System-level Transparent Checkpointing for OpenSHMEM

Rohan Garg[1⋆], Jérôme Vienne[2], and Gene Cooperman[1⋆]

[1] Northeastern University, Boston MA 02115, USA,
{rohgarg,gene}@ccs.neu.edu
[2] Texas Advanced Computing Center,
The University of Texas at Austin, TX 78758, USA,
viennej@tacc.utexas.edu

Abstract. Fault tolerance is an active area of research for OpenSHMEM programs. In this work, we present the first approach using system-level transparent checkpointing. This complements an existing approach based on application-level checkpointing. Application-level checkpointing has advantages for algorithm-based fault tolerance, while transparent checkpointing can be invoked by the system at an arbitrary time. Unlike the earlier application-level work of Hao et al., this system-level approach creates checkpoint images in stable storage, thus enabling restart at a later time or even process migration. An experimental evaluation is presented using NAS NPB benchmarks for OpenSHMEM. In order to support this work, The design of DMTCP (Distributed MultiThreaded CheckPointing) was extended to support shared memory regions in the absence of virtual memory.

Keywords: Checkpointing; fault tolerance; OpenSHMEM; process migration

1 Introduction

Checkpoint-restart is an area of research with a long history Work in this area has largely been split according to two approaches: *system-level* checkpointing and *application-level* checkpointing. System-level checkpointing typically is also *transparent*, in that it can be invoked by an external system service or by the operating system. Application-level checkpointing can also support transparent checkpointing by interposing on existing libraries.

This work presents the first system-level checkpointing solution for OpenSHMEM [9, 18]. The DMTCP (Distributed MultiThreaded CheckPointing) platform [2] is used in this approach. DMTCP directly supports checkpointing of distributed computations. This contrasts with a previous application-level approach to checkpointing by Hao et al. [15], which relies on interposing on the OpenSHMEM runtime library itself.

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

In principle, an alternative approach would be to use an implementation of OpenSHMEM on top of MPI, and then invoke system-level checkpointing of OpenSHMEM through checkpointing of the underlying MPI checkpoint-restart service. However, this is not feasible, since the current MPI implementations delegate to BLCR [10, 17] for checkpointing of a single process, and BLCR does not support the POSIX SysV shared memory objects on which most OpenSHMEM implementations depend. See Section 4 for a fuller discussion.

The barriers to supporting system-level checkpointing for OpenSHEM can be understood by reviewing the primary features of OpenSHMEM [18]. The OpenSHMEM standard is motivated by at least three extensions from shared memory between processes on a single computer to shared memory between computers on distributed hardware. SysV system calls such as shmget() and semop() must be extended to distributed hardware. And an RDMA-like technology such as InfiniBand must be used to efficiently support one-sided communication.

Specifically, the difficulties of supporting OpenSHMEM with a traditional checkpoint-restart package are three-fold.

1. shared memory objects (e.g., shmget() in SysV) were generalized from POSIX system calls on one computer to distributed hardware.
2. InfiniBand or a related RDMA technology is required. The OpenSHMEM standard [18] insists on the importance of one-sided communication: "The key feature of OpenSHMEM is that data transfer operations are one-sided in nature." [18, Sect. 2]. InfiniBand provides this.
3. synchronization primitives are generalized from APIs for inter-thread and inter-process communication [18, Sect. 8.5 and 8.8].

As mentioned earlier, an MPI-based checkpoint-restart approach relies on BLCR. Unfortunately, BLCR does not support either of items 1 or 3 above. Case 2 is supported by the various checkpoint-restart services of different MPI implementations. But Case 2 is not directly supported by a checkpointing package itself.

A significant barrier to using the DMTCP checkpointing system was the inability of DMTCP to support large shared memory regions on systems that lack virtual memory. Typically, supercomputers do not support virtual memory. An important contribution of the current work is extending the design of DMTCP to support large shared memory regions in the absence of virtual memory (see Section 3). Typical OpenSHMEM implementations require this, due to their use of SysV shared memory objects.

One can also contrast the advantages and disadvantages of the current work with the prior checkpointing work of Hao et al. [15]. Hao et al. copy the shared memory region along with privately mapped memory to the RAM of a peer process during runtime. In doing so, they protect against a single computer node failure, an important failure mode to be considered in the future exascale generation. In contrast, the current work saves into stable storage (typically a Lustre filesystem, on a supercomputer) at checkpoint time. This has the advantage that the current work supports migration of an OpenSHMEM computation to a new cluster, as well as saving a computation for restart on the same cluster at a later

time — for example, for long-running jobs on a batch system where the batch queue limits users to a maximum runtime slot of 24 hours.

The current work is based on the reference implementation of OpenSHMEM, on top of the MVAPICH2 implementation of MPI. Thus, this work also relies on the ability of DMTCP to directly checkpoint MVAPICH2. (DMTCP treats MVAPICH2 like any other distributed application, and does not rely on any MPI-specific information.)

Finally, because DMTCP does not depend on any MPI implementation, the result of this work opens the way for future support for hybrid MPI+OpenSHMEM codes. For example, MVAPICH2-X [27] provides advanced MPI features and a unified high-performance runtime for both MPI and PGAS programming models on InfiniBand clusters. MVAPICH2-X used all optimized features for communications and memory resources on Infiniband Cluster provided by the MPI library MVAPICH2 [19, 26] to improve the performance and scalability of communication on PGAS programming models [23, 22]. MVAPICH2-X supports multiples PGAS models such as Unified Parallel C and UPC++ (based on Berkeley UPC 2.20.0), OpenSHMEM (based on the OpenSHMEM reference implementation 1.0h) and Coarray Fortran (CAF) (based on Houston CAF implementation 3.0.39).

The rest of this paper is organized as follows. Section 2 briefly reviews the internals of DMTCP. Section 3 describes the places in which DMTCP needed to be extended in order to support the features of OpenSHMEM in a user program. Section 4 presents the related work. Section 5 presents an experimental evaluation, which was executed on the Stampede supercomputer at the Texas Advanced Computing Center (TACC). Section 6 then offers a conclusion and the plans for future work.

2 Review of Checkpointing

The architecture of DMTCP is described in Figure 1. A centralized DMTCP coordinator process accepts requests for checkpointing. Upon checkpoint, it sends a checkpoint message to a checkpoint thread within each user process. The checkponit thread "quiesces" the user threads, interrogates the kernel for state (e.g, open file descriptors and file offsets), and then copies the memory to a checkpoint image file. There is one checkpoint image file for each user process. See [2] for more details.

The original version of DMTCP supported only TCP-based sockets. Later, Cao et al. added support for checkpointing InfiniBand without the need to first disconnect an MPI computation from the network [8].

Two areas of novelty that are not reported elsewhere are the ability of DMTCP to checkpoint UNIX domain sockets and the ability to use leader election in order to checkpoint to correctly restore a single shared copy of a shared memory region, rather than restoring separate private memory regions on restart (one memory region for each process, or PE in the context of OpenSHMEM).

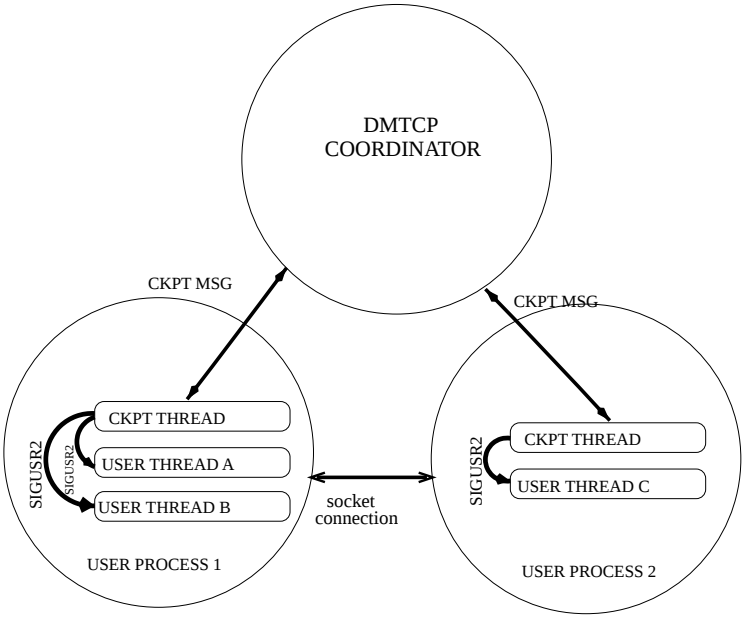

Fig. 1. The distributed architecture of DMTCP

3 Design modification of DMTCP to Support OpenSHMEM

The design of DMTCP had to be extended in three areas in order to support both checkpointing of modern MPI implementations and checkpointing of OpenSHEM. The three areas are UNIX domain sockets, SysV shared memory objects, and InfiniBand. The addition of support for InfiniBand is reported elsewhere [8]. This work describes the design of the first two capabilities.

UNIX domain sockets The original DMTCP design in 2009 [2] was sufficient to support the MPI implementations at that time. However, those earlier MPI implementations generally did not use UNIX domain sockets, and could be configured so as to avoid the use of shared memory regions for communication.

The design of support for UNIX domain sockets is similar to the TCP socket support reported in [2]. UNIX domain sockets allow one to pass a file descriptor from one process to another within the same Linux host. As with TCP sockets, one sends a "cookie" (a unique 64-bit value) through the UNIX domain socket. When the receiver reads it on the UNIX domain socket, it is known that there is no more data in the network.

SysV shared memory objects Second, the DMTCP design was extended to support SysV shared memory objects. The original DMTCP design [2] supported only BSD-style shared memory regions (using mmap and "MAP_SHARED").

Recently, support for SysV shared memory objects was added in order to support more recent MPI implementations.

Unfortunately, the design of SysV shared memory for MPI did not extend to support OpenSHMEM. OpenSHMEM requires support for large shared memory regions created by the user's application. In contrast, MPI directly creates only small shared memory regions internal to the MPI library itself, as an accelerator for communication between distinct MPI processes on the same host.

The DMTCP design depends on delegating to a single-process checkpointing package, under the name of MTCP. Each MTCP instance saves *every* shared memory region within that process, and later restore *every* shared memory region on restart. It is only at a later stage that DMTCP employs a leader election strategy to: (i) discard duplicate shared memory regions not owned by the leader; (ii) embed the leader's shared memory region within a SysV shared memory object; and (iii) send the newly created SysV shared memory object from the leader's process to all other processes on the same host.

While this approach works in most common cases, it fails when for large shared-memory areas. During the initial stages of restart, each process has mapped the shared memory region as a private region. Where virtual memory is available, this is not a problem. But on a supercomputer such as Stampede in our case, there is typically no support for virtual memory. This is because virtual memory normally resides on a hard disk, and supercomputer compute nodes generally do not have any local disks. Paging to a remote storage node on a supercomputer would produce an unacceptable performance penalty.

In order to support checkpointing of SysV shared memory regions in the absence of virtual memory, an alternative strategy was created. *Every* shared memory region is created initially as a region of zero pages. In Linux, zero pages do not require significant resources, and are easily supported even in the absence of virtual memory.

At checkpoint time, the MTCP component continues to write individual copies of the shared memory region into the process-specific checkpoint image file. But at the time of restart, instead of reading back into RAM the data of the shared memory region, MTCP simply writes the filename of the checkpoint image file for that process, and the file offset and size of the shared memory region in question. This information is written only into the first page of shared memory, and the remaining region remains as zero pages.

Finally, the same leader election strategy can be used for restart as with the existing SysV shared memory support. But in this case, the leader does not have the shared memory data resident in RAM. Instead, the leader reads the shared memory data into RAM. only at this late stage of restart, and after an appropriate host-wide barrier. All other processes wait while the leader reads the shared memory data. While this makes restart slower, this is generally acceptable, since checkpointing is the common operation, and restart is the rare operation.

OpenSHMEM and the hardware cache One of the weaknesses of the current approach concerns the OpenSHMEM support for data cache control, i.e., "mechanisms to exploit the capabilities of hardware cache". This is *not* provided by

DMTCP since that requires operating system extensions either to POSIX or to common Linux systems mechanisms such as the proc filesystem. An alternative approach that directly supports the abstractions of the OpenSHMEM library, such as [15], has the potential to use the OpenSHMEM API to save and restore information about the capabilities of the hardware cache.

4 Related Work

The OpenSHMEM standard is described in [9, 18]. Research in the area of Checkpoint-Restart for OpenSHMEM and other PGAS models is still sparse. In 2011, Ali et al. [1] proposed an application-specific fault tolerance mechanism. They achieved fault-tolerance using redundant communication and shadow copies. Hao et al. [15, 16] have proposed a more generic approach based on User Level Fault Mitigation (ULFM) using shadow memory in which the shared memory regions of peers are backed up by peers. The user code is responsible for invoking a checkpoint and for restoring correct operation during a restart.

An important distinction between the approach of Hao et al. [15] and the current work is that Hao et al. copy the shared memory region along with privately mapped memory to a peer process during runtime. This places added pressure on the network fabric and on the RAM. (The latter is significant since supercomputers typically do not support virtual memory.) In the current work, the shared memory region and privately mapped memory are copied to stable storage (often a Lustre filesystem on a supercomputer). This places added pressure on the Lustre filesystem at the time of checkpoint. Thus, each strategy has its separate advantages and problems.

Of course, a second important distinction is that the approach of Hao et al. directly support User Level Fault Mitigation (ULFM), while the current work does not directly support such a strategy.

Multiple MPI libraries support SHMEM parallel programming model. Open MPI [12] supports OpenSHMEM since version 1.7.5. In [14], Hammond et al. introduced OSHMPI [13], another implementation of SHMEM over MPI taking advantages of MPI-3 one-sided communication. As DMTCP is doing a transparent checkpoint restart, all these MPI implementations can be checkpointed and restarted transparently.

Since some implementations of OpenSHMEM are built on top of MPI, it is important to also discuss approaches to checkpointing MPI. As described earlier, such approaches split into an application-specific and system-level approach. For application-level checkpointing of MPI one notes [6, 7]. These packages provide hooks by which scientific applications on top of MPI can easily build their own checkpoint-restart routines. Such solutions add complexity at the petascale level, since they are not transparent to the end programmer.

For system-level checkpointing of OpenSHMEM, it would be tempting to employ an OpenSHMEM built on top of MPI, and then checkpoint the underlying MPI. Unfortunately, all of the checkpoint-restart services of current MPI implementations are built on top of BLCR [10, 17]. BLCR does not support the

SysV IPC objects. In particular, it does not support the POSIX-standard SysV shared memory (shm) objects [4].

Many MPI implementations provide a checkpoint-restart service based on BLCR. At the time of checkpoint, the MPI checkpoint-restart service detaches from the network, and then invokes BLCR as a single-process checkpointing utility for the individual processes. Among the MPI implementations using BLCR are OpenMPI [20], LAM/MPI [30], MPICH-V [5], and MVAPICH2 [11].

As stated above, BLCR does not support SysV shared memory objects. Hence, there is a problem if an OpenSHMEM implementation uses SysV shared memory objects (which is a common choice on a POSIX platform), and if the OpenSHMEM implementation is implemented on top of MPI. When a checkpoint is requested, the request will be passed to the checkpoint-restart service of the underlying MPI, which will delegate to BLCR. The BLCR FAQ states that "Such [SysV ipc] resources are silently ignored at checkpoint time and are not restored."

Finally, DMTCP (Distributed MultiThreaded CheckPointing) [2] provides checkpointing for general distributed computations, independently of MPI. There have also been at least three other checkpoint-restart systems that are independent of MPI and still able through Linux kernel modules to checkpoint distributed computations [21, 25, 24, 31]. However, none of these latter three appear to be under active development, and so their details are not discussed here.

Even though DMTCP operates independently of MPI, the OpenSHMEM reference implementation being used does depend on MPI. For this reason, DMTCP is checkpointing both OpenSHMEM and the MVAPICH implementation of MPI in the experiments.

5 Experimental Evaluation

5.1 Experimental Setup

The experiments have been conducted on TACC's Stampede supercomputer. Stampede is currently the # 12 supercomputer on the top500 list [32] (as of June, 2016). Stampede contains 6400 dual-socket eight-core Sandy-Bridge E5-2680 server nodes with 32 GB of memory, called "compute nodes", and 16 quad-socket eight-core Sandy-Bridge E5-4650 server nodes at 2.7 GHz with 1 TB of memory, called "large memory nodes". The nodes are interconnected by InfiniBand HCAs in FDR mode [33] and the operating system used is CentOS 6.4 with Linux kernel 2.6.32-431.el6. Experiments use the Lustre parallel filesystem version 2.5.5 on Stampede.

To do this evaluation, we use the Intel compiler version 13.0.2.146 on Stampede with the OpenSHMEM library. See [23] for a comparison of different OpenSHMEM implementations on Stampede. For the evaluation, we use a port of the NAS Parallel Benchmarks (NPB) to OpenSHMEM [29]. The NAS Parallel Benchmarks for MPI are already well-documented and widely used as a benchmark [28, 34, 3]. It consists of a suite of parallel workloads designed to evaluate

performance of various hardware and software components of a parallel computing system.

5.2 Scalability

For evaluating performance, we measure the runtime overhead, the checkpoint overhead, and the restart overhead as we scale up. The NAS BT and SP benchmarks were used to measure the scalability of DMTCP.

Table 1 shows the number of nodes used and the number of processes per node for a given number of processes (PE's). The same configuration was used for all the experiments.

Num of PE's	Num of Nodes	Processes per node
4	2	2
9	3	3
16	4	4
36	6	6
64	8	8
121	11	11
256	16	16

Table 1. Distribution of processes among nodes

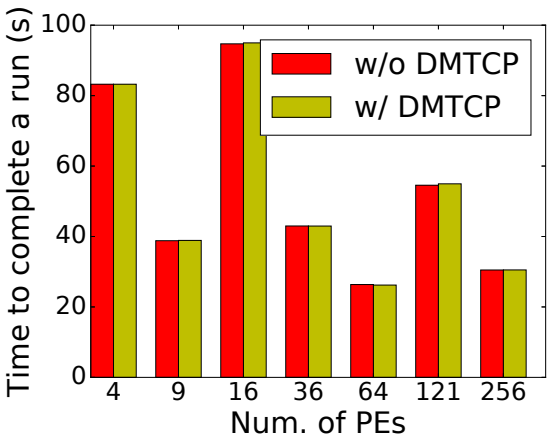

Fig. 2. Runtime overhead on OpenSHMEM NAS BT benchmark with DMTCP. BT class A was used for 4, and 9 PE's. BT class B was used for 16, 36, and 64 PE's. BT class C was used for the runs with 121 and higher PE's.

Figure 2 shows the runtime overhead imposed by DMTCP. The runtime overhead is less than 1 % in all cases. DMTCP's wrapper functions impose a negligible runtime overhead and the cost is further amortized over the duration of the run.

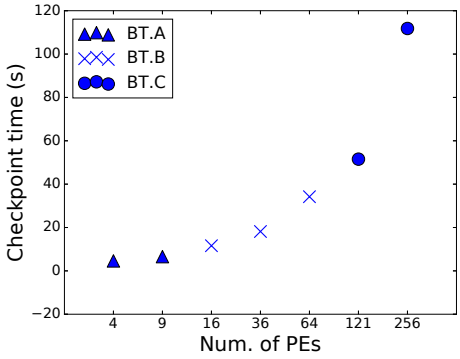

Fig. 3. Checkpoint times for OpenSHMEM NAS BT benchmark with DMTCP. BT class A was used for 4, and 9 PE's. BT class B was used for 16, 36, and 64 PE's. BT class C was used for the runs with 121 and higher PE's.

For a given number of PE's, all the runs — with and without DMTCP — were conducted on the same set of nodes to reduce the variability due to network topology and traffic.

Average checkpoint times for the NAS BT benchmark are shown in Figure 3. Five successive checkpoints were taken for a given number of processes on the same set of nodes.

Figure 4 shows the average checkpoint times for the NAS SP benchmark. Five successive checkpoints were taken for a given number of processes on the same set of nodes. The checkpoint times include the cost of synchronizing the state of distributed processes, including communications with the central checkpointing coordinator.

For both benchmarks, BT and SP, checkpoint times grow linearly with the total amount of checkpoint image data (see Figures 5 and 6). At the largest scale, 256 processes, the total data written to the disk is 2.2 TB, with an effective bandwidth of 20 GB per second.

In all the cases, the checkpoint times are dominated by the time to write the checkpoint data to stable storage, and the cost for checkpointing the state of the application is negligible.

The checkpoint image sizes for a single process for NAS benchmarks BT and SP are shown in Figures 5 and 6, respectively.

Note that the checkpoint image size is directly proportional to the number of processes sharing a computer node. For a given number of total processes, the number of processes sharing a node is shown in Table 1.

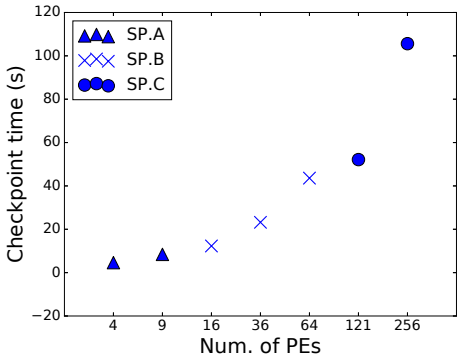

Fig. 4. Checkpoint times for OpenSHMEM NAS SP benchmark with DMTCP. SP class A was used for 4, and 9 PE's. SP class B was used for 16, 36, and 64 PE's. SP class C was used for the runs with 121 and higher PE's.

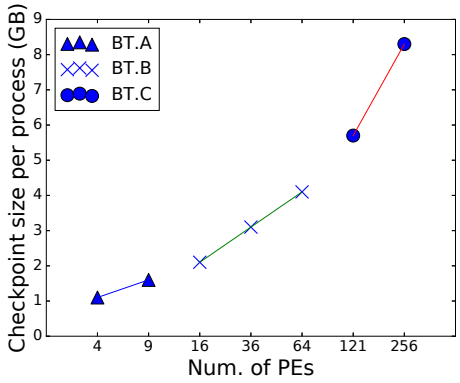

Fig. 5. Uncompressed checkpoint image sizes for OpenSHMEM NAS BT benchmark with DMTCP. BT class A was used for 4, and 9 PE's. BT class B was used for 16, 36, and 64 PE's. BT class C was used for the runs with 121 and higher PE's.

We observe that largest component, 90-97 %, in a checkpoint image is an OpenSHMEM shared-memory region, which is used for intra-node communication. Each process on a node contributes roughly 0.5 GB to the shared-memory region. The rest of the checkpoint image contains process's private memory regions.

Figures 7 and 8 show the restart times for the NAS BT and SP benchmarks, respectively, at different scales. The restart times include the cost of synchronizing the state of distributed processes, including communications with the central checkpointing coordinator.

At the scale of 16 processes and beyond, the total memory footprint of the checkpoint images required per node exceeds the available RAM on each node,

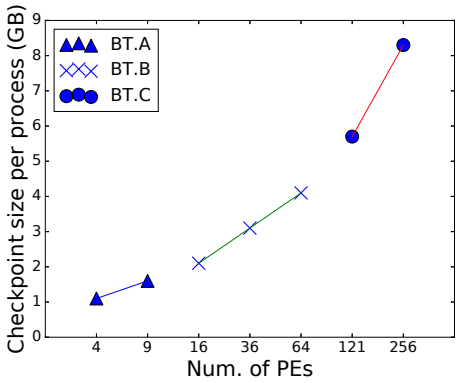

Fig. 6. Uncompressed checkpoint image sizes for OpenSHMEM NAS SP benchmark with DMTCP. SP class A was used for 4, and 9 PE's. SP class B was used for 16, 36, and 64 PE's. SP class C was used for the runs with 121 and higher PE's.

32 GB, and hence, it's not possible to directly map in the data from the checkpoint image. On restart, while restoring the memory of a process, DMTCP identifies the OpenSHMEM shared-memory memory region in its checkpoint image, reads in rest of the private data in to the memory of the process, and finally maps in the shared-memory region as *MAP_SHARED* in to the process's memory.

The restart times are nearly twice as large compared to the checkpoint times. We speculate this is because while writing the checkpoint images, Lustre buffers the checkpoint data. On restart, any buffered data must first be synchronized to the disk, transferred to each node, and then read in to the memory of each process.

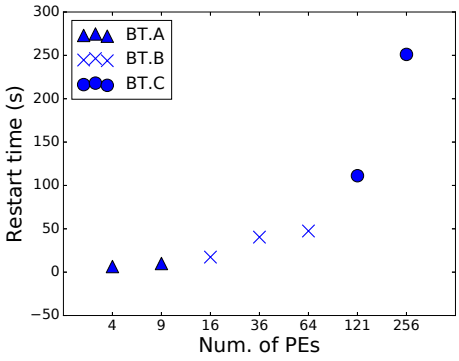

Fig. 7. Restart times for OpenSHMEM NAS BT benchmark with DMTCP. BT class A was used for 4, and 9 PE's. BT class B was used for 16, 36, and 64 PE's. BT class C was used for the runs with 121 and higher PE's.

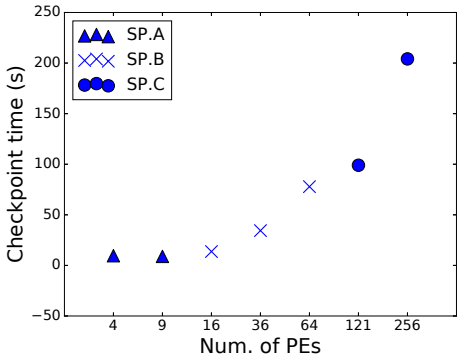

Fig. 8. Restart times for OpenSHMEM NAS SP benchmark with DMTCP. SP class A was used for 4, and 9 PE's. SP class B was used for 16, 36, and 64 PE's. SP class C was used for the runs with 121 and higher PE's.

6 Conclusion and Future Work

A system-level approach to checkpointing OpenSHMEM was presented. This approach enables one to save the state of a computation to stable storage at checkpoint time. This contrasts with the previous approach of Hao et al., in which they save to the RAM of a remote peer computer. The latter approach supports fault tolerance in the case of a single host failing, and has the potential for a fast restart, since only one computer node must be restored. In contrast, the current approach has the capability of saving the state of an entire computation for restart at a later time on the same cluster, or else for migration to a new cluster.

The current work saves the state of the shared memory region of each process to stable storage. In this case (with 16 cores supporting 16 processes (16 PEs), this can potentially place a large burden on the Lustre filesytem by saving 16 identical copies of the shared memory regions on a single host, when executing at very large scale. While this was not observed to incur significant performance penalty at the medium scale of the current experiments, it is intended to employ a leader election strategy early (at checkpoint time) in a future implementation. In this way only one copy of each shared memory region will be saved on a single host. This will significantly reduce the time to write to back-end storage. (Note that current OpenSHMEM implementations do not appear to replicate shared memory regions across hots, and so deduplication on a single host is deemed to be sufficient for good performance.)

References

1. Ali, N., Krishnamoorthy, S., Govind, N., Palmer, B.J.: A Redundant Communication Approach to Scalable Fault Tolerance in PGAS Programming Models. IEEE

Computer Society, Los Alamitos, CA, United States(US). (Feb 2011)
2. Ansel, J., Arya, K., Cooperman, G.: DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In: IEEE Int. Symp. on Parallel and Distributed Processing (IPDPS). pp. 1–12. IEEE Press (2009)
3. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, D., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS Parallel Benchmarks. The Intl. Journal of Supercomputer Applications 5(3), 63–73 (Fall 1991)
4. BLCR team: BLCR frequently asked questions (for version 0.8.5) (accessed June, 2016), `https://upc-bugs.lbl.gov/blcr/doc/html/FAQ.html#limitations`
5. Bouteiler, A., Herault, T., Krawezik, G., Lemarinier, P., Cappello, F.: MPICH-V Project: a Multiprotocol Automatic Fault Tolerant MPI. International Journal of High Performance Computing Applications 20, 319–333 (2006)
6. Bronevetsky, G., Marques, D., Pingali, K., Rugina, R., McKee, S.A.: Compiler-Enhanced Incremental Checkpointing for OpenMP Applications. In: Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS) (May 2009)
7. Bronevetsky, G., Marques, D., Pingali, K., Stodghill, P.: Automated Application-level Checkpointing of MPI Programs. In: PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 84–94. ACM Press, New York, NY, USA (2003)
8. Cao, J., Kerr, G., Arya, K., Cooperman, G.: Transparent Checkpoint-Restart over InfiniBand. In: Proc. of the 23rd Int. Symp. on High-performance Parallel and Distributed Computing. pp. 13–24. ACM Press (2014)
9. Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., Smith, L.: Introducing OpenSHMEM: SHMEM for the PGAS Community. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model. pp. 2:1–2:3. PGAS'10, ACM, New York, NY, USA (2010)
10. Duell, J., Hargrove, P., Roman, E.: The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart (BLCR). Tech. Rep. LBNL-54941, Lawrence Berkeley National Laboratory (2003)
11. Gao, Q., Yu, W., Huang, W., Panda, D.K.: Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand. In: ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing. pp. 471–478. IEEE Computer Society, Washington, DC, USA (2006)
12. Graham, R.L., Woodall, T.S., Squyres, J.M.: Open MPI: A Flexible High Performance MPI. In: Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics. Poznan, Poland (September 2005)
13. Hammond, J.: OSHMPI (06 2016), `https://github.com/jeffhammond/oshmpi`
14. Hammond, J.R., Ghosh, S., Chapman, B.M.: Implementing OpenSHMEM using MPI-3 One-sided Communication. In: OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools, pp. 44–58. Springer (2014)
15. Hao, P., Pophale, S., Shamis, P., Curtis, T., Chapman, B.: Check-Pointing Approach for Fault Tolerance in OpenSHMEM. In: OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies: Second Workshop, OpenSHMEM 2015, Annapolis, MD, USA, August 4-6, 2015. Revised Selected Papers. vol. 9397, pp. 36–52. Springer (2015)
16. Hao, P., Shamis, P., Venkata, M.G., Pophale, S., Welch, A., Poole, S., Chapman, B.: Fault Tolerance for OpenSHMEM. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models. pp. 23:1–23:3. PGAS '14 (2014)

17. Hargrove, P., Duell, J.: Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. Journal of Physics Conference Series 46, 494–499 (Sep 2006)
18. High Performance Computing Tools group at the University of Houston, Extreme Scale Systems Center, Oak Ridge National Laboratory: OpenSHMEM application programming interface (version 1.3) (accessed June, 2016), http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.3.pdf
19. Huang, W., Santhanaraman, G., Jin, H., Gao, Q., Panda, D.: Design and Implementation of High Performance MVAPICH2: MPI2 over InfiniBand (May 2007)
20. Hursey, J., Squyres, J.M., Mattox, T.I., Lumsdain, A.: The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. In: Proceedings of the 21^{st} IEEE International Parallel and Distributed Processing Symposium (IPDPS) / 12^{th} IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems. IEEE Computer Society (March 2007)
21. Janakiraman, G., Santos, J., Subhraveti, D., Turner, Y.: Cruz: Application-Transparent Distributed Checkpoint-Restart on Standard Operating Systems. In: Dependable Systems and Networks (DSN-05). pp. 260–269 (2005)
22. Jose, J., Hamidouche, K., Zhang, J., Venkatesh, A., Panda, D.: Optimizing Collective Communication in UPC (May 2014)
23. Jose, J., Zhang, J., Venkatesh, A., Potluri, S., Panda, D.K.D.: A Comprehensive Performance Evaluation of OpenSHMEM Libraries on InfiniBand Clusters. In: Openshmem and Related Technologies. Experiences, Implementations, and Tools, pp. 14–28. Springer (2014)
24. Laadan, O., Nieh, J.: Transparent Checkpoint-Restart of Multiple Processes for Commodity Clusters. In: 2007 USENIX Annual Technical Conference. pp. 323–336 (2007)
25. Laadan, O., Phung, D., Nieh, J.: Transparent Networked Checkpoint-Restart for Commodity Clusters. In: 2005 IEEE International Conference on Cluster Computing. IEEE Press (2005)
26. Laboratory, N.B.C.: MVAPICH2 (06 2016), http://mvapich.cse.ohio-state.edu/
27. Laboratory, N.B.C.: MVAPICH2-X (06 2016), http://mvapich.cse.ohio-state.edu/
28. NASA Advanced Supercomputing Division: NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html (accessed Apr, 2016)
29. Pophale, S., Nanjegowda, R., Curtis, T., Chapman, B., Jin, H., Poole, S., Kuehn, J.: OpenSHMEM Performance and Potential: A NPB Experimental Study. In: The 6th Conference on Partitioned Global Address Space Programming Models (PGAS'12). Citeseer (2012)
30. Sankaran, S., Squyres, J.M., Barrett, B., Sahay, V., Lumsdaine, A., Duell, J., Hargrove, P., Roman, E.: The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. International Journal of High Performance Computing Applications 19(4), 479–493 (2005)
31. Sudakov, O.O., Meshcheriakov, I.S., Boyko, Y.V.: CHPOX: Transparent Checkpointing System for Linux Clusters. In: IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications. pp. 159–164 (2007), software available at http://freshmeat.net/projects/chpox/
32. TOP500 supercomputer sites (Jun 2016), http://top500.org/list/2016/06/
33. Vienne, J., Chen, J., Wasi-Ur-Rahman, M., Islam, N.S., Subramoni, H., Panda, D.K.: Performance Analysis and Evaluation of InfiniBand FDR and 40GigE RoCE on HPC and Cloud Computing Systems. In: Hot Interconnects. pp. 48–55 (2012)

34. Wong, F.C., Martin, R.P., Arpaci-Dusseau, R.H., Culler, D.E.: Architectural Requirements and Scalability of the NAS Parallel Benchmarks. In: Supercomputing (1999)

Acknowledgment

We would like to thank both Kapil Arya and Jiajun Cao for many useful discussions on the internals of DMTCP, and the design of those internal components. 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.