indicates that the PPMR mechanism is able to deliver good performance to address this challenge.

VI. Related work

In the field of High Performance Computing systems, many efforts have been carried out to provide fault tolerance to MPI applications. Generally, the application state is periodically saved and used to restart the application when a failure occurs and the checkpoint is coordinated among the processors [26]. CoCheck [15], Starfish [27], LAM/MPI [28], among others, implement this class of checkpointing. These coordinated checkpoint approaches share a downside in that all processes must save their process images in a coordinated manner, which imposes a heavy burden on the IO subsystem. On the other side, message logging fault-tolerance protocols have tried to re-build the state that the application had before the failure. This is achieved by replaying the nondeterministic events (usually messages) to processes that failed. The different variants are called optimistic, pessimistic or causal [29] and are notably implemented in MPICH-V [30]. However, message logging protocols have a large memory overhead and may cause latency in the message processing.

Recent research directions have focussed on proactive Fault Tolerance proposing to migrate a process on a spare node before the failure happens. This approach can reduce the frequency to take a checkpoint and alleviate the overhead of file IO. We propose to categorize these different migration strategies according to several metrics.

Data to migrate. The data transferred during migration can be user-specified data which implies application-level checkpoint [31], or it can be the whole process images [7–9] generated by transparent system-level checkpoint like BLCR [21]. In a similar way, AMPI proposes a mechanism based on processor virtualization [6] in which a classic MPI process is represented by a user-level thread that can be migrated when a failure is predicted. With Virtual Machine (VM) migration [32–34], the entire memory used by the VM is transmitted. However, this last approach comes with the inextricable cost to migrate the complete memory content used by the guest OS.

How the data is transfered from the source to the target. The checkpoint data can be stored as a file in local/shared filesystem [31]. Simple as it is, this approach has the drawback of additional filesystem IO overhead, especially in modern multicore architectures where multiple processes running on the same node generate checkpoint data simultaneously. In such a case the concurrent IO contentions can lead to degraded IO throughput [35, 36]. A more efficient alternative is to convert it into a data stream [7, 15] which is seamlessly transmitted over the network.

Network transport used to move data. The checkpoint data can be transferred using different network transports, such as conventional socket [7] or advance network transports such as RDMA [33]. A recent study [37] with 10GigE network revealed that data copy, buffer release and driver processing were the dominant factors of data move overhead. With InfiniBand RDMA protocol such overhead can be largely alleviated.

According to this taxonomy, the new pipelined migration strategy that we presented in this paper corresponds to a process-based migration using data streaming through RDMA transport.

VII. Conclusions

In this paper we conduct extensive profiling on several process migration implementations, and reveal that inefficient IO and network transfer are the principal factors responsible for the high overhead. We have proposed and implemented Pipelined Process Migration with RDMA (PPMR) strategy into MVAPICH2 [13] to optimize the inefficient data IO and network transfer at various aspects in a process migration. Our new protocol fully pipelines data writing, data transfer, and data read on all aspects of a migration. PPMR aggregates data writings on migration source node and transfers data to target node via high throughput RDMA transport. PPMR implements an efficient process restart mechanism in the target node to restart processes from RDMA data streams. Experimental results show that PPMR achieves a 10.7X speedup to complete a process migration over the conventional approach at a moderate (8MB) memory usage. Process migration overhead is significantly minimized from 38% to 5% by PPMR when three migrations are performed in succession during application execution.

Another benefit of our Pipelined Process Migration with RDMA (PPMR) framework is that, thanks to its modular design based on FUSE, it could be easily used in other projects that perform process migration. It only requires that the checkpoint/restart library behaves like BLCR by processing the process images sequentially like streams.

As part of our future work, we plan to explore incremental checkpoint capability provided by the upcoming BLCR release to investigate its benefits in our design. We plan on investigating how the PPMR strategy can benefit other more general applications such as cluster-wide load balancing and server consolidation. We also plan on studying how the diskless cluster architectures [38, 39] and other GreenHPC systems can fully utilize and adopt out PPMR protocol to support proactive process migration for MPI jobs.

VIII. 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 #CCF-0621484, #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.

IX. Software Distribution

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

References

[1] Schroeder, Bianca and Gibson, Garth A., "A large-scale study of failures in high-performance computing systems," in *ICDSN '06*, 2006.

[2] Glosli, J. N. and Richards, D. F. and Caspersen, K. J. and Rudd, R. E. and Gunnels, J. A. and Streitz, F. H., "Extending stability beyond CPU millennium: a micron-scale atomistic simulation of Kelvin-Helmholtz instability," in *SC '07*, 2007.

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

[4] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in *Usenix Winter Technical Conference*, January 1995.

[5] E. N. Elnozahy and J. S. Plank, "Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery," *IEEE Transactions on Dependable and Secure Computing*, 2004.

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

[7] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration in HPC environments," in *SC '08*.

[8] Xiangyong Ouyang, Sonya Marcarelli, Raghunath Rajachandrasekar and Dhabaleswar K. Panda, "RDMA-Based Job Migration Framework for MPI over InfiniBand," in *Cluster*, 2010.

[9] "A Transparent Process Migration Framework for Open MPI," http://www.open-mpi.org/papers/sc-2009/jjhursey-cisco-booth.pdf.

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

[11] H. Song, C. b. Leangsuksun, and R. Nassar, "Availability Modeling and Analysis on High Performance Cluster Computing Systems," in *ARES '06*, 2006.

[12] 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.

[13] "MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE," http://mvapich.cse.ohio-state.edu/.

[14] "Open MPI:Open Source High Performance Computing," http://mvapich.cse.ohio-state.edu/.

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

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

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

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

[19] 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*.

[20] Q. Gao, W. Yu, W. Huang, and D. K. Panda, "Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand," in *ICPP '06*.

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

[22] "Filesystem in Userspace," http://fuse.sourceforge.net/.

[23] "OFED: OpenFabrics Alliance," http://www.openfabrics.org/.

[24] "FUSE: implement zero copy read," http://lwn.net/Articles/385100/.

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

[26] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems*, 1985.

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

[28] S. Sankaran and J. M. Squyres and B. Barrett etc, "The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing," *LACSI*, Oct. 2003.

[29] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," 2002.

[30] A. Bouteiller, T. Hérault, G. Krawezik, P. Lemarinier, and F. Cappello, "MPICH-V project: a multiprotocol automatic fault tolerant MPI," *IJHPCA*, 2006.

[31] S. S. Vadhiyar and J. J. Dongarra, "Self adaptivity in grid computing," *Concurr. Comput. : Pract. Exper.*, 2005.

[32] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive fault tolerance for HPC with Xen virtualization," in *ICS '07*.

[33] W. Huang, Q. Gao, J. Liu, and D. K. Panda, "High performance virtual machine migration with RDMA over modern interconnects," in *CLUSTER '07*, 2007.

[34] Scarpazza, D. P., Mullaney, P., Villa, O., Petrini, F., Tipparaju, V., Brown, D. M. L. and Nieplocha, J., "Transparent system-level migration of PGAS applications using Xen on InfiniBand," in *Cluster '07*, 2007.

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

[36] 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.

[37] Guangdeng Liao, Xia Zhu, Laxmi Bhuyan, "A New Server I/O Architecture for High Speed Networks," in *HPCA*, 2011.

[38] J. H. L. Iii and L. H. Ward, "Implementing scalable disk-less clusters using the network file system (nfs)," in *LACSI 2003*, 2003.

[39] "HPC Diskless Cluster on Sun Blade 6000," http://wikis.sun.com/display/BladeSystems/HPC+Diskless+Cluster+on+Sun+Blade+6000.