Extending Scalability of Collective IO Through Nessie and Staging

Jay Lofstead
Sandia National Laboratories
gflofst@sandia.gov

Ron Oldfield
Sandia National Laboratories
raoldfi@sandia.gov

Todd Kordenbrock
Hewlett-Packard Company
todd.kordenbrok@hp.com

Charles Reiss
University of California, Berkeley
charles@eecs.berkeley.edu

ABSTRACT

The increasing fidelity of scientific simulations as they scale towards exascale sizes is straining the proven IO techniques championed throughout terascale computing. Chief among the successful IO techniques is the idea of collective IO where processes coordinate and exchange data prior to writing to storage in an effort to reduce the number of small, independent IO operations. As well as collective IO works for efficiently creating a data set in the canonical order, 3-D domain decompositions prove troublesome due to the amount of data exchanged prior to writing to storage. When each process has a tiny piece of a 3-D simulation space rather than a complete 'pencil' or 'plane', 2-D or 1-D domain decompositions respectively, the communication overhead to rearrange the data can dwarf the time spent actually writing to storage [27]. Our approach seeks to transparently increase scalability and performance while maintaining both the IO routines in the application and the final data format in the storage system. Accomplishing this leverages both the Nessie [23] RPC framework and a staging area with staging services. Through these tools, we employ a variety of data processing operations prior to invoking the native API to write data to storage yielding as much as a 3× performance improvement over the native calls.

Categories and Subject Descriptors

D.4 [**Software**]: Operating Systems; D.4.7 [**Operating Systems**]: Organization and Design—*hierrchichal design*

General Terms

Design, Experimentation, Performance

1. INTRODUCTION

Collective IO has offered tremendous benefits for applications with moderate per process data sizes (<= 20MB) as they scale by trading additional communication time to build larger data blocks for reducing the number and increasing the size of the IO operations to storage. This reduces the total amount of time spent performing IO because the communication time is much less than the corresponding time spent writing to or reading from storage. In spite of the benefits, collective IO is not a panacea for IO performance problems. Many pieces of recent work [20, 27, 28, 13] have shown scaling collective IO can be problematic. For the MPI Tile IO benchmark, with as few as 512 processes, the data rearrangement communication overhead can dominate the actual data movement to storage time [27] taking 72% of the IO time.

Log-based formats [18, 25, 19] have demonstrated that changing the IO API or on disk storage format can achieve much better performance. Staging has been effective in improving perceived IO performance [29, 15, 22, 6, 14] through techniques like asynchronous IO. The problem with both of these approaches is the requirement to change the IO API and/or the file format used in the storage system. In some cases, one or both of these attributes cannot change. To address these situations specifically, a different approach must be used.

One approach is to create a new implementation of the IO library API that uses different IO techniques to address the performance for a particular application. The downside to this approach is that the file organization may have to change to achieve any performance benefits beyond the highly optimized implementation provided by the IO library itself. Using this new implementation of an IO library approach with a staging area offers the opportunity to maintain both the API and the file layout while performing data manipulation on far fewer resources reducing the communication overhead. By concentrating the data into fewer resources, data reorganization operations can be performed in a more localized environment reducing the communication costs. Sandia's NEtwork Scalable Service Interface (Nessie) [23] system provides a simple RPC mechanism originally developed for the Lightweight File Systems [24] project. Nessie was designed specifically for systems with native support for remote direct-memory access (RDMA) and has ports for Portals [9], InfiniBand [8], Gemini [1], and LUC [5]. Combining this mechanism with a staging area that understands the needs of the IO API leads to both flexibility in how the IO is performed and a transparent wedge that disturbs neither the host application source code nor the file format in the storage system.

Two effective uses of collective IO are the PnetCDF [17] and Parallel NetCDF IO libraries. Both have been adopted by the Community Earth System Model (CESM) [21] climate community to support their standardized netCDF data format with an API nearly identical to the earlier serial NetCDF API. While CESM uses their own PIO [12] as the user-level interface for IO calls, underneath, one of these two collective IO libraries is used for the actual to storage operations. For CESM, changing either the API or the file format is not a consideration. To address the needs of applications like CESM, we choose to leverage the Nessie framework to provide a transparent redirector to a staging area where data consolidation or simply operation caching and queuing can occur. This staging area will still use the PnetCDF API to write to storage, but from a radically reduced number of clients and with some techniques, far fewer IO operations due to data consolidation happening as a pre-processing step in the staging area.

Some other efforts that have focused on interposing a layer between the application and the storage system include the IO Forwarding Scalability Layer [7] (IOFSL), the Cray Data Virtualization Service [26] (DVS), and Kangaroo [4] for grid. The IOFSL provides services below the middleware layer where techniques such as the two-phase IO portion of collective IO are implemented. This is too low in the IO stack to be able to address the coordination communication overhead of collective IO that is addressed by this work. Systems like DVS provide a way to work around the access bottlenecks of a file system, but again do not address the collective communication overhead prior to accessing storage. Kangaroo offers a way to move data from the application to or from storage via a staging style approach. This successfully offloads the IO operations off the main compute processes, but still only intercepts the IO calls after any data rearrangement or aggregation happens as part of the collective IO process.

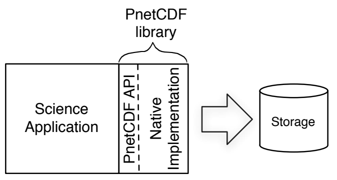
The rest of the paper is organized as follows. Section 2 contains an architectural description. Section 3 has performance results. Conclusion and future work appear in Section 4, followed by references.

2. SOFTWARE ARCHITECTURE

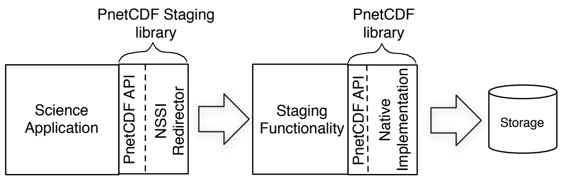
The idea explored in this paper is the possibilities of improving collective IO performance through staging while maintaining both the application IO API and the final file format in storage. Our approach is to re-implement the IO API to forward each IO request to a staging area where the data is processed prior to using the native IO library to store the data. In this case, we chose to implement the PnetCDF API because it is a mature and well-tuned IO API that effectively uses collective IO.

For our PnetCDF *staging library*, we reimplemented the PnetCDF 1.2.0 API, creating a drop-in, link-time replacement for the native PnetCDF library that uses the Nessie RPC mechanism to forward every API call to the staging area. Once the staging area has applied any requested processing to the data, it then calls the native PnetCDF API to complete the data movement to disk in the native nc5 file format. We illustrate these components in Figure 1.

The current implementation offers several options for processing the individual requests from the compute processes once they arrive in the staging area:



(a) Native PnetCDF



(b) PnetCDF Staging

Figure 1: System Architecture

1. *direct* - immediately use the PnetCDF library to execute the request synchronously with the file system
2. *caching independent* - caches the write calls in the staging area until either no more buffer space is available or the file close call is made. At that time, the data is written using an independent IO mode rather than collective IO. This avoids both coordination among the staging processes and any additional data rearrangement prior to movement to storage.
3. *aggregate independent* - similar to caching independent except that the data is aggregated into larger, contiguous chunks as much as possible within all of the server processes on a single compute node prior to writing to storage. That is, to optimize the data rearrangement performance, the movement is restricted to stay within the same node avoiding any network communication overhead.

Additional processing modes using collective rather than independent IO are available in the staging area, but have not been tested for this paper and will be examined in future work.

In addition to this PnetCDF implementation, the NetCDF API has also been implemented in the same code base. Both of these frameworks are available in the Trilinos [16] library as part of the Trios components. In this work, only the PnetCDF implementation is evaluated.

3. EXPERIMENTAL EVALUATION

The efficacy of this approach is evaluated in three parts. First, the performance of the Nessie layer is evaluated showing the scalability and performance of that infrastructure. Second, we evaluate the potential of the PnetCDF staging service by comparing it to netCDF and PnetCDF using the IOR benchmark. Finally, we evaluate the end-to-end performance of the PnetCDF staging library when applied to the IO kernel of S3D, a combustion simulation code that uses a 3-D domain decomposition.

3.1 Nessie Evaluation

```
struct data_t {
    int int_val;          /* 4 bytes */
    float float_val;      /* 4 bytes */
    double double_val;    /* 8 bytes */
};
```

Figure 2: The 16-byte data structure used for the data-movement experiments.

To measure the overhead imposed by the Nessie framework, we evaluated it on two different platforms at Sandia: Red Storm and Thunderbird. In all tests, a 16-byte structure, shown in Figure 2, is sent to validate the overheads rather than the data transfer bandwidth of the links themselves.

Red Storm is a Cray XT3 located at Sandia. At the time of testing, Red Storm had 12960 dual-core compute nodes. The compute nodes are arranged in a regular three-dimensional grid, connected with a hypertorus topology. Each node has an interconnect with a custom Cray SeaStar networking chip and a dedicated PowerPC chip. The interconnect is coupled to the processor using a HyperTransport link that has a theoretical (excluding wire protocol overhead) bandwidth of 2.8GB/s [10]. Each of the six links from each node can support 2.5GB/s, after protocol overheads. Low-level software access to the interconnects is provided through the Portals library [9], which provides a connectionless RDMA-based interface.

At the time these tests were performed, the Thunderbird system was Sandia National Laboratories largest capacity cluster. It is composed of 4,480 compute nodes, each with dual 3.6 GHz Intel EM64T processors with 6 GB of memory. Thunderbird uses an InfiniBand network with a two level CLOS topology with eight top-level core switches and 280 leaf switches (24 ports per leaf switch). Each leaf switch has 16 downlinks (16 compute nodes per leaf switch) and 8 uplinks. Thus, the network is 2-to-1 oversubscribed in terms of raw number of links.

Although Thunderbird is primarily a capacity cluster, designed for large numbers of small jobs, it is still useful as a system to evaluate performance of the InfiniBand port of Nessie and the use of staging nodes for the caching service. It is also generally more accessible than Red Storm for testing.

3.1.1 *Nessie Throughput Results*

Figure 3(a) shows the throughput of the simple data-transfer application moving data to a single staging area for 1, 4, 16, and 64 clients. The maximum observed node-to-node unidirectional bandwidth of the SeaStar network through the Portals API is around 2.1GB/s [10]. In our experiments, we achieved very close to the peak for each of the experiments, showing that the Nessie software adds very little overhead to the native transport. In addition, the increase in scale from a single client to sixty-four clients resulted in a minor performance decrease, despite the dramatic increase in the number of requests handled by the staging node.

Figure 3(b) shows the same experiments performed on Thunderbird. While we only achieve 75% of the maximum performance over the InfiniBand link, given the 80% overhead for the protocol [2], we are nearly at the peak performance of the network link.

3.2 PnetCDF Staging Evaluation

To evaluate the potential of PnetCDF staging, we measured the performance of our PnetCDF staging library when used by the IOR benchmark code. IOR (Interleave-or-random) [3] is a highly configurable benchmark code from LLNL that IOR is often used to find the peak measurable throughput of an I/O system. In this case, IOR provides a tool for evaluating the impact of offloading the management overhead of the netCDF and PnetCDF libraries onto staging nodes.

Figure 4 shows measured throughput of three different experiments: writing a single shared file using PnetCDF directly, writing a file-per-process using standard netCDF3, and writing a single shared file using the PnetCDF staging service. In every experiment, each client wrote 25% of its compute-node memory, so we allocated one staging node for each four compute nodes to provide enough memory in the staging area to handle an I/O “dump”.

Results on Thunderbird show terrible performance for both the PnetCDF and netCDF file-per-process case when using the library directly. The PnetCDF experiments maxed out at 217 MiB/s and reached the peak almost immediately. The PnetCDF shared file did not do much better, achieving a peak throughput of 3.3 GiB/s after only 10s of clients. The PnetCDF staging service, however, achieved an “effective” I/O rate of 28 GiB/s to a single shared file. This is the rate observed by the application as the time to transfer the data from the application to the set of staging nodes. The staging nodes still have to write the data to storage, but for applications with “bursty” IO patterns, staging is very effective.

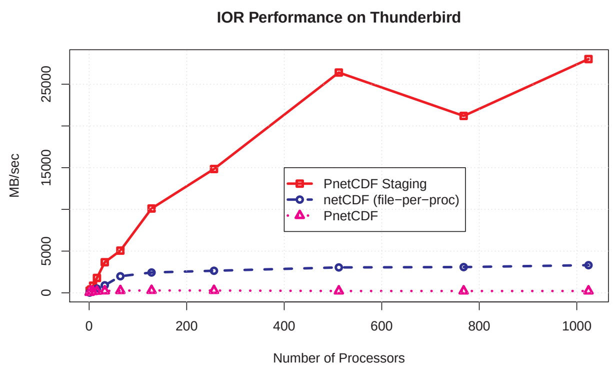
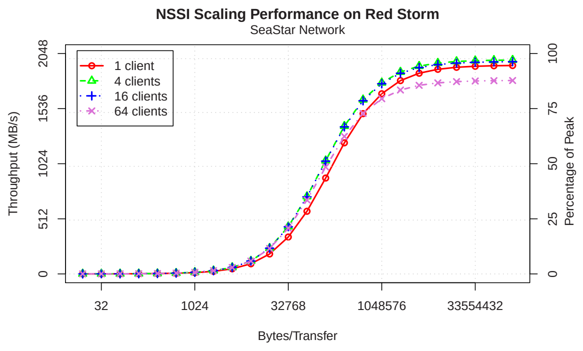


Figure 4: Measured throughput of the IOR benchmark code on Thunderbird

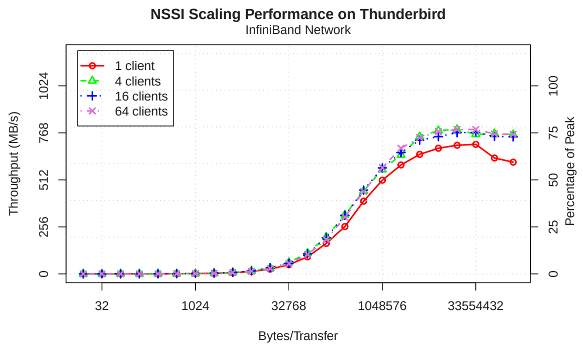
3.3 Application Evaluation of S3D using PnetCDF Staging

In the final set of experiments, we evaluate the performance of the PnetCDF staging library when used by Sandia’s S3D simulation code [11], a flow solver for performing direct numerical simulation of turbulent combustion.

All experiments take place on the JaguarPF system at Oak Ridge National Laboratories. JaguarPF is a Cray XT5 with 18,688 compute nodes in addition to dedicated login and service nodes. Each compute node has dual hex-core



(a) Nessie on Redstorm



(b) Nessie on Thunderbird

Figure 3: Measured throughput from a set of compute-node clients to a single caching server on Red Storm (upper) and Thunderbird (lower). The time includes including bandwidth excluding open time and the raw RMA get bandwidth.

AMD Opteron 2435 processors running at 2.6GHz, 16 GB RAM, and a SeaStar 2+ router. The PnetCDF version is 1.2.0 and uses the default Cray MPT MPI implementation. The file system, called Spider, is a Lustre 1.6 system with 672 object storage targets and a total of 5 PB of disk space. It has a demonstrated maximum bandwidth of 120 GB/sec. We configured the file system to stripe using the default 1 MB stripe size across 160 storage targets for each file for all tests.

In our test configuration, we use ten, 32 cubes ($32\times32\times32$) of doubles per process across a shared, global space. The data size is 2.7 GB per 1024 processes. We write the whole dataset at a single time and measure the time from the file open through the file close. We use five tests for each process count and show the best performance for each size. In this set of tests, we use a single node for staging. To maximize the parallel bandwidth to the storage system, one staging process per core is used (12 staging processes). Additional testing with a single staging process did not show significant performance differences. The client processes are split as evenly as possible across the staging processes in an attempt to balance the load.

Figure 5 shows the results of S3D using the PnetCDF library directly with the four different configurations of our PnetCDF staging library described in Section 2. In all cases measured, the base PnetCDF performance was no better than any other technique at any process count. The biggest difference between the base performance and one of the techniques is for 1024 processes using the caching independent mode at only 32% as much time spent performing IO. The direct technique starts at about 50% less time spent and steadily increases until it reached parity at 7168 processes. Both cache independent and aggregate independent advantages steadily decrease as the scale increases, but still have a 20% advantage at 8192 processes.

In spite of there only being 12 staging processes with a total gross of 16 GB of RAM, the performance improvement is still significant. The lesser performance of the direct writing method is not very surprising. By making the broadly distributed calls synchronous through just 12 processes, the calling application must wait for the staging area to complete the write call before the next process will attempt to write. The advantage shown for smaller scales shows the disadvantage of the communication to rearrange the data compared to just writing the data. Ultimately, the advantage is overwhelmed by the number of requests being performed synchronously through the limited resources.

The advantage of the caching and aggregating over the direct and base techniques shows that by queueing all of the requests and letting them execute without interruption and delay of returning back to the compute area offers a non-trivial advantage over the synchronous approach. Somewhat surprisingly, the aggregation approach that reduces the number of IO calls via data aggregation did not yield performance advantages over just caching the requests. This suggests that for the configuration of the Spider file system at least, reducing the number of concurrent clients to the IO system is the advantageous approach. Additional efforts to reduce the number of IO calls do not yield benefits.

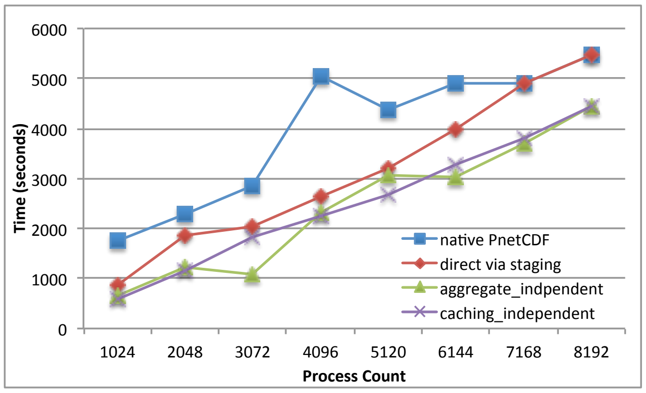


Figure 5: Writing performance on JaguarPF one staging node (12 processes)

4. CONCLUSIONS AND FUTURE WORK

This paper demonstrates that it is possible, using a very small amount of additional staging resources, to transparently improve IO performance without requiring changing the application source code. By using an efficient RPC layer, like Nessie, to communicate efficiently with a staging area, additional processing or simply changing the synchronous nature of IO requests can improve IO performance without changing the file layout in storage. The overheads added by the RPC layer are minimized on Red Storm achieving nearly 100% efficiency for the data movement. While Thunderbird was slower, it still achieved 75% efficiency for large block data movement.

The staging data processing techniques demonstrated that simply synchronously offloading the IO calls to a small staging area is not sufficient for improving performance. Different techniques can innovate in the staging area to deal with large data blocks moving to storage in a desirable format. Additional investigations into the file system parameters and varying the staging area size for different techniques will further explain the tradeoffs available for achieving improved IO performance without having to change the IO routines in the application.

This initial work is leading into some additional studies to be represented in a longer work in the near future. First, the general lack of difference between the aggregating and just caching performance suggests some file system configuration parameter is penalizing the performance of the aggregation approach. Testing is underway on RedSky at Sandia Labs, also using a Lustre file system, and is showing a different performance profile. Second, the additional tests strongly suggested by the data is to scale the number of staging nodes employed with the client count. Those tests are also underway.

Additional tests using collective versions of the aggregate independent and caching independent processing techniques are also available and being tested. These results will also be included in the next paper about this project.

5. REFERENCES

[1] Cray XE6^{TM}specifications.

[2] InfiniBand frequently asked questions.

[3] IOR interleaved or random HPC benchmark. http://sourceforge.net/projects/ior-sio/.

[4] *10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10 2001), 7-9 August 2001, San Francisco, CA, USA*. IEEE Computer Society, 2001.

[5] Cray XMT^{TM}system overview, 2008.

[6] H. Abbasi, M. Wolf, and K. Schwan. LIVE data workspace: A flexible, dynamic and extensible platform for petascale applications. In *CLUSTER '07: Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 341–348, Washington, DC, USA, 2007. IEEE Computer Society.

[7] N. Ali, P. H. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. B. Ross, L. Ward, and P. Sadayappan. Scalable I/O forwarding framework for high-Performance computing systems. In *CLUSTER*, pages 1–10, 2009.

[8] I. T. Association. InfiniBand Architecture Specification, Release 1.2, October 2004.

[9] R. Brightwell, R. Riesen, B. Lawry, and A. B. Maccabe. Portals 3.0: protocol building blocks for low overhead communication. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, April 2002.

[10] R. Brightwell, K. D. Underwood, and C. Vaughan. An evaluation of the impacts of network bandwidth and dual-core processors on scalability. In *International Supercomputing Conference*, Dresden, Germany, June 2007.

[11] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery*, 2(1):015001 (31pp), 2009.

[12] J. M. Dennis, J. Edwards, R. Loy, R. Jacob, A. A. Mirin, A. P. Craig, and M. Vertenstein. An application level parallel i/o library for earth system models. *International Journal of High Performance Computing Applications*, 2011.

[13] P. Dickens and R. Thakur. Improving collective i/o performance using threads. In *Parallel and Distributed Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, pages 38 –45, apr. 1999.

[14] C. Docan, M. Parashar, and S. Klasky. DataSpaces: An interaction and coordination framework for coupled simulation workflows. *HPDC '10: Proceedings of the 18th international symposium on High performance distributed computing*, 2010.

[15] J. Fu, N. Liu, O. Sahni, K. E. Jansen, M. S. Shephard, and C. D. Carothers. Scalable parallel i/o alternatives for massively parallel partitioned solver systems. In *IPDPS Workshops*, pages 1–8. IEEE, 2010.

[16] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams. An overview of trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[17] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. *SC Conference*, 0:39, 2003.

[18] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *CLADE 2008 at HPDC*, Boston, Massachusetts, June 2008. ACM.

[19] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu. Six degrees of scientific data: Reading patterns for extreme scale science io. In *Proceedings of The 20th International ACM Symposium on High-Performance Parallel and Distributed Computing*, San Jose, CA, June 2011. ACM.

[20] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *In Proceedings of IPDPS'09, May 25-29, Rome, Italy*, 2009.

[21] NCAR. Community earth system model, Aug. 2010.

[22] A. Nisar, W.-k. Liao, and A. Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[23] R. A. Oldfield, P. Widener, A. B. Maccabe, L. Ward, and T. Kordenbrock. Efficient data-movement for lightweight I/O. Barcelona, Spain, Sept. 2006.

[24] R. A. Oldfield, P. Widener, A. B. Maccabe, L. Ward, and T. Kordenbrock. Lightweight I/O for scientific applications. *Cluster Computing, 2006 IEEE International Conference on*, pages 1–11, 25-28 Sept. 2006.

[25] M. Polte, J. Simsa, W. Tantisiriroj, G. Gibson, S. Dayal, M. Chainani, and D. Uppugandla. Fast Log-Based concurrent writing of checkpoints. In *Petascale Data Storage Workshop, 2008. PDSW '08. 3rd*, pages 1–4, Nov. 2008.

[26] D. Wallace and S. Sugiyama. Data virtualization service. Cray User's Group, 2008.

[27] W. Yu and J. Vetter. ParColl: Partitioned collective I/O on the cray XT. *Parallel Processing, International Conference on*, 0:562–569, 2008.

[28] W. Yu, J. Vetter, and H. Oral. Performance characterization and optimization of parallel I/O on the cray XT. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–11, April 2008.

[29] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, S. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. PreDatA - preparatory data analytics on Peta-Scale machines. In *In Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium, April, Atlanta, Georgia*, 2010.

6. ACKNOWLEDGEMENTS

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

An award of computer time was provided by the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program. This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.