Visual Analysis of I/O System Behavior for High–End Computing

Chris Muelder, Carmen Sigovan, Kwan-Liu Ma
ViDi Group
University of California, Davis
Davis, CA 95616
cwmuelder/cmsigovan/klma@ucdavis.edu

Jason Cope, Sam Lang, Kamil Iskra, Pete Beckman, Robert Ross
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
copej/slang/iskra/beckman/rross@mcs.anl.gov

ABSTRACT

As supercomputers grow ever larger, so too do application run times and data requirements. The operational patterns of modern parallel I/O systems are far too complex to allow for a direct analysis of their trace logs. Several visualization methods have therefore been developed to address this issue. Traditional, direct visualizations of parallel systems, such as Gantt charts, can be applied to parallel file systems, but do they not capture domain specific properties nor scale up to modern systems. We propose a portable I/O tracing system and visualization methods to analyze the traces we have obtained. We demonstrate the effectiveness of this system on existing parallel storage systems.

Categories and Subject Descriptors

B.4.3 [**Input/Output and Data Communication**]: Interconnections (Subsystems)—*Parallel I/O*; H.5.m [**Information Interfaces and Presentation**]: Miscellaneous

General Terms

Performance

Keywords

Parallel I/O, Performance Analysis Tools, Information Visualization

1. INTRODUCTION

As the size and complexity of high-performance computing (HPC) systems continue to increase, several software layers provide support for applications to manage, coordinate and effectively use HPC resources. Developers use the tools and capabilities of these layers to create efficient and scalable scientific applications. This layered software capability and deployment approach also provides users with high-level interfaces that encapsulate the often intricate details of HPC system software.

The HPC I/O software stack comprises many layers that exhibit complex interactions. As applications issue I/O requests through this software stack, the I/O requests are handled by several software layers that transform and optimize file I/O for storage on a specific file system configuration. Obtaining an accurate and high-fidelity view of an application's I/O behavior is difficult or even impossible with current performance analysis tools. While a single I/O kernel can be distilled and analyzed from some applications, such as the FLASH-IO [8] and Chombo [6] I/O kernels, one cannot determine the overhead incurred at each software layer. While performance analysis tools, such as Darshan [4] and IPM [37], provide a high-level view for application-level I/O requests, the information provided by these tools is too broad for in-depth analysis.

An extensible set of tools that can provide a detailed representation of application I/O behavior would provide sufficient information to determine these costs. Currently, no end-to-end data collection and analysis tools provide this capability. The goal of the IOVIS project is to fill this void by providing end-to-end analysis capabilities for HPC I/O systems. IOVIS consists of data collection tools that hook into I/O system software and produce comprehensive I/O traces. Using these traces, the IOVIS visualization and analysis tools provide scalable techniques for users to distill knowledge and insight about application I/O behavior. With these tools, users can perform in-depth analyses of application I/O requests across the I/O software layers and determine why, where, and when I/O bottlenecks occur, based on an application I/O use case.

In this paper, in sections 3 and 4, we present our initial implementation of the IOVIS tool set. We describe the capabilities provided by the tools, how the tools integrate with the I/O system software components, and the implementation of several visualization techniques that enable scalable analysis of large traces. In Section 2, we present recent research related to the IOVIS project. In Section 5, we present our results using these tools and techniques. Section 6 presents our current and future areas of research. Section 7 concludes the paper with a brief summary.

LSAP'11, June 8, 2011, San Jose, California, USA.

2. RELATED WORK

There are two significant areas of related work to IOVIS. HPC software instrumentation and data collection research is related to the IOVIS tracing and data collection components. The IOVIS information visualization and analysis research is related to prior work that has developed data visualization techniques which target software performance data and metrics.

2.1 Related Data Collection Work

Tools are required to capture data on the behavior and interaction of I/O software components. HPC software instrumentation and tracing are active areas of research with a wide breadth of research topics. In our work, we adopt successful techniques rather than building new ones, and we focus our efforts on gaps in existing tools.

Some mainstream tracing solutions would not be a good fit for our purpose because they require software or hardware configurations not available on the systems we are working with. LANL-Trace [20], for instance, relies on general-purpose compute-node kernels and dynamically linked libraries, which are not available on IBM Blue Gene systems using the CNK operating systems. HPCT-I/O [12] and IOT [30] are two examples of I/O tracing toolkits developed specifically for leadership class architectures, respectively IBM Blue Gene and Cray XT. However, the results published so far have all been performed at small scale, so it is too early to say how these toolkits will function at HPC scales. TAU [34] is a flexible program and performance analysis toolkit that supports parallel tracing and has a field-proven scaling record, having been used at full scale on IBM BG/L (LLNL), Cray XT3, SGI Altix (NASA Columbia), and Linux clusters (NERSC). It is a generic tool framework that can be used for a variety of performance analysis tasks, I/O tracing included.

One successful example of generating large-scale I/O traces is the work of the Sandia Scalable I/O team, which released the traces of several parallel applications ran at a scale of 2744–6400 processes on Red Storm, a Cray XT3-class machine [31]. The traces were obtained by incorporating a lightweight tracing capability [26] into the SYSIO library [35], a user-level VFS layer linked into the applications on that platform.

Using IOVIS, we opt for a direct instrumentation approach to capture I/O operations. IOVIS traces the MPI-IO operations through the PMPI profiling interface and uses symbol renaming to capture POSIX I/O operations. This makes it easy to insert callbacks into the tracing infrastructure for each MPI-IO operation performed by the application. For other features within software components, we manually instrument software to collect data of interest. For example, we wrap the network communication calls within the PVFS2 server and client to capture information related to PVFS2 network communication.

We have performed an initial study into the problem of large-scale tracing, using in-house expertise in parallel file systems (the PVFS2 team), MPI (the MPICH team), and performance analysis (the MPI Parallel Environment (MPE) team), as well as our collaboration with the TAU team [34]. We traced MPI-IO calls made to a PVFS2 [28] volume using the PMPI interface discussed earlier. The application we used was IOR [13], a parallel application I/O benchmark. The trace events used for this work were logged by using MPE.

2.2 Related Visualization Work

Using visualization to optimize the performance of parallel systems has been explored in several ways by researchers. The communications between parallel processes and data storage servers have been researched through the analysis of access patterns [29, 39, 40]. Communication between software modules, such as client-server relationships, has also been analyzed through the use of graph-based visualizations [41]. These visual approaches are effective at analyzing network traffic, but do not provide insight into computation efficiency in a massively parallel computation environment.

One common set of visualization tools for MPI data is Jumpshot [5, 38, 22] and its predecessors (Nupshot [16] and Upshot [11]). These tools use the MPE library to intercept the MPI calls in a parallel program. They then visualize this information using Gantt charts and color coding for MPI calls. ParaGraph [10] is an older program that visualizes MPI traces collected with the MPICL library, which also uses Gantt charts, among other metrics such as overall summaries and communication graphs. Vampir [17] is a tool that combines Gantt charts and summary views. TAU's [34] visualization toolkit includes Gantt charts, a communication matrix view, and a call graph. The IPM [37] data collection framework also includes some capability for visualizing the resulting data. The visualizations show aggregate I/O rates and MPI function execution times in the form of histogram charts. Virtue [33] is the most unusual of the related works listed here in that it allows the user to to monitor the performance of an application while it is running and potentially tune it or interact with it. Virtue also incorporates virtual reality techniques, such as support for a CAVE (Cave Automatic Virtual Environment), to provide a more immersive visualization. For other parallel environments, GVU's PVaniM tool [36] and ATEMPT [19, 18] present some detailed views of communication events in a PVM (Parallel Virtual Machine) system.

Some software visualizations address the scalability issues of plots such as Gantt charts. Jerding et al. [14], Moreta and Telea [23], and Cornelissen et al. [7] use plots similar to Gantt charts to profile program execution traces, along with sub-pixel techniques to improve scalability. However, they maintain the strict ordering of the charts. In our previous work [25], we addressed some of these issues by removing the vertical constraint of Gantt charts and incorporating techniques such as high-precision alpha blending and opacity scaling similar to the work of Johansson et al. [15]. While these approaches aid in visualizing large scale communication patterns, they do not incorporate the bipartite nature of I/O communication. Here, we present a combination of these existing approaches with a bipartite detail representation and a multifocal temporal plot.

3. DATA COLLECTION APPROACH

Any system analysis is only as good as the data on which it is based. To effectively exploit visualization in order to better understand HPC I/O patterns, we must gather the right data, at the right level of detail. In this section, we describe the general features and interactions of the HPC I/O software stack and describe how trace data is collected from this software stack.

3.1 HPC I/O Software Stack

The typical HPC I/O software stack consists of multiple layers of software that provide a variety of I/O capabilities for specific application I/O patterns, system software configurations, and system architectures. Figure 1 illustrates how these software components are layered on HPC systems.

Across the majority of HPC systems, applications store data on high-performance parallel file systems. These file systems include PVFS2 [28], Lustre [3], and GPFS [32]. These file systems can be dedicated for use by a single computational resource or shared by several computational resources. The file system is often deployed

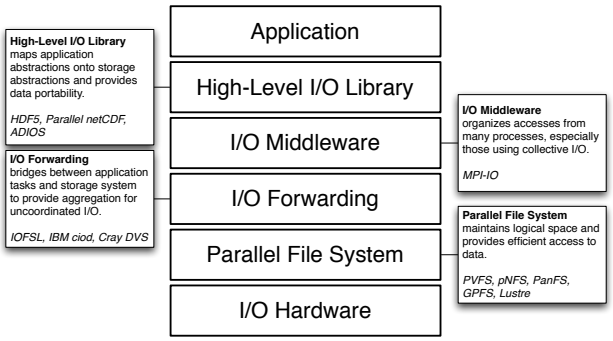


Figure 1: Configuration options for HEC system I/O.

as a dedicated HPC resource using RAID or SAN storage devices. A file system server processes application I/O requests through the file system client interface. The computation resource may run file system clients on all of its nodes or on a subset of the nodes in conjunction with I/O aggregation and forwarding tools. Examples of I/O forwarding tools include IOFSL [1, 27], IBM's ciod, and Cray's Data Virtualization Service.

At the application level, there are several application I/O interfaces. File system I/O interfaces, such as the POSIX and PVFS2 APIs, provide direct access to the file system or virtual file system. MPI-IO provides a parallel I/O interface built on top of the file system's APIs. MPI-IO coordinates and optimizes parallel I/O patterns. High-level and scientific data libraries provide mechanisms to generate self-describing and portable data sets. Examples of high-level I/O libraries include NetCDF, PNetCDF, HDF5, and ADIOS.

The overall goal for these software layers is to provide the best possible I/O performance for common HPC application I/O patterns. Achieving this goal is often a difficult task for application developers because the cost of the high-level I/O operations in the lower layers of the I/O software stack is unknown. Since these layers encapsulate interactions with lower-level software layers, the overhead associated with each layer is indiscernible. Additional information and insight about how these layers interact and what the cost of high-level operations is in subsequent layers will help isolate bottlenecks within the I/O software and identify areas of improvement for software developers. The goal of the IOVIS data collection tools is to acquire the costs of I/O operations and track the interactions of the HPC I/O software stack layers.

3.2 IOVIS Data Collection

To capture the end-to-end behavior of I/O requests using IOVIS, we integrated several instrumentation layers into the application, PVFS2 file system client, and PVFS2 file system server components. These data collection layers track the beginning and completion of file system I/O events and collect information about each event, such as event type and event data sizes. A PVFS2 storage cluster with tracing enabled is set up for instrumented applications to store data on. As the instrumented application runs, trace files are generated for each application process, each PVFS2 client process, and each PVFS2 server process. Once all application, file system client, and file system server trace data is collected, postprocessing tools are used to merge the event logs into a single file per component and convert the event logs into a trace format compatible with the visualization tool.

Each data collection layer collects information for the events initiated at that layer and adds data to track the operation execution through additional software layers. At the application layer, an application event collection library tracks and reports I/O events initiated by an application. Currently, this library supports application-level tracing of MPI-IO events using the PMPI interface. The application data collection library wraps MPI-IO calls with TAU instrumentation to track the start time, end time, file handle, data payload sizes, and a request identifier for each application I/O operation. Each process in the application generates separate collections of I/O event requests.

In order to collect I/O events at the file system client layer, the PVFS2 client was instrumented to report communication events between the application and the PVFS2 data servers. On platforms using PVFS2, the PVFS2 client facilitates communication between the application and the PVFS2 server. In order to track these client and server communications, the PVFS2 client was instrumented with TAU. Each client process tracks its communication with PVFS2 servers. When tracking these events, the client reuses the application I/O request identifier and captures the origin of the request for reporting I/O events. This allows PVFS2 client I/O requests to be tracked back to a specific application process. Additionally, a unique client identifier is tracked for every PVFS2 client operation. This client identifier is added to each PVFS2 operation sent to a server so that servers can track the origin of I/O requests.

The remaining components in the data collection layer are instrumented PVFS2 servers. This instrumentation tracks I/O events received from PVFS2 clients and data storage management events initiated by a PVFS2 server. PVFS2 servers are responsible for managing data on data storage nodes and interacting with PVFS2 clients to complete I/O requests. Therefore, the instrumentation layer must track I/O operations from the PVFS2 servers to the local storage and the communication between the PVFS2 server and PVFS2 clients. The server uses an additional TAU instrumentation layer to track network communication and storage management operations. The client and application identifiers for each request are tracked with each event so that the origin of the operation can be tracked back to the client and application processes.

We have deployed the IOVIS data collection tools on several systems. Our initial experiments were performed on Jazz, a Linux cluster located at Argonne, at a scale of around 120 application processes and 60 PVFS2 servers. The results were promising, and since then we have deployed the tracing infrastructure on several systems at the Argonne Leadership Computing Facility (ALCF). On the ALCF systems, we use the 40-rack Intrepid Blue Gene/P platform for generating application traces and the 100-node Eureka Linux cluster for generating PVFS2 server traces. Each BG/P rack contains 1024 compute nodes and 16 I/O nodes. Each compute node has a four-core 850 MHz IBM PowerPC 450 processor and 2 GB of RAM. The BG/P rack is divided into blocks (called psets) consisting of 64 compute nodes and one I/O node. Each Eureka node has two quad-core Intel Xeon processors, 32 GB of RAM, and 230 GB of local scratch storage. Eureka and Intrepid share a 10 Gbps communication network.

When tracing applications in the ALCF environment, we set up a temporary PVFS2 storage cluster on Eureka and mounted this file system on the allocated Intrepid I/O nodes. We deployed a ZeptoOS [2] operating system image for the Intrepid I/O nodes that initializes the I/O tracing environment. With this deployment, we have successfully traced applications to 16,384 processes on Intrepid and up to 32 PVFS2 I/O servers on Eureka. The applications we have evaluated in this environment include the mpi-tile-io benchmark [24], the IOR benchmark [13], the FLASH I/O kernel [8], and the Chombo I/O kernel. For the mpi-tile-io and IOR

evaluations, applications issued MPI-IO requests directly. For the FLASH I/O evaluations, we generated traces using the HDF5 [9] and Parallel NetCDF I/O [21] (PNetCDF) libraries. We generated HDF5 data using the Chombo I/O kernel. The HDF5 and PNetCDF higher-level I/O libraries issue MPI-IO requests internally, and we used our application I/O tracing library to track the I/O requests generated by these libraries.

4. VISUALIZATION APPROACH

As in our previous work [25], we start with a timeline view of the aggregate activity over all servers. From the timeline, a range of time can be selected to be shown in a second-level view. In this view, the I/O operations are plotted by the log of duration versus time, which visually clusters similar activity. From this view, a single point in time or a range of time can be selected for the detailed view, which shows the bipartite relationship between computation nodes and I/O servers, as well as a fisheye view of the selected point(s) in time.

4.1 Timeline

The timeline view depicts a stacked graph of the overall I/O activity over time. Each stacked area of the graph is associated with a type of operation, and its height represents the fraction of operations of that type in execution at a certain time. The timeline view is also used as an interface for selecting smaller time ranges to view in more detail. The selected range is indicated by the semi-transparent box shown in Figure 2. Colors are defined in Figure 3.

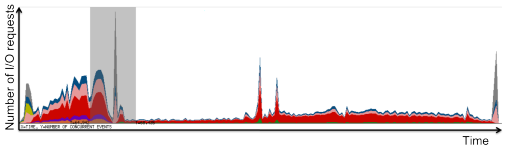


Figure 2: *Timeline of I/O activity*. The timeline provides an overview of the activity of the entire system. From the timeline, ranges of data can be selected to view in more detail.

4.2 Point-based Midlevel View

The most direct representation of the server activity is to render each operation with respect to time. Gantt charts do this, but they restrict the y-axis to represent the processes. In [25], we proposed an alternative representation. While we retain the use of the x-axis as time, we use the y-axis to represent other properties, in particular, the duration of the operation, especially on a logarithmic scale, since the durations vary over several orders of magnitude. The advantage of using duration on the y-axis is that abnormally large operations are prominently seen at the top of the plot. Since this and other y-axis mappings allow the operations to overlap, we modulate the opacity of the calls, which makes the overall intensity of the visualization represent the density of operations. Figure 4 shows an example of this. The color is mapped to the operation type as in the timeline.

The duration of the operations is already being encoded in the height, so it is redundant to also show duration on the x-axis. Instead, we use simple points to plot the duration of the operations versus either the start or end times. Similar to the line representation, dependency information is not easily visible. However, vertical and logarithmic trends clearly delimit events starting or ending simultaneously. When plotting start times versus duration, the vertical trends show simultaneous start times, and the log curves to the

Figure 3: *Color legend*. The colors used in the timeline, scatter, and fisheye plots.

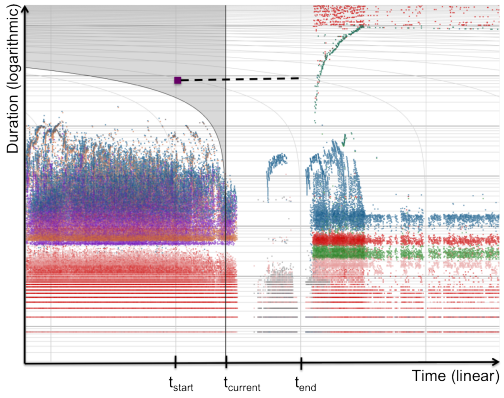


Figure 4: *Point-based plot of I/O activity*. The large point was added for illustrative purposes. The event it represents started at t_{start} and ended at t_{end}. The fact that this event was in execution at time $t_{current}$ is indicated by the point's location within the area between the logarithmic curve and asymptote corresponding to $t_{current}$.

left show simultaneous end times. When plotting end time versus duration, it is the other way around, with the log curves to the right.

One effect of a plot such as this, with time on both axes, is that a single point in time is no longer a single coordinate on the plot. At any given time, there are some number of active operations, where an operation is active if its start time is less than the current time and its end time is after the current time, that is:

$$t_{start} < t_{current} \quad and \quad t_{current} < t_{end}$$

where t_{start} and t_{end} are the start and end times of the operation and $t_{current}$ is the current time. We define t_{start} as the x-axis in the plot, so the left inequality becomes simply

$$x_{start} < x_{current}$$

which is the area left of a vertical line at $x_{current}$. But what about t_{end}, since it is not in the plot? It can simply be defined as

$$t_{end} = t_{start} + t_{duration}$$

where $t_{duration}$ is the duration of the operation, which is mapped onto the y-axis. However, the y-axis is on a logarithmic scale, so

$$y_{duration} = log_k\, t_{duration} \quad or \quad t_{duration} = k^{y_{duration}}$$

The original inequality thus becomes

$$x_{current} < x_{start} + k^{y_{duration}} \quad or \quad x_{start} > x_{current} - k^{y_{duration}}$$

which is the area to the right of a logarithmic curve asymptotically ending at $x_{current}$. A grid of these logarithmic curves is plotted in the background of Figure 4 to give a reference frame for the end times of operations. The intersection of the regions given by the inequalities defines an area in the plot that corresponds to a given point in time where every point within the region is an active operation. Similarly, this representation can be extended to select

a range of time by replacing $t_{current}$ and $x_{current}$ with the beginning and end of the selected time range. We use this representation to allow the user to select a time or region of time to view in more detail.

4.3 Opacity Scaling

When the operations are plotted with our approach, many overlap, particularly when they start or end simultaneously. A simple way to resolve this overlap is to make the calls semitransparent and use alpha blending to combine them. However, this quickly runs into limitations as the number of calls increases. First, the standard 8-bit alpha buffer only allows for a maximum overplotting of 256. Second, in order to show large numbers of overlapping events, the opacity has to be set so low that outliers are nearly invisible. To keep both the opacity of outliers high and the combined opacity of dense overlap from overflowing the alpha buffer, we utilize the opacity scaling technique of [15]. In our implementation of this technique, we first render to a high-precision density buffer D, which keeps track of the total amount of overplot, and then to a high precision color buffer C, which blends the input color information with opacity inversely proportional to the density information to result in an average color that is fully opaque. We then combine these buffers with a mapping function to render the final pixels P to the screen. We use a logarithmic mapping function,

$$P_{x,y} = C_{x,y} \times \left(o_{min} + (1 - o_{min}) \times \frac{log\left(D_{x,y}\right)}{log\left(D_{max}\right)} \right)$$

where o_{min} is a user-defined minimum opacity level and D_{max} is the maximum level of overplotting that occurred. By calculating the final opacity in this manner, we guarantee that any outliers will have at least opacity o_{min}, that no overplotting exceeds the maximum opacity, and, in the case of the logarithmic map, that the system will be able to handle many orders of magnitude of overplotting.

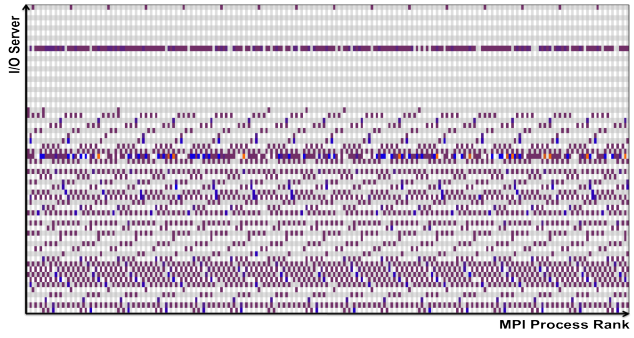


Figure 5: *Bipartite matrix of MPI ranks and I/O servers*. Cell color indicates number of operations and ranges from blue for low numbers to yellow for the highest numbers; purple coloring indicates mid-range values. The repeated patterns indicate that the processes are accessing servers according to some sort of modulo operation. Also, some servers can be seen to be under noticeably more load than others, while some servers are idle.

4.4 Matrix View

While the point-based plot provides a good view of a large number of concurrent operations, it does not convey information such as the individual server loads or the interconnection between the processes and the I/O servers. Gantt charts also do not show the interconnection network. Therefore, we have added a view to show the connections between the computation nodes and the data storage servers. Since the communication pattern is often constantly changing over the course of the job, we use this view to show the communication at instants in time or over selected durations. As we are not considering the inter-process communication here, the end-to-end communication pattern is a bipartite network, and traditional bipartite network representations can be applied. Figure 5 shows the communication network as a matrix, where the columns are the computation processes and the rows are the data storage servers. The color at the intersection of each row and column indicates the level of activity between those two entities. In the figure, we can see that two of the servers are being accessed by the majority of the compute nodes, indicating possible communication bottlenecks.

4.5 Fisheye Time Plot

Gantt charts may have some scalability issues, but are still quite intuitive. Since there are significantly fewer data storage servers than compute nodes, we can utilize Gantt charts of the servers without running out of screen space. However, there is also an issue of temporal scalability. The I/O operation durations can differ by many orders of magnitude. Showing both short and long operations in the same plot can be difficult. We therefore use fisheye zooming techniques in order to expand small operations near the focal point and shrink the much larger operations. We found that the sigmoid function was a nice fit for the zooming function, as the asymptotic properties guarantee a fixed boundary no matter how varied the inputs. So for a single focal point, we map time t to

$$x(t,t_a) = \sigma(t,t_a) = \frac{1}{1+u^{-v*(t-t_a)}}$$

where u and v are user-adjustable constants and t_a is the focal point. But our system allows the user to select a range of time, not just one focal point. To accommodate this, we decided to try a multifocal mapping function consisting of a sum of sigmoids. That is, we mapped time t to

$$x(t,t_a,t_b) = \sigma(t,t_a) + \sigma(t,t_b) = \frac{1}{1+u^{-v*(t-t_a)}} + \frac{1}{1+u^{-v*(t-t_b)}}$$

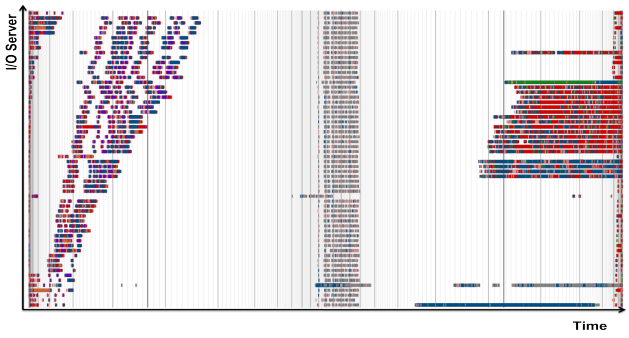


Figure 6: *Gantt chart with two fisheye zooms*. A time range was selected, and this view uses sigmoid functions around the start and end times to expand the surrounding area.

However, we discovered that when t_b and t_a got too far apart, the asymptotic nature of the sigmoid function caused data between the two end points to vanish into a horizon between the focal points. To offset this problem, we introduced a piecewise linear parameter

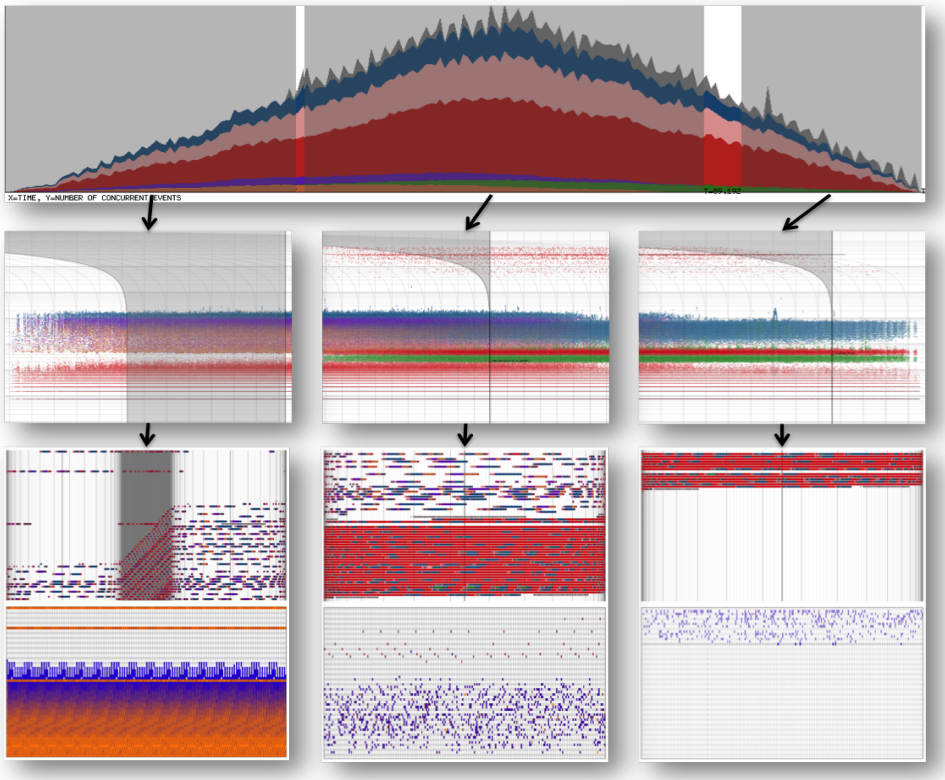


Figure 7: *IOR benchmark*. The IOR benchmark with one collective file on Jazz. There is a fairly consistent level of activity over the whole duration. However, the system takes a while to ramp up to the peak level operations and slowly falls off. This seems like a very serial trend where servers are preferentially used in order, resulting in some load imbalance. At left: early in the execution, only the first half of the servers are being used doing write operations, as the system is ramping up; the second half of the servers are idle. At middle: midway through the execution, the same pattern has moved to the second half of the servers, while the first half of the servers has finished the write operations and started on reads. At right: the last servers are finishing up the read operations while the first servers are idle.

$p(t,t_a,t_b)$ into the mapping function,

$$p(t,t_a,t_b) = \begin{cases} 0 & \text{if } t \leq t_a \\ \frac{t-t_a}{t_b-t_a} & \text{if } t_a < t < t_b \\ 1 & \text{if } t \geq t_b \end{cases}$$

The overall mapping function is

$$x(t,t_a,t_b) = (1-w) * (\sigma(t,t_a) + \sigma(t,t_b)) + w * p(t,t_a,t_b)$$

where w is another user-defined constant between 0 and 1. This allows the user to scale from completely zoomed ($w = 0$) to completely flat ($w = 1$). An example of this multifocal mapping is shown in Figure 6.

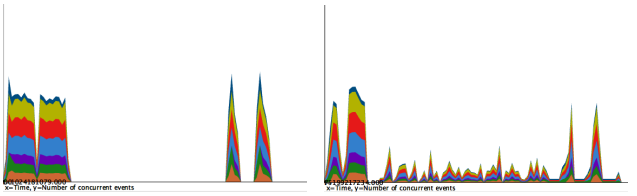
5. RESULTS

The traces generated for this paper were collected from dedicated resources where possible. This minimizes interference from nontraced applications or user activity that would distort the trace data. The dedicated environment allows us to deploy, modify and test our trace-enabled I/O software stack. When generating traces on cluster-based platforms, dedicated compute nodes are allocated for the application and for the instrumented I/O software. On the IBM Blue Gene/P system, we allocated dedicated I/O software resources on an external cluster that is similar to the production file system servers used by the ALCF. The extra compute nodes we allocate for the I/O software are accessible only by our application. The networks connecting the compute nodes with the I/O nodes are shared, systemwide resources and are subject to interference from other user activity.

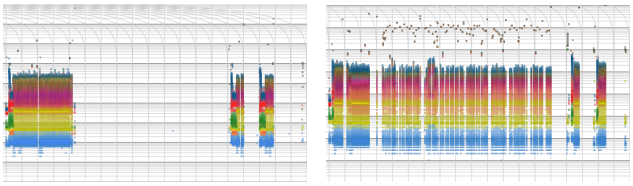
We conducted our initial experiments on Jazz, a Linux cluster at Argonne National Laboratory. These initial experiments were meant to validate our data collection process and help us test the capabilities of the visualization toolkit. We obtained an initial result from running an IOR test with a collective file shared between processes. In this test (Figure 7), the overall activity level (shown in the timeline view) was very gradual, taking a while to ramp up to the peak level of activity, then gradually falling until it finished. When we look at the trace in the more detailed views, the reason becomes clearer. The scatterplot mid-level views show us that the duration of events throughout the run was mostly constant, with communication and network requests taking significantly longer than file access operations (orange and green). This may suggest network latency problems. In the detail-views, we can see that file access patterns are very serialized, with servers being accessed in order, one by one.

We also explored the potential use of the visualization application for comparative analysis of different parallel codes or I/O frameworks, as shown in Figure 8. The cases shown are two dif-

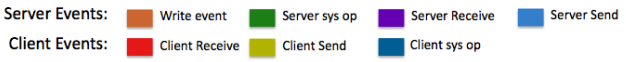
ferent runs of the FLASH I/O benchmark using the HDF5 and PNetCDF data access interfaces, both run on Intrepid's architecture and both using collectives. We can clearly see that the access patterns differ between these two interfaces. In the PNetCDF run, the peak of activity (maximum number of events in a time step) occurred in the beginning, with a pair of smaller peaks at the end of execution and many bursts of activity between.



(a) FLASH HDF5 histogram (b) FLASH PNetCDF histogram



(c) FLASH HDF5 scatterplot (d) FLASH PNetCDF scatterplot



(e) Color legend

Figure 8: Comparison views illustrating the difference in I/O patterns between HDF5 and PNetCDF with the client layer also added to the visualization. The benchmark used for both tests was FLASH, with 2048 processes, 8 I/O clients, and 2 I/O servers. The configuration of processes to compute nodes was one to one.

The HDF5 run has a similar pattern of peaks, but the time between the peaks is empty. By looking at the scatterplot views, we can see that all the peak levels of activity correspond to periods of constant activity, where events of a given type consistently take the same amount of time, which causes the striation patterns. Conversely, the valleys between peak levels of activity in the timeline correspond to gaps in the regular activity where particular events are taking longer than usual. It is notable that each gap is preceded by a write event that takes a long time (in the case of the large gap in the HDF5 run, the event is longer by several orders of magnitude). We posit that these gaps are caused by low level interruptions such as kernel tasks or disk seek times. Interestingly, the HDF5 run has one very long gap while the PNetCDF run has many, much smaller gaps, potentially indicating a difference in how the two formats buffer data when writing it to disk. Such comparative studies may be used to determine which framework is more appropriate for certain parallel codes from the point of view of I/O events and their execution times.

6. FUTURE WORK

While our approach captures end-to-end relationships well, it does not completely track the flow of data through the network. In order to identify intermediate network-based bottlenecks, more in-depth tracing of intermediate steps and an alternative representation would be necessary. Some of the traces generated by these methods can get large. While our current visualization implementation loads the entire data set at once, out-of-core techniques would be necessary for handling larger tests, which would allow for scaling to even larger systems. We have begun collecting data from I/O forwarding layers, as well as network traffic data from the Cray architecture. So far, we have primarily looked at I/O benchmarks, which may or may not be representative of common activity. Ideally, it would be better to instrument real simulation codes, or at least portions of them.

7. CONCLUSIONS

The problem of high-performance I/O optimization is complicated, but important. As the gap between processing power and I/O storage rates widens, the efficient use of the storage available will have more and more impact on the performance of the system as a whole. Here, we have presented an approach for the capture and visual analysis of I/O traces, and have applied this approach to the systems at Argonne National Laboratory. Our visualization approach has been effective in exploring and understanding the collected traces and has shown several instances where the system could be optimized, which could lead to more efficient configurations.

8. ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation through NSF-0937928 and by the Office of Advanced Scientific Computer Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. Computing time on Intrepid was provided by a U.S. Department of Energy INCITE award and an ALCF Director's Discretionary Allocation.

9. REFERENCES

[1] N. Ali, P. Carns, K. Iskra, D. Kempe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. Scalable I/O forwarding framework for high-performance computing systems. In *IEEE International Conference on Cluster Computing (Cluster 2009)*, 2009.

[2] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. Operating system issues for petascale systems. *SIGOPS Oper. Syst. Rev.*, 40(2):29–33, 2006.

[3] P. Braam. The Lustre storage architecture. http://www.lustre.org/docs/lustre.pdf, 2004.

[4] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale I/O workloads. In *2009 Workshop on Interfaces and Architectures for Scientific Data Storage*, 2009.

[5] A. Chan, W. Gropp, and E. Lusk. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, 16(2-3):155–165, 2008.

[6] Chombo - Infrastructure for adaptive mesh refinement. https://seesar.lbl.gov/ANAG/chombo/.

[7] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *J. Syst. Softw.*, 81(12):2252–2268, 2008.

[8] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement Series*, 131(1):273, 2000.

[9] HDF5. http://hdf.ncsa.uiuc.edu/HDF5/.

[10] M. T. Heath. ParaGraph: A tool for visualizing performance of parallel programs. In *Second Workshop on Environments and Tools for Parallel Sci. Comput*, pages 221–230, 1994.

[11] V. Herrarte and E. Lusk. Studying parallel program behavior with upshot. Technical Report ANL–91/15, Argonne National Laboratory, 1991.

[12] IBM's HPCT-I/O, http://domino.research.ibm.com/comm/research_projects.nsf/pages/hpct.mio.html.

[13] The IOR benchmark, http://sourceforge.net/projects/ior-sio/.

[14] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *ICSE '97: Proc. of the 19th Intl. Conf. on Software Engineering*, pages 360–370. ACM, 1997.

[15] J. Johansson, P. Ljung, M. Jern, and M. Cooper. Revealing structure within clustered parallel coordinates displays. In *InfoVis '05: Proc. of the 2005 IEEE Symposium on Information Visualization*, pages 125–132. IEEE Computer Society, 2005.

[16] E. Karrels and E. Lusk. Performance analysis of MPI programs. In J. Dongarra and B. Tourancheau, editors, *Proc. of the Workshop on Environments and Tools for Parallel Scientific Computing*, pages 195–200. SIAM Publications, 1994.

[17] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir performance analysis tool-set. In M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Tools for High Performance Computing*, pages 139–155. Springer, Berlin, 2008.

[18] D. Kranzlmüller, S. Grabner, and J. Volkert. Debugging massively parallel programs with ATEMPT. In *HPCN Europe 1996: Proc. of the Intl. Conf. and Exhibition on High-Performance Computing and Networking*, pages 806–811, London, UK, 1996. Springer-Verlag.

[19] D. Kranzlmüller and J. Volkert. Debugging point-to-point communication in MPI and PVM. *Lecture Notes in Computer Science*, 1497:265–272, 1998.

[20] LANL-Trace, http://institute.lanl.gov/data/software/#lanl-trace.

[21] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, page 39, Washington, DC, USA, 2003. IEEE Computer Society.

[22] T. Ludwig, S. Krempel, M. Kuhn, J. Kunkel, and C. Lohse. Analysis of the MPI-IO optimization levels with the PIOViz Jumpshot enhancement. In F. Cappello, T. Herault, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 213–222. Springer, Berlin, 2007.

[23] S. Moreta and A. Telea. Multiscale visualization of dynamic software logs. In *EuroVis*, pages 11–18, 2007.

[24] mpi-tile-io. http://www.mcs.anl.gov/research/projects/pio-benchmark/.

[25] C. Muelder, F. Gygi, and K.-L. Ma. Visual analysis of inter-process communication for large-scale parallel computing. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1129–1136, October 2009.

[26] N. Nakka, A. Choudhary, R. Klundt, M. Weston, and L. Ward. Detailed analysis of I/O traces of large scale applications. In *HiPC, International Conference on High Performance Computing*, December 2009.

[27] K. Ohta, D. Kempe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa. Optimization techniques at the I/O forwarding layer. In *IEEE International Conference on Cluster Computing (Cluster 2010)*, 2010.

[28] PVFS2: Parallel Virual File System, version 2, www.pvfs.org.

[29] R. Ross, T. Peterka, H.-W. Shen, Y. Hong, K.-L. Ma, H. Yu, and K. Moreland. Visualization and parallel I/O at extreme scale. *Journal of Physics*, July 2008. Proc. of DOE SciDAC Conf., 2008.

[30] P. C. Roth. Characterizing the I/O behavior of scientific applications on the Cray XT. In *PDSW '07: Proc. of the 2nd International Workshop on Petascale Data Storage*, pages 50–55, New York, NY, 2007. ACM.

[31] Sandia National Laboratories' Red Storm I/O traces, http://www.cs.sandia.gov/Scalable_IO/SNL_Trace_Data/index.html.

[32] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *First Conference on File and Storage Technologies (FAST)*, 2002.

[33] E. Shaffer, D. Reed, S. Whitmore, and B. Schaeffer. Virtue: Performance visualization of parallel and distributed applications. *IEEE Computer*, 32(12):44–51, Dec 1999.

[34] S. S. Shende and A. D. Malony. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[35] The SYSIO library, http://sourceforge.net/projects/libsysio.

[36] B. Topol, J. T. Stasko, and V. Sunderam. PVaniM: A tool for visualization in network computing environments. *j-CPE*, 10(14):1197–1222, Dec. 1998.

[37] A. Uselton, M. Howison, N. Wright, D. Skinner, N. Keen, J. Shalf, K. Karavanic, and L. Oliker. Parallel I/O performance: From events to ensembles. In *IEEE International Parallel and Distributed Processing Symposium 2010*, April 2010.

[38] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proc. of ACM/IEEE Supercomputing (SC00)*, November 2000.

[39] H. Yu and K.-L. Ma. A study of I/O techniques for parallel visualization. *Journal of Parallel Computing*, 31(2):167–183, Feb 2005.

[40] H. Yu, K.-L. Ma, and J. Welling. A parallel visualization pipeline for terascale earthquake simulations. In *Proc. of ACM/IEEE Supercomputing (SC04)*, Nov 2004.

[41] D. Zeckzer, R. Kalcklösch, L. Schröder, H. Hagen, and T. Klein. Analyzing the reliability of communication between software entities using a 3D visualization of clustered graphs. In *SoftVis '08: Proc. of the 4th ACM Symposium on Software Visualization*, pages 37–46. ACM, 2008.