LLNL-PROC-540391

Asynchronous Checkpoint Migration with MRNet in the Scalable Checkpoint / Restart Library

K. Mohror, A. Moody, B. R. de Supinski

March 21, 2012

Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS 2012)
Boston, MA, United States
June 25, 2012 through June 28, 2012

Asynchronous Checkpoint Migration with MRNet in the Scalable Checkpoint / Restart Library

Kathryn Mohror, Adam Moody, and Bronis R. de Supinski
Lawrence Livermore National Laboratory
{kathryn, moody20, bronis}@llnl.gov

***Abstract*—Applications running on today's supercomputers tolerate failures by periodically saving their state in checkpoint files on stable storage, such as a parallel file system. Although this approach is simple, the overhead of writing the checkpoints can be prohibitive, especially for large-scale jobs. In this paper, we present initial results of an enhancement to our Scalable Checkpoint / Restart Library (SCR). We employ MRNet, a tree-based overlay network library, to transfer checkpoints from the compute nodes to the parallel file system asynchronously. This enhancement increases application efficiency by removing the need for an application to block while checkpoints are transferred to the parallel file system. We show that the integration of SCR with MRNet can reduce the time spent in I/O operations by as much as $15\times$. However, our experiments exposed new scalability issues with our initial implementation. We discuss the sources of the scalability problems and our plans to address them.**

I. Introduction

As the scales of supercomputing systems grow, the systems become less reliable, because increased component counts increase overall fault rates. Applications that run on high performance computing (HPC) systems can experience mean times between failures on the order of hours or days because of hard [1] and soft errors [2]. Experts predict that exascale systems could fail as frequently as every 3-26 minutes [3], [4].

A common approach to mitigating the consequences of failures is *checkpointing*. Applications periodically save their state to checkpoint files on reliable storage, usually a parallel file system. When a failure occurs, an application can restart from its previously saved state by reading in a checkpoint file. Although checkpointing is a simple approach to tolerating failures, writing checkpoint files to a parallel file system is expensive at large scales, e.g., a single checkpoint can take on the order of tens of minutes [5], [6].

We developed the Scalable Checkpoint / Restart Library (SCR) to lower checkpointing overhead [7], [8]. SCR increases system efficiency by as much as 35%. Multi-level checkpointing systems [9], [10], such as SCR, use multiple types of checkpoints that have different levels of resiliency and cost. The highest checkpoint level writes to the parallel file system, which is slow but reliable; it can withstand a failure of an entire machine. SCR also employs faster but less resilient checkpoint levels that utilize in-system storage, such as RAM, Flash, or disk on the compute nodes, and applies cross-node redundancy schemes.

SCR is designed for globally-coordinated checkpoints that are written as a file per MPI process. These checkpoints are globally coordinated and bounded by a barrier in the SCR library at checkpoint termination, so the entire application blocks whenever a checkpoint is transferred to the parallel file system. In this work, we explore an enhancement to SCR that uses MRNet, a tree-based overlay network library [11], to transfer checkpoints from the compute nodes to the parallel file system asynchronously. This mechanism supports lower overhead recovery from more catastrophic failures.

The benefits of this enhancement are multifold. First, application and system efficiencies increase because the application can continue computing while the checkpoints are stored to the parallel file system in the background. Second, the effective load on the parallel file system decreases. SCR can throttle the rate of the asynchronous transfer to the parallel file system. Additionally, using MRNet alleviates contention for the parallel file system by reducing the number of concurrent writers from N to M, where $M \ll N$. Both of these factors mean that other concurrent users of the parallel file systems could see better overall performance due to reduced contention for parallel file system resources. Third, the MRNet infrastructure can provide additional storage locations for caching checkpoints. These levels lower restart overheads because access to the parallel file system is avoided for a wider range of failures.

In this paper, we focus on the first of these benefits: how asynchronous transfer of checkpoints can benefit applications by lowering the overhead of checkpointing to the parallel file system. We defer exploration of the other two benefits for future work.

The rest of this paper is structured as follows. In Section II, we present related research. In Section III, we give background information on SCR and MRNet. Section IV details our MRNet-based SCR implementation. Then, in Section V, we describe our experimental setup, and give results from our experiments.

II. Related Work

Several researchers have worked to lower checkpoint write overhead by caching them on compute node storage as is done by SCR. Diskless checkpointing reduces overhead by caching checkpoints in memory or other node local storage and using mirroring and parity methods for redundancy [12], [13]. Bautista-Gomez and colleagues reduce overheads by caching checkpoints on SSDs on compute nodes [14], while Dong et al. investigate the use of PCRAM [15].

Other researchers have investigated the asynchronous transfer of checkpoints to the parallel file system. Plank and Li compress checkpoints and write them asynchronously to lower overheads [16]. Ouyang et al. both aggregate checkpoint files from multiple writes and drain them asynchronously to stable storage [17]. They also explore throttling the write rate to reduce contention on the parallel file system. In FTI, Bautista-Gomez et al. utilize GPUs and dedicated FTI-MPI processes on compute nodes to hide the overhead of Reed-Solomon encoding of checkpoints [18]. Additionally, they transfer checkpoints asynchronously to the PFS in a manner similar to the Open MPI staging option [19], which uses a daemon process on the compute nodes to manage moving the data. Although not designed specifically for checkpoint/restart, data staging frameworks such as DataStager [20] and IOFSL [21] provide transparent mechanisms for asynchronous movement of data.

Our work is most similar to that of Rajachandrasekar et al. who use a data staging framework to move checkpoint data to the parallel file system through a hierarchical network [22]. They also noted the benefit of reducing the number of concurrent writers to the parallel file system. Our approach differs from this work in that we employ a generic, publically available hierarchical infrastructure for moving data, and that we combine the benefits of a multi-level checkpointing library with those of asynchronous data transfer.

III. Background

This section provides a discussion of the basic design of SCR as well as an overview of MRNet, upon which we build our SCR extension.

A. SCR

The Scalable Checkpoint/Restart (SCR) library uses storage distributed on a system's compute nodes to attain high checkpoint and restart I/O bandwidth for MPI applications. We based the design of SCR on two key observations. First, only the most recent checkpoint is needed to recover from a failure. Upon completion of the next checkpoint, the previous checkpoint can be discarded. Second, a failure typically only affects a small portion of the system, with the rest of the system still functioning normally. For instance, on the clusters on which we currently use SCR, 85% of failures disable at most one compute node [7].

Based on these observations, we designed SCR to cache only the most recent checkpoints in compute node storage and to apply a redundancy scheme to those cached checkpoints, e.g., copy them to partner nodes. Also, SCR periodically copies (flushes) a cached checkpoint to the parallel file system in order to withstand failures that disable larger portions of the system. However, a well-chosen redundancy scheme allows checkpoints to be flushed infrequently.

SCR employs a library and a set of Perl scripts to manage applications. Applications make calls into the SCR library to indicate when checkpoints are being taken and to retrieve information about where those checkpoints will be cached, e.g., RAM disk. The SCR library manages the caching of the checkpoints and application of the redundancy schemes for resiliency. The Perl scripts are executed as needed, primarily by the script used to launch the job, `scr_srun`. The `scr_srun` script has the responsibilities of querying the job's environment, checking the health of nodes in the allocation, and starting any daemon processes needed by SCR. Additionally, it starts the application and attempts to restart it after failures, assuming there are enough healthy compute nodes left in the job.

B. MRNet

MRNet is a general-purpose, software-based multicast/reduction network that can be used to build scalable tools for HPC systems [11]. Several different types of tools have been built using MRNet, including debuggers [23] and performance tools [24]. MRNet uses a tree network for scalable, flexible data aggregation from compute node-resident daemons to a front end tool control program. The flexible aggregation is accomplished with user-supplied custom data aggregation filters. In this work, we simply use the infrastructure to move checkpoints in controlled fashion from the compute nodes to the parallel file system. In other words, our filter does not aggregate checkpoints in any way, but simply passes them down the tree.

An MRNet instance is composed of a library linked into a tool's front end process, mrnet_commnodes, and a library linked into a tool's back end daemons on the compute nodes. The front end process is the root of the tree network and controls traffic through the tree structure. The mrnet_commnode processes are the interior nodes of the tree. They facilitate scalable group communications and execute data aggregation operations. The back end tool daemons are co-located with the application processes on the compute nodes and interact with the application, e.g., they initiate the transfer of checkpoint files.

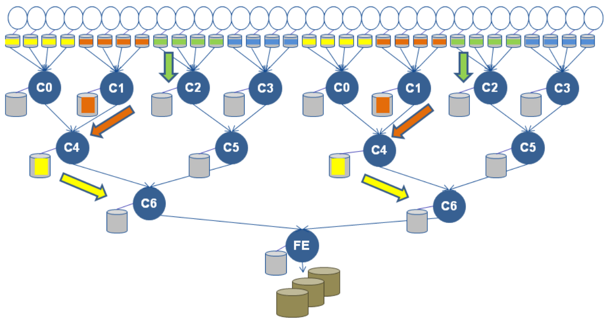

Fig. 1. Example of Moving Checkpoints Through MRNet Tree

IV. Implementation

Our implementation of SCR with MRNet uses a front end SCR/MRNet process to control SCR/MRNet back end daemons running on the compute nodes. The front end process initiates the asynchronous transfer of checkpoints by sending messages that request the files from selected daemons. It chooses the daemons to send files such that only a subset sends files at a time. This mechanism throttles the data transfer to the parallel file system. In our current implementation, the front end process performs all writes to the parallel file system (Section V-A discusses a write optimization that we are exploring). In order to use MRNet with SCR, we allocate a small number of additional nodes to the allocation so that the application processes are not perturbed by the activities of the front end or the mrnet_commnodes. We do not necessarily use a one-to-one mapping between mrnet_commnodes and additional compute nodes. Multiple mrnet_commnodes can run on a single compute node.

Figure 1 shows an example of the second in a series of logical steps of moving checkpoints with MRNet. The arrows indicate the movement that occurs in the third step. The white circles represent compute nodes. Cylinders indicate storage devices; gray cylinders are node-local storage devices, while the brown cylinders are the parallel file system. Colored rectangles in the storage cylinders represent checkpoint files. The blue circles marked with “CX” are mrnet_commnode processes, and the blue “FE” circle marked is the SCR front end.

In the first step, the front end sends a command to the daemons on nodes with yellow checkpoints to send their checkpoint files, at which point, these daemons transfer their files to the first level mrnet_commnodes labeled “C0.” In the second step (shown), the yellow checkpoints are transferred to the next level of the tree, “C4.” Simultaneously, the front end directs the daemons on nodes with orange checkpoints to send their files, and those daemons then transfer their files to the first level mrnet_commnodes labeled “C1.” In the third step, the yellow checkpoints move further down the tree to mrnet_commnodes “C6”; the orange checkpoints move to mrnet_commnodes “C4”; and green checkpoints move to mrnet_commnodes “C2.”

We modified the `scr_srun` script to execute `scr_mrnet_launcher`, which serves as the SCR/MRNet front end, instead of executing the user’s job command directly. Front end arguments indicate the nodes to use for mrnet_commnodes, parameters to set up the tree such as fanout and depth, and the original job command string. The front end uses launchMON to start the application processes and SCR/MRNET daemons. LaunchMON is an infrastructure for co-locating tool daemons on the compute nodes of a parallel job [25]. It communicates with the resource manager to identify the locations of the remote processes and to launch tool daemons scalably. Each back end daemon queries launchMON for information about the application processes on its compute node, such as executable name and process identifier, and for information about connecting to the front end process so that it can connect to the tree structure.

After the tree is connected, the front end begins to query the back end daemons for files. If the daemons have files to send, they send a message to the front end and then begin to send files. If they do not have files, they send a message indicating that they do not have files. If none of the daemons have files to send, the front end waits a specified amount of time and then again queries the back ends for files.

V. Results

We ran our experiments on Sierra, a 1,944 node, 12 core per node, Linux cluster at LLNL. We wrote all files to the Lustre parallel file system lscratchc, which is a 1.6 TB file system with a peak bandwidth of 30 GB/s from Sierra [26]. For our SCR/MRNet infrastructure, we used SCR version 1.1-8, MRNet version 3.0, and launchMON version 0.7.2.

For our experiments, we used the IOR benchmark version 2.10.2 [27]. IOR is a benchmark designed to mimic the I/O patterns of real applications for testing the performance of file systems. We used IOR to simulate an application writing checkpoints at regular intervals of ten minutes.[1] We configured IOR to write files of 48 MB per process in HDF5 format [28] for ten iterations. Additionally, we specified that IOR use barriers between the I/O operations and only do write operations.

We configured SCR to use MRNet for asynchronous transfer of checkpoints of every checkpoint set written by IOR. The MRNet fanout was set to 48, which means that there was one mrnet_commnode for every 48 backend daemons. We assigned a single core to each mrnet_commnode. For

[1]We modified the IOR source slightly due to a bug that caused the program to ignore the delay (interval) parameter. The bug was fixed in version 2.10.3.

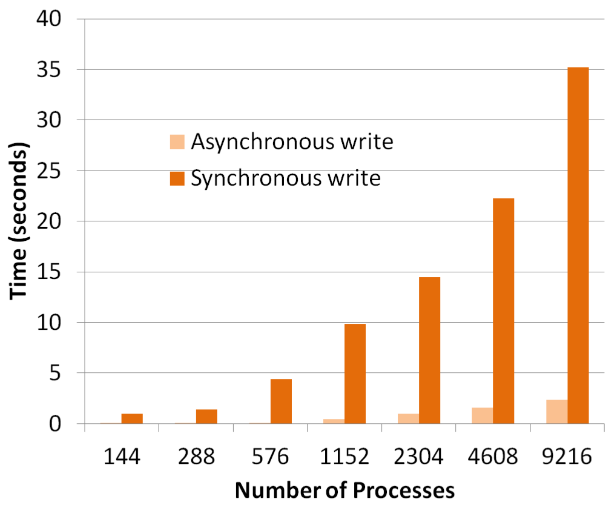

Fig. 2. I/O Time in IOR with Synchronous and Asynchronous Transfer

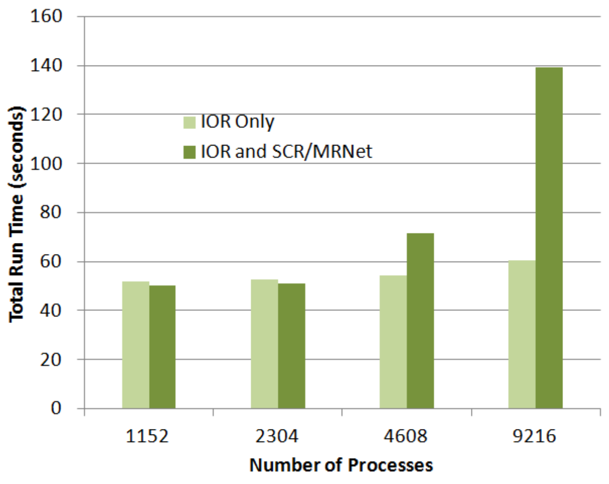

Fig. 3. Total Run Time for IOR With and Without SCR

the 9216 process runs, this amounted to a total of 770 nodes in the allocation, with 768 used by the application and two additional nodes used by mrnet_commnodes and the SCR/MRNet frontend process.

In Figure 2, we report the average I/O time over ten iterations as reported by IOR, which includes the time for opening and closing files, as well as the write time. The I/O time with synchronous checkpointing increases dramatically with increasing process count. However, when using asynchronous checkpointing with MRNet, the time increases much more slowly. In all cases, asynchronous checkpointing outperforms synchronous checkpointing, ranging from an $11\times$ difference at 144 processors to a $15\times$ difference at 9216 processors.

When inspecting the total run time of IOR when run with and without SCR/MRNet, we discovered scalability problems in our current implementation. For lower process counts, IOR with SCR/MRNet performed slightly better than IOR alone. However, as shown in Figure 3, this trend reversed at larger processor counts. The time for SCR/MRNet at 4608 processes was $1.3\times$ higher than without SCR, and for 9216 processes, it was $2.3\times$ higher.

A. Discussion

From our results, we can clearly see that SCR/MRNet can reduce I/O time in applications, especially at larger scales. However, our results for the total run time of IOR (Figure 3) show that we have more work to do in order to benefit applications in terms of total execution time. In particular, we need to investigate our implementation, and our use of the MRNet infrastructure.

One reason for the higher total run times is that in `SCR_Finalize` SCR attempts to write the final checkpoint cached on the compute nodes to the parallel file system. In our current implementation, this is still a synchronous write operation, and suffers from scalability problems.

Another major source of increased overhead in our implementation is the use of the front end process as the single writer to the parallel file system. At larger scales, the front end could not process all checkpoint files in the time it took IOR to complete its "compute cycle," after which it starts the next checkpoint. As a result, the SCR library blocks in `SCR_Complete_checkpoint` waiting for the last checkpoint to finish being written to the parallel file system before marking the next checkpoint as ready to be written. An optimization that could reduce this overhead is to alter our use of MRNet such that we use a "forest of writers" instead of a single writer. We could enable this mechanism by informing the mrnet_commnodes of their level in the tree, and at which level to perform writes to the parallel file system. Thus, we would increase the parallelism in our use of the parallel file system and increase performance.

Another source of overhead is application perturbation from the back end daemons on the compute nodes. The daemons use CPU resources as well as memory and network resources. We could alleviate the CPU and memory usage to some degree by dedicating a core on each node for the daemon process. Potentially, we could reduce the overhead resulting from the daemon's use of network resources by discovering when the application was in a compute phase and transferring the files when the application is not using the network resources.

VI. Conclusions

In this paper, we presented initial results of asynchronous checkpoint movement in the checkpointing library SCR using MRNet, a tree-based overlay network. Our goal was to increase application efficiency by removing the need for an application to block while checkpoints are transferred to the parallel file system. We showed that the integration of SCR with MRNet does significantly reduce the time spent in I/O operations, especially as application scales increase. However, we found that due to problems in our initial implementation, the overall execution time of the application increased with scale. We will investigate and correct these scalability problems in the near future.

For future work, we will investigate optimizations of our current implementation. As discussed in Section V-A, we will explore the use of a "forest of writers" to increase our write performance and to avoid blocking in `SCR_Complete_checkpoint`. We also plan to optimize the interaction of the back end daemons with the application processes to minimize application perturbation. Additionally, we plan to experiment with MRNet configuration parameters. For example, we simply used a fanout of 48 for the mrnet_commnode to daemon ratio, which may not have been the optimal choice.

Additionally, we plan to pursue other research paths. We will investigate new filters that could be used in the MRNet tree reductions. For example, checkpoints could be compressed and/or aggregated, which could offer significant gains in write performance. Another area of research is the new levels of resiliency that the MRNet infrastructure could provide for multilevel checkpointing. Using an infrastructure such as MRNet for asynchronous checkpoint movement means that multiple copies of the checkpoints are made as they move from the compute nodes through the tree to the parallel file system. However, analogously to the methods that SCR already uses for low-overhead resiliency, the additional copies of checkpoints could be cached on the nodes on which the mrnet_commnodes run; then, redundancy schemes could be applied across those. This mechanism would allow for recovery from major losses in compute nodes without necessarily needing to restart from the parallel file system.

VII. Acknowledgements

We thank the reviewers of this paper for their helpful comments which improved this paper.

This article has been authored by Lawrence Livermore National Security, LLC under Contract No. DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. LLNL-PROC-540391

References

[1] B. Schroeder and G. A. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2006, pp. 249–258.

[2] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender, "Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 329–335, September 2005.

[3] B. Schroeder and G. Gibson, "Understanding Failure in Petascale Computers," *Journal of Physics Conference Series: SciDAC*, vol. 78, p. 012022, June 2007.

[4] V. Sarkar, Ed., *ExaScale Software Study: Software Challenges in Exascale Systems*, 2009.

[5] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman, "ZOID: I/O-Forwarding Infrastructure for Petascale Architectures," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008, pp. 153–162.

[6] R. Ross, J. Moreira, K. Cupps, and W. Pfeiffer, "Parallel I/O on the IBM Blue Gene/L System," Blue Gene/L Consortium Quarterly Newsletter, Tech. Rep., First Quarter, 2006.

[7] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10*, November 2010, pp. 1 –11.

[8] "Scalable Checkpoint/Restart Library." [Online]. Available: http://sourceforge.net/projects/scalablecr/

[9] E. Gelenbe, "A Model of Roll-back Recovery with Multiple Checkpoints," in *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*, 1976, pp. 251–255.

[10] N. H. Vaidya, "A Case for Multi-Level Distributed Recovery Schemes," Texas A&M University, Tech. Rep. 94-043, May 1994.

[11] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing, SC'03*, 2003.

[12] J. S. Plank, K. Li, and M. A. Puening, "Diskless Checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, October 1998.

[13] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, "Fault Tolerant High Performance Computing by a Coding Approach," in *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005, pp. 213–223.

[14] L. A. Bautista-Gomez, N. Maruyama, F. Cappello, and S. Matsuoka, "Distributed Diskless Checkpoint for Large Scale Systems," in *CCGRID*, 2010, pp. 63–72.

[15] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Future Exascale Systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*, 2009.

[16] J. S. Plank and K. Li, "ickp: A Consistent Checkpointer for Multicomputers," *IEEE Parallel & Distributed Technology*, vol. 2, no. 2, pp. 62–67, 1994.

[17] X. Ouyang, R. Rajachandrasekar, X. Besseron, H. Wang, J. Huang, and D. Panda, "CRFS: A Lightweight User-Level Filesystem for Generic Checkpoint/Restart," in *Proceedings of the 2011 International Conference on Parallel Processing, ICPP'11*, Sept. 2011, pp. 375 – 384.

[18] L. A. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High Performance Fault Tolerance Interface for Hybrid Systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'11*, 2011.

[19] J. Hursey and A. Lumsdaine, “A Composable Runtime Recovery Policy Framework Supporting Resilient HPC Applications,” Indiana University, Tech. Rep. TR686, 2010.

[20] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, “DataStager: Scalable Data Staging Services for Petascale Applications,” in *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, HPDC’09*, 2009, pp. 39–48.

[21] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, “Scalable I/O Forwarding Framework for High-Performance Computing Systems,” in *Proceedings of IEEE International Conference on Cluster Computing and Workshops, CLUSTER’09*, 2009, pp. 1–10.

[22] R. Rajachandrasekar, X. Ouyang, X. Besseron, V. Meshram, and D. K. Panda, “Can Checkpoint/Restart Mechanisms Benefit from Hierarchical Data Staging?” in *Proceedings of the Workshop on Resiliency in High Performance Computing in Clusters, Clouds, and Grids, Resilience ’11, held in conjunction with EuroPar*, Aug. 2011.

[23] D. Arnold, D. Ahn, B. de Supinski, G. Lee, B. Miller, and M. Schulz, “Stack Trace Analysis for Large Scale Debugging,” in *Proceedings of the Parallel and Distributed Processing Symposium, 2007, IPDPS’07*, March 2007.

[24] A. Nataraj, A. Malony, A. Morris, D. Arnold, and B. Miller, “In Search of Sweet-Spots in Parallel Performance Monitoring,” in *Proceedings of 2008 IEEE International Conference on Cluster Computing*, Nov. 29 - Oct. 1 2008, pp. 69 –78.

[25] “LaunchMON.” [Online]. Available: http://sourceforge.net/projects/launchmon/

[26] “LLNL LC Parallel File Systems Summary.” [Online]. Available: https://computing.llnl.gov/tutorials/lc_resources/index.html#ParallelFileSystems

[27] “IOR Benchmark.” [Online]. Available: https://asc.llnl.gov/sequoia/benchmarks/#ior

[28] “The HDF Group.” [Online]. Available: http://www.hdfgroup.org/HDF5/