Understanding I/O Performance Using I/O Skeletal Applications

Jeremy Logan[1,2], Scott Klasky[1], Hasan Abbasi[1], Qing Liu[1], George Ostrouchov[1], Manish Parashar[3], Norbert Podhorszki[1], Yuan Tian[4], and Matthew Wolf[1,5]

[1] Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA
[2] University of Tennessee, Knoxville, Tennessee, USA
[3] Rutgers, The State University of New Jersey, Piscataway, New Jersey, USA
[4] Auburn University, Auburn, Alabama, USA
[5] Georgia Tech., Atlanta, Georgia, USA

Abstract. We address the difficulty involved in obtaining meaningful measurements of I/O performance in HPC applications, as well as the further challenge of understanding the causes of I/O bottlenecks in these applications. The need for I/O optimization is critical given the difficulty in scaling I/O to ever increasing numbers of processing cores. To address this need, we have pioneered a new approach to the analysis of I/O performance using automatic generation of I/O benchmark codes given a high-level description of an application's I/O pattern. By combining this with low-level characterization of the performance of the various components of the underlying I/O method we are able to produce a complete picture of the I/O behavior of an application.

We compare the performance measurements obtained using Skel, the tool that implements our approach, with those of an instrumented version of the original application to show that our approach is accurate. We demonstrate the use of Skel to compare the performance of several I/O methods. Finally we show that the detailed breakdown of timing information produced by Skel provides better understanding of the reasons for the performance differences between the examined I/O methods. We conclude that our approach facilitates faster, more accurate and more meaningful I/O performance testing, allowing application I/O performance to be predicted, and new systems and I/O methods to be evaluated.

1 Introduction

Understanding and optimizing the I/O performance of high-performance scientific applications is critical for the success of many of these applications. As larger and faster platforms are introduced, the higher fidelity applications that run on these platforms produce increasingly massive data sets. At the same time, the increase in computational performance of supercomputers is greatly outpacing the I/O bandwidth of these machines [1]. To achieve reasonable I/O performance despite this growing gap, increasingly sophisticated I/O techniques are required.

C. Kaklamanis et al. (Eds.): Euro-Par 2012, LNCS 7484, pp. 77–88, 2012.

The design, implementation, and subsequent optimization of these techniques require sophisticated tools that allow the performance of I/O methods to be rapidly understood.

Meaningful measurements of I/O performance on large-scale systems are currently difficult and time consuming to acquire. One approach is to instrument a representative application with timing code to measure I/O performance. This approach presents unnecessary complexity to an I/O developer since building and running the application typically requires detailed knowledge that is unrelated to the task of I/O performance measurement.

A more manageable approach is to use *I/O kernels*, such as the FLASH-IO benchmark routine [2] and the S3D I/O kernel [3], codes that include the I/O routines from a target application but which typically have had computation and communication operations removed. While providing accuracy by performing the same I/O pattern as the application, I/O kernels have significantly shorter total execution time since they do not include the application's computational workload. Furthermore, I/O kernels may not retain all of the application's dependencies on external libraries, simplifying their use on new systems. The use of I/O kernels still presents several problems, including (i) They are seldom kept up-to-date with changes to the target application; (ii) They often retain the same cumbersome build mechanism of the target application; (iii) Different I/O kernels do not measure performance the same way; (iv) An I/O kernel is not necessarily available for every application of interest.

To address the weaknesses of current approaches, we have investigated a new approach to I/O performance measurement using *I/O skeletal applications*. These are generated codes that, like I/O kernels, include the same set of I/O operations used by an application while omitting computation and communication. In contrast to I/O kernels, I/O skeletal applications are built from the information included in a high-level I/O descriptor, and do not directly duplicate any of the application source code. Our skeletal applications share the advantages of I/O kernels, providing an accurate and concise benchmark for I/O performance. Furthermore, this approach directly addresses the four problems of I/O kernels identified above: (i) Since skeletal applications are automatically generated, they are trivial to reconstruct after a change to the application; (ii) All skeletal applications share the same relatively simple build mechanism; (iii) All skeletal applications share the same flexible measurement techniques, making it easier to perform apples to apples comparisons; (iv) Given its I/O descriptor, it is a simple process to generate an I/O skeletal application for any application.

To implement our I/O skeletal approach, we have created *Skel*. Skel generates skeletal applications based on the high-level I/O descriptors used in the Adaptive IO System (ADIOS) [4]. We have extended ADIOS to include a mechanism for timing the low-level operations performed during calls to the ADIOS API functions. Integration with this augmented ADIOS library allows more detailed measurements than are typically available from I/O kernels, enabling a user to determine, for instance, how much of the total I/O time is spent performing interprocess communication or low-level I/O operations.

In the next Section, we provide an overview of the Skel tool that implements our generative approach to creating I/O skeletal applications. In Section 3, we present some brief background of declarative I/O and ADIOS. Section 4 examines the validity of the generated skeletal applications. In Section 5, we describe the use of Skel to perform a detailed analysis of three I/O methods. We describe the related work in Section 6. Finally, Section 7 concludes the paper and presents an overview of future work.

2 The Skel System

We have implemented a tool, *Skel* [5], which supports the creation and execution of I/O skeletal applications. Skel consists of a set of python modules, each of which performs a specific task. The *skel-xml* module produces the adios-config file to be used by the skeletal application. The *skel-parameters* module creates a default parameter file based on the configuration file, which is then customized by the user. Modules *adios-config* and *skel-config* interpret the inputs, the adios-config file, which already exists for users of the ADIOS XML-based API, and the parameter file, which provides all other information needed to create a skeletal application. Based solely on data from those two input files, the *skel-source* module generates the source code, in either C or Fortran, that comprises the skeletal application. The *skel-makefile* module generates a makefile for compiling and deploying the skeletal application based on a platform specific template. Finally, the *skel-submit* module generates a submission script, also based on a template, suitable for executing the skeletal application on the target platform. Due to space limitations, we refer the reader to [5] for a more detailed discussion of the design of the Skel system.

In order to obtain more detailed timings of the ADIOS I/O methods, we have extended ADIOS with a mechanism for providing such timing information. The optional, low-level timing information is collected on each application core. We have also instrumented the I/O methods of interest with customized timing instructions using our timing mechanism. The timing functionality, though simple, is designed to be extensible, so that authors of new I/O methods can instrument their codes and extend the timer set to include arbitrary timing targets.

3 Background

Our implementation of Skel depends on the use of the declarative I/O mechanism that is present in the Adaptive IO System (ADIOS) [4]. ADIOS is a componentized I/O library designed to achieve excellent I/O performance on leadership class supercomputers. It provides a mechanism for externally describing an application's I/O requirements using an XML file that is termed the *adios-config* file. The mechanism provides a clear description of the structure of the application's I/O that is separate from the application source code.

In addition to describing the application's I/O pattern, the adios-config file allows the user to toggle between the various I/O methods[1] that are offered by ADIOS. This is particularly convenient because it allows different I/O mechanisms to be substituted without the need to change or recompile the application's source code. Such flexibility is a major strength of ADIOS since it facilitates the optimization of an application's I/O performance.

3.1 I/O Methods

In this paper we focus on three of the synchronous write methods offered by ADIOS, namely `POSIX`, `MPI_LUSTRE` and `MPI_AMR` [4]. We chose these methods both to show the general applicability of the Skel tool to different I/O methods, and to explore the performance of these methods over a range of scales and applications. Each of the methods takes a different approach to handling the application's I/O. In particular, the methods each produce a different number of files as shown in Fig. 1.

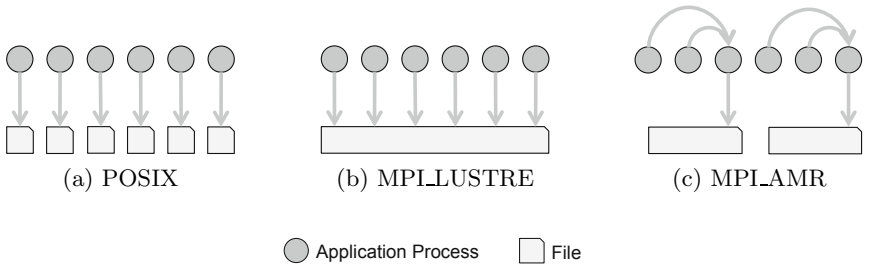


Fig. 1. Comparison of the three I/O methods

The `POSIX` method [6] was designed to take advantage of the concurrency of parallel file systems. With `POSIX`, each writing process is responsible for writing data to its own output file, as illustrated in Fig. 1a. An index file is then written by just one of the processes. So the use of the `POSIX` method results in the index file, along with a subdirectory containing all of the files written individually by the application processes.

The `MPI_LUSTRE` method, in contrast to the `POSIX` method, writes all data to a single file. Each process writes its data independently to an assigned location in the file as shown in Fig. 1b. This method is designed to work well with the striping scheme of the Lustre file system, thus the locations assigned to the writers are chosen to coincide with the beginning of a Lustre stripe, and the file's stripe size is carefully chosen so that each stripe accommodates the data from a single application process.

[1] In this paper the term *method* should be taken to mean a runtime selectable technique for performing the basic ADIOS operations (`adios_open`, `adios_write`, etc.).

Finally, the `MPI_AMR` method is a more sophisticated technique in which each of a subset of application processes acts as an aggregator for a group of peers. This method results in a collection of separate files, with one file corresponding to each of the aggregators. As can be seen in Fig. 1c, this results in fewer separate files than are produced by the `POSIX` method, but more than the single file written by `MPI_LUSTRE`. This strategy, in our experience, often achieves better throughput when scaled to large (>10,000) core counts.

These three methods all produce ADIOS-BP formatted files. ADIOS-BP is a structured, self-describing and resilient format for scientific data. The ADIOS-BP format was designed to provide efficient writing and reading [7,8] by allowing parallel I/O operations to be performed independently and supporting flexibility in the layout of data on the file system.

4 Validation of Skel

To understand whether an automatically generated skeletal application is an accurate predictor of application I/O performance, we compare performance measured with the skeletal apps with performance measured by applications and I/O kernels. We focus on two applications: CHIMERA [9] and GTS [10]. For both of these we compare the observed performance of our I/O skeletal application with that of the corresponding I/O kernel or application.

4.1 Test Platforms

Jaguar is a Cray supercomputer located at the Oak Ridge Leadership Computing Facility (OLCF) at Oak Ridge National Laboratory. Our experiments coincided with an upgrade of the Jaguar machine. The original configuration (Jaguar-XT5) consisted of 18688 compute nodes, each home to two Opteron 2435 "Istanbul" six-core processors and 16 GB of memory, and connected with Cray's Seastar interconnection network. The second configuration of Jaguar (Jaguar-XK6) had 9216 compute nodes, each containing a single 16 core AMD "Interlagos" processor and 32 GB of memory, and connected with the Gemini interconnect [11].

Sith is a 1280 core Linux cluster also located at the Oak Ridge National Laboratory. Sith consists of forty compute nodes connected with an InfiniBand interconnection network. Each node contains four 2.3 GHz 8 core AMD Opteron processors and 64 GB of memory [12].

All tests utilize Spider [13], the OLCF center-wide Lustre file system, which provides a total of 10.7 Petabytes of disk space. Spider consists of three separate Lustre file systems. Our experiments were performed using the widow1 file system, which offers 5 PB of storage space distributed over 672 OSTs.

4.2 CHIMERA

CHIMERA is a multi-dimensional radiation hydrodynamics code designed to study core-collapse supernovae [9]. An I/O kernel based on the CHIMERA application is the focus of this set of experiments. The comparison involved a weak

scaling experiment where approximately 10 MB of data was written by each core. Each test was repeated 50 times, and the results are presented in Fig. 2. Each graph shows the average observed throughput for all of the runs that used that I/O method. The error bars represent the minimum and maximum throughput that were observed during these runs.

Parameters may be used to guide some of the ADIOS methods. The `POSIX` method does not take any parameters, but both `MPI_LUSTRE` and `MPI_AMR` allow method parameters to be specified. For the `MPI_LUSTRE` tests, we used `stripe_count=160, stripe_size=10485760, block_size=10485760`. The parameters for the `MPI_AMR` tests were `stripe_count=1, stripe_size=4194304, block_size=4194304, num_aggregators=N, merging_pgs=0`. We varied the number of aggregators depending on the number of cores, keeping the aggregator count fixed at one aggregator for every 4 cores.

The results of these tests are shown in Fig. 2. The graphs show the mean throughput seen during the tests, with the error bars indicating the minimum and maximum throughput observed. Also shown are the computed relative error values for the skeletal application with respect to both maximum and average throughput.[2] The relative error of the skeletal application as compared with the CHIMERA I/O kernel is less than 10% for over 93% of the cases shown.

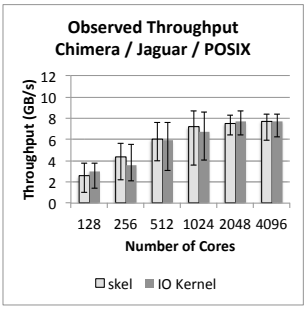
4.3 GTS

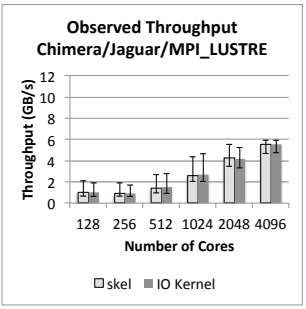
The GTS application [10] is a first principles fusion microturbulence code that studies turbulent transport of energy. GTS uses a generalized geometry to solve the realistic geometries from many fusion reactors used today. For these tests, we looked at weak scaling performance, with each core writing approximately 55 MB of data. The method parameters used for the `MPI_LUSTRE` tests were `stripe_count=160, stripe_size=57344000, block_size=57344000`. For the `MPI_AMR` tests, we used parameters `stripe_count=1, stripe_size=4194304, block_size=4194304, num_aggregators=N, merging_pgs=0`. The number of aggregators for the `MPI_AMR` method varied, with one aggregator used for every 16 processor cores.

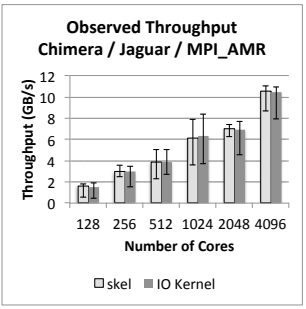
Each individual test was repeated 25 times and the results are shown in Fig. 3. Again, we have calculated the relative error for both the average and maximum throughput values. For these tests we observe that the skeletal application yields results that are within 10% of the values given by the GTS application for approximately 71% of the cases.

These results are extremely positive, particularly when we consider that skel will be most useful at larger scales with larger core counts. All cases involving 1024 or more cores, including both CHIMERA and GTS, are accurate to within 10% of the corresponding application or I/O kernel measurements.

[2] We have omitted a comparison of the minimum throughput as we have found that poor performance due to I/O contention with other jobs can lead to arbitrarily poor performance, making this measurement difficult to reproduce, even among separate runs of the same test.

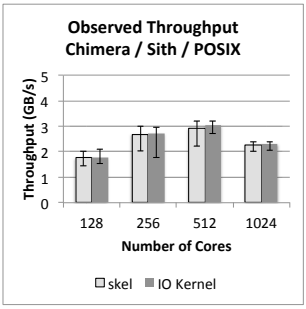


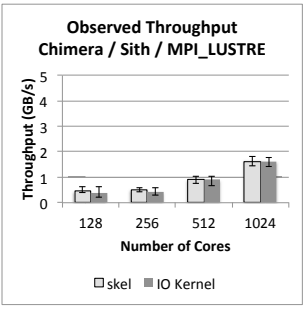


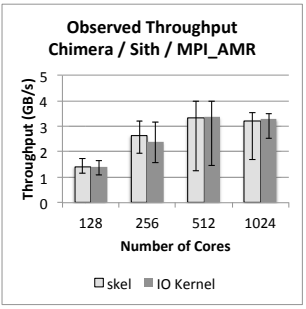


CHIMERA on Jaguar, maximum throughput % error						
Cores	128	256	512	1024	2048	4096
POSIX	1.23299306	1.16572625	-0.7565853	0.91980066	-4.3526464	0.26457144
MPI_LUSTRE	6.35053155	7.18255015	-4.5819049	-7.0988589	4.96703107	-0.9504542
MPI_AMR	-1.1075423	1.60319245	0.81126651	-6.1477011	-2.7247985	0.22539071

CHIMERA on Jaguar, average throughput % error						
Cores	128	256	512	1024	2048	4096
POSIX	-15.586524	22.9848149	1.76922079	6.63736976	-2.5146621	-0.3474594
MPI_LUSTRE	-1.1188266	-2.6544645	-9.2216744	-2.8223058	4.53511945	0.13061827
MPI_AMR	6.63407572	0.32418106	-1.2447502	-4.4013228	0.13788721	0.97436759



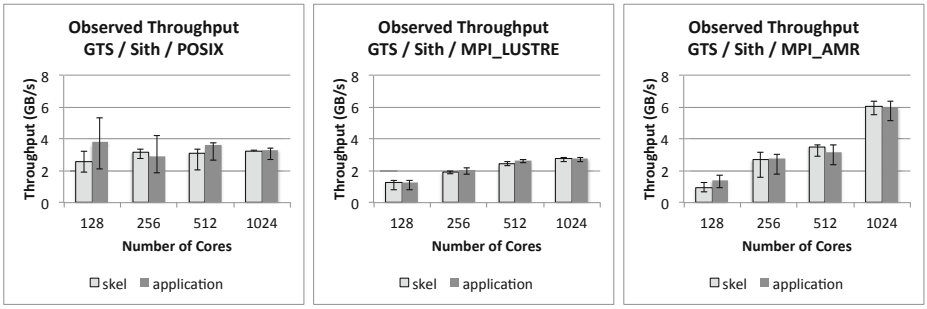




CHIMERA on Sith, maximum throughput % error				
Cores	128	256	512	1024
POSIX	-4.3197237	0.88416525	-0.0444967	0.0639165
MPI_LUSTRE	-1.0537163	-0.1010961	-0.272298	2.86825714
MPI_AMR	4.60492215	1.06613332	0.2709434	1.02694457

CHIMERA on Sith, average throughput % error				
Cores	128	256	512	1024
POSIX	0.20222178	-1.0747786	-3.9282669	-1.0447177
MPI_LUSTRE	25.8798844	19.0632635	0.10644743	0.67528339
MPI_AMR	2.00347839	9.93222729	-1.0470642	-2.0402408

Fig. 2. Results of CHIMERA comparison



GTS on Sith, maximum throughput % error				
Cores	128	256	512	1024
POSIX	-38.483027	-19.700923	-10.558645	-3.7543821
MPI_LUSTRE	-0.2314274	-9.1780889	-5.564479	-0.8277953
MPI_AMR	-26.351491	3.7610783	0.20705444	-0.6193706

GTS on Sith, average throughput % error				
Cores	128	256	512	1024
POSIX	-31.91049	8.61128093	-13.370227	-2.184351
MPI_LUSTRE	1.36451053	-5.587962	-6.9181146	-0.8563423
MPI_AMR	-32.94111	-3.6202477	9.61607405	1.06179842

Fig. 3. Results of GTS comparison

5 Using Skel to Study I/O Performance

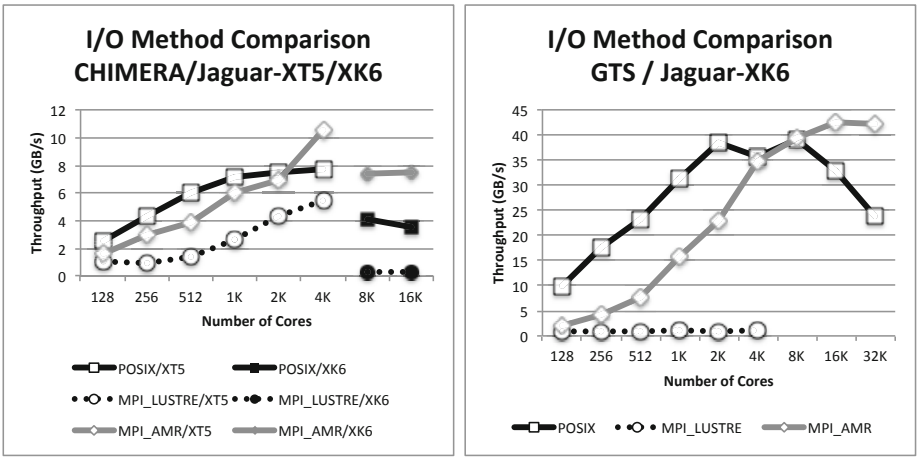
A common challenge in high-performance computing is determining which I/O method to use for a given situation. In general, the answer depends on a great many factors including application I/O pattern, frequency of I/O, storage hardware, file system type and configuration, network performance and system utilization to name a few. An intended use of Skel is to provide a mechanism for rapidly exploring I/O space in order to select an I/O method for arbitrary situations. This is useful to middleware developers for verifying and testing I/O methods, and to end users to assist in manual selection of I/O methods. In the future the mechanism should prove useful in guiding the autonomous selection of I/O methods. To illustrate this use of Skel, we have performed a comparison of three of the I/O methods available in ADIOS: `POSIX`, `MPI_LUSTRE` and `MPI_AMR`.

5.1 CHIMERA

The data from the CHIMERA skeletal application runs is shown in Fig. 4a, including two additional data points that illustrate the results of continuing these tests on the newly upgraded Jaguar-XK6. It can be seen that the `POSIX` method provides the highest throughput for most of the runs, with the `MPI_AMR` method offering better performance only for the 4096 core case. This agrees with our experience with these methods, as `MPI_AMR` has been observed to achieve better performance than `POSIX` at higher core counts. The number of cores at which it becomes advantageous to use the `MPI_AMR` method varies by application and platform, and also relies on the parameters used for `MPI_AMR`.

The CHIMERA I/O skeletal application provided us with a first glimpse at the I/O performance of Jaguar's XK6 configuration. The 8192 core and 16384 core cases were run immediately after the Jaguar-XK6 configuration became

available. It appears that there may be a slight decrease in I/O performance for the `POSIX` and `MPI_AMR` methods, but the `MPI_LUSTRE` method exhibited truly poor performance on the new hardware. We will return to this issue in the next section.



(a) CHIMERA (b) GTS

Fig. 4. Comparison of I/O methods on Jaguar

5.2 GTS

Next we compared these same three I/O methods using a skeletal application for GTS. Each run was repeated 25 times, and the results are summarized in Fig. 4b. Again we see that the `POSIX` method provides the best performance among these three methods up to 8192 cores.[3] Beyond 8192 cores, however, the `MPI_AMR` method provides greater throughput than the other methods.

A surprising result is the particularly poor performance of the `MPI_LUSTRE` method. To investigate this anomaly, we examined the more detailed timing results available from Skel. We chose a single representative run of the 1024 core GTS / `MPI_LUSTRE` experiment, and looked at the performance for each of the cores, focusing on communication and I/O timings, and sorting by total time. The result is shown in Fig. 5. It can be seen that in both cases the times for the I/O operations are roughly equal for all of the overall slowest cores. However the communication times show dramatically different behavior. On Jaguar-XK6, a few of the cores exhibit very large communication times, with the slowest taking nearly 55 seconds. This view led to a quick diagnosis of the cause of the poor performance seen with the `MPI_LUSTRE` method, an expensive global collective communication operation that is not used by the other two methods.

[3] These results are not directly comparable to those found in [5], since those tests were performed on Jaguar-XT5, and used a different file system partition.

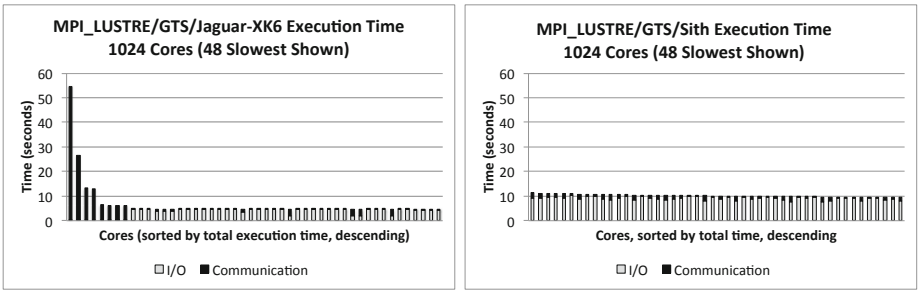


Fig. 5. Comparison of GTS execution times using MPI_LUSTRE on Jaguar-XK6 and Sith

6 Related Work

A common I/O performance measurement method is bulk I/O testing, using a tool such as IOR [14] or the NAS parallel benchmark [15, 16]. As with Skel, the bulk I/O testing process is less cumbersome, since it is not necessary to deal with the complexities of an application, however the results may not provide an accurate prediction of the I/O performance that an application would obtain [17].

There are many I/O kernels that are used to benchmark I/O performance. FLASH-IO [2], MADBench2 [18] and S3D-IO are three often-used examples. Our skeletal applications are intended to provide the utility of I/O kernels, while eliminating issues such as difficulty of use, lack of availability, and outdated versions of codes.

The Darshan project [19] is examining the I/O patterns used by applications of interest. Darshan provides a lightweight library for gathering runtime event traces for later examination. The ScalaIOTrace tool [20] also addresses the measurement and analysis of I/O performance. Similar to Darshan, it works by capturing a trace of I/O activities performed by a running application. The multilevel traces may then be analyzed offline at various levels of detail.

7 Conclusion and Future Work

In this paper, we have presented our approach for the automatic generation of I/O skeletal benchmarks, as well as our tool, Skel, which implements this approach. We have examined one application and one I/O kernel and confirmed that the measurements obtained from the I/O skeletal application provide a reasonable estimate of performance. We observed a slightly better correspondence with the CHIMERA I/O kernel than with the GTS Application. In both cases, the performance predictions produced by the skeletal application improved with larger numbers of application processes.

We have conducted a performance comparison of three of the ADIOS write methods using our I/O skeletal technique. We have shown how Skel can be

used to quickly measure the aggregate performance achieved by these methods. Furthermore, we have shown how the more detailed measurements provided by Skel can be leveraged to achieve a deeper understanding of the causes for the observed performance behavior.

In this paper we have focused on writing single restart files using synchronous I/O methods. In the future we will extend Skel to explore the task of writing smaller and more frequent analysis data. We will also investigate how to accurately measure the impact of asynchronous I/O. We expect that this will require generating some computation and communication operations to provide the same effect as those performed by the application. Finally, we will use Skel to investigate reading performance for very large data files.

Acknowledgment. This work was supported in part by the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory. Support was provided by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DEAC02- 05CH11231. Additional support was provided by the SciDAC Fusion Simulation Prototype (FSP) Center for Plasma Edge Simulation (CPES) via Department of Energy Grant No. DE-FG02-06ER54857. This material is based upon work supported by the National Science Foundation under Grant No. 1003228. Support was also provided by the Remote Data Analysis and Visualization Center (RDAV) through National Science Foundation Grant No. 0906324.

References

1. Lang, S., Carns, P., Latham, R., Ross, R., Harms, K., Allcock, W.: I/O performance challenges at leadership scale. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, pp. 40:1–40:12. ACM, New York (2009)
2. FLASH I/O benchmark routine – parallel HDF5, `http://www.ucolick.org/~zingale/flash_benchmark_io/`
3. Chen, J., Choudhary, A., De Supinski, B., DeVries, M., Hawkes, E., Klasky, S., Liao, W., Ma, K., Mellor-Crummey, J., Podhorszki, N., et al.: Terascale direct numerical simulations of turbulent combustion using S3D. Computational Science & Discovery 2, 015001 (2009)
4. ADIOS 1.3 user's manual, `http://users.nccs.gov/~pnorbert/ADIOS-UsersManual-1.3.pdf`
5. Logan, J., Klasky, S., Lofstead, J., Abbasi, H., Ethier, S., Grout, R., Ku, S.H., Liu, Q., Ma, X., Parashar, M., Podhorszki, N., Schwan, K., Wolf, M.: Skel: generative software for producing skeletal I/O applications. In: The Proceedings of D^3science (2011)
6. Lofstead, J., Zheng, F., Klasky, S., Schwan, K.: Adaptable, metadata rich IO methods for portable high performance IO. In: IEEE International Symposium on Parallel Distributed Processing, IPDPS 2009, pp. 1–10 (May 2009)
7. Tian, Y., Klasky, S., Abbasi, H., Lofstead, J., Grout, R., Podhorski, N., Liu, Q., Wang, Y., Yu, W.: EDO: improving read performance for scientific applications through elastic data organization. In: Proceedings of IEEE Cluster (2011)

8. Lofstead, J., Polte, M., Gibson, G., Klasky, S., Schwan, K., Oldfield, R., Wolf, M., Liu, Q.: Six degrees of scientific data: reading patterns for extreme scale science IO. In: Proceedings of the 20th International Symposium on High Performance Distributed Computing, HPDC 2011, pp. 49–60. ACM, New York (2011)
9. Messer, O.E.B., Bruenn, S.W., Blondin, J.M., Hix, W.R., Mezzacappa, A., Dirk, C.J.: Petascale supernova simulation with CHIMERA. Journal of Physics: Conference Series 78(1), 012049 (2007)
10. Wang, W.X., Lin, Z., Tang, W.M., Lee, W.W., Ethier, S., Lewandowski, J.L.V., Rewoldt, G., Hahm, T.S., Manickam, J.: Gyro-kinetic simulation of global turbulent transport properties in tokamak experiments. Physics of Plasmas 13(9), 092505 (2006)
11. Titan configuration and timeline, `http://www.olcf.ornl.gov/titan/system-configuration-timeline/`
12. OLCF computing resources: Sith, `http://www.olcf.ornl.gov/computing-resources/sith/`
13. Shipman, G., Dillow, D., Oral, S., Wang, F.: The spider center wide file system: From concept to reality. In: Proceedings, Cray User Group (CUG) Conference, Atlanta, GA (2009)
14. IOR HPC Benchmark, `http://sourceforge.net/projects/ior-sio/`
15. Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., Yarrow, M.: The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Tech. rep. (1995)
16. Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R., et al.: The NAS parallel benchmarks summary and preliminary results. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, pp. 158–165. IEEE (1991)
17. Shan, H., Shalf, J.: Using IOR to analyze the I/O performance for HPC platforms. In: Cray Users Group Meeting (CUG) 2007, Seattle, Washington (May 2007)
18. MADbench2, `http://crd.lbl.gov/~borrill/MADbench2/`
19. Darshan, petascale I/O characterization tool, `http://www.mcs.anl.gov/research/projects/darshan/`
20. Vijayakumar, K., Mueller, F., Ma, X., Roth, P.C.: Scalable I/O tracing and analysis. In: Proceedings of the 4th Annual Workshop on Petascale Data Storage, PDSW 2009, pp. 26–31. ACM, New York (2009)