Pattern-driven Parallel I/O Tuning

Babak Behzad
University of Illinois at Urbana-Champaign

Prabhat
Lawrence Berkeley National Laboratory

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

ABSTRACT

The modern parallel I/O software stack is complex due to the number of configurations for tuning I/O performance. Without 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 autotuning 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. If previous history is unavailable, the framework initiates model-based training to acquire efficient configurations. Our framework includes a runtime system to apply the selected configurations using dynamic linking, without the need for changing code. We test this framework using several I/O kernels extracted from real applications and demonstrate substantial I/O performance benefits.

General Terms

Parallel I/O, I/O Patterns, Autotuning, Performance Optimization, Parallel file systems

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 [22]. Since many of 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 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 [29, 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 [3]. Finding these parameters automatically is even more challenging. While autotuning has been extensively studied in optimizing computational algorithms [26, 10, 14, 25, 28, 8, 27], 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 [3]. 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]. Despite these efforts, the challenge of tuning an arbitrary I/O phase 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 define a simple key-value pair-based database to store the previously tuned I/O patterns and 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 offline and store them in the database for future use.

Overall, this paper has the following contributions:

- We provide a new definition of I/O pattern 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 the I/O patterns of an application automatically.
- We demonstrate mapping an arbitrary write pattern with existing write pattern to determine optimal configurations.
- 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 autotuning framework and present the functions of various components in the framework. We describe the experimental setup to test our framework in Section 3 and evaluate the results in Section 4. Section 5 goes over related work. Finally, we conclude the discussion in Section 6 along with our future work.

2. I/O AUTOTUNING FRAMEWORK

Figure 1 illustrates an overview of our proposed I/O autotuning framework that can address parallel I/O tuning problem. It consists of two phases: The first phase is the tuning phase which performs extraction of the I/O pattern from the application. Once the pattern is extracted, there is a look-up phase in which the pattern is queried in a database of patterns. If the pattern is found in this database, then the model associated with it is used to are suggest tuned parameters for it as XML files to be run with the application as part of the second phase, the adoption phase. This is the phase in which the application is dynamically linked with the H5Tuner library in order to set the selected tuning parameters on the fly while the application is running.

Since our previous work has shown the adoption phase in detail [1, 3], in the following subsections we describe the components of the tuning phase of the framework. 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 [17]; 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

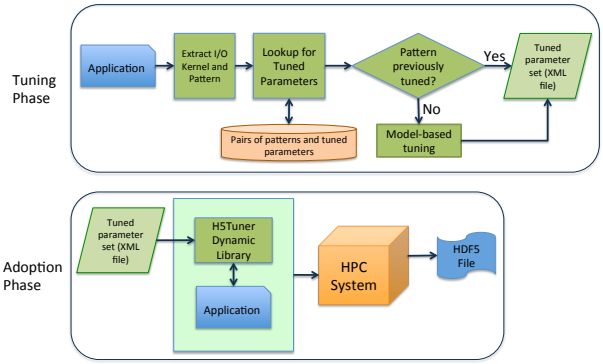

Figure 1: An overview of our I/O autotuning framework

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 next 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 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.

There are many ways of defining an I/O pattern of an

application. Following the approach of the database community, we separate the I/O pattern of an application into two categories:

- **Physical Pattern:** The physical pattern of read/write operations is related to the hardware configuration and is specific to the file system, platform, etc. These are discussed in our previous work [1, 3] and statistical models have been proposed for it. They are the models that have either linear or inverse relationship with file-system parameters such as Lustre stripe settings and MPI-IO settings. We showed that different I/O benchmarks have different relationship with these parameters and it is possible to generalize the models to take the number of processes and file sizes into account as well.

- **Logical Pattern:** Logical pattern is defined at the application level and is the focus of this paper. This is the pattern that takes the number of processes that run the application into account along with the distribution of the data between them. Higher-level I/O libraries divide the I/O operations into two categories below. We believe that in order to have a more accurate definition of the logical I/O patterns, we can utilize the same division:

  1. **Metadata:** Metadata of a high-level library includes information about the data itself such as datatypes, dimensions, etc. This also includes information about the data that user may want to save such as attributes. The size of metadata is pretty small and it is typically stored in the first part of the file.
  2. **Raw Data:** Raw data is the main data which is the bigger portion of the file and the main I/O time is spent in doing I/O operations for it. The main difference between the I/O operations of different applications exists in the access to raw data. Applications can do the I/O operations contiguously or non-contiguously. They can access the raw data in horizontal stripes or vertical stripes. They can even have random selections of this raw data. The main focus of this paper is to abstract these kinds of patterns.

As mentioned above, 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/writei [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 are going to illustrate this definition of I/O patterns with the example application we have 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. Figure 4 visualizes this decomposition.

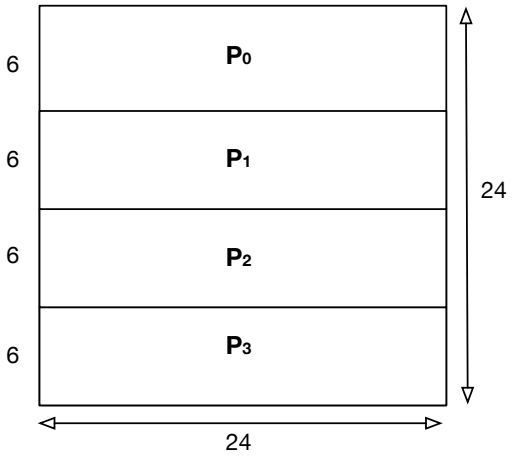

Figure 4: Decomposition of one of the dimensions in pH5Example by the four processes

2.3 I/O Patterns Repository

In order to abstract these patterns, we make use of array distribution notation that was also used in High Performance Fortran (HPF)[20]. 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, called the I/O pattern repository.

2.4 Mapping an Arbitrary Application to an I/O Pattern

As we can extract and provide a representation of an I/O pattern of HPC applications, the next question is, given an arbitrary HPC application, how can we map it to one of the existing patterns we have in our database? In this subsection we try to answer this question. The first thing one should do is to look up for the dimension of the arbitrary application and check if we have an application with the same dimension value as the arbitrary one. Then, it has to look for the distribution of the given application. The order to look for this distribution is challenging, and we propose the following solution: First check if each of the dimensions has a DEGENRATE distribution. If so, check for a BLOCK distribution for all the other dimensions. If none of the above, check for multi-dimensional BLOCK. Multi-dimensional BLOCK is the general version of BLOCK distribution which means that once the value of a dimension reaches to its maximum value, the other dimension is increased. Section 3 shows a full example of this distribution. If not, check for CYCLIC or BLOCK-CYCLIC distribution and finally if none of these apply, this is a random I/O access pattern.

The first two steps are simple to check, we just need an analysis operation which goes over every dimension of the `start` array and compares it to the previous value corresponding to the value of the previous rank. If all these numbers are the same, then it can conclude that it is a DEGENRATE distribution, if the difference is between every two consecutive element is the same, it is a BLOCK distribution. However, if these cases do not happen, step 3 should be performed and that is a multi-dimensional BLOCK. Algorithm 2.4 shows our procedure for this analysis operation which is a generalized version of step 2. This is a general version of a simple algorithm to check if numbers have a BLOCK distribution. The only difference between the general version and the simple version is that it needs a general version of `subtract` and `equal` operation. As input, the algorithm gets all the traces of `H5Sselect_hyperslab` function call. These traces have all the `start` and `count` arrays for each process. The algorithm starts by getting the first and the second element of the `start` array and uses the `gen_subtract` function to calculate the difference between these `start` arrays. It stores this in a variable named `diff` and continue with the next elements of the array in order to see if this difference stays the same in the whole array. Figure 5 shows applying this algorithm on VORPAL-IO application, which we will explain in detail in Section 3. As it can be seen in the figure, the `start` array of consecutive processes are subtracted from each other and since the `diff` value is always the same, we can conclude that VORPAL-IO has a multi-dimensional BLOCK distribution.

If none of these patterns are found in the selection values, the analyze operation will check CYCLIC and BLOCK-CYCLIC distributions too. Note that if none of these patterns can be found, we mark that application with a random I/O access pattern and will not store it in the database.

Algorithm 1 Check for Multi-Block Distribution

Input: `H5Sselect_hyperslab` traces of all the processes
Output: Whether they have a block distribution and how much is the block size
Variables:
p^i= process i
`start[ ]`: Start array in `H5Sselect_hyperslab` function
`count[ ]`: Count array in `H5Sselect_hyperslab` function
Pseudo Code:

```
prev_start ← *start[0].
cur_start ← *start[1].
diff ← gen_subtract(cur_start, prev_start).
prev_start ← cur_start.
while There are more array elements do
  update cur_start
  new_diff ← gen_subtract(cur_start, prev_start)
  if gen_equal(diff, new_diff) == false then
    is_multi_block ← false
  end if
  diff ← new_diff
  prev_start ← cur_start.
end while
return diff
```

Once the distribution of each of the datasets are found, the patterns should be queried in the database as will be shown in the next subsection. Note that the sizes in the database should not exactly match, but has to be within a

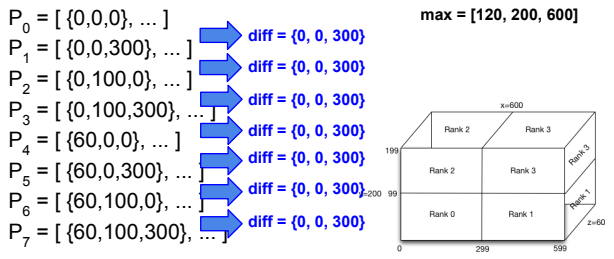

Figure 5: An Example of Generalized Operations (diff and equal) for VORPAL-IO application

threshold which we will describe later in Section 4.

2.5 Intelligent Runtime System

The runtime system we have developed consists of two main components:

1. **H5Analyze:** H5Analyze is a code we have developed based on pattern analysis provided by Tang et al. [24] for analyzing HDF5 read and write traces. Our implementation contains structures for storing information about HDF5 files, dataspaces, datasets, selections, and operations. It accepts the traces gathered by the Recorder [17] from all the processes as input and populates these information by reading all these traces. Once this information is stored, the H5Analyze code starts to execute the above-mentioned analysis on them in order to come up with the patterns and output them in HPF terminology. Figure 6 shows the output of H5Analyze on the pH5example. As it can be seen, once the correct arguments are given to H5Analyze code, it is able to find out the dimension, distribution and the size of the access pattern of each dataset.

```
$ ./H5Analyze WRITE 1 testlog/pH5example_4 4
.
.
.
I/O Pattern with HPF Terminology:
Dataset name: output/ParaEg0.h5/Data1
  - Dimension: 2
  - Distribution: <BLOCK, DEGENERATE>
  - Size: <6, 24>
Dataset name: output/ParaEg0.h5/Data2
  - Dimension: 2
  - Distribution: <BLOCK, DEGENERATE>
  - Size: <6, 24>
```

Figure 6: Output related to the I/O pattern of H5Analyze code for pH5example code

2. **Key-Value Store:** In order to store the patterns associated with their I/O performance model, ultimately we should use a database as the number of patterns increase. For now however, we are using text files as it is easier to store the patterns in text files without requiring a global database.

3. **H5Tuner Library:** If the framework is able to find a pattern close enough to an input application, then it will adopt the same model for that pattern and as completely explained in [1] it proposes the top k configuration as the k-best performing configurations for that pattern in XML files. H5Tuner library [3] is then dynamically linked to the application to set these parameters at runtime of the application. The user can control the value of k based on the amount of time they want to put on tuning the I/O phase of the application. In our experiment, $k = 20$. If the user is happy with the I/O bandwidth these top k configuration lead to, they can use that specific XML file along with H5Tuner library for that application.

4. **Modeling Component:** If the pattern of the input application is not found in the key-value store, since there is no model associated with it, the framework needs to come up with a model for it. This is the focus of our previous paper [1] and includes a training phase in which the model is trained for a set of different values for each of the parameters at different core counts. Since this component has been studied in detail in our previous work we will not go over it here. The only thing to note is that we have chosen to have a separate component for this in our framework as it may be improved over time.

3. EXPERIMENTAL SETUP

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):

1. **Edison:** Edison is a Cray XC30 system consisting 5,576 twenty-four core nodes with 64GB of memory per node. It has Cray Aries with Dragonfly topology and three Lustre file systems with aggregate bandwidth of 168 GB/s. 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. Cray's MPI library v7.0.4, HDF5 v1.8.11, and H5Part v1.6.6 were used on Edison.

2. **Hopper:** Hopper is a Cray XE6 system containing 6,384 twenty-four core nodes with 32GB of memory per node. It employs the Gemini interconnect with a 3D torus topology. We used a Lustre file system with 156 OSTs and a peak bandwidth of about 35GB/s for storing data. We used Cray's MPI library v6.0.1, HDF5 v1.8.11, and H5Part v1.6.6 for compiling the I/O kernels.

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 [16] 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. It makes use of H5Part library [4] and the variables written are for the particles and contain random data of float data type.
- **VORPAL-IO—accelerator modeling:** VORPAL[18] is an acceleration modeling and computation plasma framework developed by Tech-X Corporation. VORPAL-IO, replays only the I/O operations of VORPAL and uses H5Block to write 3D blocks of data per processor.
- **GCRM-IO—global atmospheric model:** Global Cloud Circulation Model (GCRM)[19], is a faily new atmospheric model taking large convective clouds into global climate models. GCRM-IO also uses H5Part to perform I/O operations similar to GCRM with random data.
- **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. At 512 cores, FLASH-IO creates a 122 GB checkpoint file, 11 GB centered data plotfile and 12 GB corner plotfile. At 4096 cores, it creates a 973 GB checkpoint file, 82 GB centered data plotfile, and a 92 GB corner plotfile.

Figures 7-9 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.

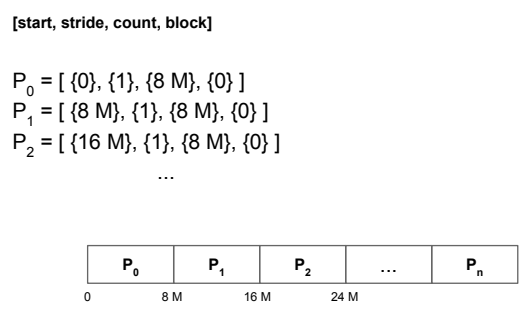

Figure 7: I/O pattern of the VPIC-IO benchmark

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 8 shows. Since

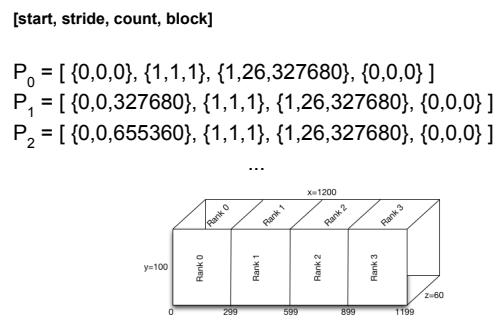

Figure 8: I/O pattern of the GCRM-IO benchmark

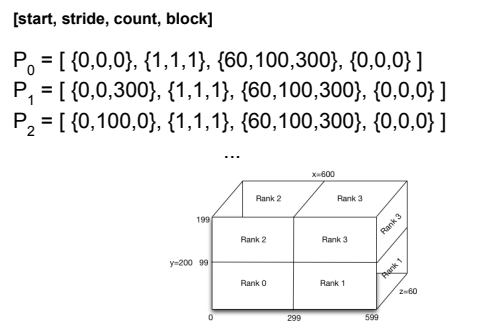

Figure 9: I/O pattern of the VORPAL-IO benchmark

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)>`

4. RESULTS

This section shows the results of our framework in four subsections: The first experiment to show that our framework is capable of identifying an I/O pattern exactly similar to what it has tuned before and configure the I/O correctly. The second one is to show that a new pattern but similar to the ones in the database is recognized and the model used for the most similar application to it in the database can lead to acceptable I/O performance. We then tune an arbitrary application that does not have any similar patterns in the database is tuned. Finally, we evaluate the overheads of our autotuning framework.

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.

For the first series of experiments, we use IOR benchmark. The second experiment uses a synthetic benchmark called Resemble-VORPAL-IO which is similar to VORPAL-IO pattern but with different block sizes. The last experiment is a whole new I/O benchmark: FLASH-IO.

4.1 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 segments (`-s 8` command line option) of IOR. VPIC-IO only does write operations and we use writeFile (`-w` option) for IOR. Since for each dataset of VPIC-IO contains 32 MB of data per processor, we use the blockSize (`-b 32m` option) of IOR along with the transfer size of 32 MB (`-t 32m` command line option).

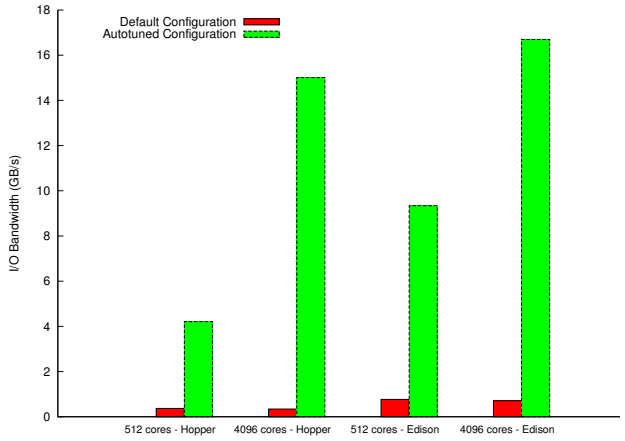

Figure 10: The I/O performance of the autotuned IOR application on Hopper and Edison compared the default configuration

Figure 10 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.

4.2 An application with similar I/O pattern

Resemble-VORPAL-IO is a synthetic benchmark generated by Record-and-Replay framework [2]. 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.

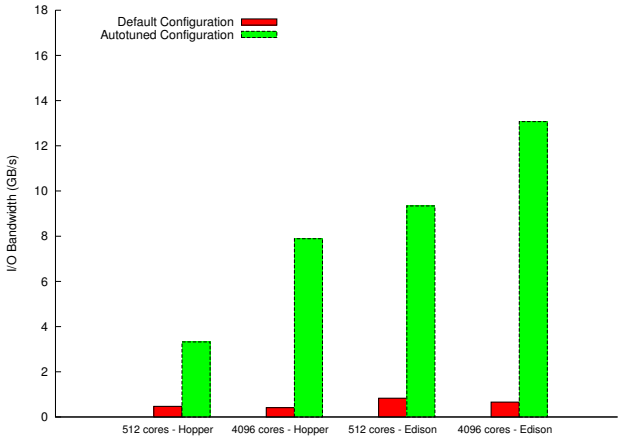

Figure 11: The I/O performance of the autotuned Resemble-VORPAL-IO application on Hopper and Edison compared the default configuration

Figure 11 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 achieved by this mechanism was 8.75 GB/s and 13.07 GB/s on the same number of cores.

4.3 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 has 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 has 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.

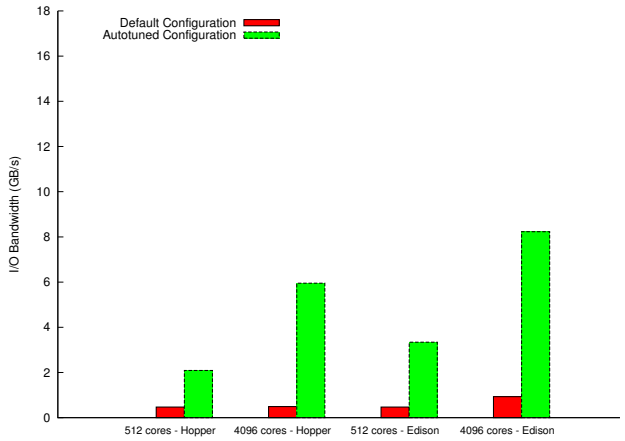

Figure 12: The I/O performance of the autotuned FLASH-IO application on Hopper and Edison compared the default configuration

Figure 12 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.

4.4 The overhead of the framework

In this subsection, we measure the overhead of the framework based on the order they appear in the architecture discussed in Section 2. Regarding the extraction of I/O pattern from an application, the framework needs at least one run of the application linked to the recorder in order to gather the traces. The overhead of the Recorder library is minimal [17]. Once the traces are gathered, H5Analyze analyzes the traces to come up with the I/O pattern. H5Analyze is a fast sequential application written in C programming language which reads in the traces and comes up with the pattern with a very fast turn-around time even at large scale trace files. The looking up phase is also fast as the number of patterns are small and comparing them to find the closest. In case the pattern of an application is found, the I/O configuration of the application is proposed with an XML file. If not, the main part of the overhead of our framework is exerted: The modeling phase. For those patterns that the framework is not able to find a match, it requires the autotuning framework to initialize the modeling process by running the application with its training set. This may require more than several hours to come up with a non-linear regression models. Once the model is developed, the framework will associate the pattern along with the new model for any future run of the application.

5. 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. [23, 21]. 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] mention that a lot of information gets lost in a typical I/O stack as the data flows between its layers. Although high-level I/O libraries contain rich information about the data structures, eventually they get down into simple offset and length pairs in the storage system. Their solution to this problem is 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] is a grammar-based I/O model in the hope of capturing and predicting I/O operations of an application. At its heart, it uses an algorithm based on Sequitur algorithm which given a sequence of symbols, builds a grammar for text compression. It supports both spatial and temporal patterns in this regard. In order to be more general, the authors use the program's stack trace as the symbols of the grammar. One strength of their approach is that it does real-time prediction as the grammar is being updated in the algorithm. 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. As [12] correctly argues about information getting lost in the flow of data in the I/O stack, but uses POSIX offset pattern matching to rediscover the information. Our approach is different in the sense that we have defined the pattern at the high-level libraries and do not lose this information. Our approach is more portable, accurate, and simpler than the POSIX version given the parallel nature of the applications.

6. CONCLUSIONS AND FUTURE WORK

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.

The main line of future work of this paper is to get it ready for production use. For this purpose, the main missing component is an implementation of a global database which one can insert and query the patterns. Additionally, as more and more applications use this framework, there is more need for a database. Another future work is to apply the same concepts to other high-level I/O libraries such as PnetCDF [15]. This is a simple addition as these libraries have the same concepts and they just differ in the function calls. Last but not least, we are thinking about an ambitious goal of pushing our framework futher to conduct real-time I/O autotuning. This means that every HPC platform will have an intelligent I/O runtime system which is able to identify the I/O pattern of different applications while they are being run and set the I/O parameters for them accordingly in order to have very good I/O performance.

7. 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.

8. 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, 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.

[3] 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.

[4] E. W. Bethel, J. M. Shalf, C. Siegerist, K. Stockinger, A. Adelmann, A. Gsell, B. Oswald, and T. Schietinger. Progress on H5Part: A portable high performance parallel data interface for electromagnetics simulations. PAC 07, 2007.

[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] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 39–, New York, NY, USA, 2003. ACM.

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

[17] 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.

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

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

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

[21] 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.

[22] 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.

[23] 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.

[24] H. Tang, X. Zou, J. Jenkins, D. A. B. II, S. Ranshous, D. Kimpe, S. Klasky, and N. F. Samatova. Improving Read Performance with Online Access Pattern Analysis and Prefetching. In *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, pages 246–257, 2014.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.