PIDX: Efficient Parallel I/O for Multi-resolution Multi-dimensional Scientific Datasets

Sidharth Kumar*, Venkatram Vishwanath†, Philip Carns†, Brian Summa*, Giorgio Scorzelli*,
Valerio Pascucci*, Robert Ross†, Jacqueline Chen‡, Hemanth Kolla‡ and Ray Grout§
* *SCI Institute, University of Utah, Salt Lake City, UT, USA*
† *Argonne National Laboratory, Argonne, IL, USA*
‡ *Sandia National Laboratory, Livermore, CA, USA*
§*National Renewable Energy Laboratory, Golden, CO, USA*
Email:sidharth@sci.utah.edu

Abstract—**The IDX data format provides efficient, cache oblivious, and progressive access to large-scale scientific datasets by storing the data in a hierarchical Z (HZ) order. Data stored in IDX format can be visualized in an interactive environment allowing for meaningful explorations with minimal resources. This technology enables real-time, interactive visualization and analysis of large datasets on a variety of systems ranging from desktops and laptop computers to portable devices such as iPhones/iPads and over the web. While the existing ViSUS API for writing IDX data is serial, there are obvious advantages of applying the IDX format to the output of large scale scientific simulations. We have therefore developed PIDX - a parallel API for writing data in an IDX format. With PIDX it is now possible to generate IDX datasets directly from large scale scientific simulations with the added advantage of real-time monitoring and visualization of the generated data.**

In this paper, we provide an overview of the IDX file format and how it is generated using PIDX. We then present a data model description and a novel aggregation strategy to enhance the scalability of the PIDX library. The S3D combustion application is used as an example to demonstrate the efficacy of PIDX for a real-world scientific simulation. S3D is used for fundamental studies of turbulent combustion requiring exceptionally high fidelity simulations. PIDX achieves up to 18 GiB/s I/O throughput at 8,192 processes for S3D to write data out in the IDX format. This allows for interactive analysis and visualization of S3D data, thus, enabling in situ analysis of S3D simulation.

I. Introduction

The increase in computational power of supercomputers is enabling unprecedented opportunities to advance science in numerous fields such as climate science, astrophysics, cosmology and material science. These simulations routinely produce large volumes of data. Analyzing this data and transforming it into useful insight is a key component of scientific discovery that is generally hindered by bottlenecks in I/O access.

The IDX format provides efficient, cache oblivious, and progressive access to large-scale scientific data by storing the data in a hierarchical Z (HZ) order [13]. The format increases the locality of data access for common queries, making it possible for scientists to interactively analyze and visualize data of the order of several terabytes [15]. IDX has been used successfully in fields such as digital photography [17], visualization of large scientific data and it is a promising approach for analysis of HPC data as well [14].

ViSUS, an IDX API, is serial in nature. This limits the use of IDX to relatively small-scale applications. To overcome the serial design of ViSUS, we developed PIDX - a parallel API to capture the data models used by HPC application and write it out in an IDX format. PIDX enables simulations to write out data in parallel in the IDX format so that scientists can interactively visualize and analyze the data. It utilizes the computation resources of each compute node to efficiently calculate the HZ ordering. It then coordinates file system access using collective communication to write the dataset in parallel. In this paper we present optimization strategies to improve the performance of aggregation in PIDX at scale. We demonstrate the efficacy of PIDX with S3D combustion simulation on the Hopper2 supercomputer at NESRC. Additionally, we demonstrate an end-to-end pipeline using ViSUS to remotely visualize this data interactively. This enables a scientist to monitor the health of their simulations which can assist in steering the simulation as well.

The remainder of this paper is organized as follows: We present relevant background information on the IDX data format in Section II. We present the PIDX API and design in Section III. We evaluate the performance of our API in Section IV using micro-benchmarks and application-level benchmarks. Next, in section V, we describe an end-to-end pipeline for S3D, from storing its simulation data using PIDX to its visualization using ViSUS. We describe relevant related work in section VI. In section VII, we present our conclusions and discuss our plans for further research.

II. ViSUS: IDX file Format

IDX enables fast and efficient access to large-scale scientific data. HZ ordering is the key idea behind IDX data format. The HZ order is calculated for each data sample

HZ Level	Start HZ	End HZ	Block Number (File Number)
0	0	0	0(0)
1	1	1	0(0)
2	2	3	0(0)
3	5	7	0(0)
4	8	15	0(0)
5	16	31	0(0)
6	32	63	0(0)
7	64	127	0(0)
8	128	255	0(0)
9	256	511	1(0)
10	512	1023	2(1) 3(1)
11	1024	2047	4(2) 5(2) 6(3) 7(3)
12	2048	4095	8(4) 9(4) 10(5) 11(5) 12(6) 13(6) 14(7) 15(7)

Table I
BLOCK AND FILE DISTRIBUTION OF EACH HZ LEVEL IN A 16^3 IDX DATA SET USING 256 ELEMENTS PER BLOCK AND 2 BLOCKS PER FILE. LEVELS 10 THROUGH 12 SPAN MULTIPLE BLOCKS.

using the spatial coordinates of that sample. All data sample points are then assigned a HZ level. The HZ level can be calculated based on an index in the HZ order using the formula $level = \lfloor \log_2(HZindex) \rfloor + 1$. Each HZ level corresponds to data at a particular level of resolution. In this formulation, with each increasing level, the number of elements increases by a factor of two. Data in an IDX file is written with an increasing HZ order. At storage, the HZ ordered data samples are grouped in blocks of a constant size. A preset number of blocks is written into each binary file. A metadata *.idx* file contains all the required associated information of the IDX file. For any IDX file, it contains the bounding box *(box)*, number of elements per block *(elements_per_block)*, number of blocks per binary file *(blocks_per_file)*, the bitmask and the filename template. Table 1, for example shows block, HZ level as well as the file layout for an IDX file of *box* (0, 0, 0 : 16, 16, 16), 256 *elements_per_block* and 2 *blocks_per_file*.

The conversion of data to IDX format can be considered as converting an N-dimensional data to one dimension. The HZ ordering and corresponding distribution of data into different levels of resolution significantly reduces lag when zooming or panning a large-scale dataset. As can be seen in Table I, block 1 contains data up to HZ level 8. In general, for each IDX file, the first block contains data up to level $\log_2(elements\_per\_block)$. Data in the first block spans the entire volume in the lowest resolution. As we access the blocks in linearly increasing order, the resolution level starts to increase and spans a smaller part of the volume. A data query to an IDX data set requires first checking the metadata to find the intersection of the queried data with the data blocks. Data can then be retrieved using lower resolution data from initial blocks or higher resolution data from later blocks. This scheme of access ensures an interactive and progressive data access.

III. PARALLEL IDX

IDX is a desirable file format for visualization of large scale HPC simulation results because of its ability to access multiple levels of resolution with low latency for interactive exploration. The existing ViSUS implementation is serial in nature, which prevents parallel simulations from writing directly into IDX format. Data could be converted as a post-processing step; however, this would significantly increase the analysis turn around time and make poor use of available parallel I/O resources available in today's leadership-class computing facilities. We have therefore developed the Parallel IDX (PIDX) library, which enables simulations to write IDX data directly [9]. PIDX coordinates data access among participating processes so that they can write concurrently to the same data set with coherent results.

In previous work [9], we successfully demonstrated the use of PIDX in coordinating parallel access to multidimensional datasets. We also explored optimizations both for parallel HZ computation and for efficient access to the underlying data files that make up an IDX dataset. While this implementation was successful with synthetic benchmarks at moderate scale (up to 512 processes), we encountered two key challenges in employing PIDX for real-world applications on leadership-class computing systems. The first is that HPC simulations typically generate data for several related multidimensional variables at each simulation time step. If we store this collection of variables in a naive manner, then we risk inefficiency from redundant HZ ordering calculations and sub-optimal file access patterns. We must therefore capture the application data model in a way that provides a complete picture of the intended data movement so that PIDX can calculate an optimal strategy for its computation and I/O phases. The second challenge is that the translation from multidimensional parallel application data into a progressive linear data format results in an extraordinary degree of noncontiguous data access, both in application memory and within the file system. This access pattern impedes scalability.

We address these challenges in PIDX using two related strategies [4]. First, we introduce a new API that tailors to the needs of HPC application data models. We then introduce an aggregation algorithm that translates the application data model into an efficient movement of data to storage. These strategies are discussed in greater detail in the following subsections.

A. Expressing HPC data models with PIDX

The first prototype implementation of PIDX presented a single `PIDX_write()` function that could be used to write a portion of a multi-dimensional dataset to disk. All data passed into this function must reside in a contiguous, row-major ordered memory region. While sufficient for simple use cases, this API was not flexible enough for more complex real-world applications. In particular, if there were multiple variables, if the variables were strided in memory, or if there were multiple samples per variable (ie, a compound vector), then transferring a complete simulation time step to IDX would require packing data into multiple intermediate buffers and issuing multiple `PIDX_write()` operations. This approach not only introduces small I/O operations to the storage system, but also limits the scope of potential I/O optimizations.

Listing 1. Enhanced PIDX API example

```
/* define variables across all processes */
var1 = PIDX_variable_global_define("var1",
    samples, datatype);

/* add local variables to the dataset */
PIDX_variable_local_add(dataset, var1,
    global_index, count);

/* describe memory layout */
PIDX_variable_local_layout(dataset, var1,
    memory_address, datatype);

/* write all data */
PIDX_write(dataset);
```

To overcome this problem, we devised an enhanced API that decouples the definition of the application's data model from the actual transfer of data to storage. This technique has proven successful in a variety of existing high level libraries such as HDF5 [1] and pNetCDF [10]. The PIDX API allows applications to store a collection of dense multidimensional variables. Each variable has its own type and can be written from an arbitrary memory layout on each process. An example of the use of this enhanced PIDX API is shown in Listing 1. The first step is to define a variable. This includes an indication of how many samples make up an instance of the variable and what MPI datatype will represent the variable. The second step is to add the variable to the dataset if the local process uses that variable. The third step is to describe how the variable is laid out in local memory, including any striding information. These initial three steps can be repeated as needed to describe all variables to be included in an IDX dataset. The final phase is the `PIDX_write()` command, which tells the library to transfer the entire dataset to storage.

This organization allows PIDX to leverage as much concurrency as possible by taking all variables into account simultaneously. It also allows PIDX to reuse HZ calculations where possible for variables that share the same dimensions. In the following subsection we will explore aggregation optimizations that take advantage of this global view of the dataset as well.

B. Aggregation strategies

Once an HZ ordering is calculated in PIDX, each process must write its local data into appropriate regions in the underlying IDX dataset. The IDX data is divided into a collection of files in a directory hierarchy to promote efficient caching during visualization [17]. These files are further broken down into blocks that each contain a contiguous set of HZ ordered samples. The blocks are sparsely populated according to the dimensions of the dataset. Each block may contain samples from many different processes in an HPC simulation, especially in the initial low resolution HZ levels. As a result, each process holds a variety of data elements that are destined for noncontiguous file regions. This access pattern problem is exacerbated if the data dimensions are not a power of two due to the fact that the IDX calculation will result in a sparse linear ordering.

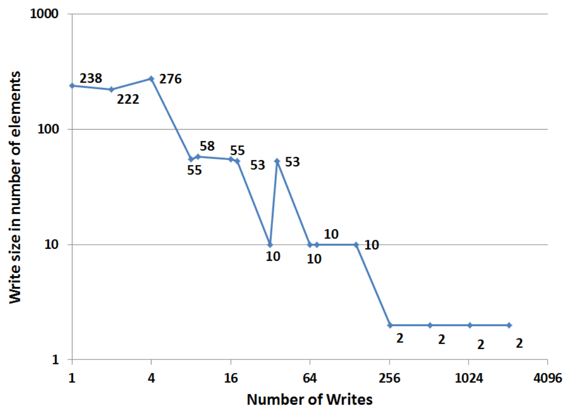

Figure 1. Histogram of file accesses produced when a single process writes a 22 x 36 x 22 volume without aggregation. Each element is 8 bytes.

Figure 1 illustrates the noncontiguous nature of the file accesses required to write a non power of two dataset using PIDX. This example shows a histogram of the number of contiguous elements (i.e. samples) in each underlying write operation needed to transfer a single 22 x 36 x 22 volume. Each element is an 8 byte double precision floating point number, yielding a total data size of roughly 136 KiB. The entire volume is written by a single process. In this example, the two largest write accesses contain 2048 elements (16 KiB), but smaller file accesses are much more common. There are 736 writes that transfer 3 elements or less to storage. This type of small write access pattern is known to scale poorly, especially on leadership-class storage systems [2].

One solution to this problem is to aggregate data before writing by using a two-phase I/O strategy [6]. In an MPI environment, this can usually be accomplished by simply issuing MPI-IO collective operations [18]. However, this approach is not applicable in PIDX. Each IDX dataset is broken into a (possibly large) collection of smaller underlying files and MPI-IO only allows collective writes to one file at a time within a given communicator. It is therefore infeasible to fully express all possible concurrent operations using MPI-IO. We instead designed an aggregation algorithm tailored specifically for IDX data sets, in which a subset of processes are responsible for combining data from all

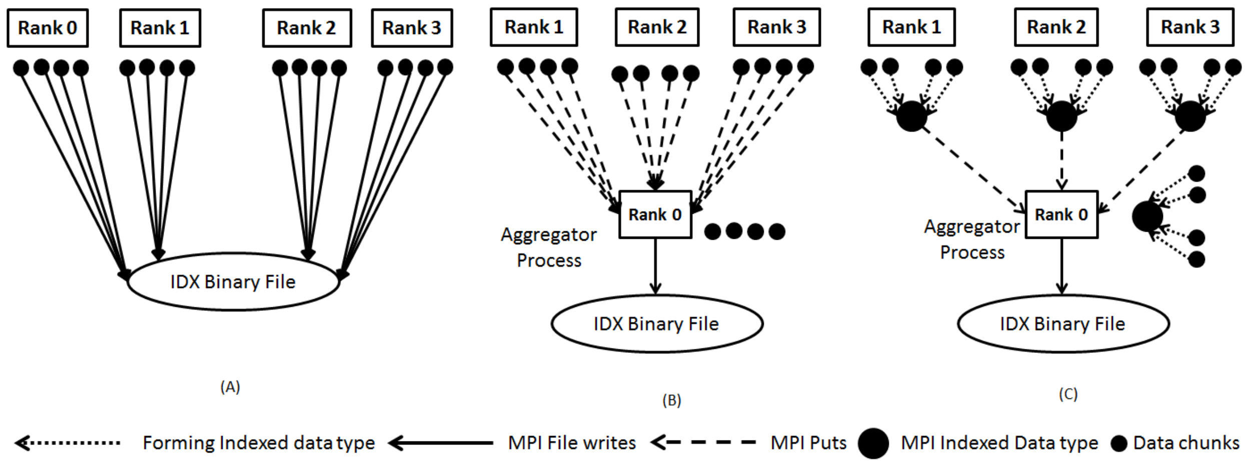

Figure 2. Schematic diagram of PIDX aggregation strategies: (A) No aggregation (B) Aggregation Implemented with RMA (C) Aggregation Implemented with RMA and MPI_Datatypes

processes into large contiguous buffers before writing to storage.

The first step in tailoring aggregation to IDX is to select appropriate aggregators. We chose aggregator processes such that each one is responsible for writing all the data corresponding to a single IDX variable to a given binary file. For example, a single variable dataset of dimensions 256^3 would produce 16 underlying files when using the default IDX parameters. 16 aggregators are therefore used to write the files, regardless of the total number of processes contributing data. If there were three variables, then there would be three aggregators per file, bringing the total number of aggregator processes to 48. The aggregators are distributed evenly among all MPI ranks in order to maximize throughput in cases where adjacent ranks share I/O resources such as forwarding nodes. The buffer size of each aggregator can be tuned using IDX-specific parameters; the default configuration results in a 64 MiB aggregation buffer for each aggregator when using double precision floating point variables. This strategy must be adjusted in corner cases where the dataset parameters would require more aggregators than can be satisfied by the job size, but the default organization offers several key advantages. Metadata overhead is minimized by having each aggregator write to no more than one file. All writes can also be perfectly block aligned in storage by shifting the offset of each variable and adjusting the IDX header information accordingly.

The second step in designing a custom aggregation algorithm for IDX was to choose the communication mechanism for transferring data from each process to the appropriate aggregator. We elected to use MPI one-sided communication for this purpose, with each aggregator presenting an RMA window in which to collect data. The clients place data directly into appropriate remote buffer locations according to a global view of the data set. This has two notable advantages over point to point communication in this context. The first is that it avoids redundant computation. The client can reuse the results of its HZ ordering calculation to determine where to write each contiguous set of samples without involving the aggregators. This leads to a second advantage, in that each process can govern how much data to transmit with each `MPI_Put()` operation according to the nature of the local IDX data and the complexity of the resulting memory access pattern. Datatypes and buffers can be constructed iteratively and broken into segments according to HZ level boundaries, datatype size, or data buffer size with no additional synchronization. `MPI_Win_fence()` is used for synchronization once all transfers are complete.

Figure 2 illustrates the evolution of the aggregation strategies used within PIDX. In Figure 2(A), each process writes its own data directly to the appropriate underlying IDX binary file, leading to a large number of small accesses to each file. In Figure 2(B), each process uses `MPI_Put()` operations to transmit each contiguous data segment to an intermediate aggregator. Once the aggregator's buffer is complete then the data is written to disk using a single large I/O operation. In Figure 2(C) we go one step further, by bundling several noncontiguous memory accesses from each process into a single `MPI_Put()` using MPI indexed datatypes. This approach reduces the number of small network messages needed to transfer data to aggregators.

C. *PIDX I/O phases*

The data model and I/O aggregation strategy presented in the preceding subsections are just two steps in the complete process of writing an IDX dataset. The entire process can be summarized as follows:

1) Describe data model (see Section III-A)
2) Create an IDX block bitmap
3) Create underlying file and directory hierarchy
4) Perform HZ encoding
5) Aggregate data (see Section III-B)
6) Write data to storage

The creation of the IDX block bitmap is a critical component of writing a large IDX dataset in parallel. This bitmap indicates which IDX blocks must be populated in order to store an arbitrary N-dimensional dataset. The maximum number of IDX blocks can be determined trivially by rounding up the global dimensions to the nearest power of two and dividing by the number of samples per block. This is only an upper bound, however. We can limit the number of files and the size of those files (especially for datasets that are just over a power of two boundary) by calculating exactly which blocks must be populated. This calculation is performed up front and stored in a bitmap indicating which blocks are used in the IDX dataset. This block bitmap is used for three purposes: to determine what files and directories need to be created within the IDX dataset, to generate the header information indicating the location of each block within each file, and to determine the correct file offset for each HZ buffer that is written to storage. In order to generate the block bitmap, the maximum number of blocks is calculated first. An inverse HZ computation is done for the starting and the ending HZ addresses of all the potential data blocks. This step yields the bounding box in x, y and z coordinates for each block. A block will contain data corresponding to the global volume only if there is an intersection between this bounding box and and the data being written by the application.

The IDX file and directory hierarchy is created by the rank 0 process in the application before any I/O is performed. In future, we plan to distribute this task among more processes in order to parallelize this activity. The HZ encoding step is performed independently on each process. In order to minimize memory access complexity, all samples are copied into intermediate buffers in a linear Z ordering. Note that the Z ordered data from each process may span multiple HZ levels or multiple files within the overall IDX dataset. Aggregation is performed as described in Section III-B. The final step is to write the HZ ordered data to disk. This step is performed using independent MPI-IO write operations. Due to explicit aggregation within PIDX there is no need for collective I/O or derived data types at this step.

IV. Evaluation

Our ultimate goal in this section is to evaluate the performance of PIDX to directly write IDX datasets for each time-step of the S3D combustion simulation.

S3D is a continuum scale first principles direct numerical simulation code which solves the compressible governing equations of mass continuity, momenta, energy and mass fractions of chemical species including chemical reactions. The computational approach in S3D is described in [3]. In the S3D code, each rank is responsible for a piece of the three-dimensional domain; all MPI ranks have the same number of grid points and the same computational load under a Cartesian decomposition. S3D has been run successfully on up to near the full size (216,000 cores) of the NCCS XT5 jaguarpf, demonstrating near linear scaling up to approximately half of the machine (approximately 120k ranks) with the current validated production code base in the weak scaling limit. Similarly, it has demonstrated near linear scaling to 120,000 cores on the CrayXE6, Hopper2, at NERSC. S3D can be compiled with support for several I/O schemes which are then selected at runtime via a configuration parameter. S3D I/O extracts just the portion of S3D concerning restart dumps, allowing us to focus exclusively on I/O characteristics. For our evaluation, we used an S3D I/O configuration wherein each process produced a 64^3 volume consisting of 4 fields (field 1 and 2 each of just 1 sample, field 3 with 3 samples and field 4 with 11 samples). This produces 32 MiB of data per process. Because PIDX is implemented in C, we developed a wrapper to facilitate the use of PIDX with S3D, which is a Fortran code. We also incorporated a custom I/O module in S3D to enable the use of PIDX as an alternative to the existing I/O schemes.

Unless otherwise noted, all experiments presented in this work were performed on the Hopper2 system at NERSC. Hopper2 has a peak performance of 1.28 Petaflops/sec, 153,216 processors cores for running scientific applications, 212 TB of memory, and 2 Petabytes of online disk storage. Hopper2's compute nodes are connected via a custom high-bandwidth, low-latency network provided by Cray. The connectivity is in the form of a *mesh* in which each node is connected to other nearby nodes like strands in a fishing net. Each network node not only handles data destined for itself, but also relays data to other nodes. The *edges* of the mesh network are connected to each other to form a *3D torus*. The custom chips that route communication over the network are know as *Gemini* and the entire network is often referred to as the *Cray Gemini Network*. The Hopper2 system has access to five different file systems which provide different levels of disk storage, I/O performance, and file permanence. Two identically configured Lustre file systems provide 25 GiB/s scratch storage for large I/O intensive jobs. We used one of these scratch file systems as the target for all benchmark datasets in our evaluation. This same file system is also used for production S3D jobs on hopper, although jobs typically use Fortran I/O as their output method.

The Lustre scratch file system used in our evaluation consisted of 26 I/O servers, each of which provides access to 6 object storage targets (OSTs). The compute node network is connected to the servers via QDR Infiniband, and the servers are in turn connected to 13 LSI 7900 disk controllers via Fibre Channel 8. Our experiments used the default Lustre striping parameters on Hopper2 in which each file is striped across two OSTs.

We first investigate the impact of PIDX file parameters on the achievable throughput. Next, we evaluate the efficacy of the I/O optimization strategies for PIDX. Finally, we compare the weak scaling performance of PIDX and Fortran I/O for S3D I/O.

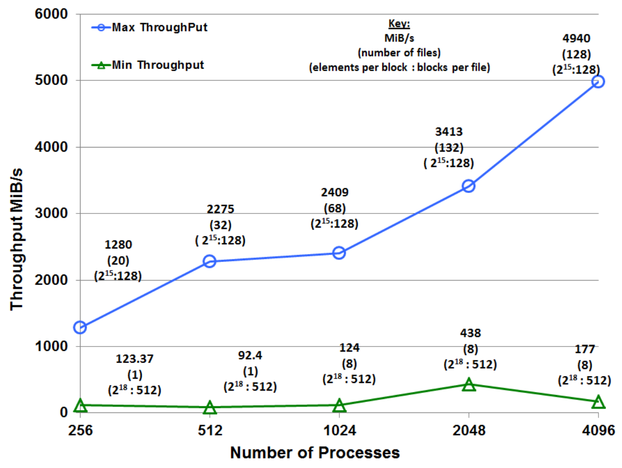

Figure 3. Impact of PIDX file parameters on achievable throughput. We see that PIDX performs best on Hopper2 with a larger number of aggregators and smaller file sizes.

A. Impact of PIDX file parameters

There are two main file parameters in PIDX that can be varied to produce the same IDX dataset with different numbers of files and different data distributions. The *elements_per_block* parameter controls the number of samples that constitute an IDX block, while the *blocks_per_file* parameter controls the number of blocks that constitute a data file within IDX. Both can be increased in order to reduce the total number of files and enlarge the size of each file. These parameters can be used to tune PIDX to reflect the characteristics of the underlying file system, as some file systems are optimized towards performance for large numbers files or performance for single shared files.

To understand the impact of these parameters, we wrote a micro-benchmark to write out a 3D volume using PIDX. In our evaluations on Hopper2, we vary the number of processes from 256 to 4096. Each process writes a 64^3 sub-volume of double precision floating point data to generate a total volume of 512 MiB (256 processes) and 4 GiB (4096 processes). We varied *elements_per_block* from 2^{15} to 2^{18} and varied *blocks_per_file* to 128, 256 and 512. Varying these parameters have a bearing on the aggregation time spent during RMA communication as well as the file write time. This means that it will take a different amount of time to generate two IDX file with same *box* but different *elements_per_block* or *blocks_per_file*.

Figure 3 depicts the maximum and minimum throughputs achieved as we vary the tuning parameters. For each data point we label the performance, the number of files generated, and the *elements_per_block* and *blocks_per_file* parameters that were used to achieve that result. As the number of files increases (smaller *blocks_per_file*), we achieve a noticeable speed up for two reasons. First, the number of aggregators is increased. Secondly, the underlying Lustre file system performs better as data is distributed across a larger number of files. We believe our design is flexible enough to be tuned to generate small number of large shared files or a large number of files depending on which is optimal for the target system. We used 128 *blocks_per_file* and 2^{15} *elements_per_block* in all further evaluations in this work.

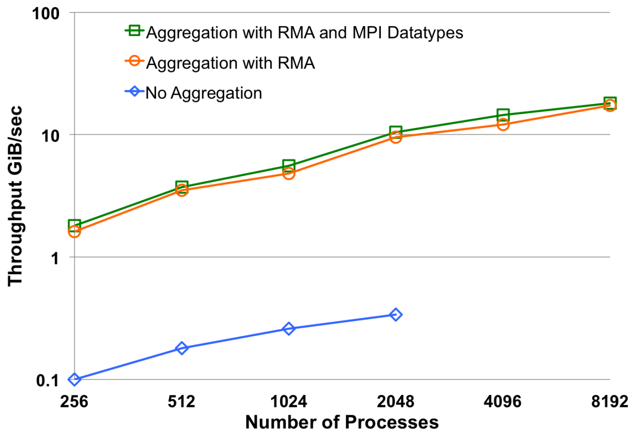

Figure 4. Throughput comparison of all the versions of the API

B. Efficacy of Aggregation on PIDX I/O performance

Figure 4 depicts performance of the aggregation mechanism in PIDX to write 10 time-steps in S3D I/O as we scale the number of processes from 256 to 8192. Each process writes out a $(64)^3$ sub-volume with 4 variables. Three cases are shown: one in which no aggregation is performed and all processes write directly to storage with MPI independent I/O, one in which aggregation is performed using a separate `MPI_Put()` operation for each contiguous region, and one in which MPI datatypes are used to transfer multiple regions using a smaller number of `MPI_Put()` operations. Aggregation with MPI datatypes yields a significant speed up for I/O performance in comparison to a scheme that uses no aggregation. At 256 processes, we achieve up to a 18-fold speed up, and at 2048 processes, we achieve up to 30-fold speed up over a scheme with no aggregation.

The aggregation strategy that utilized MPI datatypes yielded a 20% improvement over the aggregation strategy that issued a separate `MPI_Put()` for each contiguous region. An indexed datatype was used to describe all transfers to a given aggregator at each HZ level. The reason for the performance improvement is that the datatype reduced the number of small messages transferred during aggregation, therefore reducing network congestion. In future work we believe that this could be optimized further by creating datatypes that span multiple HZ levels.

C. Weak Scaling Performance

We evaluate the weak scaling performance of S3D I/O using PIDX and Fortran I/O as we vary the number of processes from 256 to 8192. In each run, S3D I/O wrote out 10 time-steps wherein each process contributed a 64^3

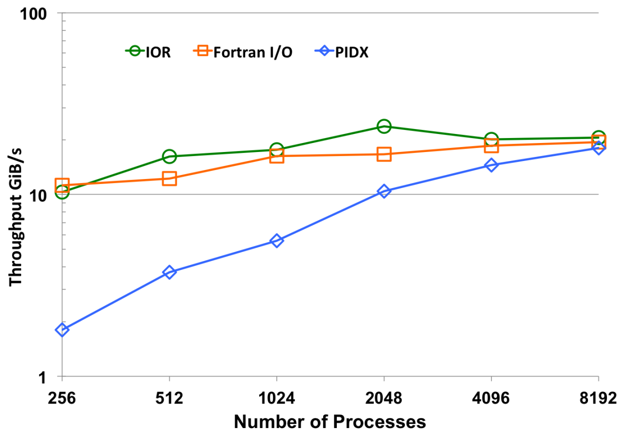

Figure 5. Weak scaling of I/O mechanisms including PIDX and Fortran I/O

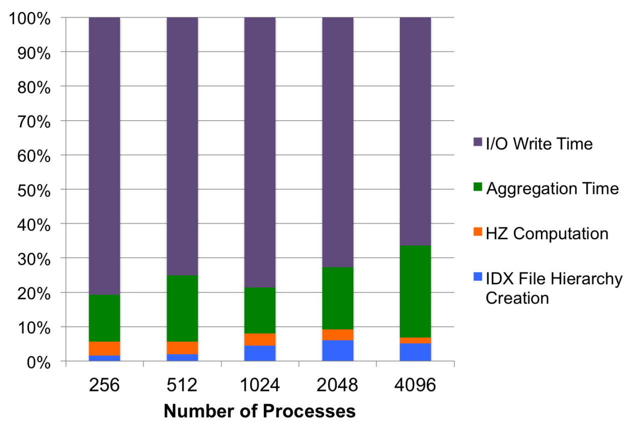

Figure 6. The proportion of time taken by the various I/O components as we scale from 256 processes (8 GiB) to 4096 processes (128 GiB)

block of double precision data (32 MiB) consisting of 4 variables. The variables were pressure, temperature, velocity (3 components) and species (11 components). In order to provide a baseline for the results obtained from PIDX and Fortran I/O, we also executed IOR tests for each process count. IOR is a general-purpose parallel I/O benchmark [16] which we configured in this case to produce a similar access pattern to that generated by S3D with Fortran I/O: each process writes one complete file without fsync. This should achieve near-optimal performance for the Lustre file system. The IOR performance gives us a measure of the maximum performance achievable for the filesystem. The S3D PIDX, S3D Fortran I/O, and IOR weak scaling results are shown in Figure 5. From the figure we see that at 8192 processes, PIDX achieves a maximum I/O throughput of 18 GiB/s (90% of the IOR throughput). As we increase the number of processes, the performance of PIDX increases as the number of files increases leading to more aggregators participating in I/O. IOR and Fortran I/O achieve similar throughput for all the process counts. This is primarily due to the fact that Fortran I/O in S3D behaves similarly to IOR test case with each process populating a unique output file. Although PIDX is not as fast as Fortran I/O for this benchmark, we can see that the results are converging towards a comparable peak performance at larger process sizes while offering significant advantages over unstructured Fortran I/O data in terms of visualization capability.

Figure 6 depicts the time taken by the various PIDX I/O components - IDX file hierarchy creation, HZ computation, aggregation time and the I/O write time. As we scale the number of processes from 256 to 4096, the amount of data written increases from 8 GiB per time-step to 128 GiB per time-step. As the IDX data size increases with the number of processes, the number of binary files required to represent the data increases. This leads to an increase in time needed to create the IDX file hierarchy. We believe that this time could be reduced by distributing the creation of the file hierarchy across a larger number of nodes, which we will confirm in future work. HZ computation is embarrassingly parallel and takes 0.17 secs as we weak scale the number of processes. We observe that the percentage of total I/O time spent on data aggregation increases with the process count due to the increasing volume of network traffic. The actual I/O write time dominates the total time. The write phase by itself achieves rates that are similar to peak rates measured with IOR, however. At 4096, if we consider just the I/O write phase performance, then we achieve 19.7 GiB/s (96% of the IOR performance).

V. Visualization Pipeline

In addition to the efficient file format, ViSUS has been shown to aid in the interactive exploration of very large datasets [13], [17], [15]. ViSUS has been extended to support a client-server model in addition to the traditional viewer. The ViSUS server uses HTTP (a stateless protocol) in order to support many clients. A traditional client/server infrastructure, where the client established and maintained a stable connection to the server, can only handle a limited number of clients robustly. Using HTTP, the ViSUS server can scale to thousands of connections.

In a typical ViSUS client-server session, the client only requests the particular region needed for display. Due to the resolution of modern displays this data is very small compared to the total size of the data and optimized to minimize latency. The Visus client keeps a number (48) of connections alive in a pool using the “keep-alive” option of HTTP. The use of lossy or lossless compression is configurable by the user. For example, ViSUS supports JPEG and EXR for lossy compression of byte and float data respectively. The ViSUS server is an open client/server architecture, therefore possible to port the server to any web server which supports a C++ module plugin (i.e. apache, IIS).

Ideally, a scientist would like to view a simulation as it is computed, in order to steer or correct the simulation

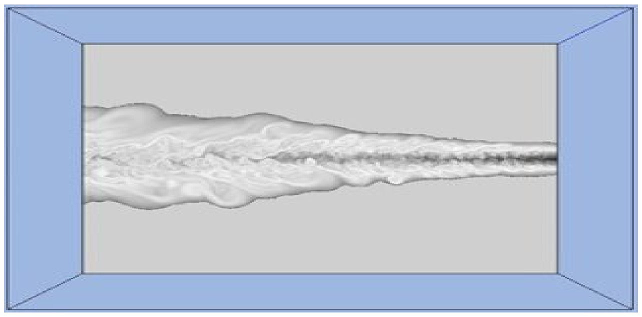

Figure 7. A 2D slice of a S3D simulation time step seen through a remote ViSUS client.

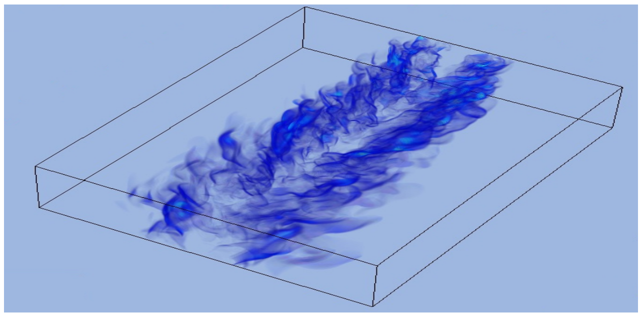

Figure 8. A volume render of an S3D simulation time step seen through a remote ViSUS client.

as unforeseen events arise. A single time-step from the S3D simulation is approximately 128 GB in size, which would be infeasible to transfer offsite. For example, in our testing the transfer of a single time-step from the Hopper2 system to our local file server took approximately 3 hours 38 minutes. In the time to transfer a single time-step, the user-scientist would have lost any chance for significant steering or correction. By using the IDX format in the simulation checkpointing, we can link this data directly with an Apache server using a ViSUS plug-in running on a log-in node for the cluster system. ViSUS can handle missing or partial data, therefore allowing the visualization of the data even as it is being written to disk by the system.

ViSUS supports a wide-variety of clients: a stand-alone application, a web-browser plug-in, or an iOS application for the iPad or iPhone. Therefore an application scientist can monitor an active simulation on practically any system without any need to transfer the data off of the computing cluster. Figures 7 and 8 show a volume render and a 2-dimensional slice of a S3D simulation time-step seen through a ViSUS stand-alone remote client. The data can be viewed remotely without moving the large simulation data off of the high-performance system.

VI. Related Work

Parallel scientific simulations often produce large volumes of data, and a variety of high level libraries have been developed to aid in structuring that data and accessing it efficiently. The Hierarchical Data Format (HDF) is one such library that allows developers to express data models in a hierarchical organization [1]. Parallel NetCDF [10] is another popular high level library with similar functionality to HDF5 but in an file format that is compatible with serial NetCDF from Unidata. Both HDF5 and PNetCDF are implemented atop MPI and MPI-IO functionality. They both leverage MPI-IO collective I/O operations for data aggregation. While HDF5 and PnetCDF are both more general-purpose and robust than PIDX, the chief advantage of PIDX is that it reorders each individual sample to in a manner that enables real-time multi-resolution visualization. PIDX is not intended to serve as a general purpose data format, but it may be suitable for use within HDF5 or PNetCDF as an alternative data layout. ADIOS is another popular library used to manage parallel I/O for scientific applications [11]. One of the key features of ADIOS is that it decouples the description of the data and transforms to be applied to that data from the application itself. ADIOS supports a variety of back-end formats and plug-ins that can be selected at run time.

Previous work in multi-resolution data formats for parallel processing environments include work from Chiueh and Katz in developing a multi-resolution video format for parallel disk arrays [5]. They leveraged Gaussian and Laplacian Pyramid transforms that were tailored to the underlying storage architecture to improve throughput. Their work focused on read-only workloads as opposed to write-heavy workloads. Chaoli, Jinzhu and Han have presented work in parallel visualization [19] of multiresolution data. Their algorithm involves conversion of raw data to a multiresolution wavelet tree and focuses more on parallel visualization rather than a generic data format as in PIDX. Many researchers previously have proposed various techniques for multiresolution encoding and rendering of large scale volumes [8], fewer studies were focused on designing parallel algorithms for generating data itself in a more interactive usable format like the IDX. The mechanism that PIDX uses to split parallel I/O data into multiple files is similar to the subfiling scheme implemented in PnetCDF [7]. That approach performs aggregation using default MPI-IO collective operations on independent communicators rather than a custom aggregation algorithm. Memik et al. also proposed a generic method for aggregating I/O to multiple files called Multicollective I/O [12]. Unlike Multicollective I/O, our approach does not expose the underlying files to the application nor does it require separate datatypes for each file.

VII. Conclusion and Future Work

There is a growing gap between data formats written out by simulations optimized for write performance and the formats required by analysis tools optimized for read performance. This gap is likely to increase as storage systems are unable to keep up with the volume of data being produced by simulations. PIDX is an effective tool to help bridge this gap by efficiently producing a cache oblivious, multi-resolution data format. In this paper, we elucidate the PIDX API design and implementation. We describe the efficacy of PIDX to perform I/O for S3D combustion application. We present PIDX performance up to 8,192 cores of the Hopper 2 supercomputer. PIDX achieves up to 18 GiB/s - 90% of the achievable IOR bandwidth. Finally, we demonstrated an end-to-end visualize pipeline to interactively visualize the datasets written out by the S3D simulation. For future work, we plan to design a tuning layer in PIDX to optimize its performance for parallel file systems including Lustre, PVFS and GPFS.

VIII. Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357 and an Argonne National Laboratory Director Fellowship. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] HDF5 home page. http://www.hdfgroup.org/HDF5/.

[2] P. Carns, K. Harms, W. Allcock, C. Bacon, R. Latham, S. Lang, and R. Ross. Understanding and improving computational science storage access through continuous characterization. In *Proceedings of 27th IEEE Conference on Mass Storage Systems and Technologies (MSST 2011)*, 2011.

[3] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. M. Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using s3d. In *Computational Science and Discovery Volume 2*, January 2009.

[4] A. Ching, A. Choudhary, W. K. Liao, R. Ross, and W. Gropp. Evaluating structured I/O methods for parallel file systems. *International Journal of High Performance Computing and Networking*, 2:133–145, 2004.

[5] T.-c. Chiueh and R. H. Katz. Multi-resolution video representation for parallel disk arrays. In *Proceedings of the first ACM international conference on Multimedia*, MULTIMEDIA '93, pages 401–409, New York, NY, USA, 1993. ACM.

[6] J. M. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel i/o via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21:31–38, December 1993.

[7] K. Gao, W. keng Liao, A. Nisar, A. Choudhary, R. Ross, and R. Latham. Using subfiling to improve programming flexibility and performance of parallel shared-file i/o. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 470 –477, sept. 2009.

[8] S. Guthe, M. Wand, J. Gonser, and W. Strasser. Interactive rendering of large volume data sets. In *Proceedings of the conference on Visualization '02*, VIS '02, pages 53–60, Washington, DC, USA, 2002. IEEE Computer Society.

[9] S. Kumar, V. Pascucci, V. Vishwanath, P. Carns, R. Latham, T. Peterka, M. Papka, and R. Ross. Towards parallel access of multi-dimensional, multiresolution scientific data. In *Proceedings of 2010 Petascale Data Storage Workshop*, November 2010.

[10] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of SC2003: High Performance Networking and Computing*, Phoenix, AZ, November 2003. IEEE Computer Society Press.

[11] J. Lofstead, S. Klasky, S. K., N. Podhorszki, and C. Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). June 2008.

[12] G. Memik, M. T. Kandemir, W. Liao, and A. Choudhary. Multicollective I/O: a technique for exploiting inter-file access patterns. *Transactions on Storage*, 2(3):349–369, 2006.

[13] V. Pascucci and R. J. Frank. Global static indexing for real-time exploration of very large regular grids. In *Conference on High Performance Networking and Computing archive proceedings of the ACM/IEEE conference on Supercomputing (CDROM)*, 2001.

[14] V. Pascucci and R. J. Frank. Hierarchical indexing for out-of-core ccess to multi-resolution data. In *Technical Report UCRL-JC-140581, Lawrence Livermore National Laboratory*, 2001.

[15] V. Pascucci, D. Laney, R. J. Frank, G. Scorzelli, L. Linsen, B. Hamann, and F. Gygi. Real-time monitoring of large scientic simulations. In *ACM Symposiumon Applied Computing03,ACMPress.*, 2003.

[16] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *Proceedings of Supercomputing*, November 2008.

[17] B. Summa, G. Scorzelli, M. Jiang, P.-T. Bremer, and V. Pascucci. Interactive editing of massive imagery made simple: Turning atlanta into atlantis. *ACM Trans. Graph.*, 30:7:1–7:13, April 2011.

[18] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, May 1999.

[19] C. Wang, J. Gao, L. Li, and H.-W. Shen. A multiresolution volume rendering framework for large-scale time-varying data visualization. In *Volume Graphics, 2005. Fourth International Workshop on*, pages 11 – 223, june 2005.