Pattern-driven Parallel I/O Tuning

Babak Behzad
University of Illinois at Urbana-Champaign

Surendra Byna
Lawrence Berkeley National Laboratory

Prabhat
Lawrence Berkeley National Laboratory

Marc Snir
Argonne National Laboratory, University of Illinois at Urbana-Champaign

ABSTRACT

The contemporary parallel I/O software stack is complex due to a large number of configurations for tuning I/O performance. Without a proper configuration, I/O becomes a performance bottleneck. As high performance computing (HPC) is moving towards exascale, poor I/O performance has a significant impact on the runtime of large-scale simulations producing massive amounts of data. In this paper, we focus on developing a framework for tuning parallel I/O configurations automatically. This auto-tuning framework first traces high-level I/O accesses and analyzes data write patterns. Based on these patterns and historically available tuning parameters for similar patterns, the framework selects best performing configurations at runtime. If previous history for a pattern is unavailable, the framework initiates model-based training to acquire efficient set of tuning parameters. Our framework includes a runtime system to apply the selected configurations using dynamic linking, without the need for changing application source code. In this paper, we describe this framework and evaluate it using multiple I/O kernels extracted from real applications and demonstrate substantial I/O performance improvement.

1. INTRODUCTION

HPC applications from various scientific domains produce and consume massive amounts of data. For example, plasma particle codes such as VPIC [5] simulating ten trillion particles can produce ≈300 TB data per time step [6]. Similarly, cosmology datasets also simulate trillions of particles producing data in the range of 10's of TB in size [21]. Since many scientific simulations need to write massive datasets to parallel storage and read them for post-processing analysis, efficient parallel write and read operations are critical to scientific discovery.

The contemporary parallel I/O software stack includes high-level I/O libraries, i.e., HDF5 and NetCDF, I/O middleware such as MPI-IO, parallel file system such as Lustre and GPFS. Each of these layers offer various configurable tuning parameters. When these configurations match "well" through all the layers, read or write operations perform efficiently. Manual characterization, tuning, and optimization of parallel I/O performance on multiple platforms have been proven to be effective [27, 13]. However, finding the right combinations of tunable parameters is complex on large-scale supercomputers because the search space is enormous. For example, on a Lustre file system using HDF5 chunking it can contain up to 336,000 possible configurations [4]. Finding these parameters automatically is even more challenging. While auto-tuning has been extensively studied in optimizing computational algorithms [24, 10, 14, 23, 26, 8, 25], applying the same techniques to parallel I/O tuning is nontrivial. One of the challenges is the sensitivity of parallel I/O performance because of interdependent parameters of various software layers. Additionally, in contrast to computational kernel tuning, where the compute nodes are not shared by other users, the parallel I/O system is shared by hundreds of applications.

In our prior work, we have shown the effectiveness of I/O tuning at multiple layers of tunable parameters using genetic algorithms [4]. We have improved the configuration search process significantly by developing an empirical performance prediction model for a selection of I/O kernels derived from real scientific simulations [1, 2]. Despite these efforts, the challenge of tuning an arbitrary I/O phase at runtime in a simulation remains an open issue. For instance, when a simulation needs to perform a large write operation, an I/O autotuning framework is required to identify the characteristics of the write operation, to find optimal tunable parameters, and to apply them at runtime without the need to stop the simulation for recompiling the simulation code with the optimal configurations.

In this paper, we address the requirements of an autotuning framework mentioned above. We first define *high-level I/O patterns* to characterize write operations. We use our tracing library to collect high-level I/O calls, such as HDF5 data model definition and write calls. This library uses binary instrumentation to redirect a set of HDF5 calls to collect the required information. We analyze these traces to obtain the I/O pattern information of a simulation's I/O phase. We then match the patterns with previously tuned I/O kernels for obtaining their optimal configurations. We provide a runtime library to apply the selected optimal configuration without the need for recompiling the code. If a matching previously tuned pattern was not available, we use our empirical prediction model to find tuning parameters at

PDSW 2015 November 15-20, 2015, Austin, TX, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-4008-3/15/11 ...$15.00.
DOI: http://dx.doi.org/10.1145/2834976.2834977.

offline and store them in the database for future use.

Overall, this paper has the following contributions:

- We provide a new representation for I/O patterns based on the traces of high-level I/O libraries, such as HDF5. This definition contains the global view of I/O accesses from all MPI processes in parallel applications.
- We develop a trace analysis tool for identifying I/O patterns of an application automatically.
- We show that using our runtime library, users can achieve significant portion of the peak I/O performance for arbitrary I/O patterns.

The remainder of the paper is structured as follows: In Section 2, we introduce our auto-tuning framework and present the functions of various components in the framework. We describe our experimental setup to test the framework and to evaluate performance improvement in Section 3. We present the related work in Section 4 and conclude the discussion in Section 5.

2. I/O AUTOTUNING FRAMEWORK

Figure 1 illustrates an overview of our proposed I/O autotuning framework. It consists of two phases: The first phase is the tuning phase, which performs extraction of the I/O pattern of an application. Once a pattern is extracted, there is a look-up phase in which the pattern is queried in a database of patterns and corresponding tuned configurations for the best I/O performance. If the pattern is found in this database, then the model associated with the pattern are stored in an XML file. In the adoption phase, the application is dynamically linked with our H5Tuner library for setting the selected tuning parameters in the XML file at runtime

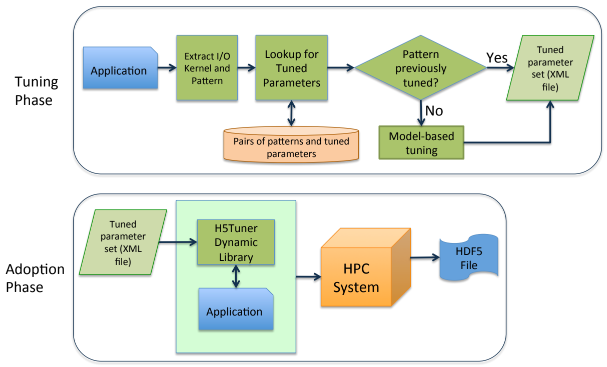

Tuning Phase
Application
Extract I/O Kernel and Pattern
Lookup for Tuned Parameters
Pattern previously tuned?
Yes
Tuned parameter set (XML file)
No
Pairs of patterns and tuned parameters
Model-based tuning
Adoption Phase
Tuned parameter set (XML file)
H5Tuner Dynamic Library
Application
HPC System
HDF5 File

Figure 1: An overview of our I/O autotuning framework

Our previous work [1, 4] describe the adoption phase in detail. This paper describes the tuning phase of the framework, on detecting I/O pattern and matching a detected pattern with the history of tuned parameter. In order to have a simpler description of these components, we use a sample parallel HDF5 application distributed along with the HDF5 source code, called `pH5Example`. The code creates two two-dimensional HDF5 datasets and writes them to a file.

2.1 I/O Traces

To be able to automatically extract the I/O activities of an application, we need to first extract the characteristics of I/O operations it is conducting. The I/O trace of an application is used towards this end. In our previous work, we have developed a multi-level I/O tracer tool, called Recorder [16]; It uses dynamic library pre-loading and intercepting I/O functions at different levels of the I/O stack. We observe that the best level of the I/O stack to define I/O patterns is at the higher-level I/O libraries such as HDF5. Therefore, we made use of the Recorder to capture all the HDF5 I/O operations of an application. At the end of one run of the application on P processes, P trace files are generated by the Recorder library. Figure 2 shows the trace file for process 0 of a four-process run of `pH5example` code. There are different function calls traced, causing to first create a HDF5 file (named `"ParaEg0.h5"`), then create two datasets (named `"Data1"` and `"Data2"`), then each process selects a hyperslab of these datasets, they write the data to them and close the file.

```
1396296304.23583 H5Pcreate (H5P_FILE_ACCESS) 167772177 0.00003
1396296304.23587 H5Pset_fapl_mpio (167772177,MPI_COMM_WORLD,
469762048) 0 0.00025
1396296304.23613 H5Fcreate (output/ParaEg0.h5,2,0,167772177) 16777216
0.00069
1396296304.23683 H5Pclose (167772177) 0 0.00002
1396296304.23685 H5Screate_simple (2,{24;24},NULL) 67108866 0.00002
1396296304.23688 H5Dcreate2 (16777216,Data1,H5T_STD_I32LE,
67108866,0,0,0) 83886080 0.00012
1396296304.23702 H5Dcreate2 (16777216,Data2,H5T_STD_I32LE,
67108866,0,0,0) 83886081 0.00003
1396296304.23707 H5Dget_space (83886080) 67108867 0.00001
1396296304.23708 H5Sselect_hyperslab (67108867,0,{0;0},{1;1},
{6;24},NULL) 0 0.00002
1396296304.23710 H5Screate_simple (2,{6;24},NULL) 67108868 0.00001
1396296304.23710 H5Dwrite (83886080,50331660,67108868,67108867,0) 0
0.00009
1396296304.23721 H5Dwrite (83886081,50331660,67108868,67108867,0) 0
0.00002
1396296304.23724 H5Sclose (67108867) 0 0.00000
1396296304.23724 H5Dclose (83886080) 0 0.00001
1396296304.23726 H5Dclose (83886081) 0 0.00001
1396296304.23727 H5Sclose (67108866) 0 0.00000
1396296304.23728 H5Fclose (16777216) 0 0.00043
```

Figure 2: A sample I/O trace generated by the Recorder for a simple parallel application called `pH5Example`

The following subsection discusses how we make use of the information in the trace files to come up with the I/O pattern of the application.

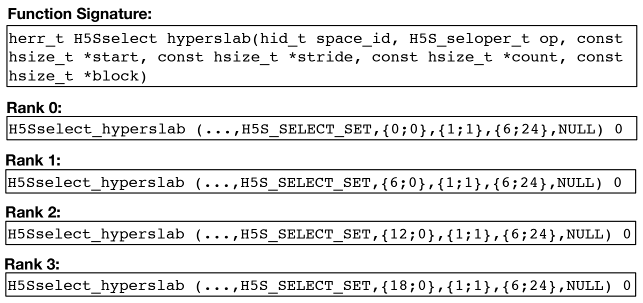
2.2 Extraction and Identification of High-level I/O Patterns

For performing automatic tuning of writing large datasets, we first need to identify the I/O pattern of the write operation. We define these patterns from observing the high-level I/O library calls, i.e., HDF5 calls.

As mentioned previously, high-level I/O libraries give us much more information in order to define and distinguish the way different applications conduct the I/O operations. One example and probably the main one is the concept of *selection* in HDF5. Selection is an important and a very powerful feature of HDF5 library that lets the developers select different parts of a file and different parts of memory in order to conduct I/O operations. It also is the main mechanism for the processes to choose different parts of the file in a parallel I/O application. Therefore, we base our definition of I/O patterns on the concept of selection. In summary, we will define the I/O pattern of an application as a coverage of the datasets based on the selections they make.

In HDF5 terminology, *hyperslab*s are portions of datasets, either a logically contiguous collection of points in a dataspace, or a regular pattern of points or blocks in a dataspace. In a parallel HDF5 program, once each process defines both the memory and file hyperslabs they execute a partial read/write [11]. In HDF5, the hyperslabs are selected using `H5Sselect_hyperslab` function. The four parameters that can be passed to this function are `start`, `stride`, `count`, and `block`: The `start` array is used by each process to specify the starting location for the hyperslab; The `stride` array specifies the distance between two consecutive selected elements or blocks. The `count` array for specifying the number of the elements/blocks to select; Finally, the `block` array specifies the size of the block selected from the dataspace.

In order to be concrete, we illustrate the definition of I/O patterns with an example application we have used in this paper. Figure 3 shows the four hyperslab selection of a parallel four-process run of `pH5Example`.



Function Signature:

```
herr_t H5Sselect hyperslab(hid_t space_id, H5S_seloper_t op, const
hsize_t *start, const hsize_t *stride, const hsize_t *count, const
hsize_t *block)
```

Rank 0:

```
H5Sselect_hyperslab (...,H5S_SELECT_SET,{0;0},{1;1},{6;24},NULL) 0
```

Rank 1:

```
H5Sselect_hyperslab (...,H5S_SELECT_SET,{6;0},{1;1},{6;24},NULL) 0
```

Rank 2:

```
H5Sselect_hyperslab (...,H5S_SELECT_SET,{12;0},{1;1},{6;24},NULL) 0
```

Rank 3:

```
H5Sselect_hyperslab (...,H5S_SELECT_SET,{18;0},{1;1},{6;24},NULL) 0
```

Figure 3: The four HDF5 hyperslab selection function calls across different ranks of a parallel four-process run of pH5Example

As it can be seen, all the processes are calling the same function with the same arguments except for `start`. The values of these `start` arrays are {0, 0}, {6, 0}, {12, 0}, and {18, 0}. The values of `count` arrays on all the ranks are {6, 24}. The call specifies that the 2D dataset is decomposed in the first dimension, with each process accessing a distinct horizontal slice.

In order to abstract these patterns, we make use of array distribution notation that was also used in High Performance Fortran (HPF)[19]. High Performance Fortran uses data distribution directives to help the programmer to distribute data between processes. Among these directives, `DISTRIBUTE` directive is used to specify the partitioning of the array data on to an abstract processor array. The basic distributions are `BLOCK`, `CYCLIC`, and `DEGENERATE`. A different distribution can be used for each dimension. Below is a short description of each of these distributions:

1. **Block Distribution:** In a block distribution, each process gets a single contiguous block of the array.

2. **Cyclic Distribution:** In a cyclic distribution, array elements are distributed in a round-robin manner. This means that the first element is on the first process, the second element on the second process and so on.

3. **Degenerate Distribution:** Degenerate distribution, represented by `*`, is basically no distribution or serial distribution. It means that all the elements of this dimension is assigned to one processor.

Using this terminology for the sample pH5Example application is straightforward. First of all, there is one HDF5 dataspace in the whole application created by the use of `H5Screate_simple()` function. It is a 2D dataspace of size 24×24. Then there are two datasets created on this dataspace named `Data1` and `Data2`. Then each of the ranks are selecting their own decomposition of the space and create two datasets of the size of the selected set as their memory dataset. Finally there are two `H5Dwrite()` function calls to write to `Data1` and `Data2`. Using HPF terminology we can abstract pH5Example as the following:

- **pH5Example:**
  `<2D, (BLOCK, *), (6, 24)>`
  `<2D, (BLOCK, *), (6, 24)>`

The advantage of this representation is that it is succinct enough in order to be stored in a key-value store as the I/O pattern repository. Currently, we are using text files to store the patterns without requiring a global database. However, as the number of patterns grow, in order to store the patterns associated with their I/O performance model, we can use a key-value store database. The schema of this database should include the dimensions of the patterns, their decompositions, their sizes, and the corresponding I/O performance model.

3. SETUP AND EVALUATION RESULTS

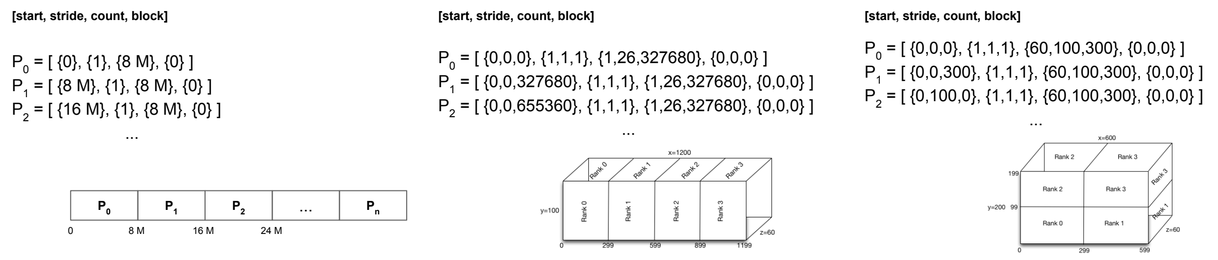
We have conducted all the experiments presented in this paper on two platforms, Edison and Hopper, located at the National Energy Research Scientific Computing Center (NERSC): Edison is a Cray XC30 system consisting $5,576$ twenty-four core Lustre file systems. We have used a Lustre partition of the file system in these experiments that has a maximum of 96 OSTs with 48 GB/s peak I/O bandwidth. Hopper is a Cray XE6 system, where we used a Lustre file system with 156 OSTs and a peak bandwidth of about 35GB/s for storing data.

In this paper we chose different I/O benchmarks and kernels. I/O kernels are simpler applications that issue the same I/O operations as a full-scale HPC applications. The four I/O kernels we have looked at are: Vector Particle-In-Cell (VPIC-IO), VORPAL-IO, and Global Cloud Resolving Model (GCRM-IO) and FLASH-IO. Below is a brief description of these I/O benchmarks.

- **IOR—I/O benchmark:** IOR [15] is an I/O benchmark developed at LLNL for the procurement of the ASCI Purple. Since it is highly-configurable and contains different I/O interfaces, it serves as one of the main HPC I/O benchmarks.

- **VPIC-IO—plasma physics:** Vector Particle-In-Cell (VPIC)[5] is a computer code simulating plasma behavior. VPIC-IO, replays only the I/O operations of VPIC application by creating a file, writing eight variables and closing the file.

- **VORPAL-IO—accelerator modeling:** VORPAL[17] is an acceleration modeling and computation plasma framework developed by Tech-X Corporation. VORPAL-IO, replays only the I/O operations of VORPAL.



```
[start, stride, count, block]

P0 = [ {0}, {1}, {8 M}, {0} ]
P1 = [ {8 M}, {1}, {8 M}, {0} ]
P2 = [ {16 M}, {1}, {8 M}, {0} ]
...
```

```
[start, stride, count, block]

P0 = [ {0,0,0}, {1,1,1}, {1,26,327680}, {0,0,0} ]
P1 = [ {0,0,327680}, {1,1,1}, {1,26,327680}, {0,0,0} ]
P2 = [ {0,0,655360}, {1,1,1}, {1,26,327680}, {0,0,0} ]
...
```

```
[start, stride, count, block]

P0 = [ {0,0,0}, {1,1,1}, {60,100,300}, {0,0,0} ]
P1 = [ {0,0,300}, {1,1,1}, {60,100,300}, {0,0,0} ]
P2 = [ {0,100,0}, {1,1,1}, {60,100,300}, {0,0,0} ]
...
```

(a) VPIC-IO (b) GCRM-IO (c) VORPAL-IO

Figure 4: I/O pattern of the (a) VPIC-IO (b) GCRM-IO (c) VORPAL-IO benchmark

- **GCRM-IO—global atmospheric model:** Global Cloud Circulation Model (GCRM)[18], is a faily new atmospheric model taking large convective clouds into global climate models.
- **FLASH-IO—high-energy density model:** FLASH I/O benchmark routine mimicks the I/O of the FLASH parallel HDF5 write operations. It has the data structures in FLASH application and writes a checkpoint file, a plotfile with centered data, and a plotfile with corner data.

Figures 4(a)-4(c) show the I/O accesses of the three applications we are considering in this work. These I/O accesses are the range of accesses based on the four parameters of the hyperslab selection. It can be observed that VPIC-IO is a 1-dimensional application and VORPAL-IO and GCRM-IO have 3-dimensional I/O accesses. We can also see how each processes are writing the same amount of data by having the same `count` arrays. The processes access different parts of the file in parallel by having different values for the `start` array.

Each process is writing a contiguous amount of data with 8 MB of size one after the other in the VPIC-IO benchmark. This is a very common and simple I/O pattern and we will see how it is abstracted. A more complex I/O access is GCRM-IO's. It is a 3-dimensional I/O benchmark decomposed only along one dimension as Figure 4(b) shows. Since only one dimension is decomposed in GCRM, we can see that the size of the whole dimension is used in the `count` array for the other two dimensions and the value of the `start` is 0.

The last I/O benchmark with the most complex I/O pattern is VORPAL-IO. It writes a 3-dimensional grid with a 3-dimensional decomposition along each of the dimensions. The size of the block that each process is writing is fixed and therefore the `count` array is the same for each of the processes. However, each of the processes have different values along the 3 dimensions of the `start` array.

Using the notation described in Section 2, we can represent our three applications as below:

- **VPIC-IO:**
  `<1D,BLOCK,8388608>`
  `<1D,BLOCK,8388608>`
  ... (5 more times) ...
  `<1D,BLOCK,8388608>`
- **GCRM-IO:**
  `<3D,(*,*,BLOCK), (1,1,327680)>`
  `<3D,(*,*,BLOCK), (1,1,327680)>`
  ... (7 more times) ...
  `<3D,(*,*,BLOCK), (1,1,327680)>`
- **VORPAL-IO:**
  `<3D,(BLOCK,BLOCK,BLOCK),(60,100,300)>`
  `<3D,(BLOCK,BLOCK,BLOCK),(60,100,300)>`
  ... (17 more times) ...
  `<3D,(BLOCK,BLOCK,BLOCK),(60,100,300)>`

We now show our results in four subsections. Note that for the results of this paper, we use all the developed models in our previous paper [1]. Therefore, there was no tuning for any application for this work and we have used the models developed for them in our previous work.

3.1 An application with the same I/O pattern

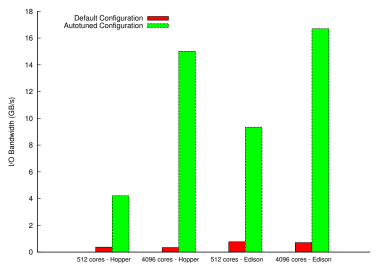
In order to have IOR issue write patterns similar to VPIC-IO, we configured it to use its HDF5 interface. Since VPIC-IO writes 8 datasets, we need to configure IOR accordingly. This is done by using 8 MB segments (`-s 8`), writeFile (`-w`), 32 MB blockSize (`-b 32m`) and transfer size of 32 MB (`-t 32m`).

Figure 5(a) shows the performance of the autotuned configuration which was proposed for IOR, as it has the same pattern as VPIC-IO, on 512 and 4096 cores of Hopper, and Edison in [1]. As mentioned before, there was no modeling effort done for this application and yet we can see that we are able to get up to 4.21 GB/s and 15.01 GB/s on 512 and 4096 cores of Hopper. On Edison these numbers are 9.34 GB/s, 16.70 GB/s.

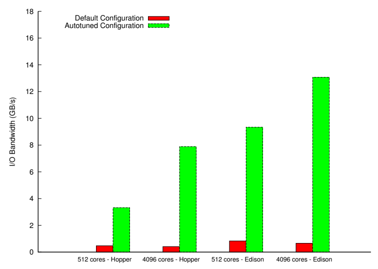
3.2 An application with similar I/O pattern

Resemble-VORPAL-IO is a synthetic benchmark generated by Record-and-Replay framework [3]. It has very similar I/O pattern to VORPAL-IO benchmark but with different block sizes of $64 \times 128 \times 256$ instead of $60 \times 100 \times 300$ of VORPAL-IO. The purpose of these experiments is two-fold: (a) To show that applications with similar I/O patterns with slight differences only in block sizes can use the same I/O configuration to obtain good I/O performance. (b) Requiring a threshold for the similarity between I/O patterns can save dramatic I/O tuning time.

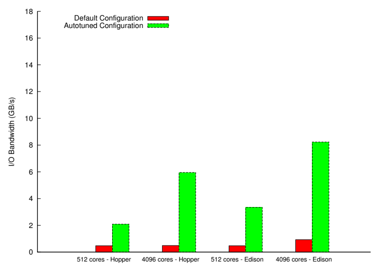
Figure 5(b) shows the performance of the autotuned configuration which was proposed for Resemble-VORPAL-IO on 512 and 4096 cores of Hopper and Edison in [1]. Similar to the previous experiment, there was no modeling effort done for this application and yet we can see that we are able to get up to 3.32 GB/s and 7.89 GB/s on 512 and 4096 cores of Hopper respectively. On Edison the highest bandwidth



(a) IOR



(b) Resemble-VORPAL-IO



(c) FLASH-IO

Figure 5: The I/O performance of the autotuned (a) IOR (b) Resemble-VORPAL-IO (c) FLASH-IO application on Hopper and Edison compared the default configuration.

achieved by this mechanism was 8.75 GB/s and 13.07 GB/s on the same number of cores.

3.3 A new application

The last experiment is designed to test an arbitrary application that has not been tuned before. For this experiment, we chose to test a well-known I/O kernel called FLASH-IO because it is popular in the HPC I/O community and also hard to tune. The same as previous experiment, we ran FLASH-IO at two scales, 512 and 4096 cores of Hopper and Edison. The way that we calculate the bandwidth for this application is a little bit different than the other ones as it produces three files. The definition of bandwidth here is basically just the sum of all the output sizes divided by the runtime of the whole I/O benchmark which is a conservative way of defining the I/O bandwidth of an application.

FLASH-IO is different from the other applications we have looked at mainly because it writes many datasets with different I/O patterns. In order to overcome this problem the framework considers the largest datasets in size and looks up for those patterns in the database. Based on the output of H5Analyze tool, FLASH-IO has 34 datasets, out of which 24 of them have the same size as the largest size of the file. On 4096 cores, this is about 40GB for each dataset. These datasets are 4D and their pattern of these dataset are also the same: `<BLOCK, DEGENERATE, DEGENERATE, DEGENERATE>`. Although the exact same pattern does not exist for this pattern, GCRM-IO has the most similar pattern to this application and therefore the framework uses the proposed configurations for GCRM-IO.

Figure 5(c) shows the performance of the autotuned configurations which was proposed for FLASH-IO based on GCRM-IO model, on 512 and 4096 cores of Hopper, and Edison by our framework. Similar to the previous experiment, there was no modeling effort done for this application and yet we can see that we are able to get up to 2.09 GB/s and 5.95 GB/s on 512 and 4096 cores of Hopper respectively. On Edison the highest bandwidth achieved by this mechanism was 3.34 GB/s and 8.23 GB/s on the same number of cores.

4. RELATED WORK

I/O patterns have been an important concept in the I/O community and several research projects have been exploiting them in different contexts. Out of these we can mention I/O Signature is a notation proposed by Byna et al [7] consisting of five dimensions of I/O operations: operation, spatial offset, request size, repetitive behavior, and temporal intervals. These are then gathered by a framework for each application and stored persistentlly for later look up in order to help prefetching. Additionally, statistical models (such as Markov models) have been proposed for a long time for being able to produce and predict I/O operations and file system performance. [22, 20]. These are then more used in the context of prefetching, caching or scheduling, as compared to our work which is tuning I/O operations in order to increase I/O bandwidth that applications gain.

In recent years, due to complexities of gaining I/O performance in modern HPC applications, I/O patterns have started to gain more attention. In particular, He et al. [12] tries to "rediscover these structures in unstructured I/O" using "gray-box" technique. In terms of framework design there are some similarities such as the way the pattern detection engine works. Additionally, Omnisc'IO [9] uses an algorithm based on Sequitur algorithm which given a sequence of symbols, builds a grammar for text compression. Most of this work uses the idea of I/O patterns with the main difference that they are based on low-level I/O layers, i.e. POSIX layer as opposed to high-level I/O layers. Our approach is more portable, accurate, and simpler than the POSIX version given the parallel nature of the applications.

5. CONCLUSIONS

Poorly tuned Parallel I/O becomes a major performance bottleneck in HPC applications that need to write or read data. This is not due to incapability of I/O subsystems, but mainly due to the complexity of its tuning. In this paper, we propose a pattern-driven autotuning framework to solve this problem. The framework consists of components to extract I/O patterns, tune configuration for the detected patterns, store them in a database of patterns associated with their I/O model, and finally map an arbitrary I/O pattern to a previously tuned model in order to improve its I/O performance. We show that using these patterns, one can tune different sets of applications ranging from the ones which have tuned before the ones which are similar to the ones before, and totally new ones.

6. ACKNOWLEDGMENTS

This work is supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center.

7. REFERENCES

[1] B. Behzad, S. Byna, S. M. Wild, M. Prabhat, and M. Snir. Improving Parallel I/O Autotuning with Performance Modeling. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, 2014.

[2] B. Behzad, S. Byna, S. M. Wild, M. Prabhat, and M. Snir. Dynamic Model-driven Parallel I/O Performance Tuning. In *IEEE Cluster 2015*, 2015.

[3] B. Behzad, H.-V. Dang, F. Hariri, W. Zhang, and M. Snir. Automatic Generation of I/O Kernels for HPC Applications. In *Proceedings of the 9th Parallel Data Storage Workshop*, PDSW '14, pages 31–36, Piscataway, NJ, USA, 2014. IEEE Press.

[4] B. Behzad, L. Huong Vu Thanh, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir. Taming Parallel I/O Complexity with Auto-Tuning. In *Proceedings of 2013 International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2013)*, SC '13, 2013.

[5] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas*, 15(5):7, 2008.

[6] S. Breitenfeld, K. Chadalavada, R. Sisneros, S. Byna, Q. Koziol, N. Fortner, Prabhat, and V. Vishwanath. Recent Progress in Tuning Performance of Large-scale I/O with Parallel HDF5. In *Proceedings of the 9th Parallel Data Storage Workshop*, PDSW '14, 2014.

[7] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp. Parallel I/O Prefetching Using MPI File Caching and I/O Signatures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 44:1–44:12, Piscataway, NJ, USA, 2008. IEEE Press.

[8] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, 2008.

[9] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross. Omnisc'IO: A Grammar-based Approach to Spatial and Temporal I/O Patterns Prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 623–634, Piscataway, NJ, USA, 2014. IEEE Press.

[10] Frigo, Matteo, Johnson, and S. G. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[11] T. H. Group. HDF5 Tutorial - Parallel Topics `http://www.hdfgroup.org/HDF5/Tutor/parallel.html`, Feb. 2011.

[12] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun. I/O Acceleration with Pattern Detection. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 25–36, New York, NY, USA, 2013. ACM.

[13] M. Howison, Q. Koziol, D. Knaak, J. Mainzer, and J. Shalf. Tuning HDF5 for Lustre File Systems. In *Proceedings of 2010 Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS10)*, Heraklion, Crete, Greece, Sept. 2010. LBNL-4803E.

[14] B. Jeff, A. Krste, C. Chee-Whye, and D. Jim. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, ICS '97, pages 340–347, 1997.

[15] LLNL. IOR `https://github.com/chaos/ior`, Feb. 2015.

[16] H. Luu, B. Behzad, R. Aydt, and M. Winslett. A multi-level approach for understanding I/O activity in HPC applications. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–5, 2013.

[17] C. Nieter and J. R. Cary. VORPAL: a versatile plasma simulation code. *Journal of Computational Physics*, 196:448–472, 2004.

[18] D. A. Randal and A. Arakawa. Design and Testing of a Global Cloud-Resolving Model. Report, 2009.

[19] H. Richardson. High Performance Fortran: history, overview and current developments. Technical report, 1.4 TMC-261, Thinking Machines Corporation, 1996.

[20] H. Simitci and D. A. Reed. A Comparison of Logical and Physical Parallel I/O Patterns. *International Journal of High Performance Computing Applications*, 12:364–380, 1998.

[21] S. W. Skillman, M. S. Warren, M. J. Turk, R. H. Wechsler, D. E. Holz, and P. M. Sutter. Dark Sky Simulations: Early Data Release. *ArXiv e-prints*, July 2014.

[22] E. Smirni and D. A. Reed. Lessons from Characterizing Input/Output Bahavior of Parallel Scientific Applications. *International Journal on Performance Evaluation*, 33:27–44, 1998.

[23] R. Vuduc, J. Demmel, and K. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005, Journal of Physics: Conference Series*, 2005.

[24] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[25] S. Williams, K. Datta, J. Carter, L. Oliker, J. Shalf, K. A. Yelick, and D. Bailey. PERI: Autotuning memory intensive kernels for multicore. In *Journal of Physics, SciDAC PI Conference: Conference Series: 123012001*, 2008.

[26] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 38:1–38:12, 2007.

[27] W. Yu, J. Vetter, and H. Oral. Performance characterization and optimization of parallel i/o on the cray xt. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –11, april 2008.