Vol.2011-ARC-197 No.18
Vol.2011-HPC-132 No.18
2011/11/28

Towards an Asynchronous Checkpointing System

KENTO SATO,[†1] ADAM MOODY,[†2] KATHRYN MOHROR,[†2]
TODD GAMBLIN,[†2] BRONIS R. DE SUPINSKI,[†2]
NAOYA MARUYAMA[†1] and SATOSHI MATSUOKA [†1,†3]

The overall failure rate of HPC systems is increasing because the number of components is growing. Checkpoint/Restart, the most common technique to tolerate these faults, enables an application to restart from the last checkpoint even if a failure happens while the application is running. However, writing large checkpoint files may impact application runtime, depending on the bandwidth of the file systems to which checkpoints are written. To minimize the impact, we propose an asynchronous checkpointing system to write checkpoints to the file system in the background. This system uses extra nodes to drain a checkpoint from compute nodes using RDMA (Remote Direct Memory Access) to minimize CPU usage. Our preliminary evaluation shows that our asynchronous checkpointing system reduces checkpointing impact with runtime increases of CPU-bound applications under 1% compared to not checkpointing to a parallel file system.

1. Introduction

The computational power of High Performance Computing (HPC) systems is growing exponentially, driven by compute-intensive scientific simulations. Simulation enables researchers to conduct fine-grained scientific simulations. However, the overall failure rate of HPC systems increases with the size of the HPC system. For example, TSUBAME2.0, ranking 5th in the Top500 list [1] (June 2011), experienced about 1,500 failures over the past one year rainging from a memory error to whole rack failure [2]. This means that a failure occurred every 6 hours. Moreover, in exascale systems, MTBF is projected to shrink to only a few minutes [3]. Without a viable resilience strategy, it will be impossible for an application to continue to run for even one day on such a large machine. Thus, resilience in HPC is becoming more important than ever as we plan for exascale systems.

†1 Tokyo Institute of Technology
†2 Lawrence Livermore National Laboratory
†3 National Institute of Informatics

Checkpointing is an indispensable fault tolerance technique, commonly used by HPC applications that run continuously for hours or days at a time. A *checkpoint* is a snapshot of application state that can be used to restart execution if a failure occurs. Generally, to survive a devastating failure, such as a multi-node or whole-rack failure, checkpoints are periodically written to stable, reliable storage, such as a parallel file system (PFS) [4–6]. For example, TSUBAME2.0 experienced a whole rack failure about 30 times over the past one year due to failures of switches or water cooling systems [2]. In addition, Coastal, Hera and Atlas clusters at Lawrence Livermore National Laboratory (LLNL), reported 191 failures out of 871 runs of a production application, a total of over 5 million node-hours. 15% of the failures requiered the application to restart from the checkpoint in the PFS [7]. Thus, applications are required to write their checkpoints to the PFS.

When checkpointing large-scale systems, tens of thousands of compute nodes write checkpoints to PFS, and low I/O throughput to the PFS becomes a bottleneck. Thus, the overhead of checkpointing becomes very large. However, the I/O throughput is usually unstable and varied over the execution, since the PFS is shared with other compute nodes and users, which makes performance estimation difficult.

Multi-level checkpointing is a promising approach for addressing these problems. This approach uses multiple storage levels, such as RAM, local disk and the PFS, according to the different degrees of resiliency and the cost of checkpointing in those storage levels. The Scalable Checkpoint/Restart (SCR) library[7,8] is a multi-level checkpointing implementation for MPI applications. SCR enables MPI applications to use storage distributed among a system's compute nodes and on the PFS to attain high checkpoint performance. By taking frequent, inexpensive node-local checkpoints, and less frequent, high-cost checkpoints to the PFS, applications can achieve high resilience and better efficiency. However, even with less frequent checkpointing to the PFS, overhead of writing to the PFS can still dominate overall application runtime. Moreover, because the huge numbers of compute nodes write checkpointings concurrently, large write operations burden the PFS and are themselves a major source of failures. Thus, it is critical to achieve reliable application runs while minimizing the overhead of checkpointing to the PFS and failure rates due to overburdening the PFS.

To reduce the checkpointing overhead and the workload to the PFS, we propose an asynchronous checkpointing system in which an application can continue to run while checkpoints are transferred to the PFS. We use agents running on additional nodes to

asynchronously transfer checkpoints from compute nodes to the PFS using Remote Direct Memory Access (RDMA). This approach reduces application overhead because the application does not have to block whie writing to the PFS, and RDMA transfers minimize CPU interference. In addition, since the additional nodes are independent from the compute nodes, the additional nodes can control I/O rate to reduce the probability of a failure in the PFS without affecting an application on compute nodes. In this paper, we focus on minimizing the checkpointing overhead of applications. Our preliminary experiments show that our asynchronous checkpointing system can impose less than 1% overhead on the runtime of a CPU-bound benchmark when using a sufficient number of additional nodes for transferring checkpoints.

2. The Scalable Checkpoint/Restart Library (SCR)

Our asynchronous checkpointing system is developed as an extension to the SCR library[7),8)] . and is intended to transfer SCR checkpoints to the PFS. The SCR library is an application-level checkpoint/restart library that has been used in production applications at LLNL. It enables MPI applications to use distributed storage, (including RAM disk, SSD, and the PFS) for multi-level checkpoint/restart. SCR is designed to support globally-coordinated checkpoints written primarily as a file per MPI process, a common checkpointing technique used by large-scale codes.

2.1 Checkpint/Restart Scheme

SCR uses distributed storage for checkpointing to increase reliability while reducing overhead. SCR first writes each checkpoint to local storage such as a RAM disk, a local disk, or an SSD. Generally, since older checkpoints are rarely used unless the subsequent checkpoints are corrupted, SCR caches a user-defined number of the most recent checkpoints in local storage and deletes older checkpoints. SCR also applies redundancy schemes, e.g. simple checkpoint copy to a partner node, to potentially recover checkpoints after a failure of one or more compute nodes. Moreover, for more crucial failures, SCR also periodically copies (flushes) a cached checkpoint to the PFS based on a specified redundancy scheme.

On a restart after a failure, SCR tries to restart from the most recent checkpoints in cache. If it cannot restart from the cached checkpoints, then SCR inspects the most recent checkpoint in the PFS. It then fetches and restarts from the checkpoint. Thus, when the application restarts, it must use the same number of processes as were used in the application run that wrote the checkpoint.

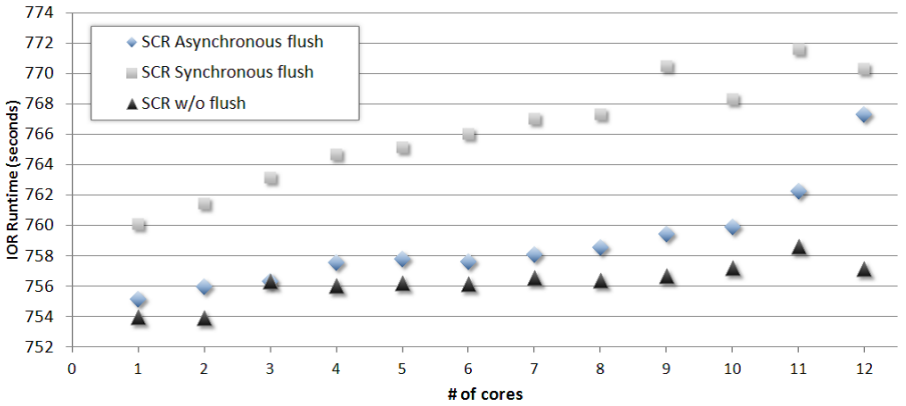

Fig. 1 IOR Runtime on a 12-core machine with SCR

2.2 SCR Flush

As mentioned, to withstand catastrophic failures, such as a rack-level failure and an outage in datacenters, SCR copies (flushes) a checkpoint from a node-local storage to a PFS with user-specified frequency. As an example of the flush, if a user specifies the checkpoint flush frequency as 10, SCR creates checkpoint directories on the specified path in the PFS, and flushes every 10 checkpoints ($10, 20, \ldots 10 \times n$ iterations). When an MPI application writes its checkpoint to node-local storage, it obtains the full path and file name the application must use to write the checkpoint through SCR library so that SCR knows which node-local checkpoints were written and what redundancy schemes were applied to them, and additionally which checkpoints should be flushed to the PFS. After the end of the checkpoint, the application is required to inform SCR that the current checkpoint is finished, and then SCR starts its flush operation. SCR supports two types of flush operations: synchronous and asynchronous. When SCR copies a checkpoint to the PFS synchronously, it blocks the application until the copy has completed. In large-scale computations on tens of thousands or more compute nodes, the total checkpoint size can reach multiple terabytes. Thus, an application may block for ten minutes or more while the copies complete, so synchronous flushes can application runtime. SCR also supports an asynchronous flush in which it starts the flush but immediately returns

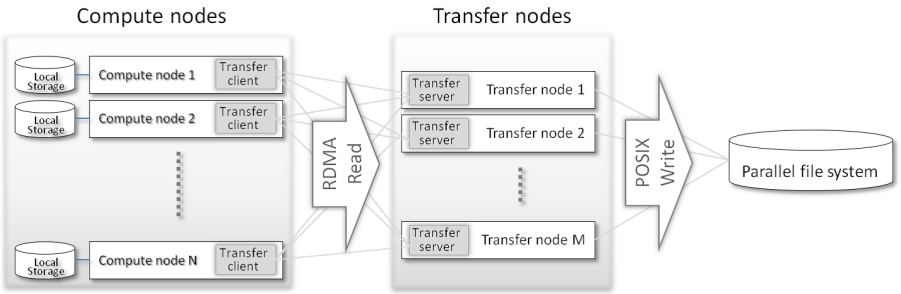

Fig. 2 Our asynchronous checkpointing system

control to the application. SCR then copies a cached checkpoint to the PFS. During the flush, an external and independent process is running in the background to copy the checkpoint to the PFS asynchronously, so the application is not blocked while SCR copies the cached checkpoint. At some later point, SCR checks whether the flush has completed properly. Thus, SCR guarantees that the flushed checkpoint is written to the PFS without missing checkpoint data.

Although SCR's asynchronous flush is expected to be effective, the external process uses CPU time, which impacts application runtime. Fig.1 shows the runtime of the IOR benchmark[9] using SCR with synchronous flush, asynchronous flush, and no flush. We used the Sierra machine at LLNL, a 1944-node Linux cluster with 12 cores per node. The result shows that the IOR runtime increases even if asynchronous flush is used, and especially when the full 12 cores per node are used by IOR. Motivated by this result from current SCR asynchronous flush implementation, we developed a more lightweight asynchronous checkpoint system that does not affect application runtime.

3. Asynchronous Checkpointing System

3.1 Architecture

Our asynchronous checkpointing system has two types of nodes: *compute nodes* and *transfer nodes* (Fig.2). The compute nodes are a group of nodes on which an application is executed. In addition, a *transfer client* process runs on each compute node. Once SCR finishes flushing a checkpoint, it calls a function of the transfer client library. The transfer client then sends a request to the *transfer server* to read a checkpoint. The transfer nodes are a group of nodes that read a checkpoint from compute nodes and write to the PFS. A transfer server process running on a transfer node uses RDMA to read a remote checkpoint data located in memory region on compute nodes and write the checkpoint to the PFS. Generally, each transfer node handles multiple compute nodes; the number of transfer processes our system employs depends on the number of compute nodes in the run. Because it uses additional nodes to transfer checkpoint data, our asynchronous checkpointing system has the following advantages:

(1) **Stable checkpint write performance:** A PFS is generally accessed concurrently by not only a large number of processes in a single application, but additionally by processes in other user's jobs. With our asynchronous checkpointing system, compute nodes write their checkpoints through transfer nodes. Thus, an application is not directly affected by spikes in the I/O workload of other users.

(2) **Minimal CPU usage:** On a flush operation, even asynchronous checkpointing can impact application runtime as shown in Fig.1. With RDMA, the asynchronous checkpointing system drains a checkpoint from compute nodes to a PFS while minimizing the impact on the application runtime.

(3) **Overall data center I/O balancing:** Checkpointing is one of the most I/O intensive operations, and one application's checkpoint bandwidth can affect other applications' I/O performance and potentially cause PFS failures. Since transfer nodes, which are independent from compute nodes, write checkpoints to a PFS, I/O control can be applied to solve the problem without directly throttling I/O rate in compute nodes.

In this paper, we focus on minimizing CPU usage and consider an I/O throttling technique for optimizing overall data center I/O as future work.

3.2 RDMA checkpoint transfers

We implemented an RDMA checkpoint transfer system for asynchronous checkpointing based on the SCR library. The existing SCR asynchronous flush reads a checkpoint from a local storage to a buffer and directly writes the checkpoint from the buffer to the PFS (Fig.3). We extended this implementation to drain a checkpoint from compute nodes to the PFS through a transfer node using RDMA transfers. The other checkpoint management, such as version, checkpoint location and redundancy scheme, relies on the original SCR library. Fig.4 describes the architecture. The transfer client and server processes run on compute nodes and transfer nodes, respectively. These processes transfer

Vol.2011-ARC-197 No.18
Vol.2011-HPC-132 No.18
2011/11/28

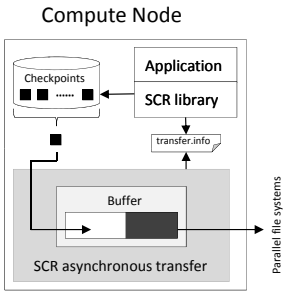

Fig. 3 Original SCR asynchronous flush

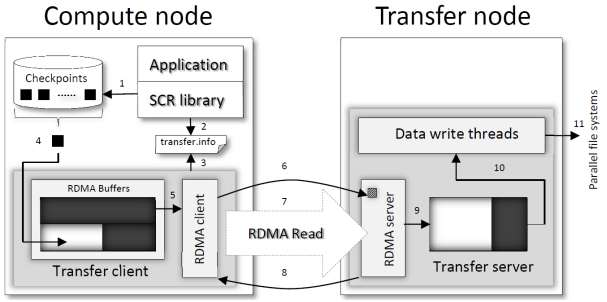

Fig. 4 Proposed asynchronous flush client/server using RDMA

and write checkpoints to the PFS.

We explain how a checkpoint on a compute node is written to the PFS through a transfer node. For simplicity, we assume that the SCR library writes checkpoints to local storage according to a particular redundancy scheme. The local storage can be RAM disk, a hard disk drive, or SSD (Step 1 in Fig.4). After several checkpoints are written to local storage, SCR writes transfer information into a file called *transfer.info* to request the transfer client to transfer the checkpoint according to specified flush frequency (Step 2). The transfer.info file includes the source and destination paths from and to which the checkpoint should be flushed. The transfer client process periodically checks the transfer.info file to see if SCR has issued the request (Step 3). If the transfer client detects the request in transfer.info, the transfer client process reads the checkpoint from the source path and writes to local RDMA buffer space (Step 4). Once the transfer client fills the buffer with checkpoint data, it calls an RDMA client function to send out the chunk of checkpoint data in the buffer (Step 5). Since the RDMA client function returns control immediately, the transfer client process can read the next chunk to one of the buffer entry in the buffer pool (Fig.4 shows the double-buffering case) while the RDMA client transfers checkpoint chunks to the transfer server. The RDMA client issues a request for the chunk transfer to an RDMA server (Step 6). When the RDMA server running on the transfer node receives a transfer request, an RDMA read request is issued to the transfer client to read the remote buffer space into a local buffer, which then sends an acknowledgement to the RDMA client. The RDMA client issues RDMA read requests until all chunks are sent to the transfer node (Step 7-9). Finally, the data writer threads write the checkpoint to the PFS in parallel with RDMA reads by the RDMA server (Step 10-11).

An RDMA operation can read or write remote memory regions up to only a few MB of data at a time. Therefore, a checkpoint chunk is also divided into smaller chunks and the RDMA server reads the remote chunks one by one. To transfer checkpoints from thousands of compute nodes with fewer transfer nodes, a transfer server can concurrently handle RDMA requests from multiple transfer clients. However, a large amount of incoming checkpoint data from multiple compute nodes can overflow a buffer in a transfer node. To avoid buffer overflow, if buffered checkpoint data exceeds a specified buffer size limit, the transfer nodes throttle the RDMA read rate to balance between incoming checkpoint data from compute nodes and outgoing data to the PFS. Thus, to avoid incoming data rate from being saturated by outgoing data rate, an appropriate number of transfer nodes needed to be determined.

4. Evaluation

We developed and evaluated the asynchronous checkpointing system on the Sierra cluster at LLNL using a parallel file system designed to deliver a peak performance of 30GB/s. Each Sierra node has two 2.8 GHz 6-core Intel Xeon 5660 processors (12 cores in total) and 24GB of memory, and nodes share a QLogic IBA7322 QDR InfiniBand HCA (4X) interoconnect. We used the IOR benchmark[9] developed by LLNL, which measures I/O performance using POSIX, MPI I/O, or HDF5 with several access patterns. IOR has a main loop that executes read/write operations followed by a sleep-delay interval. The number of loop iterations is specified by the user along with the number of seconds for the delay interval. We ran IOR to write a 200 MB per-process ckeckpoint files to local storage through the HDF5 API. IOR repeats this write operation three times with a 250 second sleep interval between writes, for 750 seconds of delay in total. We wrote node-local checkpoints to RAM disk (/tmp) and flushed them to a Lustre PFS [4] . To emulate a compute-bound scientific application writing checkpoints periodically, we replaced the sleep-delay with a CPU-intensive loop based on the $do\_work()$ function in the ATS (APART Test Suite) [10]. The $do\_work()$ function repeatedly generates a random index using the C $rand()$ function, copies a value of one array at the index to another array at the same index a specified number of times. We calibrated this computation

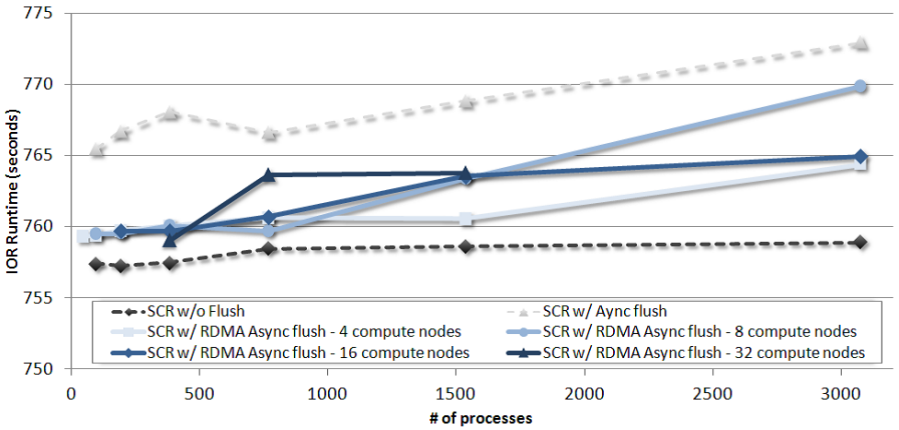

Fig. 5 IOR Runtime varying number of compute nodes

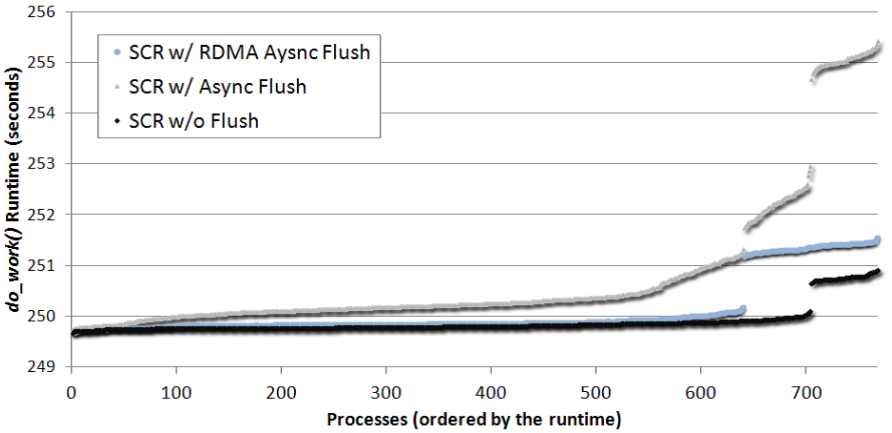

Fig. 6 The $do\_work()$ function runtime

time to take 250 seconds.

Fig.5 shows IOR runtime with an increasing number of MPI processes. The MPI processes occupy all cores on computes node, i.e., 12 MPI processes per compute node. With **SCR w/o Flush**, SCR writes checkpoints to local storage, but never flushes them to the PFS. With **SCR w/ Async flush**, SCR uses the original implementation with a 200MB buffer to flush each checkpoint to the PFS after IOR writes it. **SCR w/ RDMA Async flush** is the proposed asynchronous checkpointing with two 200MB buffers in the transfer client and a 20GB buffer in the transfer server. We vary the number of compute nodes that a transfer node handles from 2 to 32 in increasing powers of 2. The first asynchronous flush in both **SCR w/ Async flush** and **SCR w/ RDMA Async flush** starts at the end of the first checkpoint to local storage. Thus, the first iteration is not affected by asynchronous checkpointing. As presented in Fig.5, asynchronous checkpointing with RDMA can flush checkpoints to the PFS with less runtime overhead that the original SCR asynchronous checkpointing. Since the asynchronous checkpointing uses RDMA to drain checkpoint data from compute nodes to the PFS, our system uses less CPU time than the original one and reduces the impact on the application.

Fig.6 shows a compute intervals by the $do\_work()$ function, which is sorted by the runtime, of each MPI process for the run with 768 processes. Processes causing runtime gaps are distributed across compute nodes equally, i.e. the differences are not due to a small set of defected or slower compute nodes. We found that the original asynchronous checkpointing scheme impacts $do\_work()$ runtime more than our new mechanism, and the runtime using our asynchronous checkpointing is much closer to that without flushes. We observed CPU usage of our asynchronous checkpointing process running on a compute node is only about 0.1-0.3% on average. As for the jump in runtime for **SCR w/o Flush**, we also observed that the Lustre RPC client process, which is called *ptlrpcd*, used 0.1% of CPU over the execution. We expect that the gap in runtime performance was caused by ptlrpcd.

Fig.5 also shows that the runtime with **SCR w/ RDMA Async flush** tends to increase with the number of computes nodes that a transfer node handles. In the current implementation, an RDMA client process polls a memory region to detect an acknowledgement from an RDMA server of completed RDMA read. Since a transfer node, which handles a larger number of compute nodes, take longer before sending the reply, the RDMA client inspects the memory region more often, which impacts the $do\_work()$ runtime more heavily.

We also investigated runtime differences with the different number of compute nodes that a transfer node handles. As presented in Fig.7, we observed that IOR runtime increases significantly with 64 compute nodes and is much longer than with **SCR w/ Async flush**. The time increases because one checkpoint flush starts before the prior

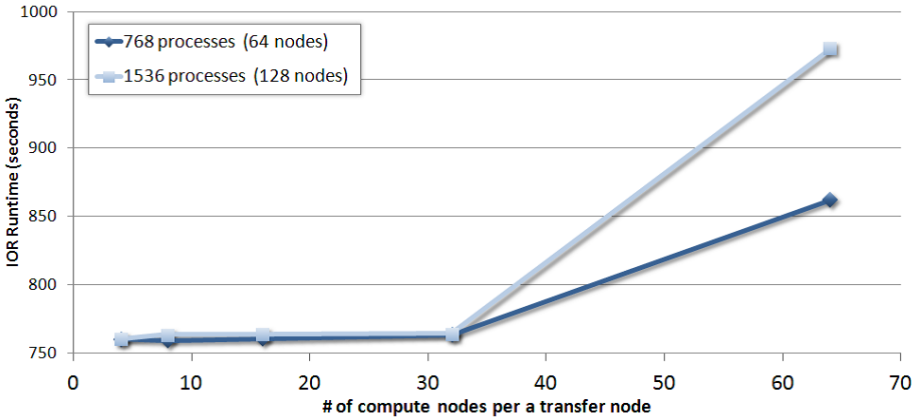

Fig. 7 IOR Runtime increasing number of compute nodes per transfer node

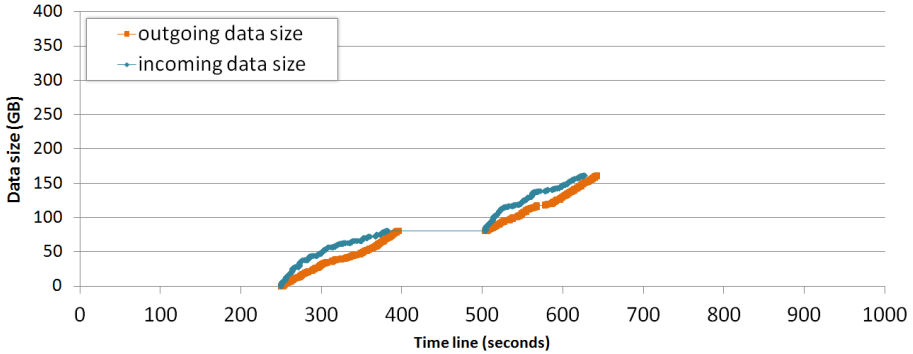

Fig. 8 incoming/outgoing data size of a transfer node's buffer with 32 compute nodes

flush completes when there many compute nodes handled by the transfer node. SCR does not start the next flushing untile the prior one completes. When multiple compute nodes share a transfer node, the throughput, at which a compute node can send checkpoint data to the transfer node, decreases accordingly. This can lead to flush operations not completing before the next one begins.

Fig.8 and 9 shows the impact of varying incoming and outgoing checkpoint data size in a transfer node that handles 32 and 64 compute nodes respectively. As shown in

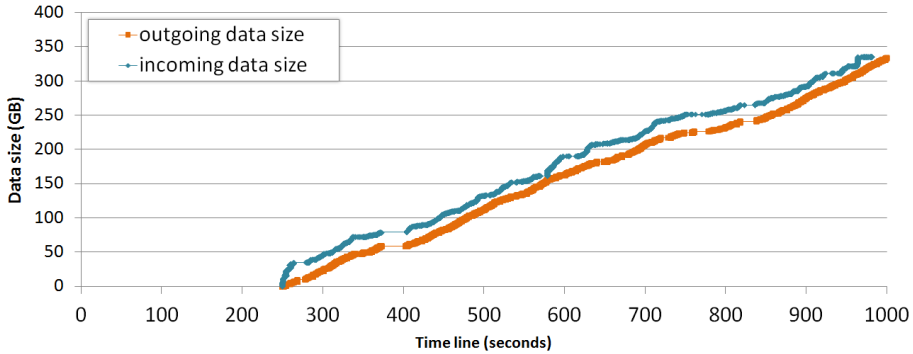

Fig. 9 incoming/outgoing data size of a transfer node's buffer with 64 compute nodes

both figures, the incoming data rate is saturated by the outgoing data rate since the RDMA servers throttle an RDMA read to balance the incoming and outgoing data rates when buffered checkpoint data exceeds 20GB. However, with 32 compute nodes, the rate was about 605 MB/s. Since total checkpoint size of 32 compute nodes is 76.8GB (= 200MB x 12 processes x 32 nodes), the flush operation completes within 250 seconds. Alternatively, the incoming data rate with 64 compute nodes was about 458 MB/seconds, which means that about 335 seconds is required to receive 153.6 GB (= 200MB x 12 processes x 64 nodes). When we consider the I/O throttling to reduce the probability of a failure in the PFS, the incoming data rate is expected to be saturated with the less number of compute nodes. Therefore, we found that performance analysis and modeling of our asynchronous checkpointing system are required to determine the appropriate number of transfer nodes given the number of compute nodes, the checkpoint size, network and effective I/O throughput. In addtion, we used the $do\_work()$ function of ATS, which is a CPU-bound function. However since our asynchronous checkpointing system shares a network bandwidth, it is expected to affect network-bound applications. We consider these problems to be future work.

5. Related Work

Multi-level checkpointing [7),11)] is a promising technique for fault-tolerant execution that utilizes different levels of storages such as RAM, SSD, hard disk and PFS depending on redundancy and resiliency requirements. Under a well-chosen redundancy scheme,

multi-level checkpointing enables an application to be reliable while maintaining the performance. Moody et al. [7] modeled and optimized the frequency based on collected failure rates and checkpointing costs of each level by using predictable a Markov model. Bautista-Gome et al.[11] proposed multi-level checkpoint using local SSDs and an underlying PFS. The proposed technique uses Reed-Solomon (RS) encoding to avoid recovery from a costly PFS. Generally, since checkpointing to and restarting from a PFS are relatively costly operations compared to those from and to local storages, the PFS is used less often in these techniques. However, an increasing failure rate by a recent growing number of components in HPC systems requires checkpointings to a PFS more frequently. Thus, with even multi-level checkpointing, the lightweight checkpointing to a PFS is crucial for exascale systems.

Significant research explores I/O performance in PFS. Lustre[4], PVSF[5] and GPFS[6] exhibit high I/O performance by using multiple dedicated I/O nodes. This enables applications to read/write data in parallel. However, since these file systems are shared by tens of thousands compute nodes, it is difficult to scale to a larger number of compute nodes writing checkpoints concurrently.

Asynchronous I/O has long been used to hide I/O bottlenecks. [12–14] These techniques enable applications to parallelize I/O and computation, resulting in increased CPU utilization and enhanced I/O performance. Patrick et al.[12] presented a comprehensive study of different techniques of overlapping I/O, communication, and computation, and showed the performance benefits of asynchronous I/O. Nawab et al.[13] asynchronously transfer multiple striped TCP data streams to increase I/O performance in Grid environments significantly. Asynchronous staging service using RDMA proposed by Hasan et al.[14] is the closest research to this work. The authors achieved high I/O throughput by using additional nodes. As we observed, when an additional node handles a large number of compute nodes, buffer can overflow and an optimal number of compute nodes that a additional node handles needs to be determined. However, the comprehensive study on the problem is not shown and also the solution is not presented given the number of compute nodes.

6. Conclusion and Future Work

We have proposed an asynchronous checkpointing system based on the SCR library. Our approach uses extra nodes to drain checkpoint data from compute nodes to a PFS using RDMA. Our preliminary experiments showed that the asynchronous checkpointing system can transfer checkpoint data with less than a 1% of runtime increase for CPU-bound applications with an appropriate number of transfer nodes.

As future work, we will further analyze and model our asynchronous checkpoint system to determine the optimal number of transfer nodes for a number of compute nodes, the checkpoint size, and network and I/O throughputs. We will also consider an I/O throttling in transfer nodes to reduce PFS failures. due to heavy load.

Acknowledgments This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-509152.

References

1) TOP500 Supercomputing Sites, http://www.top500.org/.
2) TSUBAME 2.0 - MONITORING PORTAL, http://mon.g.gsic.titech.ac.jp/.
3) Schroeder, B. and Gibson, G.A. Understanding failures in petascale computers, *Journal of Physics: Conference Series*, Vol.78, No.1, pp.012022+ (online), DOI:10.1088/1742-6596/78/1/012022 (2007).
4) Lustre: A scalable, high-performance file system, http://wiki.lustre.org/index.php/Main_Page.
5) Haddad, I.F. PVFS: A Parallel Virtual File System for Linux Clusters, *Linux J.*, Vol.2000 (online), available from ⟨http://portal.acm.org/citation.cfm?id=364654⟩ (2000).
6) Schmuck, F. and Haskin, R. GPFS: A Shared-Disk File System for Large Computing Clusters, *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, USENIX Association, (online), available from ⟨http://portal.acm.org/citation.cfm?id=1083349⟩ (2002).
7) Moody, A., Bronevetsky, G., Mohror, K. and de Supinski, B.R. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System, *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, Washington, DC, USA, IEEE Computer Society, pp.1–11 (online), DOI:10.1109/SC.2010.18 (2010).
8) Scalable Checkpoint / Restart Library, http://sourceforge.net/projects/scalablecr/.
9) IOR HPC benchmark, http://sourceforge.net/projects/ior-sio/.
10) Gerndt, M., Mohr, B. and Träff, J. Evaluating OpenMP Performance Analysis Tools with the APART Test Suite, *Euro-Par 2004 Parallel Processing* (Danelutto, M., Vanneschi, M. and Laforenza, D., eds.), Lecture Notes in Computer Science, Vol.3149, Springer Berlin / Heidelberg, Berlin, Heidelberg, chapter20, pp.155–162 (online), DOI:10.1007/978-3-540-27866-5_20 (2004).
11) Bautista-Gome, L., Komatitsch, D., Maruyama, N., Tsuboi, S., Cappello, F. and Matsuoka, S. FTI: high performance Fault Tolerance Interface for hybrid systems, *Proceedings of the*

Vol.2011-ARC-197 No.18
Vol.2011-HPC-132 No.18
2011/11/28

2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, Seattle, WS, USA (2011).

12) Patrick, C.M., Son, S. and Kandemir, M. Comparative evaluation of overlap strategies with study of I/O overlap in MPI-IO, *SIGOPS Oper. Syst. Rev.*, Vol.42, pp.43–49 (online), DOI:10.1145/1453775.1453784 (2008).

13) Ali, N. and Lauria, M. Improving the Performance of Remote I/O Using Asynchronous Primitives, pp.218–228 (online), DOI:10.1109/HPDC.2006.1652153.

14) Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K. and Zheng, F. DataStager: scalable data staging services for petascale applications, *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, New York, NY, USA, ACM, pp.39–48 (online), DOI:10.1145/1551609.1551618 (2009).