System-level Scalable Checkpoint-Restart for Petascale Computing

Jiajun Cao*
Northeastern University
jiajun@ccs.neu.edu

Kapil Arya
Mesosphere, Inc.
kapil@mesosphere.io

Rohan Garg*
Northeastern University
rohgarg@ccs.neu.edu

Shawn Matott
CCR: Center for Computational Research
State University of New York at Buffalo
lsmatott@buffalo.edu

Dhabaleswar K. Panda
The Ohio State University
panda@cse.ohio-state.edu

Hari Subramoni
The Ohio State University
subramon@cse.ohio-state.edu

Jérôme Vienne
Texas Advanced Computing Center
The U. of Texas at Austin
viennej@tacc.utexas.edu

Gene Cooperman*
Northeastern University
gene@ccs.neu.edu

***Abstract*—Fault tolerance for the upcoming exascale generation has long been an area of active research. One of the components of a fault tolerance strategy is checkpointing. Petascale-level checkpointing is demonstrated through a new mechanism for virtualization of the InfiniBand UD (unreliable datagram) mode, and for updating the remote address on each UD-based send, due to lack of a fixed peer. Note that InfiniBand UD is required to support modern MPI implementations. An extrapolation from the current results to future SSD-based storage systems provides evidence that the current approach will remain practical in the exascale generation. This transparent checkpointing approach is evaluated using a framework of the DMTCP checkpointing package. Results are shown for HPCG (linear algebra), NAMD (molecular dynamics), and the NAS NPB benchmarks. In tests up to 32,752 MPI processes on 32,752 CPU cores, checkpointing of a computation with a 38 TB memory footprint in 11 minutes is demonstrated. Runtime overhead is reduced to less than 1%. The approach is also evaluated across three widely used MPI implementations.**

I. Introduction

Scalability of checkpointing for petascale and future exascale computing is a critical question for fault tolerance on future supercomputers. A stream of publications by researchers has been concerned with this question of fault tolerance for future supercomputers [1]–[5].

System-level transparent checkpointing has been avoided at larger scale in HPC because of the need for a full-memory dump. For example, a 2014 report [4] on software resilience presents a typically pessimistic outlook for pure full-memory checkpoints:

> "The norm in 2009 was to store the application state on remote storage, generally a parallel file system, through I/O nodes. Checkpoint time was significant (often 15-30 minutes), because of the limited bandwidth of the parallel file system. When checkpoint time is close to the MTBF, the system spends all its time checkpointing and restarting, with little forward progress. Since the [MTBF] may be an hour or less on exascale platforms, new techniques are needed in order to reduce checkpoint time." [4]

Nevertheless, prior work on transparent, system-level checkpointing is only used at moderate-scale (e.g., 128 MPI processes in [6], [7]). The single-node checkpointing package BLCR [8], [9] is used in combination with the checkpoint-restart service of a given MPI implementation such as- [10] for Open MPI or [11] for LAM/MPI. In this approach, the checkpoint-restart service temporarily tears down the InfiniBand network, and delegates the single-process checkpointing to BLCR. This approach does not scale, since BLCR does not support SysV shared memory objects [12]. Most modern MPI implementations require such shared memory for efficient communication among MPI processes on the same node to avoid the delay in going through kernel system calls.

Moreover, an important work on transparent, system-level checkpointing is [13], which supported only InfiniBand RC (reliable connection) mode. While that result sufficed for earlier MPI implementations, modern MPI implementations require InfiniBand UD for optimal performance when running with more than about 64 processes. This is because a pure point-to-point RC mode implementation would require up to n^2 connections for n MPI ranks (for n MPI processes). MPI requires InfiniBand UD for the scales considered in this work, such as 32,752 MPI processes on 32,752 CPU cores. Setting up $(32,752)^2$, or nearly 1 billion, point-to-point RC connections is unacceptable both due to large memory resources and long times for initialization.

Advances in transparent checkpointing on large-scale supercomputers depend on the fundamental problem of transparent checkpointing over InfiniBand: how to save or replay "in-flight data" that is present in the InfiniBand network at the time of checkpointing, while at the same time not penalizing standard runtime performance. In particular, we need to address (a) how to enable transparent checkpointing support for InfiniBand UD (unreliable datagram) mode; and (b) how to reduce the excessive runtime overhead seen at larger scales. This second

* This work was partially supported by the National Science Foundation under Grant ACI-1440788.

issue of runtime overhead affects performance even when no checkpoints are taken. The earlier result [13] had shown runtime overhead to grow as high as 1.7% ith 2K cores. When scaling to 4K cores on the Stampede supercomputer in this work, overhead then grow to an unacceptable 9% (see Section III-B for a discussion and solution).

A. Contributions

The primary contribution of this paper is to demonstrate the practicality of petascale system-level checkpointing through the use of full-memory dumps. In order to achieve this, DMTCP [14] was used as a vehicle for checkpointing. We have extended the designs for DMTCP software to have a tight interaction with modern MPI runtimes by taking advantage of some important scalability features. The proposed enhancements are along these directions:

1) This is the first checkpoint support for a hybrid InfiniBand communication mode that uses both reliable connection (RC) and unreliable datagram (UD). A hybrid RC/UD mode provides better performance than a pure connection-oriented RC mode, and is a commonly used optimization among modern MPI implementations. See Section III-A for details.
2) A secondary contribution is to lower the runtime overhead for checkpointing RC mode connections themselves. Previous work supported only RC mode [13], using runtime tracing of InfiniBand send messages. The runtime overhead was shown to be 1.7% for 2K cores (see Table 2 in [13]), which grew to 9% for 4K cores in current experiments on Stampede. We use a new checkpoint-time strategy that reduces runtime overhead to under 1% even for many cores (see Section IV-C2).

B. Significance of this Work

The current work represents an advance in the state-of-the-art. By transparently supporting both InfiniBand RC and UD mode, this work demonstrates a pure system-level checkpoint over 32,752 CPU cores at the petascale Stampede supercomputer [15] in just 10.9 minutes, during which 38 TB are saved to stable storage on a Lustre filesystem. In contrast, an earlier report [4] described the 2009 state-of-the-art for checkpointing to be 15–30 minutes for a supercomputer from that era, and had argued that checkpointing times would increase even further from the times of that era.

Of course checkpointing through full-memory dumps is just one component of a software resilience strategy of the future, and is compatible with other complementary approaches. These include multi-level checkpointing [16], incremental checkpointing, partial restart, mitigation of silent data corruption (SDC), tuning of checkpoint frequencies [17], [18], and alternate approaches to error prevention, prediction, tolerance, detection, containment and recovery (forward or backward) [4], [5].

C. Going beyond the petascale HPC systems

Going beyond the petascale level presented here, there is an important question of scalability to support future exascale computing. In order to address this, we enunciate a simple formula, the Checkpoint Fill-Time Law, for predicting the checkpoint time using fundamental specifications for a given supercomputer (see Section III-D). This law predicts an ideal checkpoint time, and underestimates the checkpoint time for two real-world applications (HPCG and NAMD) by as much as a factor of ten. Nevertheless, this formula predicts that SSD-based exascale supercomputers of the future will enable checkpointing through a full-memory dump in just 1.6 minutes (ideally), or a real-world 16 minutes if one extrapolates using the same factor of ten that is seen at the petascale level.

In order to gain confidence in the predictions for an SSD-based supercomputer, we also tested on a single SSD-based computer in Table I. A 3 GB checkpoint image was created there in 7.2 seconds (and restart required 6.2 seconds). This is an I/O bandwidth of 416 MB/s, close to the ideal bandwidth of 500 MB/s for SATA 3.0 interface. Since 3 GB is 2.3% of the 128 GB SSD disk, the predicted ideal checkpoint time is 2.3% of 4.3 minutes, or 5.9 seconds. So, the predicted time of 5.9 seconds compares well with the actual time of 7.2 seconds.

D. A Remark Beyond Research Contributions

In addition to the research contributions above, we were surprised to discover a counter-intuitive practical issue in checkpointing at petascale levels. Simply launching a new computation was found to be excessively slow with 8K cores, and was found to fail at 16K cores (see Section III-C). This was tracked down to limitations in the hardware/software system. The simple act of creating 16K cores was found to overwhelm the hardware/software system on the Stampede supercomputer. In discussions with sysadmins, we found that in the emphasis on InfiniBand over Ethernet meant that each rack at Stampede was provided with just a single 10 Gb Ethernet backbone from each rack. Hence, this appears to have led to longer delays in the processing of Ethernet by the kernel at larger scales, and we directly observed processes receiving a SIGKILL signal from the kernel at 16K cores.

E. Organization of Paper

The rest of this paper is organized as follows. The relevant background on Lustre, DMTCP, and the various modes used by MVAPICH2 are presented in Section II. Section III describes the methodology used to achieve petascale level and some implications for extending checkpointing to the next level. The experimental evaluation is presented in Section IV. Section V describes the scalability issues associated with petascale checkpointing. The related work is presented in Section VI, and conclusions appear in Section VII.

II. Background

The following three subsections review three critical components that affect the performance in the experiments: the MPI implementation (MVAPICH2 at TACC, and Intel MPI and Open MPI at CCR — see Section IV), DMTCP itself as the checkpointing software, and Lustre as the back-end filesystem.

A. MVAPICH2

We highlight MVAPICH2 [19] as the MPI used in the majority of experiments. Other MPI implementations typically have similar features to those described here. MVAPICH2 uses the TCP/IP-based Process Management Interface (PMI) to bootstrap the InfiniBand end-points using InfiniBand RC. While PMI is the most straightforward way to establish InfiniBand connectivity, it leads to poor startup performance due to the n^2 point-to-point connections referred to in the introduction. For MPI jobs with more than 64 processes, MVAPICH2 also uses the lazy establishment of "on-demand" connections using InfiniBand UD [20].

B. DMTCP

Distributed MultiThreaded CheckPointing (DMTCP) [14] provides a framework for coordinated checkpointing of distributed computations via a centralized coordinator. Each client process of the application communicates with the coordinator via a TCP socket. A checkpointing library is injected into each process of the target application. This library creates a checkpoint thread in each process, to communicate with the coordinator and to copy process memory and other state to a checkpoint image.

The coordinator employs global barriers to synchronize checkpoint/restart among multiple nodes, and it provides a publish-subscribe scheme for peer discovery (e.g., discover new TCP peer addresses for InfiniBand id during restart). These are used in combination with wrappers around library functions to build plugin libraries. The plugin libraries are injected along with the checkpoint library. They serve to translate real ids into virtual ids seen by the application, and to update the virtual address translation table with the new real ids that are seen on restart [21]. This virtualization capability operates below the level of the MPI library. A new plugin capability for this work serves to virtualize the InfiniBand UD mode (see Section III-A).

C. Lustre

The Lustre filesystem at Stampede plays a critical role in supporting high-bandwidth writes of checkpoint image files. Lustre [22] is a parallel object-based filesystem in widespread use that was developed to support large-scale operations on modern supercomputers. Lustre attains high I/O performance by simultaneously striping a single file across multiple Object Storage Targets (OSTs) that manage the system's disks. Lustre clients run the Lustre file system and interact with OSTs for file data I/O and with metadata servers (MDS) for namespace operations. The Lustre protocol features authenticated access and write-behind caching for all metadata updates.

III. Issues for Petascale Checkpointing and Extrapolation to Exascale Checkpointing

In the first subsection, we discuss a key barrier to petascale checkpointing and its solution: support for InfiniBand UD mode. In the nature of lessons learned, we also present two additional and unexpected barriers to scalability within the context of running on the Stampede supercomputer: excessive runtime overhead at large scale, and the lack of support for processes employing many TCP sockets.

Finally, the scalability of this approach for future exascale supercomputers is a key concern. The key question here is the write bandwidth to the storage subsystem for a full-memory dump from RAM. Section III-D presents a simple, empirical model, the Checkpoint-Fill-Time Law, to extrapolate trends, and predicts that with the adoption of SSD for secondary storage in supercomputers (and with hard disks being relegated to tertiary storage), expected checkpoint times in the exascale generation are estimated at 1.6 minutes, ideally, and 16 minutes in real-world applications.

A. Checkpointing Support for UD (Unreliable Datagrams)

Recall from Section II-A that the InfiniBand UD communication mode is for connectionless unreliable datagrams. Newer versions of MPI use a hybrid RC/UD scheme for balancing performance with the memory requirements for the queue pairs. Thus, transparent checkpointing of modern MPI requires support for UD and in particular for hybrid RC/UD mode (in which both types of communication operate in parallel).

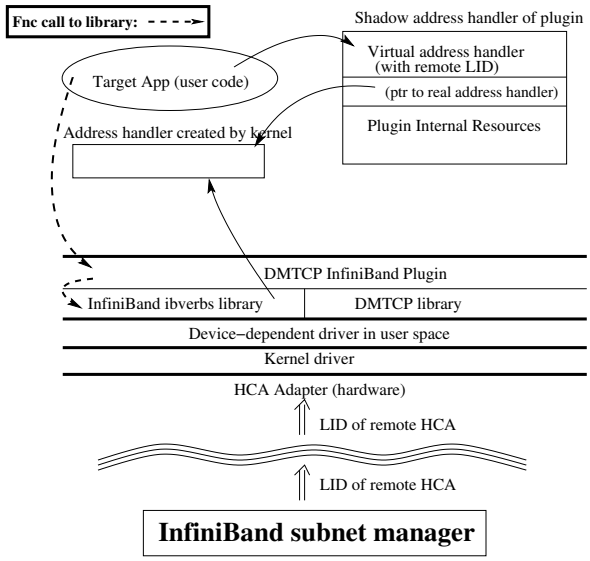


Fig. 1: Virtualization of address handler and LID (local id) of remote HCA (hardware channel adapter).

The key to checkpointing UD is to virtualize remote address of the queue pairs, so that the actual address can be replaced by a different address after restart. Figure 1 presents an overview of the situation, to accompany the detailed description that follows.

The approach here maintains a translation table between virtual and actual addresses, and is implemented using DMTCP plugins [21]. Further, on each UD-based send, the InfiniBand LID (local identifier) must also be updated for a possibly different remote queue-pair address.

In detail, each computer node includes an InfiniBand HCA (Host Channel Adapter). The HCA provides hardware support

for a *queue pair*, which refers to a send-receive pair of queues. Unlike the connection-oriented RC communication mode, UD has no end-to-end connections. So a local queue pair can send messages to different remote queue pairs.

The problem of virtualizing a remote UD address is made more difficult because of the dynamic change of the address of the remote queue pair, which is identified by a unique pair (LID (local identifier), qp_num (queue pair number)). The LID is assigned by the subnet manager, which is unique only within the local subnet, while the queue pair number is assigned by the hardware driver, which is unique only within the local HCA. Since both fields can change after restart, we need to virtualize both fields in order to identify the remote UD address.

At the time of restart, all previously created UD queue pairs (as well as the ones created after restart) will send their address pairs to the checkpoint coordinator. After the application resumes, each node must discover the (remote-LID, queue-pair-number). It was decided to do this by querying the checkpoint coordinator at runtime prior to each UD-based send for the latest destination LID. Although this adds to the runtime overhead, UD is not used as the primary communication channel and so this runtime querying overhead is negligible in practice.

Note that UD presents a very different situation from the older RC mode work in [13]. For RC mode it's safe to build the connection at restart, because the peer won't change. But the destination of a given LID can change after restart. In the extreme case, each UD-based send can change its peer queue-pair address, and so there's no fixed peer. Instead, we are forced to patch the right remote address on every send.

Finally, the UD protocol refers to an address handler (for which the remote LID is just one field). So, instead of virtualizing the remote LID, we must virtualize the address handler (AH). Hence, we create wrapper functions around all InfiniBand library calls that refer to an AH. Whenever an AH is created, we also create a shadow AH. Thus, the application code only sees pointers to our shadow AH, and our wrapper functions make corresponding modifications to the actual AH struct that is passed to the InfiniBand library. On restart, the shadow AH is redirected to a new, actual AH constructed by the InfiniBand library. In particular, this technique can account for any hidden fields in the actual AH, or other examples of data hiding that an InfiniBand library may use.

B. Reducing RC-mode Runtime Overhead

In testing on the LU.E benchmark, we saw runtime overhead rise to 9% for 4K CPU cores compared to the 1.7% runtime overhead at 2K cores reported by [13]. This was due to the non-scalable tracing of send/receive requests required by the InfiniBand checkpointing code to shadow hardware state because InfiniBand devices don't provide a way to "peek" at the current state.

We address this scaling problem by updating the model by relaxing some of the guarantees around send/receive queues. Instead of computing the exact number of pending send messages at checkpoint time, we poll the receive queues for a "reasonable" amount time during checkpointing. If a message arrives during this time, we wait again. If no messages arrive, we assume that there are no more messages in flight. For practical purposes, most message will arrive in the first time window. There might be a small number of messages arriving in the second time window if there is a slow network switch. Since the InfiniBand network is quiesced at this point (because all processes are going through checkpoint), no new messages are being scheduled to send. In our experiments, we used a "one-second window" for draining in-flight messages and noticed that all "pending" messages arrived within the first window. No messages arrived in the second window.

C. TCP Congestion Limits during MPI Initialization

While startup time was reasonable for DMTCP with 8K clients, when running with 16K clients, some of the clients randomly died. We were able to diagnose this by creating an artificial application unrelated to DMTCP. The application similarly created a TCP connection between a coordinator and each client. We observed a SIGKILL being sent to each of the random processes that died. Since standard user-space tools failed to indicate the sender of the SIGKILL, The behavior was reproducible: DMTCP ran well with 8K clients, but was never observed to run with 16K clients.

In discussions with the sysadmins, they pointed out that there was a single 10 Gb Ethernet backbone from each rack, since the cluster emphasized InfiniBand over Ethernet. we speculate that the Linux kernel had sent the SIGKILL due to excessive delays seen by the kernel on top of an overloaded Ethernet backbone.

We then implemented two standard solutions. First, a "staggered sleep" (i.e., network backoff) was used to avoid bursts of communication over the network when initializing the TCP sockets. This worked. However, Stampede also sets a per-user limit of 16K sockets per process (which can be individually overridden by the system administrator on a per-user basis).

So, in order to scale to 16K MPI processes and beyond, we then employed a second solution. We created a new mode using a two-level tree of coordinators. Specifically, a "sub-coordinator" process was created on each computer node to relay messages to the main coordinator. In certain cases, the messages of clients were aggregated by the sub-coordinator into a single message in order to optimize network overhead. As shown in section IV, the launch time improved significantly when using this two-level tree of coordinators.

D. SSDs as a Disk Replacement in the Exascale Generation

As is well known, the bottleneck for a transparent checkpoint employing a full-memory dump is the sustained write bandwidth (sustained transfer rate) to the storage subsystem. This observation yields a simple equation for predicting the ideal checkpoint time for a full-memory dump of all of the aggregate RAM. The relationship is well known, and we formalize it here as the Checkpoint-Fill-Time Law. We assume

Name	Year intro.	Storage_{RAM}	$\text{Storage}_{disk/SSD}$	Ratio	Assumed disk size	Assumed single disk bandwidth	Single disk fill time (min.)	Ideal ckpt time (min.)
Stampede (TACC)	2014	205 TB	10 PB	0.02	2 TB ??	100 MB/s	333	6.7
Jaguar (ORNL)	2009	598 TB	10.7 PB	0.056	1 TB	100 MB/s	167	9.4
Titan (ORNL)	2012	710 TB	10.7 PB	0.066	1 TB	100 MB/s	167	11.0
Sunway TaihuLight	2016	1,311 TB	??	0.05 ??	3 TB ??	100 MB/s	500	25.0 ??
CCR (UB)	2015	1.728 TB	500 TB	0.0035	4 TB	100 MB/s	666	2.3
SSD-based 4-core node	2014	16 GB	128 GB	0.125	4 TB ??	500 MB/s	4.3	4.3
Theoretical Exascale	2020	??	??	0.1 ??	4 TB ??	4 GB/s ??	16	1.6 ??

TABLE I: Predictions using the Checkpoint-Fill-Time Law for a full-memory dump (checkpoint size = Storage_{RAM}). Since the size of disks used by the storage nodes is often not reported, it is estimated at half of the largest size disk at the time of introduction of the computer.

here a write bandwidth (transfer rate) of 100 MB/s for a *single* disk.

$$\begin{aligned}\text{CkptTime} &= \text{Storage}_{RAM} \,/\, \text{Bandwidth}_{disks} \\ &= \text{Storage}_{RAM} \,/\, (\text{Number}_{disks} \times 100 \text{ MB/s}) \\ &= \frac{\text{Storage}_{RAM}}{\text{Storage}_{disks}} \times \frac{\text{Storage}_{disks}}{\text{Number}_{disks}} \,/\, 100 \text{ MB/s} \\ &= \frac{\text{Storage}_{RAM}}{\text{Storage}_{disks}} \times \text{SingleDiskFillTime}_{disk}\end{aligned}$$

Similarly, we can write down such a law for an SSD.

$$\text{CkptTime} = \frac{\text{Storage}_{RAM}}{\text{Storage}_{SSDs}} \times \text{SingleDiskFillTime}_{SSD}$$

There are many inaccuracies in this law. As a minor example, this formula ignores the existence of redundant disks in a RAID configuration. The aggressive 100 MB/s transfer rate is meant to account for that. More seriously, a back-end storage subsystem such as Lustre includes a back-end network that usually cannot support the full bandwidth of the aggregate disks. A back-end storage subsystem is optimized for typical write loads, which are only a fraction of the maximum write load with all compute nodes writing simultaneously. Finally, this law is not intended to be used for small checkpoint images, since this results in small write blocks that are inefficient for use with disks, and since there is a large variation in perceived checkpoint time for small writes due to interference by larger jobs simultaneously using I/O.

Some examples of predictions are shown in Table I. The goal of this table is to make a crude prediction on expected checkpoint times for a future exascale supercomputer based on SSDs. As will be seen in Section IV-B, the predictions of the law for Stampede running both HPCG and NAMD are approximately ten times faster than what is seen experimentally (after accounting for the fact that the HPCG computation uses only 1/3 of the available nodes and only 2/3 of the available RAM per node).

In extrapolating a future exascale SSD-based supercomputer, the formula predicts a full-memory dump time of 1.6 minutes. We assume a ratio of RAM to SSD size of 0.1 instead of the historical 0.02 or 0.05 for disks, since SSD is more expensive than disk. It is assumed that SSD will be used for secondary storage and disk for tertiary storage. If we accept a factor of ten difference between ideal, theoretical time and real-world time (in keeping with the ten-fold penalty seen for HPCG and NAMD in Section IV-B, then this extrapolation predicts a real-world 16 minutes checkpoint time for exascale computing.

The specification of a future SSD at 4 TB with write bandwidth of 4 GB/s is based on an extrapolation from current SSDs. At this time, the high-end PCI Express (PCIe) 3.0-based SSDs can achieve 1.5-3.0 TB/s of sequential writes. The PCIe 3.0 interface limits one today to 8 GB/s. (The upcoming PCIe 4.0 promises 16 Gigatransfers/s that typically translate to 16 GB/s.) We conservatively assume a bandwidth of 4 GB/s and a 4 TB storage for for commodity SSDs, four years from now. This is in keeping with Flash density trends in [23, slide 4] and with [24].

IV. Experimental Evaluation

We evaluate our approach for: (a) ability to checkpoint real-world applications (see section IV-B); scalability trends across a large range in the number of cores used (see section IV-C); and (b) applicability to diverse environments (see section IV-D).

A. Setup

The experiments were run on the Stampede supercomputer [15] at TACC (Texas Advanced Computing Center). As of this writing, Stampede is the #10 supercomputer on the Top500 list [25]. In all cases, each computer node was running 16 cores, based on a dual-CPU Xeon ES-2680 (Sandy Bridge) configuration with 32 GB of RAM.

Experiments use the Lustre filesystem version 2.5.5 (see Section II-C) on Stampede. InfiniBand connections run over a Mellanox FDR InfiniBand interconnect. A lower bandwidth Ethernet connection is available for TCP/IP-based sockets. For all the experiments, uncompressed memory dumps were used.

The largest batch queue normally available at Stampede provides normally for 16K CPU cores, but special permission was obtained to briefly test at the scale of 32,752 CPU cores. Hence, the maximum scale was 2,047 nodes with 16 CPU cores.

At 32,752 CPU cores, the tests use one-third of the compute nodes of Stampede. This is sixteen times as many cores as the largest previous transparent checkpoint of which we are aware [13]. The Stampede supercomputer is rated at 5.2 PFlops (RMAX:sustained) or 8.5 PFlops (RPEAK:peak). [25]. Hence, we estimate our usage during

this checkpoint experiment (using Xeons only) to be a large fraction of a petaflop.

The experiments at the largest scale were done using a reservation in which our experiments had exclusive access to up to one-third of the nodes of Stampede. The system administrators were careful to monitor our usage during this period, to ensure that there was no interference with the jobs of other users. The administrators observed a peak bandwidth of 116 GB/s to the Lustre filesystem, when we were writing checkpointing image files at large scale (16K through 32K CPU cores). The system administrators of Stampede also observed that they did not find any disturbance in the workflow of other users during this time.

B. Experiments with Real-world Software

Two software experiments reflect real-world experience with many CPU cores and with large memory footprints. The High Performance Conjugate Gradients (HPCG) benchmark represents a realistic mix of sparse and dense linear algebra [26], and is intended to provide a good "correlation to real scientific application performance" [27]. The tests with the molecular dynamics simulation NAMD then represents performance for an application not based on linear algebra.

1) Evaluation of HPCG: Table II shows checkpoint and restart times for HPCG at the scale of 8K, 16K, 24,000, and 32K CPU cores. The 24,000 and 32K core cases were run with special permission of the system administrators (since at Stampede, the largest standard batch queue supports only 16K cores), In all cases, the aggregate size of the checkpoint images per node is 19.2 GB, representing almost two-thirds of the 32 GB RAM available on each node. The bandwidth for writing checkpoints progressively diminishes with larger size computations, except at the largest scale of 32,752 cores. This last case was run during a maintenance period, with presumably little writing associated with that ongoing maintenance.

Note that the Checkpoint Fill-Time Law predicts an ideal checkpoint time of 6.7 minutes (see Table I). At 16K cores, the checkpoint represents 9.4% of the total $6,400$ nodes $\times$ 32 GB, and thus, the observed checkpoint time is ten times larger than the predicted ideal time of 0.628 minutes. When pushing Lustre beyond its standard configuration, the checkpoint for 32,752 cores represents 19.2% of the total RAM of $6,400 \times 32$ GB, and thus, the observed checkpoint time of 10.9 minutes is eight times larger than the predicted ideal time of 1.29 minutes.

Compared to the checkpoint times, the restarts times are nearly twice as large. We believe this is because when writing the checkpoint images, Lustre buffers the checkpoint data. On restart, the checkpoint data needs to be synchronized to the disk, transferred to each node, and then read into the memory of each node.

2) Evaluation of NAMD: Table III shows the results of NAMD for 8K and 16K CPU cores. The input parameters are taken from a NAMD-based petascale study of biomolecular simulations [28]. Comparing these results with those for HPCG, we see that the write bandwidth for Lustre depends primarily on the number of CPU cores issuing a sustained write, and does not vary significantly even though the total checkpoint sizes are smaller for NAMD. The I/O bandwidths for NAMD correspond roughly with HPCG, and hence the actual checkpoint times observed also compare to the ideal times of Table I in approximately the same ratio. (For NAMD with 8K processes, the 260 MB per process appears to be a little small for ideal Lustre I/O.0

Num. of processes	Checkpoint time (s)	Restart time (s)	Total ckpt size (TB)	Write (ckpt) bandwidth (GB/s)
8192	136.1	215.3	9.4	69
16368	367.4	706.6	19	52
24000 ([1])	634.8	1183.8	29	46
32752 ([2])	652.8	2539.05	38	60

NOTE: Executed with special permission[1]; and during Stampede maintenance[2] (mostly exclusive access to the cluster).

TABLE II: Checkpoint and restart trends for HPCG; checkpoint image size for each process is 1.2 GB, with 16 images generated on each computer node.

Num. of processes	Checkpoint time (s)	Restart time (s)	Total ckpt size (TB)	Ckpt (write) bandwidth (GB/s)
8192	41.4	111.4	2.1	51
16368	157.9	689.8	9.8	62

TABLE III: Checkpoint and restart trends for NAMD with 8K and 16K cores (one MPI process per core); checkpoint image size for each process is 260 MB and 615 MB, respectively, with one checkpoint image per process.

C. Scalability

Num. of processes	Launch time (s)
1K	0.3 - 7.5
2K	0.8 - 10.5
4K	3.2 - 86.7
8K	29.2 - 87.9
16K	99.3 - 120.8
16K (*)	15.2 - 21.6

(*) Launch time for 16K processes with checkpointing using tree of coordinators.

TABLE IV: Launch time for different number of processes running with checkpointing.

1) Launch Overhead: Table IV shows the overhead incurred when starting the NAS/NPB LU benchmark [29](NPB 3.3.1, Class E) under our proposed approach. Recall that the approach uses a centralized coordinator for coordinating checkpointing among distributed processes and the coordination messages are sent over TCP/IP network. As a result, we observe that the launch time scales with the number of processes because of the increased load on the TCP/IP network. The large variation in the launch time is due to the congestion on the TCP/IP network that is also used at Stampede for administrative work. The launch time

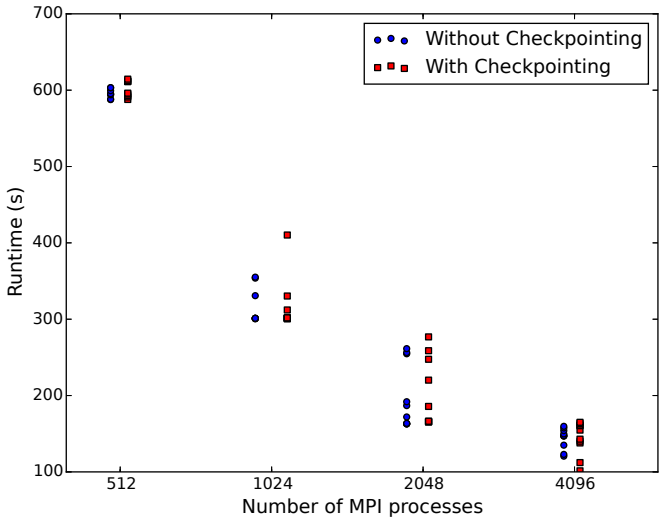


Fig. 2: Runtime overhead for LU.E

improves by up to 85% at the scale of 16K processes when we switch to using a tree of coordinators. Each compute node runs an additional sub-coordinator process that aggregates and relays requests from the processes on the same node to a root coordinator. This reduces the network connections by a factor of 16.

Num. of processes	Runtime (s) (natively)	Runtime (s) (w/ checkpointing support)	Overhead (%)
512	596.6	601.4	0.8
1024	316.2	317.8	0.5
2048	197.6	201.9	2.2
4096	144.0	144.1	0.1

TABLE V: Runtime overhead for NAS benchmark LU.E (class E): Times are native (without checkpointing support) and with checkpointing support.

2) Runtime overhead: Figure 2 demonstrates the small overhead of executing with our approach. To minimize the variation in communication overhead due to network topology, for a given problem size, the experiments with and without checkpoint support were run on the same set of nodes. Even for a fixed set of nodes, we observed a large variation in the runtimes in successive runs. We attribute this to the network congestion on the InfiniBand backend used by Lustre.

Table V shows that the average runtime overhead is less than 1% in all cases, except for 2K processes, where it is 2.2%. Since runtime overhead returned to 0.1% with 4K processes, we speculate that the run with 2K processes suffered from interference by other users on that day.

As noted in the introduction, before introducing the optimizations of Section III-B, we had observed a runtime overhead of 9% with DMTCP for 4K cores. The runtime overheads reported here are an important advance.

3) Checkpoint-Restart Trends: The scalability trends up to 16K CPU cores are demonstrated for the NAS LU benchmark in Table VI.

Num. of processes	Checkpoint time (s)	Restart time (s)	Checkpoint size per process (MB)
1024	14.5	15.8	428
2048	24.2	20.6	342
4096	33.7	36.9	300
8192	65.8	107.6	280
16368	131.8	514.7	285

TABLE VI: Checkpoint and restart trends for LU.E

The checkpoint overhead can be divided into two parts: the communication between the compute processes and the central coordinator, and the work to write the checkpoint images. Since there are only a few small messages that are sent to coordinate checkpointing, the checkpoint time is dominated by the time Lustre takes to write the checkpoint images.

Notice that the time to write a checkpoint image differs by up to 99% at the scale of 16K processes, even though the sizes of their images are the same. This difference increases with the number of processes. We attribute this to Lustre creating and writing meta-data for each file.

The restart overhead also consists of two parts: the time to build the connection with the coordinator, and the time to read into the memory the checkpoint images. In the case of restart, it is the work to build all the connections that dominates the trend. It follows the same trend as in Section IV-C1, since it uses the same model to build the connections.

D. Diversity

In this section, we demonstrate the support for different applications as well as different MPI implementations. Apart from LU, we test three other NAS benchmarks: BT, SP, and FT. In addition, we test the Scalable Molecular Dynamics (NAMD) real-world application [30]. We also show the support for two other popular MPI implementations: Open MPI [31] and Intel MPI [32].

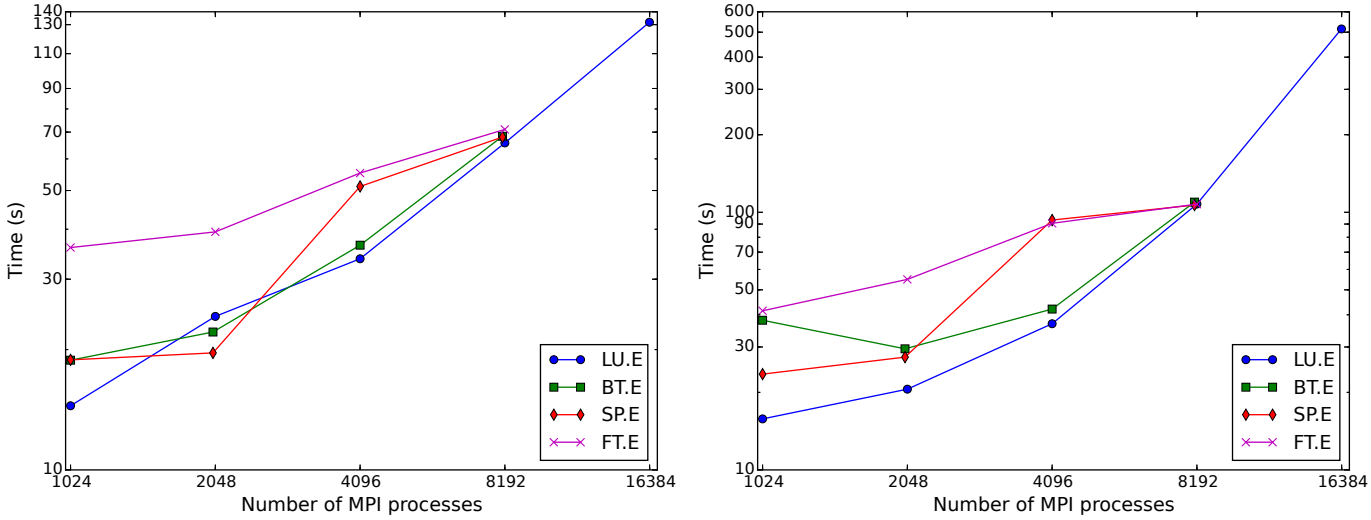
1) Evaluation of NAS/NPB Benchmarks: The performance of BT, SP, and FT is demonstrated for up to 8k CPU cores. Together with LU, the checkpoint times and the restart times are shown in Figures 3a and 3b, respectively.

The memory overhead of the proposed approach is negligible compared with the memory footprint of the application. As a result, for a given scale the checkpoint times and the restart times roughly correspond the checkpoint image sizes, regardless of the application type.

MPI implementation	Checkpoint time (s)	Restart time (s)	Checkpoint size per process (MB)
Intel MPI	298.9	191.8	775
Open MPI	299.7	128.5	520

TABLE VII: Comparison of two MPI implementations; LU.E for 500 processes

2) Evaluation of Different MPI Implementations: The experiments with Open MPI and Intel MPI were done at the Center for Computational Research at the State University of New York, Buffalo [33]. The results shown in Table VII



(a) Checkpoint Times

(b) Restart Times

Fig. 3: Checkpoint and Restart Times for Four NAS Benchmarks (note the log-log axes)

demonstrate that the proposed approach is MPI-agnostic. While the restart times roughly correspond to checkpoint image sizes, the checkpoint times don't. This is attributed to filesystem backend caching.

V. Discussion of Scalability Issues

A. Centralized Coordinator

Our approach uses a single-threaded central coordinator. It appears that this design does *not* insert a central point of contention. In these experiments, the checkpoint coordinator was always run on a separate compute node with no competing processes. Total network traffic on each socket was estimated to be a total of 20 KB during a checkpoint-restart. This traffic was primarily related to the publish-subscribe database maintained by the InfiniBand checkpointing code. Nevertheless, the CPU load was always measured at less than 5% of the time for one CPU core.

Separately, the approach uses TCP sockets to communicate with the peer processes. This represents a design flaw at the petascale level. Two issues were encountered. First, the use of multiple TCP writes without an intervening read forced us to invoke `TCP_NODELAY` to turn off Nagle's algorithm [34]. Second, there was a need at larger scale to use a staggered sleep (network backoff) during initialization of TCP sockets, so that the many peers would not overwhelm the operating system (or possibly the switch hardware) in a burst of requests to create new socket connections.

Additionally, most Linux-based operating systems include a limit on the number of socket connections per process. Our implementation needed to be extended with a tree-of-coordinators so that the many peers connecting to the coordinator would not exceed this limit.

B. Better support from the InfiniBand device drivers

As discussed in III-B, the shadow send/receive queues provide stronger correctness guarantees but impose a significant runtime overhead. The proposed alternative is to use a heuristic-based approach with relaxed correctness guarantees. A third alternative is possible if the InfiniBand device driver can provide an API to "peek" into the hardware to learn the current state of the send/receive queues. While being non-destructive, the peek operation could significantly simplify the logic around draining and refilling of the send/receive queues without imposing a runtime overhead.

C. Fast Restart using Demand-paging

During restart, there is an opportunity to use "mmap" to map the checkpoint image file back into process memory (RAM) on-demand. Instead, all memory was copied from the checkpoint image file to process memory (RAM) during restart. With "mmap", the restart could be significantly faster for a certain class of application that have a smaller working set. This would allow for some overlap of computation and demand-paging generated file I/O. Further, there is less of a "burst" demand on the Lustre filesystem. This mode was not used, so that the worst-case time for restart could be directly measured.

VI. Related Work

To the best of our knowledge, the largest previous checkpoint was carried out by Cao et al. [13]. That work demonstrated transparent checkpoint-restart over InfiniBand RC (but not UD mode) for the first time. Scalable results were demonstrated for the NAS NPB LU benchmark for 2048 MPI processes over 2048 CPU cores. That work mostly used local disk rather than Lustre, showing a maximum I/O bandwidth of 0.05 GB/s when using the local disks of 128 nodes. (The one

Num. of processes	Ckpt time (s)	Rst time (s)	Ckpt size (MB)
1024	18.8	38.1	675
2025	22.1	29.5	480
4096	36.5	42.1	368
8100	68.3	109.5	331

(a) BT.E

Num. of processes	Ckpt time (s)	Rst time (s)	Ckpt size (MB)
1024	18.9	23.5	634
2025	19.6	27.4	452
4096	51.28	93.3	368
8100	68.0	106.7	332

(b) SP.E

Num. of processes	Ckpt time (s)	Rst time (s)	Ckpt size (MB)
1024	36.0	41.5	1200
2048	39.4	54.9	703
4096	55.3	90.6	488
8192	71.1	107.5	385

(c) FT.E

TABLE VIII: Checkpoint and restart trends for various NAS benchmarks

example with Lustre over 512 processes reported an I/O bandwidth of just 0.1 GB/s.) The previous work was demonstrated solely for Open MPI using RC mode, while today most MPI implementations also take advantage of InfiniBand UD mode during initialization.

The most frequently used packages for system-level transparent checkpointing today are BLCR [8], [9], CRIU [35], Cryopid2 [36], and DMTCP [14]. Only DMTCP and BLCR are used for checkpointing MPI computations. DMTCP is the only one of the four that supports transparent checkpointing of distributed computations, and so it supports our current MPI-agnostic approach toward checkpointing.

In contrast, BLCR is also often used for checkpointing MPI, but only in combination with an MPI-specific checkpointing service such as [10] for Open MPI or [11] for LAM/MPI. BLCR can only checkpoint the processes on a single node. Hence, an MPI-specific checkpointing service temporarily tears down the InfiniBand network, and then uses BLCR [8], [9] to checkpoint individual nodes as standalone computations. Afterwards, the InfiniBand connections are re-built.

DMTCP is preferred over the combined use of BLCR with an MPI implementation-specific checkpointing service for two reasons: (a) It is MPI-agnostic, operating without modification for most MPI implementations; and (b) the alternative checkpointing service that tears down the network can incur long delays when re-initializing the InfiniBand connections upon resuming the computation and hence limits its performance.

There have been several surveys of the state of the art for software resilience in the push to petascale and then exascale computing [1]–[5]. One of the approaches is FTC-Charm++ [37], which provides a fault-tolerant runtime base on an in-memory checkpointing scheme (with a disk-based extension) for both Charm++ and AMPI (Adaptive MPI). Three categories of checkpointing are supported: uncoordinated, coordinated, and communication-induced.

Because of the potentially long times to checkpoint, a multi-level checkpointing approach [16] has been proposed. The key idea is to support local fault tolerance for the "easy" cases, so that a global checkpoint (potentially including a full-memory dump) is used as a last resort. Since restart from a global checkpoint are needed less often, such checkpoints may also be taken less often.

A popular application-level or user-level mechanism is ULFM (user-level failure mitigation). By applying recovery at the user-level, they offer different recovery models, such as backward versus forward, local versus global and shrinking versus non-shrinking. [38] reviews the ULFM model, and adds an application-level model based on global rollback.

Finally, rMPI (redundant MPI) has been proposed for exascale computing [39]. This has the potential to make checkpointing less frequent, and thus allow for longer times to checkpoint. The authors write, "Note that redundant computing ... reduces the overhead of checkpointing but does not eliminate it." The authors provide the example of a fully-redundant application for which Daly's equation [40] predicts a run for 600 hours without failure over 50,000 *nodes* with a 5-year MTTI/node. [39, Figure 12] (MTTI is mean-time-to-interrupt.)

VII. Conclusion

The need for a fault-tolerance solution for exascale computing has been a long-time concern [1]–[5]. This work has demonstrated a practical petascale solution, and provided evidence that the approach scales into the exascale generation. Specifically, system-initiated full-memory dumps for three modern MPI implementations over InfiniBand have been demonstrated. This required virtualization of InfiniBand UD, since the previous simpler InfiniBand RC point-to-point mode did not support modern MPI implementations at scale.

Testing on real-world-style applications of NAMD and HPCG stressed large memory footprints. The current Lustre filesystems successfully supported many-terabyte full-memory dumps. A simple formula in Section III-D allowed for extrapolation to future SSD-based exascale computers. The predicted ideal checkpoint time was 1.7 minutes, which extrapolates to under 17 minutes (ten-fold increase) after comparing the ideal formula against current supercomputers.

In particular, special permission was received to run HPCG with 32,752 CPU cores (one-third of the Stampede supercomputer), and a 38 TB checkpoint image was created in 10.9 minutes. The system administrator manually monitored a similar run with 24,000 cores, and reported that it did not affect the normal use of I/O by other concurrent users.

Acknowledgment

We would like to acknowledge the comments and encouragement of Raghu Raja Chandrasekar in integrating DMTCP with MVAPICH2. We would also like to acknowledge Zhixuan Cao, Peter Desnoyers, and Shadi Ibrahim for helpful feedback. We also acknowledge the support of the Texas Advanced Computing Center (TACC) and the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575.

We especially would like to acknowledge the assistance of Tommy Minyard and Bill Barth from TACC in helping set up and monitor the large experiment there. Also, we acknowledge the resources at the University at Buffalo Center for Computational Research (www.buffalo.edu/ccr) that were used in performing a portion of the numerical experiments.

REFERENCES

[1] E. N. Elnozahy and J. S. Plank, "Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery," *IEEE Trans. on Dependable and Secure Computing*, vol. 1, no. 2, pp. 97–108, 2004.

[2] F. Cappello, "Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 212–226, 2009.

[3] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *International Journal of High Performance Computing Applications*, pp. 374–388, 2009.

[4] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing Frontiers and Innovations*, vol. 1, no. 1, pp. 5–28, 2014.

[5] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing failures in exascale computing," *International Journal of High Performance Computing Applications*, 2014.

[6] F. Liu and J. B. Weissman, "Elastic job bundling: An adaptive resource request strategy for large-scale parallel applications," in *Proc. of Int. Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. ACM, 2015.

[7] F. Shahzad, M. Wittmann, T. Zeiser, G. Hager, and G. Wellein, "An evaluation of different I/O techniques for checkpoint/restart," in *IEEE 27th Int. Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. IEEE, 2013, pp. 1708–1716.

[8] J. Duell, P. Hargrove, and E. Roman, "The design and implementation of Berkeley Lab's Linux checkpoint/restart (BLCR)," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-54941, 2003.

[9] P. Hargrove and J. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters," *Journal of Physics Conference Series*, vol. 46, pp. 494–499, Sep. 2006.

[10] J. Hursey, T. I. Mattox, and A. Lumsdaine, "Interconnect agnostic checkpoint/restart in Open MPI," in *Proc. of the 18th ACM int. Symp. on High Performance Distributed Computing*. ACM, 2009, pp. 49–58.

[11] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, 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, 2005.

[12] BLCR team, "BLCR frequently asked questions (for version 0.8.5)," accessed July, 2016, https://upc-bugs.lbl.gov/blcr/doc/html/FAQ.html#limitations.

[13] J. Cao, G. Kerr, K. Arya, and G. Cooperman, "Transparent checkpoint-restart over InfiniBand," in *Proc. of the 23rd Int. Symp. on High-performance Parallel and Distributed Computing*. ACM Press, 2014, pp. 13–24.

[14] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *IEEE Int. Symp. on Parallel and Distributed Processing (IPDPS)*. IEEE Press, 2009, pp. 1–12.

[15] "TACC Stampede user guide - TACC user portal," https://portal.tacc.utexas.edu/user-guides/stampede, accessed July, 2016, 2016.

[16] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *2010 Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'10)*. IEEE Press, 2010, pp. 1–11.

[17] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. L. Scott, "An optimal checkpoint/restart model for a large scale high performance computing system," in *IEEE Int. Symp. on Parallel and Distributed Processing*. IEEE Press, 2008, pp. 1–9.

[18] D. Tiwari, S. Gupta, and S. S. Vazhkudai, "Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems," in *44th Annual Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE Press, 2014, pp. 25–36.

[19] D. K. Panda, K. Tomko, K. Schulz, and A. Majumdar, "The MVAPICH project: Evolution and sustainability of an open source production quality MPI library for HPC," in *Int. Workshop on Sustainable Software for Science: Practice and Experiences, held in conjunction with Int. Conference on Supercomputing (SC'13)*, November 2013.

[20] S. Chakraborty, H. Subramoni, J. Perkins, A. A. Awan, and D. K. Panda, "On-demand connection management for OpenSHMEM and OpenSHMEM+MPI," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, May 2015, pp. 235–244.

[21] K. Arya, R. Garg, A. Y. Polyakov, and G. Cooperman, "Design and implementation for checkpointing of distributed resources using process-level virtualization," in *IEEE Int. Conf. on Cluster Computing (Cluster'16)*. IEEE Press, 2016, pp. 402–412.

[22] Open Scalable File Systems, Inc., "Lustre file system: OpenSFS," http://opensfs.org/lustre/.

[23] J. H. Yoon, "3D NAND technology implications to enterprise storage applications," in *Proc. of Flash Memory Summit*, 2014, Memory Technology, IBM Systems Supply Chain; http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2015/20150811_FM12_Yoon.pdf.

[24] P. Desnoyers, "Future trends in SSD," private communication, July, 2016.

[25] "TOP500 supercomputer sites," http://top500.org/lists/2015/11/, Nov. 2015.

[26] J. Dongarra, M. Heroux, and P. Lusczek, "HPCG benchmark," http://hpcg-benchmark.org/, HPCG 3.0, 2015.

[27] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems," *Sandia Report, SAND2013-4744*, vol. 312, 2013.

[28] J. C. Phillips, Y. Sun, N. Jain, E. J. Bohm, and L. V. Kalé, "Mapping to irregular torus topologies and other techniques for petascale biomolecular simulation," in *Proc. of Int. Conf. for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 81–91.

[29] NASA Advanced Supercomputing Division, "NAS parallel benchmarks," http://www.nas.nasa.gov/publications/npb.html, accessed July, 2016, 2016.

[30] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[31] Open MPI Team, "Open MPI: Open source high performance computing," https://www.open-mpi.org/.

[32] Intel, Inc., "Intel MPI library," https://software.intel.com/en-us/intel-mpi-library.

[33] "Center for Computational Research," https://www.buffalo.edu/ccr.html, accessed July, 2016, 2016.

[34] S. Cheshire, "TCP performance problems caused by interaction between Nagle's algorithm and delayed ACK," May 2005, http://www.stuartcheshire.org/papers/NagleDelayedAck/.

[35] CRIU team, "CRIU," accessed July, 2016, http://criu.org/.

[36] M. O'Neill, "Cryopid2," accessed July, 2016, http://sourceforge.net/projects/cryopid2.

[37] G. Zheng, L. Shi, and L. V. Kalé, "FTC-Charm++: An in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *IEEE Int. Conf. on Cluster Computing (Cluster'04)*. IEEE Press, 2004, pp. 93–103.

[38] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, B. R. de Supinski, K. Mohror, and H. Pritchard, "Evaluating and extending user-level fault tolerance in MPI applications," *International Journal of High Performance Computing Applications*, 2016.

[39] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, and T. Brightwell, "rMPI: Increasing fault resiliency in a message-passing environment," Sandia National Laboratories, Tech. Rep. SAND2011-2488, Apr. 2011, http://prod.sandia.gov/techlib/access-control.cgi/2011/112488.pdf.

[40] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303–312, 2006.