Skel: Generative Software for Producing Skeletal I/O Applications

Jeremy Logan*, Scott Klasky*, Jay Lofstead‡,
Hasan Abbasi*, Stephane Ethier‖, Ray Grout††, Seung-Hoe Ku**, Qing Liu*,
Xiaosong Ma§, Manish Parashar¶, Norbert Podhorszki*, Karsten Schwan†, Matthew Wolf†
*National Center for Computational Science, Oak Ridge National Laboratory, Oak Ridge, TN
¶Department of Electrical & Computer Engineering, Rutgers, The State University of New Jersey, Piscataway, NJ
**Courant Institute of Mathematical Sciences, New York University, New York, NY
§Department of Computer Science, North Carolina State University, Raleigh, NC
‖Plasma Physics Laboratory, Princeton University, Princeton, New Jersey
††National Renewable Energy Laboratory, Golden, CO
†College of Computing, Georgia Tech, Atlanta GA
‡Sandia National Laboratories, Albuquerque, NM

***Abstract*—Massively parallel computations consist of a mixture of computation, communication, and I/O. As part of the co-design for the inevitable progress towards exascale computing, we must apply lessons learned from past work to succeed in this new age of computing. Of the three components listed above, implementing an effective parallel I/O solution has often been overlooked by application scientists and was usually added to large scale simulations only when existing serial techniques had failed. As scientists' teams scaled their codes to run on hundreds of processors, it was common to call on an I/O expert to implement a set of more scalable I/O routines. These routines were easily separated from the calculations and communication, and in many cases, an I/O kernel was derived from the application which could be used for testing I/O performance independent of the application. These I/O kernels developed a life of their own used as a broad measure for comparing different I/O techniques. Unfortunately, as years passed and computation and communication changes required changes to the I/O, the separate I/O kernel used for benchmarking remained static no longer providing an accurate indicator of the I/O performance of the simulation making I/O research less relevant for the application scientists.**

In this paper we describe a new approach to this problem where I/O kernels are replaced with skeletal I/O applications automatically generated from an abstract set of simulation I/O parameters. We realize this abstraction by leveraging the ADIOS [1] middleware's XML I/O specification with additional runtime parameters. Skeletal applications offer all of the benefits of I/O kernels including allowing I/O optimizations to focus on useful I/O patterns. Moreover, since they are automatically generated, it is easy to produce an updated I/O skeleton whenever the simulation's I/O changes. In this paper we analyze the performance of automatically generated I/O skeletal applications for the S3D and GTS codes. We show that these skeletal applications achieve performance comparable to that of the production applications. We wrap up the paper with a discussion of future changes to make the skeletal application better approximate the actual I/O performed in the simulation.

I. Introduction

As simulations begin to scale to hundreds of thousands of processors, I/O is becoming an increasingly painful area because an ever increasing percentage of time is spent in I/O, communication, and memory movement causing less time to be spent on the calculations. Recently, the Department of Energy has funded three large-scale co-design centers whose aim is to use three application areas, Nuclear Energy, Combustion, and Extreme Materials, as a basis to co-design the fundamental science applications, the exascale hardware, and the associated data analytic capabilities in a holistic system. To realize this vision, teams need to create skeletal and compact applications for use in hardware simulators to better understand the way that algorithms, hardware, and science impact one another. An existing set of such small applications is Mantevo [2], [3] created by Sandia Labs. This package contains miniapplications that embody essential performance characteristics of key applications, minidrivers that exercise Trilinos [4] math pacakges, and paramaterizable application proxies that can be calibrated to mimic the performance of a large scale application. Similar to the Mantevo effort, *skeletal applications* are small scale applications which remove the basic floating point calculations but retain the basic communication and I/O in the code. *Compact applications* add a portion of science kernels to these skeletal applications but are generally simple in terms of lines-of-code and software complexity. Decoupling the I/O from the application allows hardware simulators to execute these instructions in a reasonable amount of time facilitating the co-design of the hardware with the algorithms. The inclusion of I/O in skeleton and compact applications improves the usefulness of these tools over what Mantevo currently provides and will more completely represent application activities on potential exascale hardware designs. By keeping these I/O kernels current, these data movement activities can be more accurately represented when modeling new systems.

We believe that a phase transition is necessary for the parallel I/O community. "Custom" I/O kernels such as Flash-IO [5] and MADbench2 [6] should be replaced with automatically created I/O kernels representing the current state of application I/O tasks. Our reasoning is three-fold. First, I/O kernels may be difficult to extract from complex simulations. This barrier

limits the number of kernels to teams that can spend the time and effort to extract a kernel that accurately represents the I/O patterns of the science application. Second, I/O kernels can be difficult to use as their interfaces vary widely between kernels and they are sometimes derived from the already complex simulation interfaces with few changes to make them accessible to anyone but an application expert. Third, I/O kernels quickly become dated and are rarely updated. For example, Argonne National Laboratory maintains a list of 14 parallel I/O benchmarks[1] (updated June 22, 2011) for use by the I/O and co-design communities. Digging deeper into the benchmarks themselves reveals, for example, Flash-IO was last updated 6-22-2001; Chombo, QIO, and SSCA, point to pages not found; and PIO/IOR, MaDbench, and MPI-IO were last updated in 2006. All of these benchmarks quickly become outdated as the I/O demands of simulations evolve and yet they are still used in many publications to illustrate the performance of new I/O methods.

While having a consistent, stable set of benchmarks is certainly desirable for easier apples-to-apples comparisons between different papers, for tests using these benchmarks to be meaningful for scientific application developers making selections for their I/O routines, the benchmarks must be kept relatively current. Moving from static benchmarks extracted from scientific applications to automatically generated skeletal applications achieves this goal. Each generated skeletal application has a version associated with it that can be reported along with the results. This achieves both keeping the benchmarks current and affording easier apples-to-apples comparisons.

In order to create I/O skeletal applications, we have worked closely with many application teams. In fact, several of the application authors are co-authors of this paper. These application scientists agree with the benefit of such skeletal applications and illustrate the seriousness of our team to create I/O kernels that are accurate, automatically created, and openly delivered to the science community. In order to realize our vision, we have utilized the ADIOS framework, which allows I/O experts the ability to create new I/O techniques and try them on any system or with any ADIOS integrated application without changing the source code.

Briefly, in an effort to illustrate how ADIOS features support Skel, we recap ADIOS here. Our team created ADIOS to simplify I/O on the largest supercomputers in the world for application teams. Our goal was to reduce the I/O complexity while delivering "extreme" I/O performance. ADIOS allows scientist to create groups of variables that they want to write out on each process, termed a *Process Group* (PG). Each PG can be written out synchronously or asynchronously using the host of different transport methods that ADIOS provides. Most simulations that use ADIOS have an external XML file, such as the excerpt from the GTC XML configuration file shown in Figure 1. In this illustration, one large array, *electrons*, will be written to the file system using the *MPI* transport method. This

[1]http://www.mcs.anl.gov/~thakur/pio-benchmarks.html

```
<adios-group name="particles"
       coordination-communicator="comm">
  <var name="mype" type="integer"/>
  <var name="nparam" type="integer"/>
  <var name="pes" type="integer"/>
  <var name="nparam*pes" type="integer" />
  <var name="nparam*mype" type="integer" />
  <var name="ntracke" type="integer" />

  <global-bounds dimensions="nparam*pes,ntracke"
                 offsets="nparam*mype,0">
    <var name="electrons" type="real"
         dimensions="nparam,ntracke"/>
  </global-bounds>
</adios-group>
<transport method="MPI" group="particles"/>
```

Fig. 1. GTC I/O descriptor

is a synchronous method in which each participating process arranges data in local, self-describing chunks, writing them to storage in parallel. While this "chunked" style of output is commonly believed to give poor reading performance, extensive testing of this data organization has shown [7], [8], [9] that it allows for arbitrary access patterns for reading and, in general, can give unprecedented read performance for the most common read access patterns used in scientific codes. ADIOS contains many different I/O methods from many different institutions, including MPI, MPI-AMR, MPI-lustre, PHDF5, NETCDF-4, POSIX, DataSpaces, DataTap, NSSI, and EDO. These were created at ORNL, Sandia, Georgia Tech, Rutgers University, and Auburn University. A central goal was to allow any I/O scientist the ability to create new ADIOS transport methods that can be used by the application scientist by just changing the transport method entry in the XML from MPI in the above example to any other transport. Application scientists can also publicly release their XML file to I/O scientist to afford them the ability to optimize their I/O on different platforms. [10]

One of our goals is to abstract I/O APIs away from the actual implementation to allow any I/O scientist the ability to create new transport methods and try them with production science codes. ADIOS uses a generative programming technique [11] to convert the XML code into ADIOS write statements to perform the actual output. Recently this functionality was extended with a new application called *skel* that affords simulation of the I/O patterns of a science code by creating full skeletal applications based on an application's external XML file. The creation of the skeletal applications is steered by an additional XML file containing test parameters. This file is generated by skel and is initially populated with default values that can be tuned by the user. Using only the application specific information in these two files, the skel tool creates source code to mimic the application's I/O operations. Since the XML file describing the I/O is used when running applications that use ADIOS, application scientist can archive the additional XML file describing the input parameters as well as their actual input files so that we can automatically

create new I/O skeletal applications that provide a consistent interface from one application, such as the GTC fusion code, to another, such as the S3D combustion code.

The rest of this paper is organized as follows. First we describe our general vision for leveraging I/O skeletal applications and discuss some of the differences and similarities between traditional I/O kernels and the skeletal applications. We then give a description of the way we generate these skeletal applications discussing not just *writing* I/O kernels, but also *reading* the patterns most common to large scale scientific computing. We then compare and contrast the performance of running the skeletal application to the actual I/O performance for the simulation. Later we look at the work related to this research and compare and contrast this with our techniques. Finally we examine the pros and cons of our approach and explore future work that is necessary to remove bottlenecks in I/O performance paving the way for progress towards extreme scale computing.

II. I/O Skeletal Applications

Initially, the emergence of skeletal applications will have a profound effect on the ability of I/O middleware developers to quickly and accurately test the performance of I/O methods in a context of actual science application usage scenarios. By reducing the time required to evaluate new methods, more time will be available to advance the methods themselves resulting in more efficient I/O and better utilization of the available I/O resources.

The skeletal applications will also improve over time by better approximating the application's I/O performance. The generation process will become simpler with skeletal application parameters being extracted from output files and application traces automatically. Integrating application traces with skel would also provide more realistic control of the timing of I/O operations allowing the skeletal application to mimic the jitter present in the simulations and give a better representation of the amount of time available to move data asynchronously between output events.

We envision a repository containing various I/O descriptors that are available to developers wishing to test a new I/O implementation against realistic I/O requirements. We also envision a database of performance measurements representing a selection of skeletal applications that have been tested on a variety of computing platforms. Such a database would initially assist users in selecting an optimal I/O method for a particular I/O pattern. In the future, intelligent middleware will consult the database to find the optimal method without the user's involvement.

The work presented here represents the first steps in a ripe new research area. For now, we see this project as producing a comprehensive set of skeletal applications representing the I/O patterns of actual scientific simulations. This set of skeletal applications will share a uniform interface, will be easily kept up to date, and will accurately reflect the I/O performance of the original simulations in their evolving forms.

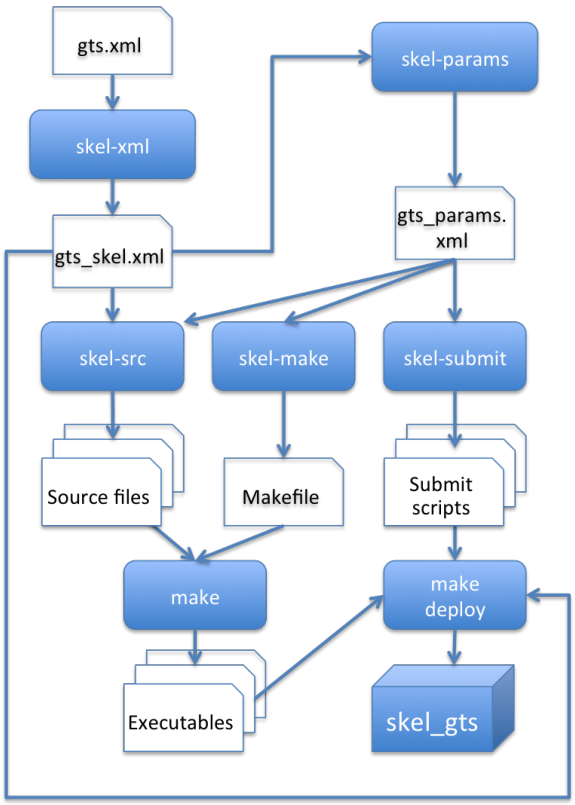

Fig. 2. Skel system overview

III. The Skel System

We have developed skel, a software tool used to create, deploy, and execute skeletal I/O applications. This section details the design of skel and illustrates how it is used to generate skeletal I/O codes.

A. System Structure

Figure 2 illustrates the process of using skel to produce skeletal applications. The system requires only two inputs: the original xml file from the application of interest and a parameter file that the user populates with the details of the tests that are to be run. Except for supplying these input files, generation of a skeletal application using skel requires very little user participation.

Processing begins with a call to skel-xml to prepare the IO descriptor for use with the generated skeletal application. This involves aligning the file with the target language (currently C or Fortran), reordering dimensions as needed. Variable names are rewritten by replacing any characters that would be invalid in the target language, such as mathematical operators and struct accessors, and renaming any variables that conflict with a skel variable or a language keyword, e.g., `enum`, a reserved word in C, is used as a variable name in the XGC1 code. This produces an I/O descriptor that can be safely used for generating the parameter file and the skeletal application source files as well as for execution of the skeletal application.

Next, a parameter file is produced from the app_skel.xml file. The generated parameter file contains reasonable defaults for most of the scalar variables used as array sizes that are typically produced dynamically by the application and do not appear in the XML descriptor. The file also contains a default set of tests including each of the I/O groups in the input file. We expect that users will typically make changes to the parameter file to reflect the particular tests to be performed. We discuss the details of the parameter file in the next section.

Once the parameters have been edited to the user's satisfaction, the output files that make up the skeletal application are generated. The commands skel-src, skel-make, and skel-submit are used (as shown in Figure 2) to produce, respectively, the skeletal application source code (in C or Fortran), makefile, and submission scripts. The generated makefile contains targets for compiling the skeletal application and deploying the relevant files to the target filesystem as well as for performing the earlier tasks of producing the skel XML file and parameter file. In fact, we have written a bootstrapping script that creates an initial project makefile, which, once created, hides all of the complexity of the various skel calls behind a few simple make targets. Thus, despite the apparent complexity of the diagram, the tool is very easy to use, requiring only the ability to edit the XML code in the parameter file.

B. Skeletal application parameters

We have chosen to place all settings and parameters for the automatically generated codes into a single XML parameter file. This contrasts with many I/O benchmarks that provide control via command line arguments or using a flat configuration file. Though XML is often viewed as a bulky format, it offers several advantages that we felt justified its use. First, we are building from an existing XML file for the I/O specification so using XML to specify parameters means that we are not dealing with an additional format. Second, the amount of data required for the parameters is relatively small making the overhead of the text-based XML encoding inconsequential. Third, the Measurement data from the skeletal applications is also generated as XML. In fact, we embed the skel parameter data directly in the results file providing a record of exactly what tests generated those results. This is simple to do with XML and the parameters are simple to extract should the tests need to be repeated.

The parameters to the skeletal application can be divided into two categories. The first category includes the details about the application. These details consist mainly of values for some of the application variables. Generally the skeletal application is not concerned with values of simulation variables, but only with the size and memory layout of data. However, some of the application's scalar variables will typically be used to represent the size of stored arrays. It is these values that are important in specifying the I/O pattern. Since the values of these variables are not located in the I/O descriptor, we must supply them as parameters. This category also includes parameters to specify values for the array elements in the skeletal application. By initializing arrays to known values, such as MPI rank or array location, it is possible to verify the correctness of the underlying I/O method.

The other category of parameters includes those related to the execution of the skeletal application. Skel users may specify any number of individual tests by providing the test type (e.g., write, read), groups to be included in the test, I/O method and corresponding parameters, and directions for handling output files. The specified tests are collected into batches for which individual submission scripts are created by skel-submit. Batches are further parameterized by core count and target platform.

```
<skel-params>
  <adios-group name="restart">
    <scalar name="inum" value="1000000" />
    <scalar name="inphase" value="8" />
    <array name="sp__pct__gid" dims="inum"
           fill-method="rank" />
    <array name="sp__pct__phase"
           dims="inum,inphase"
           fill-method="rank" />
  </adios-group>

  <batch name="write_read" cores="1024"
      target ="pbs_cray">
    <test type="write" method="POSIX"
          group="restart" />
    <test type="read_all" group="restart" />
  </batch>
</skel-params>
```

Fig. 3. An excerpt from an XGC1 parameter file

An excerpt from a parameter file is shown in Figure 3. The example shows parameters for testing the restart group for the XGC1 simulation. Here we provide values for the two scalars that are used as array dimensions. Other scalars in the restart group are omitted from the parameters file. The `fill-method` attribute specifies that the arrays be filled with the rank of the writing process. Finally we specify a single batch consisting of two tests: one write test and one read test. We note that the structure of the parameters file is provided by skel and the user needs only to fill in the desired values.

C. Code generation

Implemented in Python, skel's approach to code generation is simple yet powerful. Skel uses a combination of template text for the fixed parts of the skeletal application and simple traversals of the data in the XML input files to generate application specific code in the target language. Generation of each source file is done by iterating over the parts of the target application emitting appropriate code for each part. For instance, in the case of C files, this includes:

- include statements
- main function and MPI setup
- variable declarations and memory allocation
- I/O code interspersed with the associated timing code
- release allocated memory
- finalization and closing code block

We use the Python minidom parser for simple parsing of each XML files into a document object model (DOM) and provide an abstract representation of the XML content by using wrapper classes to hide the details of the DOM from the rest of the code.

Each test specified in the parameter file initiates the generation of a separate executable that implements the test. Thus, the Makefile generated for the skeletal application must provide instructions for building the executable for each test. Likewise, each batch in the parameter file induces the generation of a submission script that includes code to run the corresponding executable for each test in the batch. The deploy target in the generated Makefile copies all of the executables, submission scripts, and XML descriptors into a user specified directory from which test will be run.

D. Specifying read patterns

While writing is typically done just one way, that is by writing out all values for every variable in a group, reading patterns of interest are considerably more varied [7]. In the case of a restart file, it is often the case that it is necessary to read exactly the same pattern that was written using the same number of processors. However, there may be times when the restart data must be read by a different number of processes, for instance when the number of available processors has decreased since a restart was written, and we want to continue the execution of a simulation using fewer cores.

For analysis data however, the situation is more complicated. Analysis and visualization work is typically performed on a significantly smaller number of processors than were used to generate the data. And quite often the focus is on some subset of the data, either a subset of the variables in a group, or a spatial subset of some or all of the variables, e.g., a single plane in a 3D space, or a sub-volume.

So getting a true picture of read performance will require looking at particular read scenarios that are common for an application and the read scenarios of interest may vary between applications. Thus, we are incorporating into skel a flexible mechanism that allows various read tests to be specified. We discuss this here in some detail, but note that, at the time of this writing, we have not begun to gather data on read performance.

Read patterns in skel are being implemented as an extension to the parameters file. For the `type` attribute of the `test` element, we allow, in addition to `write`, two other options: `read` and `read_all`. Use of `read_all`, illustrated in Figure 3, indicates that the file is being read by the same number of processes and using the same pattern as was used when the file was written.

The `read` option allows more general read tasks to be specified. Figure 4 shows two examples of read pattern specification using data from the S3D simulation. The figure contains a batch with two read tests specified. The first test specifies that the skeletal application should read data for a partial plane for the single variable `OH`. In the second test we specify a subvolume to be read for the three related variables `uvel`, `vvel` and `wvel`. By allowing read operations to be specified using a combination of variables, offsets, and sizes, we offer a great deal of flexibility capable of supporting all of the most commonly used read patterns.

```
<batch name="read_methods" cores="96"
       target ="pbs_ib">

  <!-- Read a partial plane for one variable-->
  <test type="read" group="restart">
     <read_var name="OH" offset="320,400,960"
               size="480,360,1" />
  </test>

  <!-- Read a subvolume  for several related
       variables-->
  <test type="read" group="restart">
     <global-bounds offset="80,80,120"
                    size="320,240,240" >
       <read_var name="uvel"  />
       <read_var name="vvel"  />
       <read_var name="wvel" />
     </global-bounds>
  </test>
</batch>
```

Fig. 4. S3D read patterns

IV. Performance Comparison

Our experience in I/O applications comes from working directly with many codes. The first code that we placed ADIOS was the Gyrokinetic Toroidal Code (GTC) [12], a first principles fusion microturbulence code that studies turbulent transport of energy. Related to GTC is the GTS code [13], which uses a generalized geometry to solve the realistic geometries from many fusion reactors used today. Similarly, the XGC1 code studies fusion microturbulence around the edge of a plasma. All three of these codes use a Particle In Cell (PIC) method for solving the fundamental equations used in the simulation. The Pixie3D code is a fusion Magneto Hydrodynamic code that has also incorporated ADIOS in daily use to study larger scale instabilities in fusion reactors. The S3D combustion code is a Direct Numerical Solver (DNS) code that has very different properties than the PIC codes. It writes more frequently than PIC codes, but much less data per process. Likewise, the PMCL3D is an earthquake code that produces an extremely large amount of data per process making efficient I/O essential for checkpoint-restarts and analysis. We used the ADIOS descriptors from each of these applications to develop and test skel. In the following sections we provide performance comparisons with two of these applications, GTS and S3D.

Our experiments were conducted using Jaguar [14], a Cray XT5 machine with 18,688 compute nodes, each containing 2 hex-core AMD Opteron processors. Jaguar is located at the Oak Ridge Leadership Computing Facility (OLCF), and is connected to Spider, a large center-wide Lustre file system. As host to our target applications, Jaguar was a natural choice for the initial testing of skel.

A. GTS

We chose the GTS application for our first performance tests. The goal was to see how well our GTS skeletal application was able to mimic the IO performance of GTS. To accomplish this, we performed a weak scaling experiment using the GTS application and then used skel to create several GTS skeletal application tests corresponding to the GTS weak scaling runs. These tests were performed using core counts from 128 to 2048 coinciding with powers of two. Each core produced approximately 55 MB of data resulting in a total data size ranging from 6.9 GB for 128 cores up to 112 GB for the 2048 core run. The ADIOS *POSIX* method was used for all cases.

The results of these tests are shown in Figure 5. As can be seen, the measured performance of the skeletal application closely resembles that of GTS. As time was limited, these results reflect only a single run for each data point. As we continue to refine skel, we will perform much more detailed testing.

B. S3D

We employed a similar strategy to evaluate skel with S3D. We began with a strong scaling test using the S3D I/O Kernel. Then, we produced the corresponding skeletal application tests using skel. Here we varied the core counts between 128 and 1024, again in powers of two. The strong scaling tests each produced a fixed data size of approximately 1.7 GB distributed across all cores. Again, the ADIOS *POSIX* method was used.

Results of the S3D testing is shown in Figure 6. Once again, we observe that the relative performance of the S3D I/O kernel and the S3D skeletal application are similar showing roughly the same qualitative behavior, but with some variation in observed throughput, particularly in the 256- and 512-core cases. We find these preliminary results promising, and hope to develop a better understanding of any differences in performance as we continue to develop and test skel.

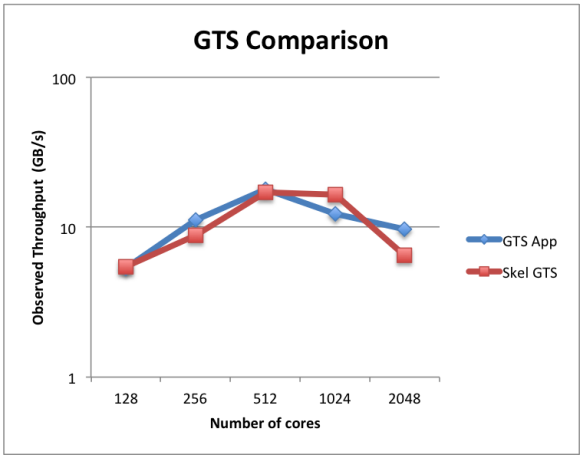

Fig. 5. GTS Weak Scaling Comparison

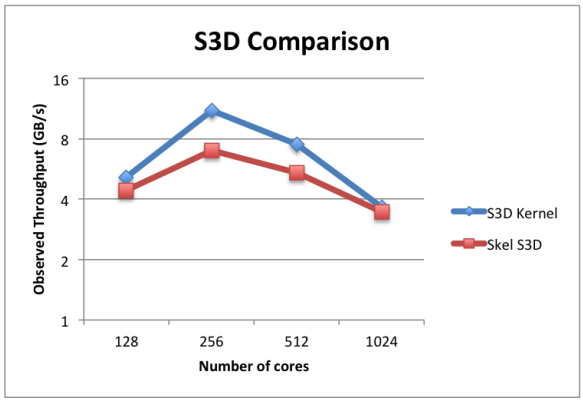

Fig. 6. S3D Strong Scaling Comparison

V. Related Work

IOR [15] is a versatile tool for measuring I/O performance. It is widely used to determine the raw performance capabilities of a system. IOR provides a great deal of configurability, however it contains no mechanism to mimic the I/O pattern of an application. It is left to the user to determine how a particular IOR configuration relates to the performance of a specific application [16]. In contrast, our aim is to minimize options available to the user while insuring that the tests consist of realistic access patterns that reflect the style of I/O that is performed by applications of interest.

There are a number of I/O kernel benchmarks that are derived from HPC applications, including FLASH-IO [5], MADBench2 [6], S3aSim [17] and S3D-IO to name a few. As with our skeletal applications, this style of benchmark aims to capture the I/O pattern of the application, often utilizing an I/O kernel taken directly from the application. There are several drawbacks to this approach that are directly addressed by skel. The first is the lack of a consistent user interface among these application derived benchmarks. Another issue is that the benchmarks are seldom kept up to date with changes to the application on which they are based. Finally, these derived benchmarks are manually coded, making it a nontrivial process to create a new benchmark for an application where one does not already exist.

The Darshan project [18] is examining the I/O patterns used by applications of interest. Darshan provides a lightweight library for gathering runtime information about the I/O being performed by an application, with one goal of the project being to collect this information from a large number of users. Data that can be collected includes a large number ($\sim$100) of statistics, many relating to low level POSIX or MPI-IO events.

The ScalaIOTrace tool [19] 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. Furthermore, traces may be executed outside of the application using a replay engine.

Both ScalaIOTrace and Darshan are useful for understand-

ing the I/O behavior of an application that is executed with the I/O method in question as the traces reflect not only the application's behaviors, but also the effects of the I/O middleware in use during execution. For testing new I/O middleware methods or keeping traces up to date with application changes, use of either of these tools would require applications to be executed again using the new methods to acquire a relevant trace. Our skel system does not require execution of an application, only a high level descriptor of the I/O being performed. The skeletal application could even be built *before* the application in question has been developed in order to test the behavior of existing I/O methods with the new application in advance.

VI. Conclusion and Future Work

We have presented skel, a new tool for generating skeletal I/O applications, and demonstrated its ability to generate skeletal applications based on a variety of scientific simulation codes. We have, for two of those codes, provided preliminary tests that show that our automatically produced skeletal applications mimic simulation performance reasonable well.

We have outlined several advantages to using skeletal applications to measure performance characteristics of I/O middleware. First, they are easily generated from a high-level I/O descriptor, such as the one used by ADIOS. They are easily updated when applications change, thus skeletal applications are much more likely to be kept up to date with changes in the applications on which they are based. Finally, skel provides a consistent interface across all generated skeletal applications making them easy to use and lessening the burden of porting benchmark codes to different platforms.

In the future we plan to work on refining the performance of the skeletal applications to more closely resemble the I/O performance of the original simulations. We plan to investigate the integration of other sources of application behavior data, possibly including I/O traces and application output files.

I/O benchmarks quite often can be used to accurately illustrate potential I/O performance on new systems for the versions of the applications they represent, but numerous problems exist for using them to represent the prouction. For example, many file systems use a form of asynchronous data movement, which can dramatically interfere with the application's communication patterns. Our work in I/O staging [20] revealed that asynchronous data movement can interfere with MPI collectives and other interprocess communication. We quite often see this performance degradation on systems that have the concept of "dirty" cache, such as Lustre, which does a majority of data output after the I/O requests are finished. Accurate I/O skeletal applications will eventually have to take this into effect if we are truly able to see I/O interference effects of asynchronous data movement techniques. I/O performance is also greatly affected by I/O interference [21] from other clients of the file system and great care most be taken to overcome these effects to get the best average I/O performance.

VII. 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.

References

[1] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and integration for scientific codes through the Adaptable IO System (ADIOS)," June 2008. [Online]. Available: http://www.adiosapi.org/uploads/clade110-lofstead.pdf

[2] M. A. Heroux, "Design issues for numerical libraries on scalable multicore architectures," *Journal of Physics: Conference Series*, vol. 125, no. 1, p. 012035, 2008. [Online]. Available: http://stacks.iop.org/1742-6596/125/i=1/a=012035

[3] M. Heroux, D. Doerfler, P. Crozier, J. Willenbring, C. Edwards, A. Williams, M. Rajan, E. Keiter, H. Thronquist, and R. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep., 2009.

[4] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.

[5] "FLASH I/O benchmark routine – parallel HDF5," http://www.ucolick.org/~zingale/flash_benchmark_io/.

[6] "MADbench2," http://crd.lbl.gov/~borrill/MADbench2/.

[7] 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 symposium on High performance distributed computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/1996130.1996139

[8] M. Polte, J. Lofstead, J. Bent, G. Gibson, S. A. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, M. Wingate, and M. Wolf, "...and eat it too: high read performance in write-optimized HPC I/O middleware file formats," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, ser. PDSW '09. New York, NY, USA: ACM, 2009, pp. 21–25. [Online]. Available: http://doi.acm.org/10.1145/1713072.1713079

[9] Y. Tian, S. Klasky, H. Abbasi, J. Lofstead, R. Grout, N. Podhorski, Q. Liu, W. Yandong, and Y. Weikuan, "EDO: improving read performance for scientific applications through elastic data organization," in *Proceedings of IEEE Cluster 2011*, to appear.

[10] "ADIOS 1.3 user's manual," http://users.nccs.gov/~pnorbert/ADIOS-UsersManual-1.3.pdf.

[11] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen, "Generative programming and active libraries," in *Selected Papers from the International Seminar on Generic Programming*. London, UK: Springer-Verlag, 2000, pp. 25–39. [Online]. Available: http://portal.acm.org/citation.cfm?id=647373.724187

[12] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney, "Grid -based parallel data streaming implemented for the gyrokinetic toroidal code," in *SC*, 2003, p. 24.

[13] W. X. Wang, Z. Lin, W. M. Tang, W. W. Lee, S. Ethier, J. L. V. Lewandowski, G. Rewoldt, T. S. Hahm, and J. Manickam, "Gyrokinetic simulation of global turbulent transport properties in tokamak experiments," *Physics of Plasmas*, vol. 13, no. 9, p. 092505, 2006. [Online]. Available: http://link.aip.org/link/PHPAEN/v13/i9/p092505/s1&Agg=doi

[14] "Jaguar," http://www.olcf.ornl.gov/computing-resources/jaguar/.

[15] "IOR HPC Benchmark," http://sourceforge.net/projects/ior-sio/.

[16] H. Shan and J. Shalf, "Using IOR to analyze the I/O performance for HPC platforms," in *Cray Users Group Meeting (CUG) 2007*, Seattle, Washington, May 2007.

[17] "Avery Ching, S3aSim," http://users.eecs.northwestern.edu/~aching/research_webpage/s3asim.html.

[18] "Darshan, petascale I/O characterization tool," http://www.mcs.anl.gov/research/projects/darshan/.

[19] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, "Scalable I/O tracing and analysis," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, ser. PDSW '09. New York, NY, USA: ACM, 2009, pp. 26–31. [Online]. Available: http://doi.acm.org/10.1145/1713072.1713080

[20] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "DataStager: scalable data staging services for petascale applications," in *HPDC*, 2009, pp. 39–48.

[21] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing variability in the IO performance of petascale storage systems," in *Supercomputing*. IEEE, 2010, pp. 1–12.