2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Boosting Application-specific Parallel I/O Optimization using IOSIG

Yanlong Yin[1] yyin2@iit.edu
Surendra Byna[2] sbyna@lbl.gov
Huaiming Song[1] hsong20@iit.edu
Xian-He Sun[1] sun@iit.edu
Rajeev Thakur[3] thakur@mcs.anl.gov

[1]Department of Computer Science, Illinois Institute of Technology, Chicago, Illinois
[2]Computational Research Division, Lawrence Berkeley National Lab, Berkeley, California
[3]Mathematics and Computer Science Division, Argonne National Lab, Argonne, Illinois

***Abstract*—Many scientific applications spend a significant portion of their execution time in accessing data from files. Various optimization techniques exist to improve data access performance, such as data prefetching and data layout optimization. However, optimization process is usually a difficult task due to the complexity involved in understanding I/O behavior. Tools that can help simplify the optimization process have a significant importance. In this paper, we introduce a tool, called IOSIG, for providing a better understanding of parallel I/O accesses and information to be used for optimization techniques. The tool enables tracing parallel I/O calls of an application and analyzing the collected information to provide a clear understanding of I/O behavior of the application. We show that performance overheads of the tool in trace collection and analysis are negligible. The analysis step creates I/O signatures that various optimizations can use for improving I/O performance. I/O signatures are compact, easy-to-understand, and parameterized representations containing data access pattern information such as size, strides between consecutive accesses, repetition, timing, etc. The signatures include local I/O behavior for each process and global behavior for an overall application. We illustrate the usage of the IOSIG tool in data prefetching and data layout optimizations.**

Keywords-Parallel I/O, I/O characterization, data access pattern, I/O optimization

I. INTRODUCTION

As high performance computing (HPC) is moving towards exa-scale, efficient usage of resources in the large-scale machines is a critical requirement. Efficient usage typically translates to faster scientific discovery and to lower energy consumption. Improving data access performance plays a significant role in making parallel computers efficient. Since many scientific applications deal with large amounts of data, making parallel file I/O efficient has an enormous impact on making parallel applications execute faster. Typically, execution time of a parallel program includes the time spent on computation, communication among processes, and data I/O. In many data intensive applications, I/O performance is usually a significant bottleneck leading to wastage of CPU cycles and the corresponding wasted energy consumption. In HPC systems, the gap between computing capacity and I/O performance keeps increasing because of highly diverse growth rates of storage devices and processors. As the number of processing cores in large-scale clusters increase, the insatiable desire for accessing more data continues to grow. Hence, improving data access performance is the key for improving efficiency of HPC applications at exa-scale.

The first step towards efficient data accesses is to understand their behavior. A few tools exist for profiling communication and computation overheads in parallel applications [1] [2] [3] [4]. However, there is a serious lack of tools for analyzing parallel I/O performance in a comprehensive manner and for converting the analyzed data into information that optimization techniques can use. The existing I/O analysis tools [5] [6] [7] [8] [9] have limited scope of I/O characterization. Few of these tools [5] [8] collect a lot of trace information about I/O calls and leave it for programmers to understand. These tools do not provide the much needed analysis step to gain a clear insight into I/O characteristics. Without the analysis step, although some I/O traces are available, they just sit idle in some server and are not useful for improving the efficiency. A few other tools [6] [7][9] provide partial understanding of I/O behavior but also require programmer involvement in performing optimizations. The latter category of tools aims towards reducing overhead and resource requirement in collecting information about I/O calls by retrieving few details and infrequently. While they achieve the low resource usage goal, they can only provide little insight into I/O behavior.

We aim to develop an I/O characterization tool, which gives comprehensive understanding of the I/O behavior of parallel applications and paves a path towards automatic optimization of data access. MPI-IO and parallel file systems are widely adopted in HPC systems to reduce the negative impact of the I/O gap as well as for ease of use. While MPI-IO and file systems bring I/O performance to an acceptable level, there is a significant scope for optimizing overall performance of parallel I/O. Many optimization strategies have been proposed for data read, such as data prefetching, two-phase collective I/O, data sieving and data requests scheduling and for data placement and organization, data replication, and data distribution.

Most existing I/O optimizations can benefit from knowing I/O behavior of an application. In many occasions, making the optimal design of performance improvements or choosing optimal system configuration for performance tuning requires application-specific information. For example, in a data prefetching enabled system, untimely or useless prefetching happens from time to time, which harms I/O performance. Knowing the application's data access pattern, the prefetcher can avoid untimely and useless prefetching. Section II describes more details on this example.

Noticing the widespread demand for retrieving parallel I/O access patterns of applications, we developed IOSIG tool that helps users to understand the I/O characteristics of their

978-0-7695-4691-9/12 $26.00 © 2012 IEEE
DOI 10.1109/CCGrid.2012.136

applications precisely. We motivate the use of I/O patterns for various optimizations and then describe the design and development of the IOSIG tool. The rest of this paper is organized as follows. We discuss the motivation for using I/O access pattern information in I/O read and write optimizations in Section II. Section III describes methodologies including classification of data access patterns, I/O signature notation, and design and implementation of IOSIG software. Section IV evaluates IOSIG toolkit. Section V exhibits several optimizations using IOSIG. We discuss related works in Section VI and conclude the paper in Section VII.

II. MOTIVATION

This section exhibits three I/O optimization techniques to show the usage of I/O access information and a characterization tool that gives comprehensive understanding of parallel program's I/O behaviors.

A. Accurate and timely data prefetching

Data prefetching is a proven effective way to improve data access performance and is widely used in many layers of computer storage system hierarchy. Typically, execution of many scientific applications includes multiple data access phases and computation phases. Data prefetching improves performance by overlapping application's data access phases and computation phases. The prefetching thread needs to predict what data the application will request and fetches the data from storage devices to local buffer (or a collective cache in parallel I/O system [10]) before the application issuing the actual requests.

Effectiveness of prefetching is measured by its coverage and accuracy. The coverage is the percentage of cache misses that were avoided by enabling prefetching. Accuracy refers to whether the prefetched data in cache is used by the application. If prefetching loads wrong data, or loads correct data too late, it does not mask the I/O latency but brings more useless data into the buffer, wasting I/O bandwidth and possibly replacing useful data in the local buffer. If prefetching loads the data too early, the cache may be polluted as useful data might be replaced.

Clearly, prefetching accuracy is directly affected by the accuracy of prediction on what data to prefetch and when to prefetch, and so is I/O performance. If a prefetching thread is aware of I/O characteristics of an application and uses that information effectively, application performance improves. Since prefetching method at runtime has to be faster in decoding future I/O accesses, I/O behavior has to be represented in a straightforward yet comprehensive representation. In Section III.B, we define such representation, called *I/O signatures*.

B. Determining the optimal data layout

In modern parallel file systems [11], data are typically distributed over multiple storage nodes in a round robin fashion, to take advantage of parallel accesses. This round robin data layout is most widely used because it can provide acceptable I/O performance for many scenarios. In PVFS2, it is the default data layout method namely "simple striping".

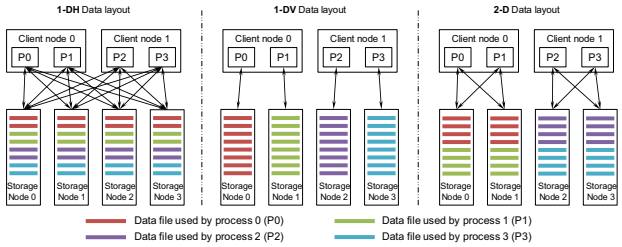

Figure 1. Different data layouts cause different interactions.

However, in some cases, it yields poor performance because "the number of storage nodes" is not the only parameter affecting I/O performance. The number of processes, the request size, offsets of requests, etc., also affect I/O latency. Parallel file systems do provide more than one data layout methods to advanced users for choosing optimal layout configurations. We name three most popularly adopted data layout methods as 1-DH, 1-DV, and 2-D data layout. As showed in Fig. 1, 1-DH data layout is the simple striping method and distributes data across all storage nodes. 1-DV data layout refers to the policy that data to be accessed by each I/O client process is stored on one storage node. 2-D data layout refers to the policy in which data to be accessed by each process is stored on a subset (called storage group) of storage nodes.

However, the problem is that, choosing the best data layout for an application is difficult or even impossible sometimes to make layout decision without knowing an application's I/O characteristics. The "best" data layout method for some application means while adopting this data layout, the given application's overall data access cost (measured in time) is minimum. Different data layouts result in different data access behaviors between clients and servers. Different interactive behaviors result in different data access cost measured in time.

In the previous work [12], we developed a mathematical model to investigate the data access cost for the data requests under different data layouts and application's different data access behaviors. To find an optimal data layout for some given application with lowest data access cost, the model analyzes the cost for each single request and calculates the overall cost by summing all single costs up. As a result, in this approach one part of the input to the model is data access pattern information, which also brings the demand to the proposed IOSIG tool.

C. Application-specific adaptive data layout

As mentioned in Section II.B, parallel file systems typically use simple striping. In addition, the stripe size is typically a fixed value. This may be acceptable for applications with some fixed data access patterns, however, cannot guarantee sustained performance improvements for various data access patterns. Assume we have an application with the data access pattern showed in Fig. 2, that is, the application access the first contiguous 4MB of each contiguous 16MB data segments in the whole file from the beginning. Fig. 2 also shows four different parallel file

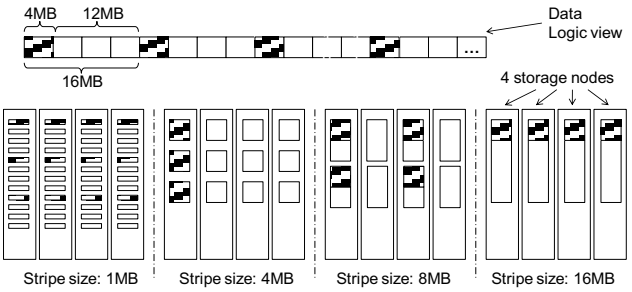

Figure 2. Only some specific stripe size can ensure optimal performance for a given data access pattern.

systems configured with "simple striping" but with different stripe size: 1MB, 4MB, 8MB, and 16MB. In the 1MB case, with each accessed 4MB data block distributed over four storage nodes, maximum parallelism and balanced workload are obtained, but there is too much network overhead. In the 4MB case, all the accessed data are on the first storage node and there is no inter-node parallel access at all. In the 8MB case, the first half of each stripe on storage node 0 and 2 will be accessed, which creates a more reasonable arrangement with respect to the 4MB case. While stripe size is 16MB, the first one fourth of all stripes will be accessed, which results in the best situation with the highest degree of parallelism and balanced workload obtained at the same time, without creating too much network overhead like the 1MB case.

An easily observed fact is that, for one data access pattern, only some stripe size can ensure optimum performance in parallel I/O systems. This observation is also verified to be true with experimental results in [13]. More noteworthy, in order to find this optimal stripe size, it requires to know the application's data access patterns. To cover the case where one single application may have various different data access patterns in different phases of its execution, in the previous work [13], we have proposed "application-specific adaptive data layout", which also needs the knowledge of an application's data access patterns.

III. IOSIG Tool Design and Development

IOSIG is a toolkit designed to reveal I/O behavior of parallel applications. We develop IOSIG with the goal of letting optimization strategies use knowledge of I/O characteristics at runtime with a low overhead. IOSIG software consists of two components 1) trace collector and 2) trace analyzer. These two independent tools also represent the two basic steps of the approach for I/O characterization, respectively. The first step is to trace the I/O operations of some application during its runtime and save all the traced operations histories into trace files. The second is to perform offline analysis on the trace files generated in the previous step, to produce data access pattern information. The trace analyzer produces I/O characteristics in the form represented in *I/O signatures* defined in our previous work [14]. The produced *I/O signatures* are easy to be read by humans or automatic optimization tools for capturing I/O behavior and for improving data access performance. The rest of this section describes the design and implementation of the trace collector, introduces I/O signatures, and provides details of the trace analyzer.

TABLE I. TRACED MPIIO FILE READ OPERATIONS.

	Non-collective	**Collective**
Blocking	MPI_File_read_at, MPI_File_read, MPI_File_read_shared	MPI_File_read_at_all, MPI_File_read_all, MPI_File_read_ordered
Non-blocking	MPI_File_iread_at, MPI_File_iread, MPI_File_iread_shared	MPI_File_read_at_all_begin/end, MPI_File_read_all_begin/end, MPI_File_read_ordered_begin/end

A. Trace collection

1) Traced file operations

The Message Passing Interface (MPI) [15] is a widely used programming model for easy and effective communication among processes of HPC applications. The MPI2 standard also defines a set of routines for transferring data to and from external storage, called MPI-IO. The MPI-IO library is also widely used in HPC applications as the basic file operation interface. IOSIG trace collector captures MPI-IO calls and records their information.

For file open, close, and seek operations, the corresponding MPI-IO routines are MPI_File_open, MPI_File_close, and MPI_File_seek. For file read and write operations, MPI-IO provides multiple variations of routines, considering HPC applications' needs, such as collective, non-collective, blocking, and non-blocking file operations. Table I shows all the MPI-IO read calls that IOSIG trace collector traces, and the write calls are identical in form using "write" to replace "read".

The trace collector captures the described MPI-IO routines by using the Profiling MPI interface (PMPI) to MPI. The Profiling MPI interface reroutes MPI calls to user defined instrumentation wrapped around the basic MPI calls. The new functionality is available as a static library, and a developer who wants to trace I/O behavior can link the library to any MPI-IO based application. Other than this simple linking step, there is no need for changing the code. Instrumentation is performed if an application is linked with the trace library. The instrumentation overhead is minimal with the use of PMPI. There is zero instrumentation overhead on the application if the library is not linked. This flexibility allows developers to use the library while debugging for performance and discard the linking step during the production runs.

2) Operation-level traced information

The trace collector gathers information of the file operations performed by each parallel process, i.e. the collector generates one file for each process, where each file operation trace contains the following information.

a) MPI rank and process ID of the process that performs the operation. We record this information to distinguish the source of the event because different processes may have different data access behavior.
b) File ID that identifies the file that the operation manipulates. One process may access more than one file with different ID or it may access the same file

more than one time, recording file ID can ensure a clear description of all the possible cases. The ID need to be unique for different files. Simple combination of process ID, MPI rank, file handler ID, and the time value can generate a unique ID.

c) Absolute file offset and request size of the data access, both in bytes. The size and the offset are the two most critical parameters for a data access operation.
d) Name of the invoked I/O routine, such as MPI_File_read. From a routine's name, we can get more information such as its style on synchronization and collaboration and this information is useful for the post analysis and also for pinpointing performance bottlenecks.
e) The I/O routine's invoking time, which is, the elapsed time from the starting of the whole application to the point when the routine is invoked. We gather this information for the optimizations where time information is necessary such as prefetching needs to make decisions on "when to prefetch data".
f) The I/O routine's end time - Using this end time together with the corresponding start time, we can find the time spent for performing an I/O operation. We can also calculate the percentage of the total I/O time in the total execution time, which gives users more hints on the level of application's I/O intensity and help the user to determine the application's performance bottlenecks.
g) Mapping between unique File IDs and file paths. This information is useful when the analyzer needs to pinpoint the exact file, for example, to analyze the correlation of files and processes.

B. Trace analysis

I/O accesses in parallel computing typically follow some patterns due to iterative computations. We classify these patterns into local and global patterns. The local patterns explain how each process accesses data and the global patterns provide an overview of the I/O characteristics of an application. We have classified local file access patterns in [14]. In this section, we introduce local and global access patterns and I/O signatures and describe the implementation of the trace analyzer.

1) Data access patterns

Based on study of various parallel I/O benchmarks, we classified I/O accesses based on various parameters, including spatial locality, size of accesses, temporal locality, iterative behavior, and I/O operations. Fig. 3 shows the classification of local I/O access patterns. The spatial locality is divided into contiguous, non-contiguous, and combinations of contiguous and non-contiguous patterns. The non-contiguous patterns are further divided based on byte order distance between successive accesses. Some data accesses may just occur once, and some other accesses may repeat multiple times in the same pattern. The repetitive behavior occurs often in loop codes. Request size plays a significant role in striping factor, stripe size, and the number of requests going to an I/O server. In our classification, we regard data accesses as small accesses if request size is less than a page size, medium if it is equal to the page size, and large if it is more than the page size. Temporal behavior is difficult to define. We classify based on intervals between accesses, which can be fixed or random. I/O operations are divided based on read and write.

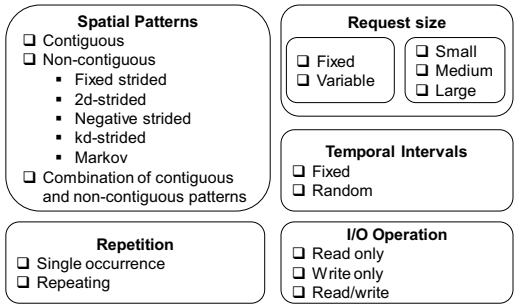

Figure 3. I/O access pattern classification.

2) Local I/O Signature

In order to be used by optimization strategies at runtime and for easier understanding of I/O behavior, we developed I/O signature notations, which are compact and informative. Local I/O signatures include trace signatures and pattern signatures. A trace signature contains the detailed information of all I/O operations it covered. The pattern signature provides an abstract description of the trace signature. In form, a trace signature looks as follows.

{I/O operation, file_id, initial position, dimension, ([{offset pattern}, {request size pattern}, {pattern of number of repetitions}, {temporal pattern}], [...]), # of repetitions}

It stores information of an I/O operation, starting offset, depth of a spatial pattern, temporal pattern, request sizes, and repetitive behavior. In some instances, offsets, request sizes, timing, and number of repetitions also contain a pattern. Random temporal patterns are not captured in the trace signature.

A pattern signature contains all the five factors of the classification and looks as follows.

{I/O operation, <Spatial pattern, Dimension>, <Repetitive behavior>, <Request size>, <Temporal Intervals>}

These signatures represent compressed I/O trace files in the presence of regular access patterns. However, when accesses are random, signatures are of limited use. It still can be beneficial to construct only a pattern signature for random accesses without constructing a trace signature. Automatic prefetching and data layout optimization methods can use pattern signatures for deciding whether to perform optimizations and use trace signatures for deciding the optimization parameters.

3) Global I/O Signature

Local I/O signature represents the information of a single process's data access patterns. By analyzing all the local I/O Signatures of the same application, we are able to acquire global I/O signature that represents the data access pattern of the whole application.

Global I/O signature is necessary because in some cases local I/O signatures cannot provide the whole picture of application's data access, thus may not be able to help users

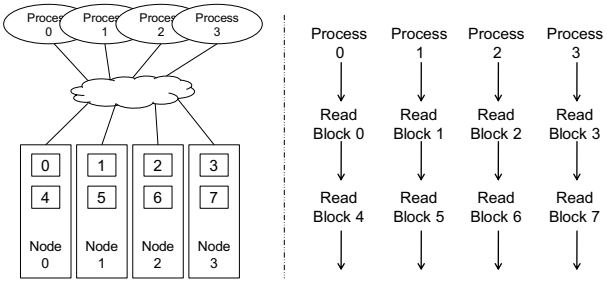

Figure 4. An example that shows that global I/O signature is necessary to find optimal data layout.

to make a correct decision while optimizing I/O performance for the whole application. For example, showed in Fig. 4, some parallel file systems have four storage nodes and an application has 4 processes that need to access a shared file distributed over all storage nodes. The request size is equal to the stripe size. The processes have the same data access pattern, fixed strided but with different offsets. So process *i (0≤i≤3)* accesses data stripe numbered *4(n-1)+i* for its *n*th access. In other word, process 0 accesses data stripe 0, 4, 8 …; process 1 accesses data stripe 1, 5, 9…; and so on. By examining the data accesses of a single process, we can find that the accessed data locate on one single storage node, which is not a balanced workload with four storage nodes available. Thus, it is necessary to find a better stripe size and rearrange the data. However, by considering the data accesses of all four processes, the whole application's workload achieves balance and the data accesses become contiguous spatially. Hence, global I/O signature that includes the data access information for the whole application can help choosing the best data layout. A global I/O signature includes the following information.

a) The total number of processes.
b) Each global data access pattern along with the corresponding processes IDs and the starting and end time information. A global data access pattern may be different from those process-level patterns it covers. As shown in Fig. 4, the global data access pattern is of contiguous type. The time information is necessary because there might be multiple "global data access patterns" in the same execution phase.
c) File sharing information: Summary of the Process ID information in local patterns provides what processes share files. This information helps users to identify the data used by multiple processes and to apply I/O optimization such as creating multiple replications of the data for reading it fast and in parallel.

4) Implementation of the trace analyzer

We develop the trace analyzer in Python to perform offline trace analysis. The trace analyzer utilizes a basic "template matching" approach to recognize data access patterns from trace files. Each trace file can be regarded as a sequence or a list of file operation records. The analyzer uses a cursor to mark its progress during the analysis. It starts from the first record and move forward to examine all records until reaching the end of the list. During scanning, the analyzer picks a predefined access pattern as the template, to check whether the pattern matches the records around the cursor. If the pattern matches, the cursor moves forward along with the same pattern and continues in the trace until the match does not hold. If there is no match for the first template, the analyzer switches to other templates and scans again. If the analyzer fails to find a match for all templates, it skips the current record, moves the cursor forward, and starts over the matching at the new position.

Besides local and global I/O signatures, the analyzer also generates several other outputs. 1) A figure showing the offset patterns. 2) A figure showing the request size patterns. 3) A histogram figure of data reads and writes on time axis, where for each file operation, the x-axis represents time spent on a file operation and the y-axis represents the actual bandwidth. 4) A "protobuf" [16] based output file containing the identified I/O Signatures. While I/O signature form explained above is human readable, we write the signatures using protocol buffers format for automatic optimization systems. The protocol buffers format developed by Google [16] encodes structured data in an efficient and extensible way.

Both the trace collector and the trace analyzer work at application level and require no complex installation, which is a significant advantage for users without the super user permissions in large-scale machines.

IV. SOFTWARE EVALUATION

We evaluate the IOSIG toolkit on a 65-node SUN Fire Linux-based cluster, in which there are 64 computing nodes and one head node. All the nodes are equipped with HDDs, and are connected with Gigabit Ethernet and InfiniBand interconnections. The evaluations include the overhead and resource consumption by the IOSIG software. We analyzed performance of parallel I/O benchmarks, such as MPI-TILE-IO, IOR benchmark, and PIO-BENCH.

A. Runtime overhead

The trace collection library needs to be linked with an application in order to take effect. The tracing library linked application generates the trace files during its execution. The goal of this evaluation is to show that the overhead of IOSIG tool is negligible. To measure the overhead of the trace collector, we find the difference of execution times of the application with and without linking the tracing library. Fig. 5 and Fig. 6 show the trace collection overhead in running IOR and MPI-TILE-IO benchmarks, respectively. In the figures, we can observe that the trace collection overhead is very low. The bars in the graph compare the execution times of the original application and the application linked with IOSIG trace collection utility. We run the benchmarks with varying number of processes on 8 client nodes, accessing data from 16 storage nodes configured as one parallel file system with PVFS2. For IOR benchmark, the overhead is 1%, and for MPI-TILE-IO benchmark it is around 2% in most cases and below 6% overall. These results show that the overhead introduced by the trace collector is remarkably low and acceptable.

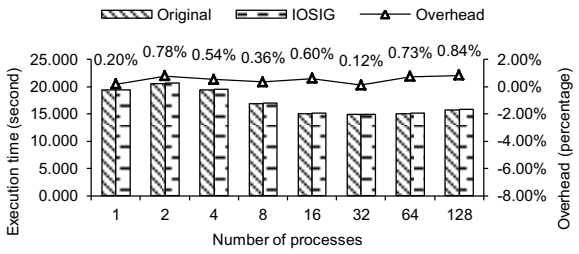

Figure 5. Trace collection overhead with IOR benchmark.

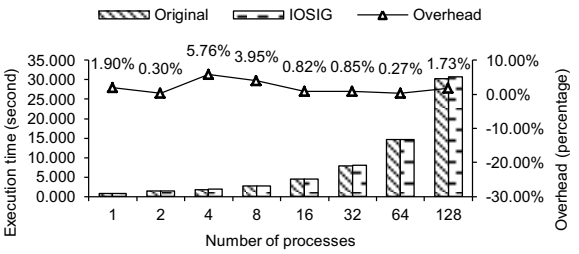

Figure 6. Trace collection overhead with MPI-TILE-IO benchmark.

B. Scalability

The experimental results in Fig. 5 and Fig. 6 also show the scalability of IOSIG tracing library. As the number of processes increases, the trace collection overhead remains low. We tested both IOR and MPI-TILE-IO benchmarks while increasing the number of processes (1 to 128). We observe that the percentage of overhead remains negligible. This is because that the library adopts a totally distributed approach, where each MPI process only generates its own trace file, and there is no communication or synchronization among processes.

C. Trace file size

The size of each I/O event record is around 100 bytes. The size of each trace file is proportional to the number of I/O events it includes. For a test using PIO-BENCH with 4 processes accessing an 8GB file, the generated trace files contains 5296 file operation records and their combined sizes are 514490 bytes, thus 102 bytes per record.

D. Analysis performance

Execution time of the offline trace analysis is also proportional to the number of traced records. In other words, the time complexity is *O(n)*. During our tests, the analyzer takes less than 2 seconds to finish analyzing the mentioned trace file with 5296 records on a single core of the client nodes of our system. When necessary, this process can be parallelized, such as, using shell scripts to start multiple trace analyzers working on multiple trace files simultaneously.

E. Memory footprint

The trace collector writes one record for each I/O routine to the trace files during the routine's execution. Nowadays, local file systems usually adopt buffering mechanism that saves the written data in a buffer and flush the data back to the trace file occasionally in order to avoid too many small writes to storage devices. The underlying local file system client manages the writing buffer. Hence the trace collector itself does not consume much memory.

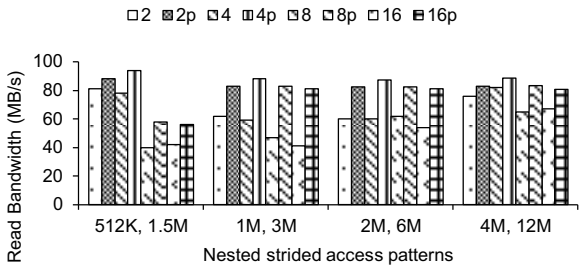

Figure 7. Bandwidth of PIO-Bench, with Nested Strided pattern on NFS. The numbers in the legend labels are the numbers of processes, and the suffix "p" in the labels indicates the cases where prefetching is enabled.

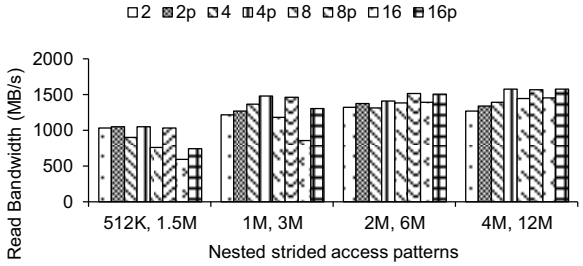

Figure 8. Bandwidth of PIO-Bench, with Nested Strided pattern on PVFS. The numbers in the legend labels are the numbers of processes, and the suffix "p" in the labels indicates the cases where prefetching is enabled.

The trace analyzer maintains a size-limited queue to store the I/O events that are under analysis. As mentioned in Section III.B, the analyzer uses "template matching" method to detect predefined patterns in the queue of I/O events retrieved from trace files. To avoid the program using too much memory, we limited the size of the queue to less than 5000 file operations. Using python environment to run the analyzer on the trace files with 5296 records, during several executions, we observe that the memory consumption for the whole python environment is less than 40MB, which is acceptable as the memory resource is not scarce for an offline analysis.

V. Optimizations using I/O Signatures

This section presents two I/O optimizing techniques that benefit from IOSIG for their optimizations, to provide a better illustration for IOSIG's effectiveness and practicality.

A. I/O Signature based data prefetching

We described in this subsection how knowledge of data access patterns helps improve the performance of data prefetching in parallel I/O system.

PIO-BENCH provides the testing ability with several different I/O access patterns. We set the access pattern as "nested strided (read)", work units as 100, and request size as 4096 bytes. We compile PIO-BENCH to link it with IOSIG

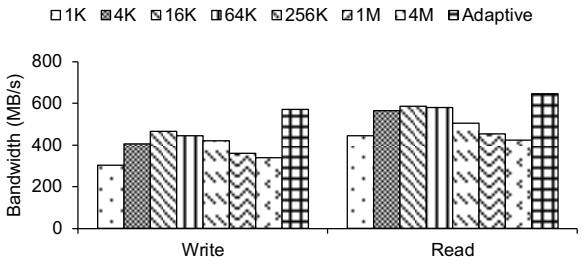

Figure 9. Average bandwidth in Ethernet environment. The sizes in the legend labels are stripe sizes, and "Adaptive" means the adaptive stripe size in the adaptive data layout scheme.

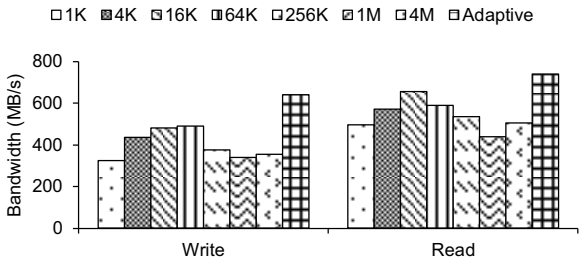

Figure 10. Average bandwidth in InfiniBand environment. The sizes in the legend labels are stripe sizes, and "Adaptive" means the adaptive stripe size in the adaptive data layout scheme.

tracing library and then run it with four MPI processes. We conduct this test on two different file systems, NFS and PVFS2. From generated traces, IOSIG trace analyzer produces the following local trace signature.

{MPI_READALL, file_id, initPos, 2, ([{initPos, 32768, 1}, 4096, 1], [{initPos+8192, 32768, 1}, 4096, 1, fixed_interval], 100), 1}

The signature indicates that each process starts its file operations from offset *initPos*, reads 4096 bytes, seek forward by 4096 bytes (so the second read's offset is *initPos+8192*), read 4096 bytes again, seek 20480 bytes, and then repeat the above two sets of reads and seeks 100 times. In this circumstance with default read-ahead prefetcher, when prefetching size is 4096 bytes, the prefetching cache's hit ratio is 0. In order to make the hit ratio as large as 100%, prefetching size needs to be larger than 20480+4096=24576 bytes (assuming every prefetching occurs timely), but only 25% of fetched data gets accessed, which is inefficient. With the help of data access pattern information included in I/O signatures, it is possible to make data prefetching highly accurate and timely, especially for those applications that have complicated data access patterns. Fig. 7 and Fig. 8 show the I/O performance improvements by enabling the I/O signature based data prefetching in the parallel I/O system. The average performance gain on NFS is around 36%, and that on PVFS for larger strides is around 20% [14].

B. Access pattern-aware adaptive data layout

In section II, we described how configuration of stripe sizes in parallel file systems affects the overall I/O performance (see Fig. 2). An application accessing a large data set may have different access patterns in different phases of its execution or on different data segments. In order to achieve optimal I/O performance, we need to choose data layouts adaptively, by changing stripe sizes for different parts of the application depending on access patterns.

We use 4 IOR benchmarks with different configurations chained together, to simulate a large I/O intensive application doing both reads and writes with different access patterns. Each IOR runs on its own data, and the request sizes are 1KB, 4KB, 64KB, and 1MB, respectively specified by IOR command line options. Take the 64KB read test with IOR as an example, the local trace signature is: *{MPI_READAT, file_id, 0, 1, ([{0, 65536, 1}, 65536, 1, fixed_interval], 65536), 1}*, which means that IOR does a contiguous read operation, the size of each single read is 64KB and the total number of reads is 65536. The other IOR's access patterns are almost the same, only with different request sizes and numbers.

With default configuration of PVFS2 and without knowing the I/O Signatures, all data used by this simulated application are distributed over all storage nodes with a uniform default strip size. With the knowledge of I/O Signatures, we determined that the optimal stripe sizes of different data sets were 4 KB, 16 KB, 64 KB, and 1 MB, respectively. We tested the overall I/O bandwidth for reads and writes, with different uniform strip sizes and the adaptive data layout that allows different optimal strip size for different data sets. Fig. 9 and Fig. 10 show the results with the adaptive data layout selection strategy. The performance improvement of write operations is between 25% and 101%, and that of read operations is between 9% and 71% compared with the default layout strategies where the stripe size is static and fixed for all the data sets.

VI. RELATED WORK

Reed et al. have studied and categorized common data access pattern of parallel applications [17] [18] [19], including the global data access pattern.

Carns et al. of Argonne National Lab have designed and developed Darshan to explore I/O characteristics of HPC application that ran on IBM Blue Gene/P series of computers where Darshan gets deployed full time [6] [20]. To keep runtime overhead minimum, Darshan traces several accumulative or statistics information and access patterns of application or files, and does not provide details any single file operation. While Darshan is useful for understanding the I/O behavior, IOSIG goes further in providing signatures that optimization strategies can directly utilize.

There are a few tools focusing on tracing data I/O events, like HPCT-IO [5], LANL-Trace [8], IOT [7], and ScalaIOTrace [9], but these tools do not provide enough comprehensive trace analysis or data access representation.

HPC community also developed several general tracing and profiling tools with analysis and visualization functionalities, such as TAU [1], jumpshot [2], Periscope [3], Upshot [21], and EPILOG [4]. These tools mainly focus on profiling and analyzing an individual application's parallel processing performance, MPI messaging between computing nodes and processes/threads, and I/O behaviors in main

memory layer instead of file storage layer. IOSIG is comprehensive in terms of providing tracing capability along with the analysis of patterns and signature representations.

VII. CONCLUSIONS

High performance computing is moving towards exa-scale, and the HPC community has proposed and adopted many I/O optimization techniques to ease the widely recognized I/O bottleneck and to make large-scale machines more efficient. One requirement of efficient I/O optimizations is to understand applications I/O behavior. Many existing I/O optimizations can benefit from knowing I/O access patterns of an application. We exhibit three optimizations of this type to explain the need for an I/O characterization tool that gives comprehensive understanding of the I/O behavior of parallel applications and paves a path for optimization of data access. In this paper, we describe the work we have done on meeting this requirement. We present IOSIG tool that helps users to understand the I/O characteristics of their applications precisely. IOSIG works in two steps, 1) to trace file operations during one application's runtime and generate trace files and 2) to perform analysis on the generated trace files to get the application's data access patterns presented in I/O Signatures. Comparing with existing I/O characterization tools, IOSIG has several advantages. 1) It is light-weight. 2) It produces traces that are more detailed. 3) It can detect local and global data access patterns from trace files. 4) It works in application level, which is a valuable advantage for users without the administrator's permission in large-scale machines. The software evaluation proves that IOSIG keeps the overhead at a minimal level with common resource requirements. We explain the usage of IOSIG tool through two existing parallel I/O optimization strategies.

In the future, we plan to extend IOSIG with the ability to identify the data I/O intensity of HPC applications quantitatively.

ACKNOWLEDGMENT

The authors are thankful to and Dr. Robert Ross and Samuel Lang of Argonne National Laboratory for their constructive and thoughtful suggestions toward this work. The authors are also grateful to anonymous reviewers for their valuable comments and suggestions. This research was supported in part by National Science Foundation under NSF grant CCF-0621435 and CCF-0937877, and in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

REFERENCES

[1] Shende, S. and Malony, A. D., "The tau parallel performance system," *International Journal of High Performance Computing Applications,* vol. 2, pp. 287--311, 2006.

[2] Zaki, O., Lusk, E., Gropp, W., and Swider, D., "Toward scalable performance visualization with jumpshot," *International Journal of High Performance Computing Applications,* vol. 13, no. 3, pp. 277-288, 1999.

[3] M. Gerndt and M. Ott, "Automatic performance analysis with periscope," *Concurrency and Computation: Practice and Experience,* vol. 22, no. 6, pp. 736-748, 2010.

[4] B. Mohr and F. Wolf, "KOJAK - a tool set for automatic performance analysis of parallel," in *Proc. of the European Conference on Parallel Computing*, 2003.

[5] S. Seelam, I.-H. Chung, D.-Y. Hong, H.-F. Wen, and H. Yu, "Early experiences in application level I/O tracing on Blue Gene systems," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2008.

[6] Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., and Riley, K., "24/7 Characterization of petascale I/O workloads," in *IEEE International Conference on Cluster Computing*, 2009.

[7] P. C. Roth, "Characterizing the I/O behavior of scientific applicationson the Cray XT," in *Proceedings of the 2nd InternationalWorkshop on Petascale Data Storage*, 2007.

[8] "HPC-5 open source software projects: LANL-Trace," [Online]. Available: http://institute.lanl.gov/data/software/#lanl-trace.

[9] Vijayakumar, K., Mueller, F., Ma, X., and Roth, P. C., "Scalable I/O tracing and analysis," in *Proceedings of the 4th Workshop on Petascale Data Storage*, 2009.

[10] Liao, W.-keng, Ching, A., Coloma, K., and Choudhary, A., "An implementation and evaluation of client-side file caching for MPI-IO," in *Proc. of the IEEE International Parallel and Distributed Processing Symposium*, 2007.

[11] Carns, P.H., Ligon, W.B. III, and Ross, R.B., "PVFS : a parallel file system for linux clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[12] H. Song, Y. Yin, Y. Chen, and X.-H. Sun, "A cost-intelligent application-specific data layout scheme for parallel file systems," in *Proc. of the 20th International ACM Symposium on High Performance Distributed Computing*, 2011.

[13] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "A segment-level adaptive data layout scheme for improved load balance in parallel file systems," in *Proc. of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2011.

[14] 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 *International Conference for High Performance Computing Networking Storage and Analysis (SuperComputing)*, 2008.

[15] Gropp, W., Lusk, E., and Skjellum, A., Using MPI: portable parallel programming with the message passing interface, MIT Press, 1999.

[16] Google Inc., "Protocol buffers - Google's data interchange format," Google Inc., 2008.

[17] Madhyastha, T.M. and Reed, D.A., "Learning to classify parallel input/output access patterns," *IEEE Transactions on Parallel and Distributed Systems,* vol. 13, no. 8, 2002.

[18] Madhyastha, T.M. and Reed, D.A., "Exploiting global input/output access pattern classification," in *ACM Press.*, 1997.

[19] Madhyastha, T.M. and Reed, D.A., "Input/output access pattern classification using hidden markov models," in *Workshop on Input/Output in Parallel and Distributed Systems*, 1997.

[20] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross., "Understanding and improving computational science storage access through continuous characterization," in *27th IEEE Conference on Mass Storage Systems and Technologies*, 2011.

[21] V. Herrarte and E. Lusk, "Study parallel program behavior with Upshot," MCS Division, Argonne National Laboratory, 1991.