J Supercomput (2015) 71:840–870
DOI 10.1007/s11227-014-1329-0

IOPro: a parallel I/O profiling and visualization framework for high-performance storage systems

Seong Jo Kim · Yuanrui Zhang · Seung Woo Son · Mahmut Kandemir · Wei-keng Liao · Rajeev Thakur · Alok Choudhary

Published online: 22 November 2014

Abstract Efficient execution of large-scale scientific applications requires high-performance computing systems designed to meet the I/O requirements. To achieve high-performance, such data-intensive parallel applications use a multi-layer layer I/O software stack, which consists of high-level I/O libraries such as PnetCDF and HDF5, the MPI library, and parallel file systems. To design efficient parallel scientific applications, understanding the complicated flow of I/O operations and the involved interactions among the libraries is quintessential. Such comprehension helps identify

S. J. Kim (✉) · M. Kandemir
Pennsylvania State University, University Park, PA 16802, USA
e-mail: seokim@cse.psu.edu

M. Kandemir
e-mail: kandemir@cse.psu.edu

Y. Zhang
Intel Corporation, Santa Clara, CA 95054, USA
e-mail: yuanrui.zhang@intel.com

S. W. Son
University of Massachusetts Lowell, Lowell, MA 01854, USA
e-mail: seungwoo_son@uml.edu

W. Liao · A. Choudhary
Northwestern University, Evanston, IL 60208, USA
e-mail: wkliao@eecs.northwestern.edu

A. Choudhary
e-mail: choudhar@eecs.northwestern.edu

R. Thakur
Argonne National Laboratory, Argonne, IL 60439, USA
e-mail: thakur@mcs.anl.gov

I/O bottlenecks and thus exploits the potential performance in different layers of the storage hierarchy. To profile the performance of individual components in the I/O stack and to understand complex interactions among them, we have implemented a GUI-based integrated profiling and analysis framework, *IOPro*. IOPro automatically generates an instrumented I/O stack, runs applications on it, and visualizes detailed statistics based on the user-specified metrics of interest. We present experimental results from two different real-life applications and show how our framework can be used in practice. By generating an end-to-end trace of the whole I/O stack and pinpointing I/O interference, IOPro aids in understanding I/O behavior and improving the I/O performance significantly.

Keywords MPI-IO · Parallel file systems · Parallel NetCDF · HDF5 · I/O software stack · Code instrumentation · Performance visualization

1 Introduction

Users of HPC systems often find an interesting situation: it is not the CPU, memory, or network that restricts the performance of applications, but the *storage systems*. In fact, the prior research [1–13] shows that I/O behavior can be a dominant factor that determines the overall performance of many HPC applications. Therefore, understanding the parallel I/O operations and the involved issues is critical to meet the requirements for a particular HPC system and/or decide I/O solutions to accommodate expected workloads.

Unfortunately, understanding parallel I/O behavior is not trivial as it is a result of complex interactions between hardware and a number of software layers, collectively referred to as the *I/O software stack*, or simply *I/O stack*. Figure 1 illustrates a typical I/O stack used in many (if not most) HPC systems. Note that this figure is intended to present the software layers and a logical view; it is not meant to illustrate the physical connectivity and configuration of an I/O stack. At the lowest level is the storage hardware consisting of disks, controllers, and interconnection network connecting multiple physical devices. At this level, data are accessed at the granularity of blocks across multiple physical devices such as in a RAID array. Above the storage hardware are the parallel file systems, such as Lustre [14], GPFS [15], PanFS [16], and PVFS [17]. The roles of the parallel file system are to manage the data on the storage hardware, present the data as a directory hierarchy, and coordinate accesses to files and directories in a consistent fashion. The MPI-IO library [18], part of MPI-2 [19], sits, as a middleware, on top of the parallel file systems. It provides a standard I/O interface and a suite of optimizations including data caching and process coordination [1–6].

While the MPI-IO interface is effective and advantageous because of its performance and portability, it does not support structured data abstraction for scientific applications. To provide such structured data format, high-level I/O libraries (e.g., Parallel netCDF [20] and HDF5 [21]) are added on top of MPI-IO. These high-level libraries allow application programmers to better describe how their applications access shared storage resources. As shown in Fig. 1, a parallel I/O application may directly call the MPI-IO library or a POSIX I/O function to access the disk-resident

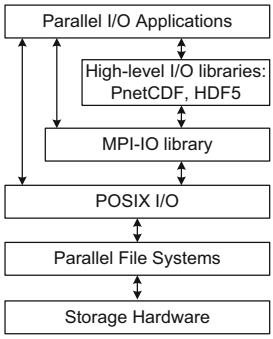

Fig. 1 I/O software stack

data sets. Alternatively, large-scale, data-intensive applications may exercise several layers of the I/O stack. Since the interactions among these layers are complex and unpredictable, understanding and characterizing those interactions must precede performance tuning and optimization for the HPC applications.

One approach to understanding I/O behavior is to let application programmers or scientists instrument the I/O software stack manually. Unfortunately, this approach is extremely difficult and error prone. In fact, instrumenting even a single I/O call may necessitate modifications to numerous files from the application to multiple I/O software layers below. Worse, a high-level I/O call from the application program can be *fragmented* into multiple calls (subcalls) in the MPI library, which is severely challenging. Since many parallel scientific applications today are expected to run on large-scale systems with hundreds of thousands of processes to achieve better resolution, even collecting and analyzing trace information from them is laborious and burdensome.

Motivated by these observations, we have developed a performance analysis and visualization framework for parallel I/O, called *IOPro*. Instead of manually instrumenting source code of applications and other components of the I/O stack, IOPro takes as input the description of the target I/O stack and the application program, *automatically* generates the *instrumented* I/O stack to trace the specified I/O operations, and compiles and builds it. Next, it runs the application with detailed configuration information for I/O servers (PVFS2 in our case) and an MPI process manager, *mpiexec*. Then, it collects and analyzes the trace log data and presents detailed statistics based on user-specified metrics of interest.

A unique aspect of IOPro is to provide an integrated profiling and analysis environment for the entire parallel I/O software stack. Our implementation is designed to profile the performance of individual components in the I/O stack. Using the profiled results, the bottleneck is identified and a proper solution is applied to eliminate the bottleneck and, thus, to improve the performance. Section 6.3 explains how IOPro is used to provide an appropriate I/O strategy in detail. In addition, IOPro can work with different I/O stacks and user-provided probe code. For instance, with the user-specified probes, it can trace parallel I/O in Blue Gene/P systems that deploy the I/O forwarding scalability layer (IOFSL) [22]. Also, it can provide a reconfigurable setup for the I/O stack. Last but not least, it provides a hierarchical view for parallel I/O.

In our implementation, every MPI I/O call has a unique identification number in the MPI-IO layer and is passed to the underlying file system with trace information. This mechanism helps associate the MPI I/O call from the application with its subcalls in the file system layer systematically. Using this information, our framework visualizes detailed I/O performance metrics for each I/O call, including latency, throughput, estimated energy consumption, and the number of I/O calls issued to and from servers and clients.

We believe that IOPro is a powerful and useful tool for scientists and application programmers as well as performance engineers. For the scientists and application programmers who do not have an in-depth knowledge of underlying complexities of emerging HPC systems, it can provide detailed I/O statistics that helps them understand the characteristics of I/O from the perspective of the applications. Using the performance measurements of the underlying I/O stack, more optimized code can be implemented. For the performance engineers, it enables customized instrumentation for more detailed performance measurements. Therefore, IOPro can enable insights into the complex I/O interactions of scientific applications and provide an adaptive I/O strategy.

The rest of this paper is organized as follows. Related work is discussed in Sect. 2. In Sect. 3, we discuss challenges in characterizing I/O performance of HPC systems. Section 4 gives an overview of our approach to I/O instrumentation. Section 5 elaborates on the details of code instrumentation, computation methodology, and query usages. An experimental evaluation of the proposed tool is presented in Sect. 6, followed by concluding remarks in Sect. 7.

2 Related work

There exists prior research in profiling performance and diagnosing the related problems in large-scale distributed systems. In this section, we discuss the work related to static/dynamic instrumentation, and tracing and profiling frameworks.

2.1 Static/dynamic instrumentation

Over the past decade, a lot of static/dynamic code instrumentation tools have been developed and tested that target different machines and application domains. Static instrumentation generally inserts probe code into the program at compile time. Dynamic instrumentation, on the other hand, intercepts the execution of an executable at different points of execution and inserts instrumentation code at runtime. ATOM [23] statically instruments the binary executable through rewriting at compile time. FIT [24] is an ATOM-like static instrumentation tool but aims at retargetability rather than instrumentation optimization. HP's Dynamo [25] monitors an executable's behavior through interpretation and dynamically selects "hot instruction traces" from the running program. DynamoRIO [26] is a binary package with an interface for both dynamic instrumentation and optimization. PIN [27] is designed to provide a functionality simulator to the ATOM toolkit; but, unlike ATOM, which instruments an executable statically by rewriting it, PIN inserts the instrumentation code dynamically

while the binary executable is executing. Dyninst [28] and Paradyn [29] are designed for dynamic instrumentation to reduce the overheads incurred during instrumentation.

2.2 Tracing and debugging

Tools such as CHARISMA [30], Pablo [31], and tuning and analysis utilities (TAU) [32] collect and analyze file system traces [33]. Paraver [34] is designed to analyze MPI, OpenMP, Java, hardware counter profiles, and operating system activity. Open|SpeedShop [35] is targeted to support performance analysis of applications. Kojak [36] aims at the development of a generic automatic performance analysis environment for parallel programs, and Stack Trace Analysis Tool (STAT) [37] is designed to help debug large-scale parallel programs.

2.3 Large-scale distributed system tracing

To understand complex system behavior, Magpie [38] automatically extracts a system's workload during execution and produces a workload model. This work has been extended to datacenters [39]. Fay [40] provides dynamic tracing of distribute systems for user- and kernel-mode operations in x86-64 Windows systems. Lee et al. [41] proposed the dynamic probe class library API for large-scale systems, extended by DynInst. Darshan [42] captures I/O behavior such as I/O rates, transaction sizes, and I/O library usage in HPC applications. Vampir [43] provides an analysis framework for MPI applications, and IOPin [44] performs the runtime profiling of parallel I/O operations in HPC systems.

Our work differs from the aforementioned efforts in that we provide an integrated profiling and analysis environment that uses source code analysis to *automatically instrument the entire I/O stack*. Also, unlike some of the existing profiling and instrumentation tools, our proposed tool can work with different I/O stacks. Supported by end-to-end tracing functionality, it presents various analytical performance metrics, such as latency, throughput, energy consumption, and call information, to investigate detailed I/O behavior using specified query formats, on the fly.

3 Background

In this section, we discuss the challenges in characterizing the I/O performance of modern HPC systems. We also explain the importance of the collected performance metrics and their usage to improve the I/O performance.

3.1 Challenges

Modern HPC systems comprise multiple entities such as high-level I/O libraries (e.g., PnetCDF and HDF5), the MPI library as a middleware, and POSIX on top of the underlying parallel file systems. When a scientific application runs on large-scale systems with hundreds of thousands of processes, its operation is often complex and

difficult to understand. Frequently, application I/O calls can be *optimized* in the middle I/O layer to achieve better performance. Also, the high-level I/O calls from applications can break down into multiple calls in the MPI library, which make it extremely challenging to understand and reason about.

Most of the previous research in this area focuses on presenting performance metrics for the given applications. However, these statistics only reflect quantitative information at each layer of the I/O stack rather than a deep understanding of the I/O interaction and association from the application through the multiple libraries to the underlying parallel file system. Instead, our scheme provides a qualitative approach to associate high-level I/O from the application with the operations in the underlying parallel file system by *automatically* injecting the probe code, and visualizes the user-provided metrics of interest for better understanding. As a result, it helps scientists and system engineers profile and improve the performance of applications running on deep storage hierarchies. We want to emphasize that, while in principle a knowledgeable user can manually instrument an I/O stack, in practice this is very difficult due to the complex interactions between different layers, which makes it very challenging to pass/propagate values/metrics of interest across the layers and accumulate results.

3.2 Performance metrics

Depending on I/O demands and data access patterns, a given parallel application may require bounded execution time, relatively low throughput, or both. In many parallel applications, the requests from different processes are frequently interleaved and merged into contiguous portions of the file to reduce the high I/O latency. When such an optimization, broadly referred to as *collective I/O*, is used, all the joined processes broadcast and exchange the information related to the I/O request. If the I/O access patterns of all processes are contiguous and can benefit from collective I/O, an aggregator process can access disk-resident data by two-phase I/O: (1) redistribution of data to the processes (communication phase) and (2) a single, large, contiguous access to data (I/O phase) in case of write operation. This method has the additional cost of interprocess communication, but it can significantly reduce the I/O time. Although collective I/O is performed to improve I/O latency, the performance of collective I/O can be significantly affected by the critical path from the process to the server. For example, if the process on the critical path has a small size of temporary buffer needed for two-phase I/O, frequently copies the data into the buffer, and communicates other processes for redistribution, it can degrade the performance. In this case, the critical path from the aggregator process to the server dominates the overall application performance. Also, the I/O server on the critical path can be a major bottleneck in certain situations such as explosion of I/O requests to the server, network hardware failure, or faulty I/O server. Since the I/O operations interfere with each other during the execution of multiple applications, it is also important to figure out the number of I/O operations issued and the server(s) the I/O requests from the applications target. In case of burst I/O to the server, by setting MPI hints the application can perform I/O operations without striping data to the bottleneck I/O server. Using our framework, therefore, users can easily/automatically generate the *instrumented I/O stack* to capture

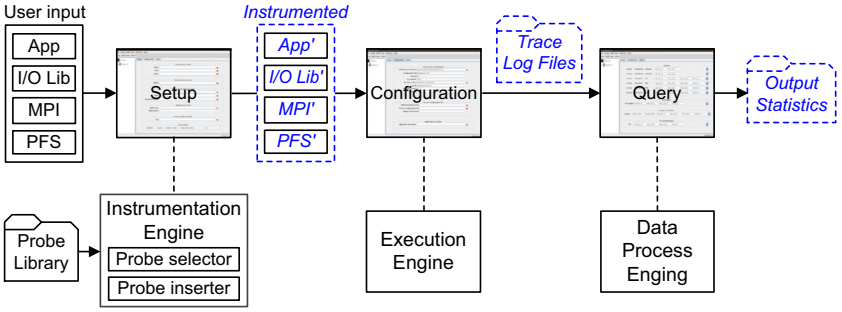

Fig. 2 Overview of IOPro. It takes as input an application program and I/O stack information, and builds an instrumented I/O stack to profile I/O operations, separately from the original I/O stack. After configuring the PFS server (PVFS in our case) and MPI program launcher, *mpiexec*, it runs the application program. The query analyzer then collects trace log files and returns the statistics based on the metrics of interest

latency, throughput, and I/O call access information that affect the performance, and analyze those metrics by visualization.

4 High-level view of instrumentation, execution, and visualization

In this section, we first give a high-level view of IOPro. As shown in Fig. 2, IOPro consists of three main components: *instrumentation engine*, *execution engine*, and *data processing engine*. Each of these components works with its corresponding front-end (i.e., setup view, configuration view, and query analyzer, respectively), as will be explained in the following subsections.

4.1 Instrumentation engine

To provide an automated I/O tracing functionality for parallel applications, IOPro accepts the necessary information from the setup view (Fig. 3). This information includes the directory locations of an application and the I/O software stack such as the parallel file system (e.g., PVFS), the MPI library (e.g., MPI-IO), and the high-level I/O library (e.g., HDF5). It also takes the location of trace log files generated by each layer of the I/O stack. As shown in Fig. 3, an instrumented file for a high-level I/O library is automatically chosen, depending on the selected high-level I/O library. In the example, H5FDmpio.c would be instrumented when targeting HDF5. In addition, the `make` option "`make -f Makefile.bb flash_benchmark_io`" is given to compile the FLASH I/O benchmark [45]. Further, if desired, a trace option can be chosen here to track a specific operation and code range (i.e., write, read, or both) and application source code lines (1–5,000 in this case). Note that the current implementation targets a user-level parallel file system. Unlike other system-level parallel file systems such as Lustre, GPFS, and PanFS, PVFS2 clients and servers can run at user level.[1] Therefore,

[1] PVFS2 also supports an optional kernel module that allows a file system to be mounted as in other file systems.

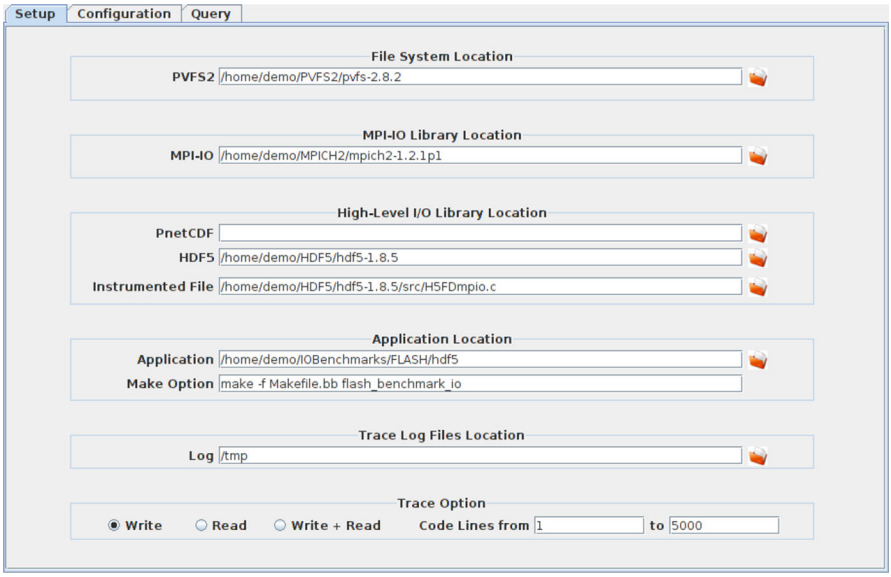

Fig. 3 The front-end (setup view) of the instrumentation engine of IOPro

we can easily implement the functionality to trace and profile I/O operations in a hierarchical fashion, without kernel modifications that are normally not allowed in system-level file systems.

As a back-end of setup view, an instrumentation engine consists of a probe selector and a probe inserter. In this context, a *probe* is a piece of code being inserted into the application code and I/O software stack (e.g., in the source code of the high-level I/O library, the MPI library, and PVFS2), which helps us collect the requested statistics. Using the user-provided information from the setup, the instrumentation engine inserts the probe into the appropriate locations in the I/O stack automatically, and generates an instrumented version of PVFS2, the MPI-IO library as well as the high-level I/O library. More details are provided in Sect. 5.

4.2 Execution engine

After creating the instrumented I/O stack and the application program successfully in the previous stage, the execution engine builds and compiles them. Also, as in Fig. 4, the front-end (configuration view) in the execution engine takes information about the file systems, storage locations, endpoints that each server manages, metadata servers, and I/O servers. Using the user-provided information, it creates a global PVFS2 server configuration file (e.g., `fs.conf`). In general, PVFS2 servers are deployed using this global configuration file shared by all PVFS2 servers. Figure 4 shows an example of the front-end of the execution engine where the user-provided information is taken to run the application. In this example, bb18 is configured as a metadata server and bb05, bb06, bb07, and bb08 as I/O servers. The 512 MPI processes specified in the

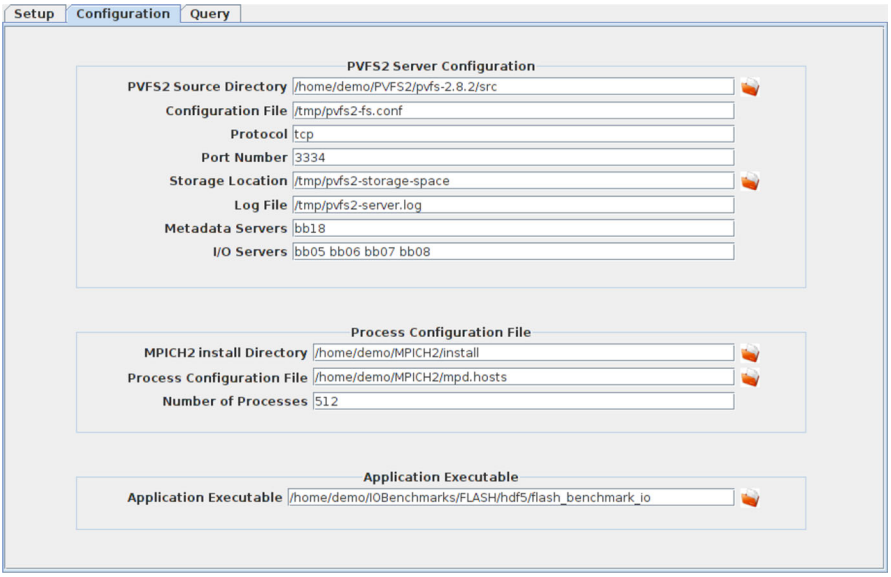

Fig. 4 The front-end (configuration view) of the execution engine

mpd.hosts file that has node information would be launched by mpiexec to run the executable flash_benchmark_io.

We want to emphasize that the instrumented I/O stack is separately built from the non-instrumented one. Therefore, the application can run either on the instrumented I/O stack or on the non-instrumented (original) I/O stack by setting `LD_LIBRARY_PATH`.

4.3 Data process engine

After running the application with the user-provided information in the execution engine, the data process engine collects all trace log files from each layer of the target I/O stack. Table 1 lists a representative set of high-level metrics that can be profiled

Table 1 Statistics that can be analyzed by IOPro

I/O latency experienced by each I/O call in each layer (MPI library, client, server, or disk) in I/O stack
Average I/O access latency in a given segment of the program
Throughput achieved by a given I/O read and write call
Disk power consumption incurred by each I/O call
Number of disk accesses made by each I/O call
Amount of time spent during inter-processor communication in executing a collective I/O call
Number of I/O nodes participating in each collective I/O

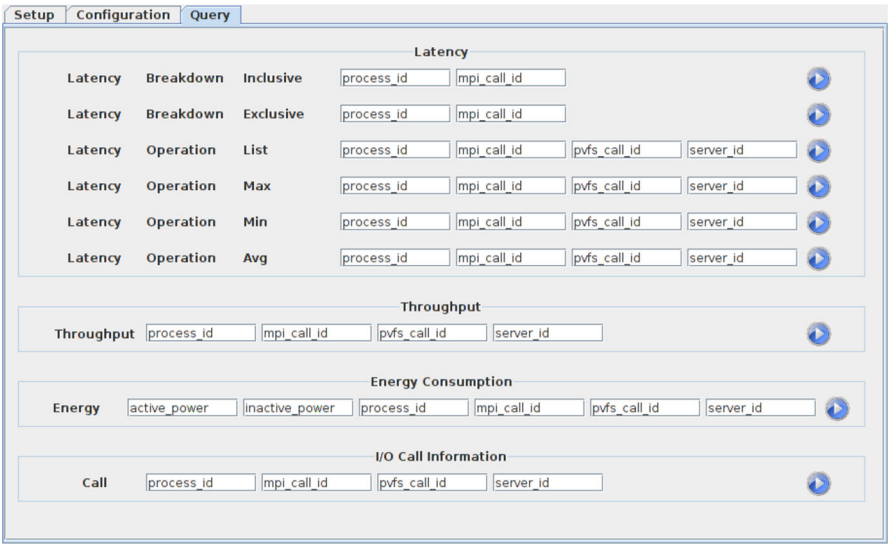

Fig. 5 The front-end (query analyzer view) of data processing engine

and visualized by our prototype of IOPro. Based on the user's query taken in the front-end of the data process engine (Fig. 5), the data process engine calculates the statistics using the collected trace log files, returns the performance metrics, and visualizes it for further investigation. The detailed query specification is discussed later in Sect. 5.4.

5 Technical details

In this section, we go over the code instrumentation component of IOPro and the use of probes, the configuration of the servers, the role of the query analyzer, and various sample queries.

5.1 Code instrumentation

Using the information provided in the setup view (Fig. 3), IOPro automatically patches PVFS, the MPI library, and the high-level I/O libraries, such as PnetCDF and HDF5, as a preparation for code instrumentation. Using the probe library that maintains probe template codes for PVFS, MPI, PnetCDF, and HDF5, the instrumentation engine generates actual probes that contain the trace log file location. In this context, a *probe* is a piece of code inserted into the I/O software stack software to help collect the required statistics. IOPro then creates a probe location file from a provided template file (as in Listing 1) that specifies the appropriate location in the MPI library and PVFS where the probes should be inserted. The syntax given in Listing 1 is for illustrative purposes and is based on an initial prototype of IOPro that is currently under development. Probe selector, a sub-component of the instrumentation engine, parses the probe location file and extracts the location information for the probe code to be inserted. Using

Listing 1 A sample template file specifying probe locations. In this template file, five different probes are specified for the application, MPI I/O library, PVFS client, PVFS server, and disk layers, including probe names and location information to be inserted as well as file names to be instrumented in the I/O stack.

```
[−1;App;common;Lib/APP/APP-common-probe1;−n 0]
[−1;App;write;Lib/APP/APP-write-probe;−l MPI_File_write(;before]
[−1;App;write;Lib/APP/APP-write-probe;−l MPI_File_write_at(;before]
[−1;App;write;Lib/APP/APP-write-probe;−l MPI_File_write_at_all(;before]
[−1;App;write;Lib/APP/APP-write-probe;−l MPI_File_write_all(;before]
[−1;App;read;Lib/APP/APP-read-probe;−l MPI_File_read(;before]
[−1;App;read;Lib/APP/APP-read-probe;−l MPI_File_read_at(;before]
[−1;App;read;Lib/APP/APP-read-probe;−l MPI_File_read_at_all(;before]
[−1;App;read;Lib/APP/APP-read-probe;−l MPI_File_read_all(;before]
[0;MPI;latency;Lib/MPI-IO/MPI-start-probe;−n 73;src/mpi/romio/mpi-io/read.c]
[0;MPI;latency;Lib/MPI-IO/MPI-end-probe;−n 164;src/mpi/romio/mpi-io/read.c]
[0;MPI;latency;Lib/MPI-IO/MPI-start-probe;−n 75;src/mpi/romio/mpi-io/read_all.c]
[0;MPI;latency;Lib/MPI-IO/MPI-end-probe;−n 120;src/mpi/romio/mpi-io/read_all.c]
[0;MPI;read;Lib/MPI-IO/MPI-rw-probe1;−n 158;src/mpi/romio/adio/common/ad_read_col 1.c]
[0;MPI;read;Lib/MPI-IO/MPI-rw-probe2;−n 730;src/mpi/romio/adio/common/ad_read_col 1.c]
[0;MPI;latency;Lib/MPI-IO/MPI-start-probe;−n 73;src/mpi/romio/mpi-io/write.c]
[0;MPI;latency;Lib/MPI-IO/MPI-end-probe;−n 171;src/mpi/romio/mpi-io/write.c]
[0;MPI;latency;Lib/MPI-IO/MPI-start-probe;−n 75;src/mpi/romio/mpi-io/write_all.c]
[0;MPI;write;Lib/MPI-IO/MPI-rw-probe2;−n 526;src/mpi/romio/adio/common/ad_write_c oll.c]
[0;MPI;write;Lib/MPI-IO/MPI-rw-probe1;−n 679;src/mpi/romio/adio/common/ad_write_c oll.c]
[1;Client;latency;Lib/PVFS/client-start-probe;−n 377;src/client/sysint/sys-io.sm]
[1;Client;latency;Lib/PVFS/client-end-probe;−n 402;src/client/sysint/sys-io.sm]
[2;Server;latency;Lib/PVFS/server-start-probe;−n 152;src/server/io.sm]
[2;Server;latency;Lib/PVFS/server-end-probe;−n 5270;src/io/job/job.c]
[3;Disk;latency;Lib/PVFS/disk-read-start-probe;−n 190;src/io/flow/flowproto-bmi-t
 rove/flowproto-multiqueue.c]
[3;Disk;latency;Lib/PVFS/disk-read-end-probe;−n 1010;src/io/flow/flowproto-bmi-tr
 ove/flowproto-multiqueue.c]
[3;Disk;latency;Lib/PVFS/disk-write-start-probe;−n 1343;src/io/flow/flowproto-bmi
 -trove/flowproto-multiqueue.c]
[3;Disk;latency;Lib/PVFS/disk-write-end-probe1;−n 1514;src/io/flow/flowproto-bmi-
 trove/flowproto-multiqueue.c]
```

the extracted probe location information, the probe inserter automatically inserts the appropriate probes into the proper locations in the I/O stack.

Figure 6 illustrates how the instrumentation engine works. In this figure, *IOCallIDs* is a small array that contains information of each layer such as the MPI I/O call ID, PVFS client ID, PVFS server ID, disk operation ID, I/O type, and the start timestamp and the end timestamp of each layer. When IOCallIDs are passed from the upper layer to the layers below, the inserted probes extract the information from them and generate the trace log files with latency statistics at the boundary of each layer.

Note that a high-level MPI I/O call can be fragmented into multiple small subcalls. For example, in two-phase I/O [6], which consists of an I/O phase and a communication phase, tracing an I/O call across the boundaries of the layers in the I/O stack is not trivial. In our implementation, each call has a unique *identification number* in the current layer and passes it to the layers below. This helps us associate the high-level call with its subcalls in a hierarchical fashion. It also helps analyze trace log data

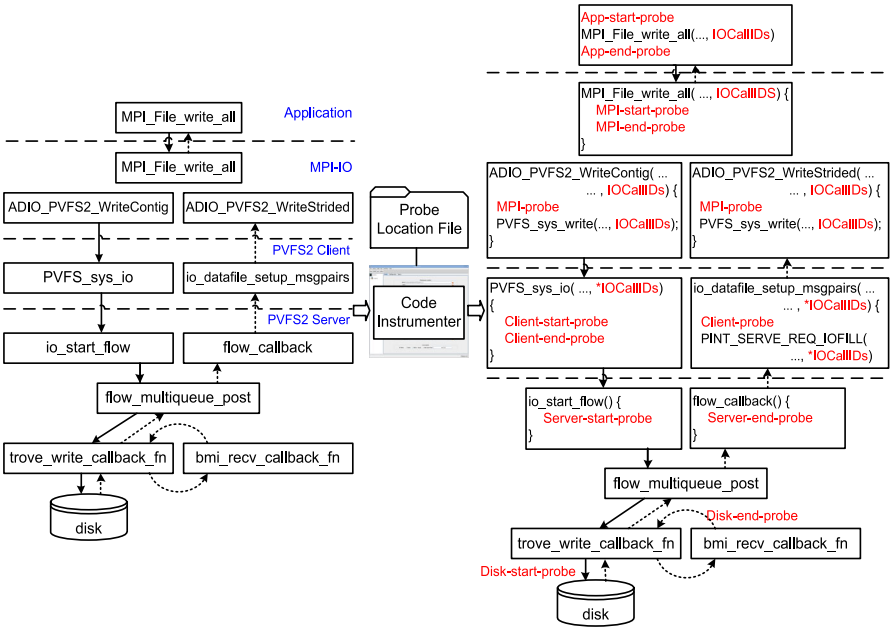

Fig. 6 Illustration showing how probes are inserted into the different layers of the I/O stack components by the instrumentation engine. *Left* I/O call flows when `MPI_File_write_all()` is issued. *Right* the instrumented I/O stack

by combining the statistics that come from different layers in a systematic way (for example, all the variables that hold latency information at different layers are associated with one another using these IDs).

In the PVFS server, a unique data structure, called *flow_descriptor*, maintains all the information to perform requested I/O operations from the PVFS clients. This structure is used by our tool. In Fig. 6, for example, the *Server-start-probe* inserted into the PVFS server layer extracts the necessary information passed from the PVFS client and packs it into the flow_descriptor. Since the flow_descriptor is passed to the entire PVFS server, the probes inserted in the server can extract the necessary information from it and manipulate the statistics to trace I/O calls without much difficulty.

5.2 Configuration of the execution environment

After PVFS is installed, the user specifies which nodes in the cluster will serve as metadata servers and I/O nodes. The user also determines how many MPI processes will be used to run the application. Unfortunately, manually configuring the PVFS servers and running the parallel application on them can be very tedious and challenging. Instead of manual configuration, our tool provides a simple mechanism to specify configuration of running environments (see Fig. 4). It takes the configuration metrics for the servers such as metadata server(s) and I/O server(s) as well as a filename storing this configuration, protocol, port number, storage location, and a log filename for each

Table 2 Accepted query format

Latency	Breakdown	Inclusive	***process_id***	***mpi_call_id***		
Latency	Breakdown	Exclusive	***process_id***	***mpi_call_id***		
Latency	Operation	List	***process_id***	***mpi_call_id***	***pvfs_call_id***	***server_id***
Latency	Operation	Max	***process_id***	***mpi_call_id***	***pvfs_call_id***	***server_id***
Latency	Operation	Min	***process_id***	***mpi_call_id***	***pvfs_call_id***	***server_id***
Latency	Operation	Avg	***process_id***	***mpi_call_id***	***pvfs_call_id***	***server_id***
Thru.	***process_id***	***mpi_call_id***	***pvfs_call_id***	***server_id***		
Energy	***active_p***	***inactive_p***	***process_id***	***mpi_call_id***	***pvfs_call_id***	***server_id***
Call	***process_id***	***mpi_call_id***	***pvfs_call_id***	***server_id***		

Bold italic indicates user input

server. It also takes a filename that specifies the host machine(s) from which the MPI job launcher, *mpiexec*, is launched and the number of processes (or clients) to be used for running the given application.

This simple configuration method also provides us with the flexibility of running a single application and/or multiple applications using different configuration options *without* recompilation of the instrumented I/O stack. For example, we can easily run the application program(s) on the instrumented I/O stack with different combinations of configurations such as (1) running the same application but varying the number of metadata server(s), I/O server(s), or PVFS clients; (2) running different applications on the same configuration; (3) different mixes of the previous two; and (4) running multiple applications on the same configuration and/or a different one.

5.3 Computation methodology

After running the application program, the data process engine collects all trace log files from each layer of the I/O stack. Based on the user's queries, it then processes the trace log and returns the corresponding statistics. As shown in Fig. 5, our current implementation provides functionalities to analyze latency, throughput, estimated energy consumption, and the number of calls issued from clients to servers. We want to emphasize however that, if desired, IOPro can be easily extended to accommodate additional/new statistics. Table 2 shows the input query formats accepted by the current implementation of our tool. The detailed description of our query will be given in Sect. 5.4.

Figure 7 illustrates the computation of latency and throughput. For each I/O call, the I/O latency value computed at each layer is the *maximum* value of the I/O latencies from the layers below it.

$$Latency_i = Max(Latency_{i-1}A, Latency_{i-1}B, Latency_{i-1}C) \quad (1)$$

However, the computation of I/O throughput in Fig. 7b is additive; in other words, I/O throughput at any layer is computed by summing the sizes of data coming from the layers below below it.

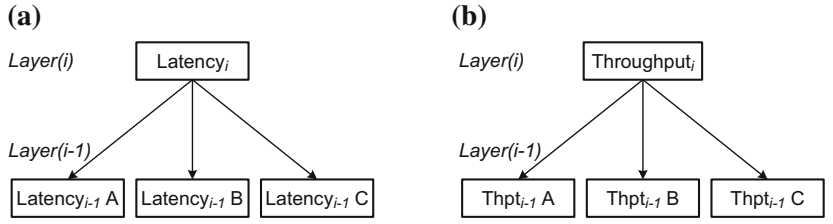

Fig. 7 Computation of latency and throughput. I/O latency computed at each layer is equal to the maximum value of the I/O latencies obtained from the layers below it. In contrast, I/O throughput is the sum of I/O throughput coming from the layers below. **a** Computation of I/O latency, **b** computation of I/O throughput

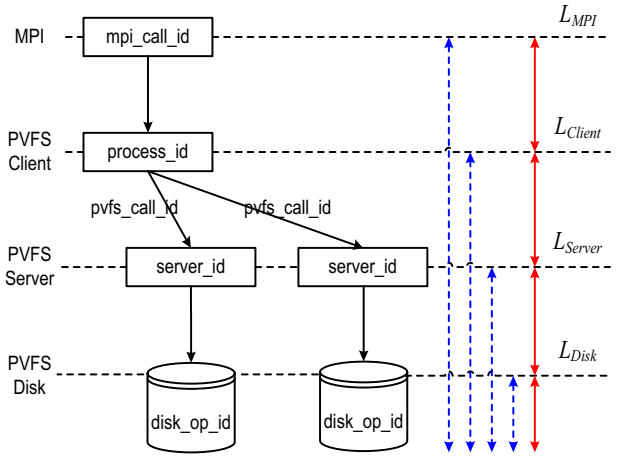

Fig. 8 Inclusive (*dotted arrow*) and exclusive (*solid arrow*) latency computations

$$Throughput_i = \sum(Thpt_{i-1}A, Thpt_{i-1}B, Thpt_{i-1}C) \tag{2}$$

To compute (estimate) the energy consumption for I/O call, we employ the power model described in [46].

In our work, *inclusive* latency means the time spent in the current layer, which includes the latency in the layers below. *Exclusive* latency is the time spent in the current layer and excludes the sublayers. That is, it can be calculated from inclusive latency by subtracting the sublayer latency from the current layer. Figure 8 demonstrates how the inclusive and exclusive latencies are computed (the dotted arrows denote inclusive latency, and the solid arrows indicate exclusive latency). The figure also shows the employed tracing mechanism, which identifies and distinguishes I/O calls at each layer. Each layer generates a unique ID such as *process_id*, *mpi_call_id*, *pvfs_call_id*, and *server_id* when an I/O call is passed. This unique number is cumulatively carried down to the sublayers. All information for the I/O calls passed through the entire I/O stack is stored in the last layer. By matching and identifying these IDs, one can easily relate the high-level MPI I/O call to the subcalls.

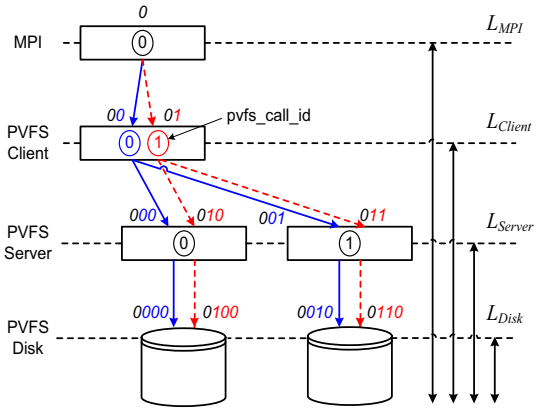

Fig. 9 Computation of inclusive latency

Figure 9 shows the computation of inclusive latency in more detail. When, for instance, a collective I/O call is issued, this I/O call can be fragmented into multiple I/O calls in the MPI library if the size of requested I/O is larger than that of the buffer in the MPI library. For example, in the figure, mpi_call_id *0* is fragmented into two pvfs_call_id's *0* and *1*. In the PVFS client layer, each split I/O call has its own ID, *00* and *01* for the mpi_call_id 0, respectively. When these calls reach servers 0 and 1, the cumulative trace information is *000* and *001* for cumulative ID 00 (blue line), and *010* and *011* for ID 01 (red one). This relationship is maintained until the end of the I/O stack is reached. Therefore, for mpi_call_id 0, the inclusive latency computed at the PVFS client layer is

$$Latency_{client} = \sum(L_{00}, L_{01}), \tag{3}$$

and the inclusive latency at the PVFS server layer is

$$Latency_{server} = \sum(Max(L_{000}, L_{001}), Max(L_{010}, L_{011})), \tag{4}$$

where L denotes latency. Exclusive latency, on the other hands, can be calculated as shown in Fig. 8.

5.4 Query model

As listed in Table 2, the current implementation of our tool provides four user metrics to be analyzed: latency, throughput, energy, and call information. Below, we discuss the details of our queries (metrics in square brackets are the user-provided input).

– Latency Breakdown Inclusive [*process_id*] [*mpi_call_id*]
This query returns the inclusive latency information given process_id and mpi_call_id. For example, the query 'Latency Breakdown Inclusive [0–1] [1–10]' returns the inclusive latency for the mpi_call_id 1 to 10 issued from the process 0 and 1 to all servers in breakdown fashion, as described in Sect. 5.3. This is also applied to compute exclusive latency.

- Latency Operation List [*process_id*] [*mpi_call_id*] [*pvfs_call_id*] [*server_id*]
  This returns all latency information, listing detailed latency statistics for all matching process–server combinations. For example, a query such as Latency Operation List [0–4] [1–10] [–] [1–3] returns all combinations for mpi_call_id 1 to 10 issued from the process 0 to 4 to the server 1 to 3. In this case, all possible combinations are 15. In the parameter pvfs_call_id, "-" means all. By default, pvfs_call_id is set to "-" for simplicity since it is implicitly fragmented depending on the size of the I/O request.
- Latency Operation Max/Min [*process_id*] [*mpi_call_id*] [*pvfs_call_id*] [*server_id*]
  This is similar to the list latency format except that it returns the maximum/minimum latency. For example, Latency Operation Max [0–4] [1–10] [–] [1–3] returns the maximum latency for mpi_call_id 1 to 10 issued from processes 0 to 4 to servers 1 to 3. Unlike list latency, this shows only the maximum latency among the given servers and the corresponding server number. Note that this query provides the latency statistics from the process's and server's points of view. More specifically, from the process's point of view, we can easily identify in which server a given mpi_call_id experiences the maximum latency. From the server's point of view, we can identify the process that has the most latency in that server. Also, unlike inclusive/exclusive latency, it presents detailed latency, not in breakdown fashion. For example, if an mpi_call_id 0 is split into ten subcalls, it returns the maximum latency among all ten individual subcalls.
- Latency Operation Avg [*process_id*] [*mpi_call_id*] [*pvfs_call_id*] [*server_id*]
  This returns the average latency given the ranges of processes, MPI I/O calls, and servers for each MPI I/O call.
- Throughput [*process_id*] [*mpi_call_id*] [*pvfs_call_id*] [*server_id*]
  This calculates disk throughput in each PVFS server for each mpi_call_id from the process's and server's points of view.
- Energy [*active_power*] [*inactive_power*] [*process_id*] [*mpi_call_id*] [*pvfs_call_id*] [*server_id*]
  This calculates the estimated energy consumption in server class disks [46]. It also plots both the process's and the server's views. Here, *active_power* is the amount of power consumption (in watts) when the status of disk is active; *inactive_power* is the power consumption when the disk is not active.
- Call [*process_id*] [*mpi_call_id*] [*pvfs_call_id*] [*server_id*]
  This returns statistics about the number of issued I/O calls from processes to servers. Using this query, one can detect which I/O server is suffering the most from the I/O requests.

6 Evaluation results

Most of the GUI components of IOPro have been implemented in Java and the JFreeChart library. Our implementation was evaluated on the Breadboard cluster at Argonne National Laboratory. In our experiments, we built an I/O stack with pvfs-2.8.2, mpich2-1.2.1p1, pnetcdf-1.2.0, and hdf-1.8.5. Then, IOPro automatically generated an instrumented version of this I/O stack using IOPro. Note that we opted to use

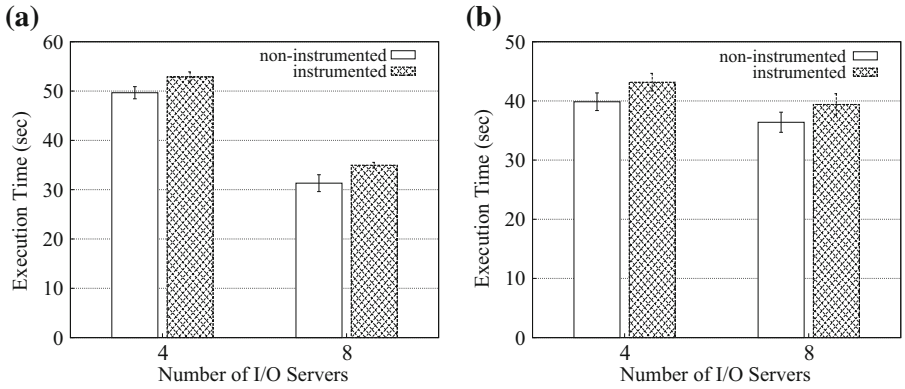

Fig. 10 Average execution time comparison. Two benchmarks run on 512 MPI processes with 1 metadata server and various number of I/O servers. It can be seen that, in both benchmarks, the overheads caused by our implementation are about 8.5 %. **a** FLASH I/O, **b** S3D I/O

Table 3 Overhead comparison

Benchmark	I/O server	Non-instrumented (s)	Instrumented (s)	Overhead (%)
FLASH I/O	4	49.66	52.85	6.4
	8	31.32	34.96	11
S3D I/O	4	39.86	43.16	8.3
	8	36.39	39.40	8.3

PVFS, the user-level parallel file system, so that we could easily implement a tracing and profiling mechanism without kernel modifications.

To evaluate the overhead caused by our implementation, we measured the average execution time after 20 iterations, running two I/O intensive benchmarks, S3D I/O and FLASH I/O. In each run, we dropped cache both in the servers and in compute nodes to minimize the effect on cache. Figure 10 compares the average execution time of two benchmarks running with 512 MPI processes and various number of I/O servers on a non-instrumented I/O stack and an instrumented one. The result shows that the overhead from all combinations is approximately 8.5 %, on average. Table 3 presents the detailed statistics.

To demonstrate the capabilities of IOPro, we next present the detailed results with two benchmarks.

6.1 FLASH I/O

FLASH I/O benchmark is the I/O kernel of the FLASH application [45], a block-structured adaptive mesh hydrodynamics code that solves fully compressible, reactive hydrodynamic equations, developed for studying nuclear flashes on neutron stars and white dwarfs. The computational domain is divided into small blocks that are distributed across different MPI processes. The FLASH block is a three-dimensional

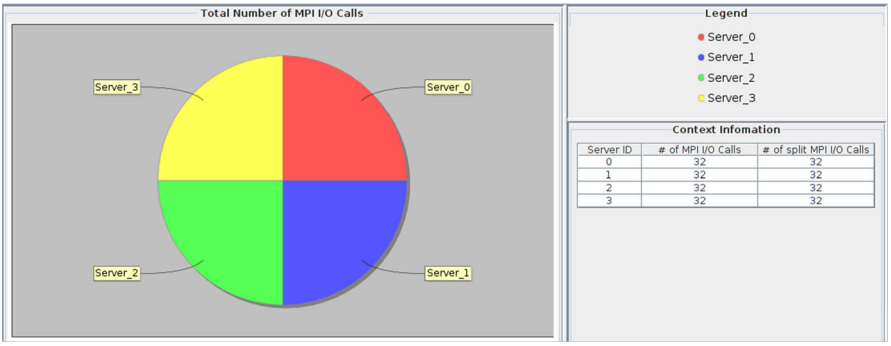

Fig. 11 Number of I/O calls issued to all servers from one process using the HDF5 interface. We see that each server receives the same number of calls from that process

array and there are 80 blocks on each MPI process. Each block contains inner blocks with additional four element of guard cells to hold the state variables of the neighboring blocks. The inner block surrounded by guard cells has 24 data array variables, e.g., density, velocities, energy, and pressure. Every process writes these blocks into a checkpoint file using 24 collective I/Os, in a manner that the checkpoint file appears as the data for variable 0 up to variable 23. FLASH I/O generates one checkpoint file and two visualization files that contain centered and corner data. FLASH I/O works with both the PnetCDF and HDF5 interfaces to save data and metadata in each high-level I/O format.

In our evaluation, we ran the FLASH I/O benchmark on 1 metadata server, 4 I/O servers, and 512 MPI processes using the HDF5 interface. We configured a $8 \times 8 \times 8$ block size in X–Y–Z dimensions. In this experiment, FLASH I/O produces a 3.8-GB checkpoint file and two visualization files (329 and 466.8 MB, respectively). From Fig. 11, we see that 32 collective I/O calls without fragmentation are evenly issued to all servers from one process. The checkpoint file is generated by the first 24 I/O calls and two visualization files are created by the following 4 I/O calls, respectively.

Figure 12 illustrates the inclusive latency of the FLASH I/O benchmark from Process_0 to process 30 among all 512 processes. The total time in the figure presents the global time spent in each layer to run the application program, using different color legends. The context information gives the detailed latency statistics in the MPI I/O library, PVFS client, PVFS server, and server disk layers for each process. We observe that the latency in the MPI I/O library and the PVFS layer is unevenly distributed among the processes. For example, the most time spent in the MPI library for the I/O requests is approximately 11.51 s, in process 16, and the least is about 8.97 s in process 1. We observe that the overhead in the MPI library is relatively small. In MPI-IO, the default collective buffering scheme is set to *automatic*, that is, MPI-IO uses heuristics to determine whether it enables the optimization. Since FLASH I/O accesses noncontiguous data, rarely exploiting data exchanges and optimization for collective I/O, MPI-IO disables collective buffering and automatically converts collective I/O requests to independent I/O requests. We find that the latency in the MPI I/O library increases about 14 % because of the communication overhead when force-

PID	MPI	Client	Sever	Disk
0	11.028	8.982	7.497	7.491
1	8.969	7.061	6.126	6.12
2	11.093	9.055	7.999	7.993
3	11.057	9.175	7.65	7.644
4	9.813	7.894	6.395	6.387
5	10.078	8.282	7.01	7.005
6	10.565	8.731	6.426	6.42
7	11.404	9.64	8.206	8.199
8	9.499	7.585	6.646	6.64
9	9.327	7.55	6.001	5.941
10	10.854	8.927	8.017	8.008
11	11.038	9.243	7.805	7.8
12	10.374	8.523	7.393	7.388
13	10.026	8.238	7.114	7.107
14	10.365	8.607	7.317	7.311
15	9.813	8.129	6.473	6.467
16	11.506	9.514	8.115	8.109
17	10.043	8.316	7.288	7.281
18	10.917	9.097	7.959	7.953
19	10.954	9.18	7.414	7.408
20	10.07	8.187	6.776	6.769
21	10.468	8.768	7.481	7.475
22	10.484	8.619	6.973	6.966
23	10.464	8.658	7.128	7.123
24	10.412	8.579	7.512	7.506
25	10.218	8.566	7.128	7.122
26	11.158	9.421	8.17	8.164
27	9.518	7.931	6.174	6.168
28	10.375	8.561	7.538	7.533
29	10.712	9.033	7.892	7.76

Fig. 12 Inclusive latency values for the FLASH I/O benchmark labelfig

fully enabling collective I/O. Since all the joined processes exploit independent I/O for running the application, the completion time for each substantially differs.

Figure 13 compares the maximum and the minimum latency for all I/O requests issued to all servers from Process_0 to Process_30. Unlike in Fig. 13a, we observe a bigger latency gap between the client layer and the server (the green portion) in Fig. 13b. We also notice the latency difference spent in the server and the disk between Fig. 13a, b. If the data lie on the data block, but is smaller and not fit into it, it take less time to write the smaller portion of the data. Figure 14 plots more detailed latency statistics for mpi_call_id 18 through 23. The difference between the maximum and minimum latency of process 16 (in Fig. 13) is caused by the latency in the server from those I/O calls, as shown in Fig. 14. Note that the number on the server legend (the green bar) in Fig. 14 is the server ID. The I/O request (18-0) from the process, for example, spends the maximum amount of time in server 0 and has minimum latency in server 3 even if it stripes 64 KB data over all servers.

Disk throughput from mpi_call_id 0 through 23 from process 16 is plotted in Fig. 15.

Here, we observe the I/O characteristics of FLASH I/O. Although FLASH I/O issues collective I/O requests to write checkpoint files, MPI-IO disables them and automatically converts them to independent I/O requests because the data are noncontiguous. We also notice that collective buffering rather degrades the performance of the FLASH I/O. Based on this observation, the optimized code can be implemented in a way to exploit the potential benefits of collective I/O. As seen in Fig. 14, the latencies of the I/O calls in the specific servers are higher than the others. Therefore, the application and I/O stack can be tuned to reduce those variances.

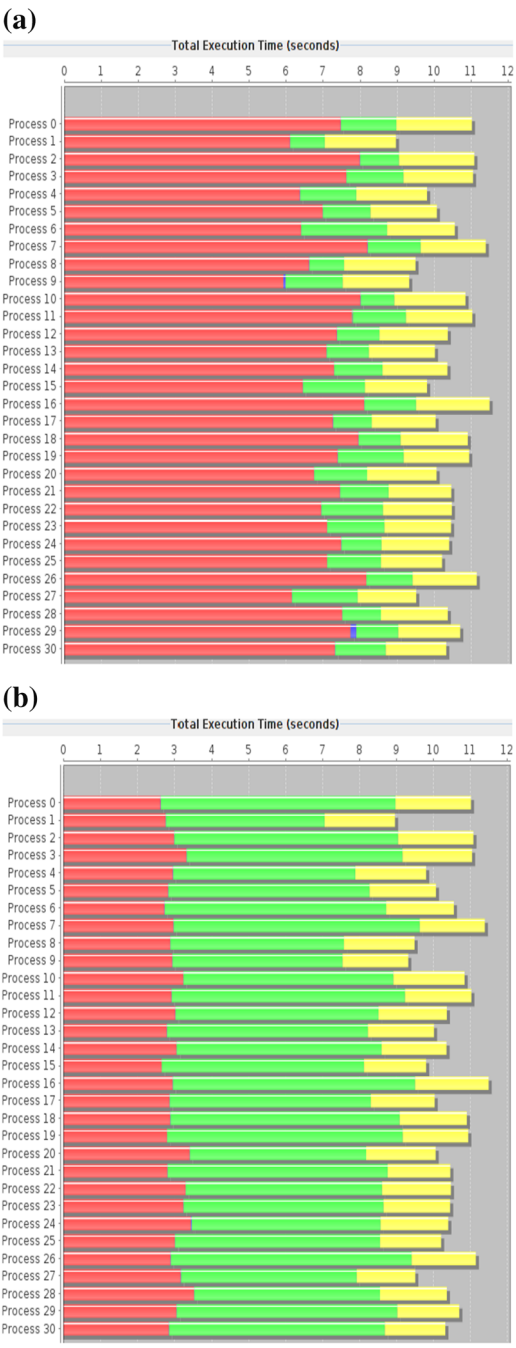

Fig. 13 Total maximum and minimum latency from all processes. In both figures, the time spent in the PVFS client layer is the same, but the time spent in the PVFS server and disk is different. **a** Maximum latency of FLASH I/O, **b** minimum latency of FLASH I/O

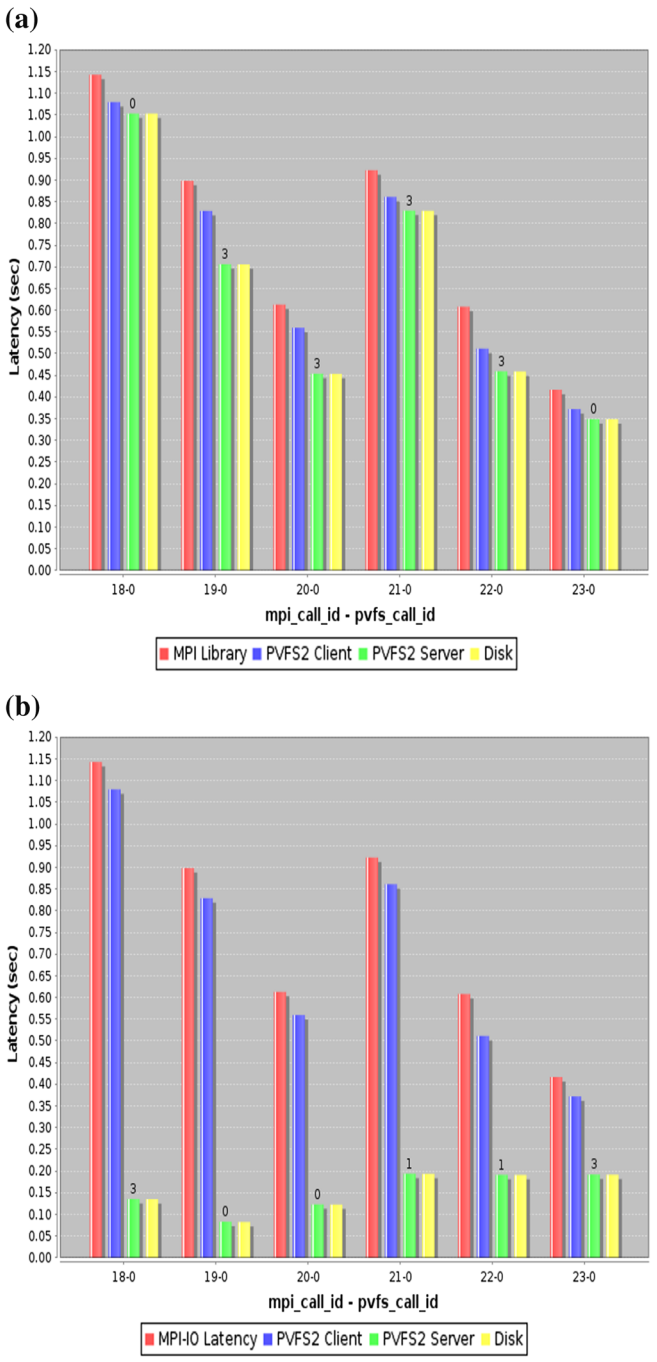

Fig. 14 Maximum and minimum latency from the perspective of Process_16 for mpi_call_id ranging from 18 to 23. **a** Process 16's maximum latency, **b** process 16's minimum latency

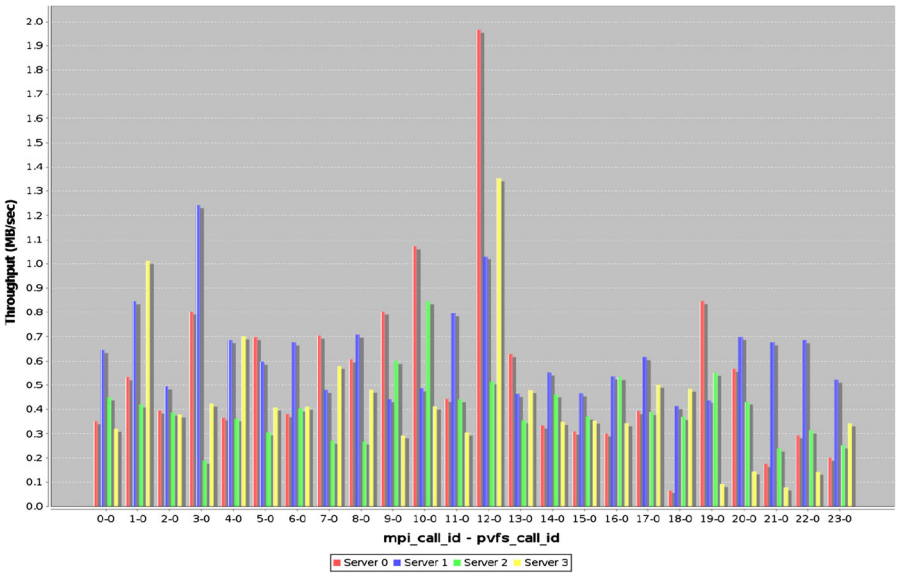

Fig. 15 Disk throughput for mpi_call_id 0 to 23 to write a checkpoint file from process 16 to 4 servers

6.2 S3D I/O

The S3D I/O benchmark is a parallel turbulent combustion application, named S3D [47], developed at Sandia National Laboratories. Using a direct numerical simulation, S3D solves the fully compressible Navier-Stokes, total energy, species, and mass continuity equations coupled with detailed chemistry. A checkpoint is performed at regular intervals; its data consist primarily of the solved variables in 8-byte, three-dimensional arrays. This checkpoint data can be used to obtain several physical quantities of interest. Therefore, most of the checkpoint data is maintained for later use. At each checkpoint, four global arrays—representing the variables of mass, velocity, pressure, and temperature—are written to files.

Among those four arrays, pressure and temperature are three-dimensional arrays while mass and velocity are four dimensional. All four arrays share the same size for the lowest three spatial dimensions X, Y, and Z and are partitioned among the MPI processes along with X–Y–Z dimensions. For the three-dimensional arrays, the subarray of each process is mapped to the global array in block partitioning of X–Y–Z dimensions. For the four-dimensional arrays, the lowest X–Y–Z dimensions are partitioned as same as the three-dimensional arrays, but the fourth dimension is not partitioned. For the arrays of mass and velocity, the length of fourth dimension is 11 and 3, respectively.

S3D I/O supports MPI-IO, PnetCDF, and HDF5 interfaces. In our evaluation, we configured 1 metadata server and 8 I/O servers and ran S3D I/O on 512 MPI processes with the PnetCDF interface. We maintain the block size of the partitioned X–Y–Z dimensions as $400 \times 200 \times 200$ in each process. With this configuration, S3D

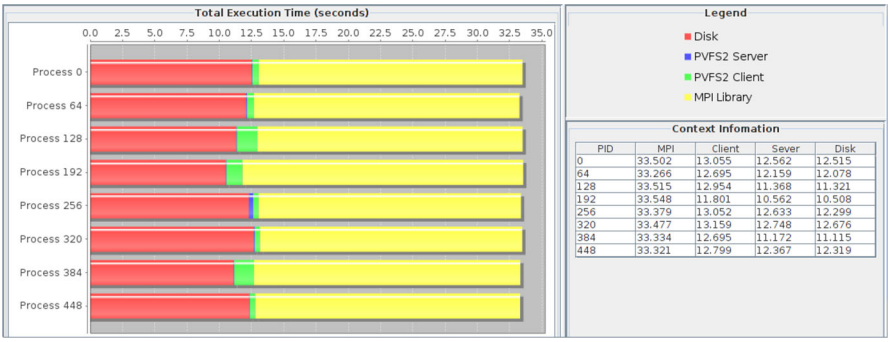

Fig. 16 Inclusive latency of the S3D I/O benchmark

I/O produces three checkpoint files, 1.9 GB each. The average execution time on the instrument I/O stack with 8 I/O servers is presented in Fig. 10b.

Figure 16 shows the inclusive latency generated by the query analyzer. For a collective write in S3D I/O, a subset of MPI tasks (called *aggregator*) in each compute node communicates with other processes to exchange data and writes a large chunk of data into a temporary buffer. After that, the aggregator in each node ships the I/O request to the destination I/O servers. In our configuration, we have 8 aggregator processes to perform the actual I/O operation (Fig. 16). We observe that each process spends about 33.4 s (on average) in the MPI I/O library and that most of the time spent in the server layer is for disk operations. We also notice a latency gap between the MPI I/O library and the PVFS client layer (the yellow portion in Fig. 16). In S3D I/O, all the joined processes heavily exchange the data for optimization, such as two-phase I/O [6] and data sieving [5]. This optimization and synchronization result in the overhead in the MPI library.

Figure 17a, b plots inclusive latencies from the perspective of Process_320 and Process_192 that have a maximum and minimum latency in the disk, respectively. In both plots, the time spent in the MPI library for mpi_call_id *0*, *4*, and *8* is relatively longer than that for the other I/O calls. In general, S3D I/O produces three checkpoint files using 12 collective I/O calls, and these files are generated by call ids 0–3 (first file), 4–7 (second file), and 8–11 (third file). The first I/O call (0, 4, and 8) in each checkpoint file initially opens the checkpoint file and writes the mass variable to it. Recall that, among the four arrays of mass, velocity, pressure, and temperature, mass and velocity are four-dimensional arrays whose length of the fourth dimension is 11 and 3, respectively. Since the mass array is the largest, it takes longer to be written into each checkpoint file. In the same reason, the last I/O call (3, 7, and 11) to write velocity takes relatively longer time than to write pressure and temperature. In Fig. 16, the total time difference of disk operation between Process_320 (12.68 s) and Process_192 (10.51 s) is mainly caused by mpi_call_id *4* (3.31 vs. 1.99) and mpi_call_id *8* (4.14 vs. 3.03) in Fig. 17a, b.

Generated using the max query format, Fig. 18 presents detailed I/O information ranging from from MPI_call_id 0 to 3 that create the first checkpoint file. Here, mpi_call_id 0 spends considerable time in the MPI I/O library to open the checkpoint

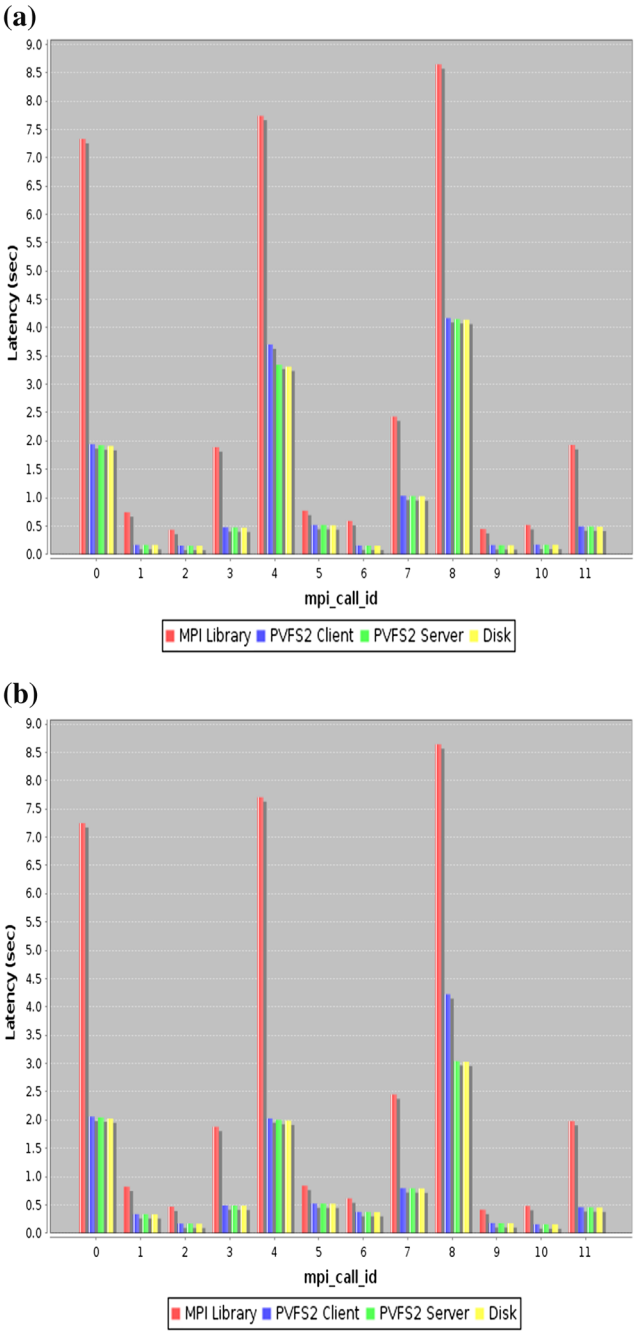

Fig. 17 Inclusive latency from the perspective of Process_320 and Process_192. The total time difference for disk operation between them is mainly caused by mpi_call_id 4 (3.31 vs. 1.99 s) and 8 (4.14 vs. 3.03 s) in **a**, **b**. **a** Process 320's inclusive latency, **b** process 192's inclusive latency

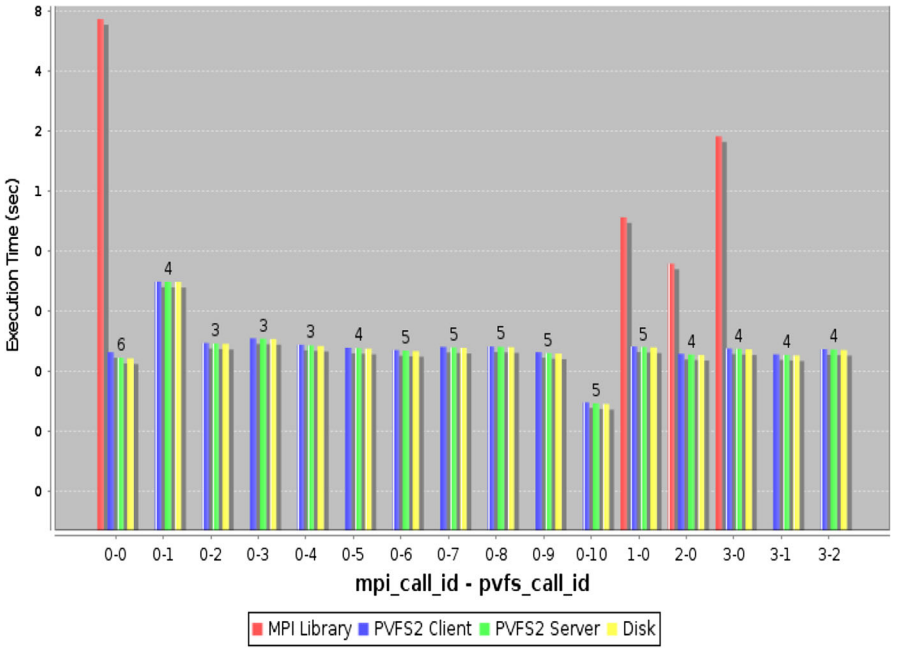

Fig. 18 Detailed latency from mpi_call_id 0 to 3 when creating the first checkpoint file in Process_320. The *x* axis is a pair of (*mpi_call_id - pvfs_call_id*) and the *y* axis is the execution time in log scale. Here, mpi_call_id *0* and mpi_call_id *3* are fragmented into 11 subcalls (*pvfs_call_id*) and 3 in the MPI library, respectively. The number on the PVFS server (*green bar*) in the *figure* indicates the server ID where the I/O call has the maximum value

file and write data into it. Since the size of the requested I/O to write the mass array is bigger than the buffer in the MPI library, this I/O call is split into multiple subcalls. In a typical collective I/O, all processes communicate with one another and exchange access information among all processes and reorganize I/O requests for better performance. After this step, all participating processes issue the I/O requests but cannot send the next I/O requests until all finish their I/O requests. In Fig. 18, mpi_call_id 0 is fragmented into eleven subcalls from (*0–0*) to (*0–10*) when writing the mass array whose length of the fourth dimension is 11, and mpi_call_id 3 three subcalls (*3–0*), (*3–1*) and (*3–2*) to write velocity whose length of the fourth dimension is 3, respectively. The latency difference between the MPI library layer and the PVFS layer, in mpi_call_id 0, is caused by communications and data exchanges as well as by synchronizations among the split I/O requests.

Note that the inclusive latency is computed by summing all maximum values of the corresponding split calls from the I/O call in the given process(es). Further, the maximum latency shows more detailed information for the split calls, if any, such as individual maximum values and the server ID having a maximum among the server's given ranges. Therefore, the inclusive latency for mpi_call_id 0 is calculated by adding the maximum values of the split calls for this I/O call in Fig. 18. Figure 19 plots the disk throughput from the perspective of the server 0. Among 8 aggregator processes,

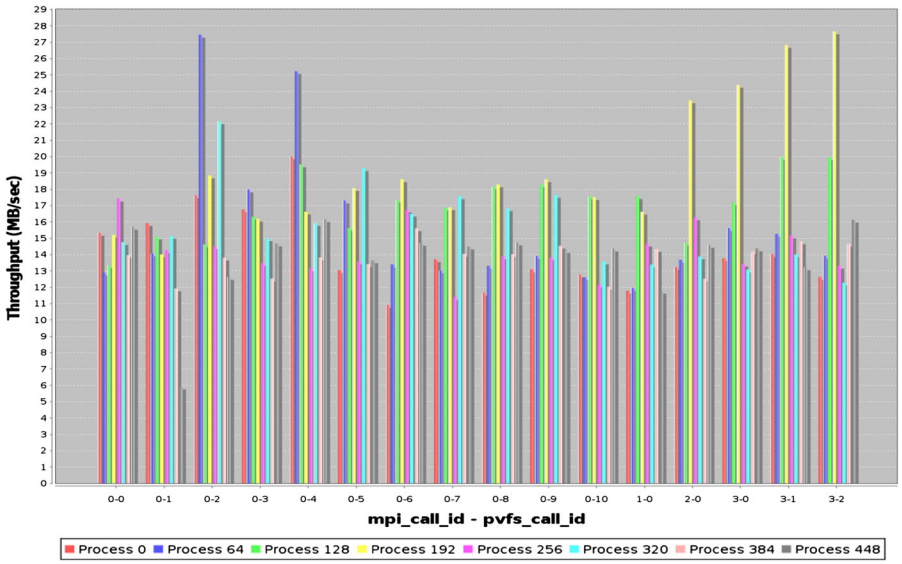

Fig. 19 Disk throughput to server 0 from all 8 aggregator processes for mpi_call_id 0 to 3 to create the first checkpoint file. The *x* axis is a pair of (*mpi_call_id - pvfs_call_id*). Here, call id pairs of (*2–0*), (*3–0*), (*3–1*), and (*3–2*) from the Process_192 have maximum disk throughput in the server 0

Process_192 has a maximum throughput (23.44 MB/s, 24.39, 26.83, and 27.67, respectively) for (2–0), (3–0), (3–1), and (3–2).

Unlike FLASH I/O, all the joined processes heavily exchange data to do optimization before sending I/O requests to the PVFS server in S3D I/O. In addition to optimization, communication and synchronization among the processes cause the overhead in the MPI library. Based on this understanding, scientists and application programmers can customize the existing code to reduce the overhead, specifically in mpi_call_id 0, 4, and 8 at the application level. Also, performance engineers may improve the performance in the MPI I/O library and disk operation at the system level.

6.3 Case study: pinpointing I/O interference in the concurrent execution of multiple applications

In HPC systems that share I/O system resources across processes, interference occurs when multiple applications access a storage resource, which in turn, causes substantial I/O performance degradation. To simulate this real case scenario and to profile detailed metrics in such a situation, first, we separately run two benchmarks, S3D I/O and a synthetic benchmark and measure the execution of each benchmark, as a baseline experiment. We run S3D I/O with the same configuration as in Sect. 6.2, using 512 MPI processes with 1 metadata server and 8 I/O servers. Here, the synthetic benchmark accesses data in row, column, and block fashion, and generates a 2-GB checkpoint file. At this time, we run the synthetic benchmark on 64 MPI processes, but it only stripes data over 1 I/O server, by setting MPI hints. After that, we run both benchmarks at the

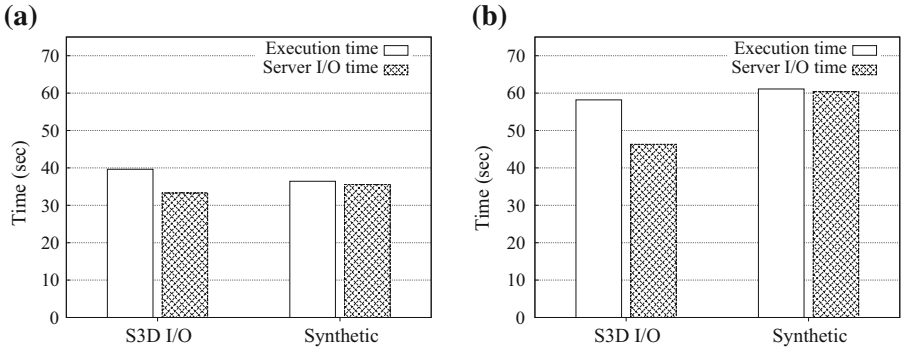

Fig. 20 Comparison of the execution time and the maximum I/O time in servers. In Fig. 20a, the execution time of S3D I/O and the synthetic benchmark is 39.63 s and 36.43, and, in Fig. 20b, 56.18 and 61.11, respectively. In both the experiments, the corresponding detailed I/O server time and striped data size are described in Tables 4 and 5. **a** Running individually, **b** running concurrently

Table 4 Baseline: S3D I/O detailed server I/O time and striped size of data

Server 0	Server 1	Server 2	Server 3	Server 4	Server 5	Server 6	Server 7
33.29 s	33.28	33.27	33.28	33.28	33.27	33.29	33.28
732 MB	732 MB	732 MB	732 MB	732 MB	732 MB	732 MB	732 MB

Table 5 Running S3D I/O with the synthetic benchmark in interference

Server 0	Server 1	Server 2	Server 3	Server 4	Server 5	Server 6	Server 7
46.31 s	46.22	46.26	46.27	46.24	46.21	46.15	46.20
2.73 GB	732 MB	732 MB	732 MB	732 MB	732 MB	732 MB	732 MB

same time so that I/O operations are interfered in each other. S3D I/O accesses 8 I/O servers to write data and the synthetic benchmark only stripes 1 I/O servers among 8 I/O servers. The compute node was not overlapped when running concurrently in this experiment.

Figure 20 compares the execution time and the I/O time in the server when each benchmark runs separately. In Fig. 20a, considered as the baseline, the execution time and the maximum server I/O time are 39.63 s and 33.29 in S3D I/O and 36.43 and 35.64 in the synthetic benchmark, respectively. Table 4 presents detailed metrics in S3D I/O. When I/O operations are interfered, as shown in Fig. 20b, the execution time increases upto 56.2 s in S3D I/O and 61.1 in the synthetic benchmark (see Table 5). Therefore, the overhead of the execution time caused by I/O interference are 42 and 68 %, respectively. In this scenario, the data from S3D I/O are evenly striped to 8 I/O servers, about 732 MB each. At the same time, the synthetic benchmark accesses one of the 8 I/O servers to write a 2-GB checkpoint file. This I/O server is a bottleneck and causes the degradation of the overall I/O performance in both applications.

Table 6 Running S3D I/O with the synthetic benchmark *without* interference

Server 0	Server 1	Server 2	Server 3	Server 4	Server 5	Server 6	Server 7
35.64 s	32.74	32.72	32.72	32.72	32.74	32.73	32.73
2 GB	837 MB	837 MB	837 MB	837 MB	837 MB	837 MB	837 MB

Based on this observation, a new I/O strategy can be adopted to prevent the interference. By setting up the MPI hints not to stripe the bottleneck I/O server, thus striping data only to 7 I/O servers, the execution time S3D I/O is 42.35 s, and the I/O time and striped data size in server are presented in Table 6. Note that the execution time of S3D I/O with 7 I/O servers increases about 7 % compared to the baseline. By ensuring that the two applications do not interfere with each other however, one can eliminate performance degradation.

7 Conclusions

Performance analysis and visualization is an important step in understanding I/O behavior, which is a result of complex interactions in the I/O stack. Performing manual code instrumentation is often difficult and extremely error prone. Even building the I/O stack and configuring the running environment for application benchmarks are not trivial because of the scale of the current HPC systems. Moreover, collecting and analyzing trace data from them is challenging and daunting task. To alleviate these difficulties, we have developed a parallel I/O profiling and visualizing framework, IOPro. Our tracing utility uses existing MPI I/O function calls and therefore adds minimum overhead to the execution time of applications. Our framework provides multiple metrics to analyze and investigate detailed I/O behavior, including latency, throughput, energy consumption, and call information. The results from these metrics contribute to evaluating and explaining the parallel I/O behavior.

We used two application benchmarks, S3D I/O and FLASH I/O, to evaluate our implementation of IOPro. Our experiments demonstrate different I/O behaviors in each application: S3D I/O exchanges data among the joined processes to do optimization and synchronization in the MPI library whereas FLASH I/O rarely does such optimization. Although both applications issue collective I/O requests to write the checkpoint files, the characteristics of I/O are different in each benchmark. Using the performance information depending on the I/O behavior, the application programs can be optimized to improve the performance. Also, customized instrumentation can be performed to get more detailed performance statistics in the I/O stack.

Lastly, we show that, when multiple applications interfere each other due to sharing I/O system resources, our framework can be used to profile detailed performance metrics, aid in understanding complex I/O behavior, and detect the issue that degrades the performance. Based on the gleaned information, the user can then employ an appropriate solution.

Acknowledgments We would like to thank the anonymous reviewers for their comments in improving this paper.

References

1. Ching A, Choudhary A, Coloma K, Liao W-K, Ross R, Gropp W (2003)Noncontiguous I/O accesses through MPI-IO. In: Cluster computing and the grid, (2003) Proceedings. CCGrid 2003. 3rd IEEE/ACM international symposium. IEEE, pp 104–111
2. Ma X, Winslett M, Lee J, Yu S (2003) Improving MPI-IO output performance with active buffering plus threads. In: Parallel and distributed processing symposium, 2003. Proceedings international. IEEE, p 10
3. Coloma K, Choudhary A, Liao W-K, Ward L, Russell E, Pundit N (2004) Scalable high-level caching for parallel I/O. In: Parallel and distributed processing symposium, 2004. Proceedings of 18th international. IEEE, p 96
4. Liao W, Ching A, Coloma K, Choudhary A (2007) An implementation and evaluation of client-side file caching for MPI-IO. In: IEEE international parallel and distributed processing symposium. IEEE, p 49
5. Thakur R, Gropp W, Lusk E (1998) Data sieving and collective I/O in ROMIO. In: Proceedings of the seventh symposium on the frontiers of massively parallel computation. Published by the IEEE Computer Society, pp 182–189
6. Del Rosario J, Bordawekar R, Choudhary A (1993) Improved parallel I/O via a two-phase run-time access strategy. ACM SIGARCH Comput Archit News 21(5):31–38
7. Shan H, Antypas K, Shalf J (2008) Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In: Proceedings of the 2008 ACM/IEEE conference on supercomputing. IEEE Press, p 42
8. Zhang X, Davis K, Jiang S (2010) IOrchestrator: improving the performance of multi-node I/O systems via inter-server coordination. In: Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis. IEEE Computer Society, pp 1–11
9. Lofstead J, Zheng F, Liu Q, Klasky S, Oldfield R, Kordenbrock T, Schwan K, Wolf M (2010) Managing variability in the IO performance of petascale storage systems. In: Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis. IEEE Computer Society, pp 1–12
10. Song H, Yin Y, Sun X-H, Thakur R, Lang S (2011) Server-side I/O coordination for parallel file systems. In: Proceedings of 2011 international conference for high performance computing, networking, storage and analysis. ACM, p 17
11. Zhang X, Davis K, Jiang S (2011) QoS support for end users of I/O-intensive applications using shared storage systems. In: Proceedings of 2011 international conference for high performance computing, networking, storage and analysis. ACM, p 18
12. Zhang X, Davis K, Jiang S (2012) Opportunistic data-driven execution of parallel programs for efficient I/O services. In: Parallel and distributed processing symposium (IPDPS) (2012) IEEE 26th international. IEEE, pp 330–341
13. Chen Y, Sun X-H, Thakur R, Song H, Jin H (2010) Improving parallel I/O performance with data layout awareness. In: Cluster computing (CLUSTER), 2010 IEEE international conference. IEEE, pp 302–311
14. Schwan P (2003) Lustre: building a file system for 1000-node clusters. In: Proceedings of the 2003 Linux symposium, vol 2003
15. Schmuck FB, Haskin RL (2002) GPFS: a shared-disk file system for large computing clusters. In: FAST, vol 2, p 19
16. Welch B, Unangst M, Abbasi Z, Gibson G, Mueller B, Small J, Zelenka J, Zhou B (2008) Scalable performance of the Panasas parallel file system. In: Proceedings of the 6th USENIX conference on file and storage technologies. USENIX Association, p 2
17. Carns P, Ligon III W, Ross R, Thakur R (2000) PVFS: a parallel file system for Linux clusters. In: Proceedings of the 4th annual Linux showcase & conference, vol 4. USENIX Association, pp 28–28
18. Thakur R, Gropp W, Lusk E (1999) On implementing MPI-IO portably and with high performance. In: Proceedings of the sixth workshop on I/O in parallel and distributed systems. ACM, pp 23–32

19. Gropp W, Huss-Lederman S, Lumsdaine A, Lusk E, Nitzberg B, Saphir W, Snirv (1998) MPI - the complete reference, The MPI-2 extensions, vol 2
20. Li J, Liao W, Choudhary A, Ross R, Thakur R, Gropp W, Latham R, Siegel A, Gallagher B, Zingale M (2003) Parallel netCDF: a high-performance scientific I/O interface. In: Proceedings of the 2003 ACM/IEEE conference on supercomputing. IEEE Computer Society, p 39
21. The HDF Group (1997-2014) Hierarchical Data Format, version 5. http://www.hdfgroup.org/HDF5/
22. Ali N, Carns P, Iskra K, Kimpe D, Lang S, Latham R, Ross R, Ward L, Sadayappan P (2009) Scalable I/O forwarding framework for high-performance computing systems. In: IEEE international conference on cluster computing and workshops, 2009. CLUSTER'09. IEEE, pp 1–10
23. Srivastava A, Eustace A (1994) ATOM: a system for building customized program analysis tools. ACM 29(6):196–205
24. De Bus B, Chanet D, De Sutter B, Van Put L, De Bosschere K (2004) The design and implementation of FIT: a flexible instrumentation toolkit. In: Proceedings of the 5th ACM SIGPLAN–SIGSOFT workshop on program analysis for software tools and engineering. ACM, pp 29–34
25. Bala V, Duesterwald E, Banerjia S (2000) Dynamo: a transparent dynamic optimization system. In: ACM SIGPLAN Notices, vol 35, no. 5. ACM, pp 1–12
26. Bruening DL (2004) Efficient, transparent, and comprehensive runtime code manipulation. Ph.D. dissertation, Massachusetts Institute of Technology
27. Luk C, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi V, Hazelwood K (2005) Pin: building customized program analysis tools with dynamic instrumentation. In: ACM SIGPLAN notices, vol 40, no. 6. ACM, pp 190–200
28. Source O. Dyninst: An application program interface (api) for runtime code generation. Online, http://www.dyninst.org
29. Hollingsworth JK, Niam O, Miller BP, Xu Z, Gonçalves MJ, Zheng L (1997) MDL: a language and compiler for dynamic program instrumentation. In: Proceedings of parallel architectures and compilation techniques. IEEE, pp 201–212
30. Nieuwejaar N, Kotz D, Purakayastha A, Ellis S, Best M (1996) File-access characteristics of parallel scientific workloads. Parallel Distrib Syst IEEE Trans 7(10):1075–1089
31. Simitci H (1996) Pablo MPI instrumentation user's guide. Department of Computer Science, University of Illinois
32. Moore S, Wolf F, Dongarra J, Shende S, Malony A, Mohr B (2005) A scalable approach to MPI application performance analysis. Recent advances in parallel virtual machine and message passing, interface
33. Moore S, Cronk D, London K, Dongarra J (2001) Review of performance analysis tools for mpi parallel programs. In: Recent advances in parallel virtual machine and message passing interface. Springer, pp 241–248
34. Pillet V, Labarta J, Cortes T, Girona S (1995) Paraver: a tool to visualize and analyze parallel code. In: Proceedings of WoTUG-18: transputer and occam developments, vol 44, pp 17–31
35. Open |SpeedShop. http://www.openspeedshop.org/wp/
36. Mohr B, Wolf F (2004) KOJAK—a tool set for automatic performance analysis of parallel programs. Euro-Par 2003 parallel processing, pp 1301–1304
37. Arnold D, Ahn D, De Supinski B, Lee G, Miller B, Schulz M (2007) Stack trace analysis for large scale debugging. In: IEEE international parallel and distributed processing symposium. IEEE, p 64
38. Barham P, Donnelly A, Isaacs R, Mortier R (2004) Using Magpie for request extraction and workload modelling. In: OSDI, vol 4, p 18
39. Sigelman BH, Barroso LA, Burrows M, Stephenson P, Plakal M, Beaver D, Jaspan S, Shanbhag C (2010) Dapper, a large-scale distributed systems tracing infrastructure, Google research
40. Erlingsson Ú, Peinado M, Peter S, Budiu M, Mainar-Ruiz G (2012) Fay: extensible distributed tracing from kernels to clusters. ACM Trans Comput Syst (TOCS) 30(4):13
41. Lee GL, Schulz M, Ahn DH, Bernat A, de Supinski BR, Ko SY, Rountree B (2007) Dynamic binary instrumentation and data aggregation on large scale systems. Int J Parallel Program 35(3):207–232
42. Carns P, Latham R, Ross R, Iskra K, Lang S, Riley K (2009) 24, 7 characterization of petascale I, O workloads. In: Cluster computing and workshops, (2009) CLUSTER'09. IEEE international conference. IEEE, pp 1–10
43. Nagel WE, Arnold A, Weber M, Hoppe H-C, Solchenbach K (1996) VAMPIR: visualization and analysis of MPI resources

44. Kim SJ, Son SW, Liao W-K, Kandemir M, Thakur R, Choudhary A (2012) IOPin: runtime profiling of parallel I/O in HPC systems. In: High performance computing, networking, storage and analysis (SCC). SC Companion: IEEE, pp 18–23
45. Fryxell B, Olson K, Ricker P, Timmes F, Zingale M, Lamb D, MacNeice P, Rosner R, Truran J, Tufo H (2000) FLASH: an adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. The Astrophys J Suppl Ser 131:273
46. Gurumurthi S, Sivasubramaniam A, Kandemir M, Franke H (2003) DRPM: dynamic speed control for power management in server class disks. In: Computer architecture, (2003) Proceedings of 30th annual international symposium. IEEE, pp 169–179
47. Sankaran R, Hawkes E, Chen J, Lu T, Law C (2006) Direct numerical simulations of turbulent lean premixed combustion. In: Journal of physics: conference series, vol 46. IOP Publishing, p 38