Efficient Data Restructuring and Aggregation for I/O Acceleration in PIDX

Sidharth Kumar,* Venkatram Vishwanath,† Philip Carns,† Joshua A. Levine,* Robert Latham,† Giorgio Scorzelli,* Hemanth Kolla,‡ Ray Grout,§ Robert Ross,† Michael E. Papka,† Jacqueline Chen,‡ Valerio Pascucci*

*Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT, USA
†Argonne National Laboratory, Argonne, IL, USA
‡Sandia National Laboratories, Livermore, CA, USA
§National Renewable Energy Laboratory, Golden, CO, USA

Abstract—**Hierarchical, multiresolution data representations enable interactive analysis and visualization of large-scale simulations. One promising application of these techniques is to store high performance computing simulation output in a hierarchical Z (HZ) ordering that translates data from a Cartesian coordinate scheme to a one-dimensional array ordered by locality at different resolution levels. However, when the dimensions of the simulation data are not an even power of 2, parallel HZ ordering produces sparse memory and network access patterns that inhibit I/O performance. This work presents a new technique for parallel HZ ordering of simulation datasets that restructures simulation data into large (power of 2) blocks to facilitate efficient I/O aggregation. We perform both weak and strong scaling experiments using the S3D combustion application on both Cray-XE6 (65,536 cores) and IBM Blue Gene/P (131,072 cores) platforms. We demonstrate that data can be written in hierarchical, multiresolution format with performance competitive to that of native data-ordering methods.**

I. Introduction

Techniques for I/O of large-scale simulation data have strong demands, requiring scalability for efficient computation during simulation as well as interactivity for visualization and analysis. Hierarchical representations have been shown to provide this level of progressive access in an efficient, cache-oblivious manner. One such format, IDX, relies on a hierarchical Z order (HZ order) that reorders standard row-major data into a one-dimensional format based on locality in any arbitrary dimension, at various level-of-detail resolutions [1], [2]. While successful APIs such as ViSUS have adopted IDX to fields such as digital photography and visualization of large scientific data [3], [1], these are serial in nature. Extending them to parallel settings, the Parallel IDX (PIDX) library [4], [5] provides efficient methods for writing and accessing standard IDX formats by using a parallel API.

The levels of resolution in HZ order are logically based on even powers of 2, typical of level-of-detail schemes. If the original dataset is a power of 2 in each dimension (i.e., $2^x \times 2^y \times 2^z$), then the resulting ordering will be dense. However, many scientific simulations, such as S3D [6], Flash [7], and GCRM [8], do not normally produce datasets with even, power-of-2 dimensions. For clarity in exposition throughout the paper, we will use the term *irregular* to describe datasets that have non-power-of-2 dimensions (e.g., $22 \times 36 \times 22$) and *regular* to describe datasets that have power-of-2 dimensions (e.g., 32^3).

Naive parallel HZ encoding of both regular and irregular datasets results in noncontiguous file access. This problem was addressed in previous work [5] with aggregation strategies that perform some amount of network I/O before disk I/O. However, parallel IDX encoding of irregular datasets results in an additional challenge: not only is the file access noncontiguous, but the memory buffer each process used for encoding is sparse and noncontiguous as well. Even if aggregation is used to improve the file access pattern, the memory buffer access pattern results in too many small messages sent on the network during aggregation and too much wasted memory on each process for PIDX to achieve acceptable performance for irregular datasets.

In this paper we present an algorithm to enable efficient writing of irregular IDX datasets from a parallel application. This algorithm extends the two-phase writing (aggregation and disk I/O) used for regular data [5] with a three-phase scheme that restructures irregular data in preparation for aggregation. In our work we have used both synthetic microbenchmarks and the S3D production combustion simulation code to evaluate PIDX on irregular data. We also perform a case study to understand the effects of two of the important parameters that influence the performance of PIDX API - the power-of-2 block size used for restructuring and the number of blocks per file. We present a comparison of PIDX performance across two leadership systems: Hopper, a Cray XE6 at NERSC, and the Intrepid Blue Gene/P at Argonne National Laboratory. We compare the performance of PIDX with other leading popular output methods such as PnetCDF and Fortran I/O for a production application. Our results show both weak and strong scaling at 65,536 processes (Hopper) and 131,072 processes (Intrepid). The results also illuminate some of the fundamental challenges involved with performing efficient disk I/O with a minimal amount of network overhead.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

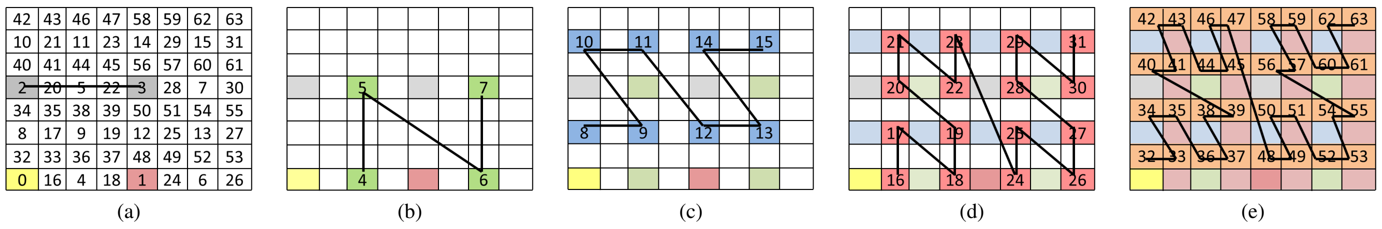


Fig. 1. HZ ordering (indicated by box labels and curves) at various levels for an 8x8 dataset. (a) levels 0, 1, and 2. (b)-(e) levels 3-6, respectively.

II. Related Work

Several I/O libraries are available to aid in structuring and efficiently accessing large-scale scientific simulation data. Parallel NetCDF [9] and HDF5 [10] are two prominent examples. The library HDF5 allows developers to express data models in a hierarchical organization. While HDF5 and PnetCDF are both more general-purpose and more widely used than PIDX, the chief advantage of PIDX lies in its ability to enable real-time, multiresolution visualization. Both HDF5 and PnetCDF leverage MPI-IO collective I/O operations for data aggregation. PIDX instead has a custom MPI one-sided communication phase for data aggregation. In PIDX parallel I/O data is split across multiple files. The number and size of the files can be tuned with the help of some IDX parameters. A similar subfiling scheme was implemented for PnetCDF in [11] with an approach that performs aggregation by using default MPI-IO collective operations on independent communicators rather than a custom aggregation algorithm. ADIOS is another popular library used to manage parallel I/O for scientific applications [12]. A key feature of ADIOS is that it decouples the description of the data and transforms to be applied to that data from the application itself.

Wang, Gao, and Shen have presented work in parallel visualization [13] 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. By comparison, PIDX attempts to solve the multiresolution visualization problem by first solving the storage problem in a way that is amenable to both efficient data I/O and hierarchical data access for analysis.

The PIDX library described in this work maintains full compatibility with the IDX file format used by ViSUS. IDX is a desirable file format for visualizing large-scale high-performance computing (HPC) simulation results because of its ability to access multiple levels of resolution with low latency for interactive exploration [1], [2]. HZ ordering is the key idea behind the IDX data format. An HZ order is calculated for each data element by using the spatial coordinates of that element (see Fig. 1). An HZ level (corresponding to a level of resolution) is assigned to all positions. With each increasing level, the number of elements increases by a factor of 2. Data in an IDX file is written with an increasing HZ order.

Each data element in the IDX format may be a single value (1 data sample) or a compound type, such as a 3D vector (3 data samples). Elements that are contiguous in HZ space are grouped together into fixed-size blocks. IDX blocks are in turn grouped together into a collection of files. The files are distributed over a directory hierarchy with a metadata file at the top level used to describe various properties of the data set, such as the element type and bounding box. In this work, we fix the elements-per-block parameter at 2^{15} but vary the blocks-per-file parameter in order to alter the number of files and level of inter-file concurrency used to write an IDX data set in parallel.

III. Existing techniques for writing IDX data in parallel

In our previous work [4], we demonstrated the use of PIDX in coordinating parallel access to multidimensional, regular datasets, and we explored various low-level encoding and file access optimizations. In a subsequent work [5], we introduced a new API that enabled more efficient expression of HPC data models. We also introduced a two-phase I/O strategy [14]. Each client first performed a local HZ encoding of its own data. The HZ-encoded data was then aggregated to a subset of processes that in turn wrote the data to disk. The aggregation phase transformed the small-sized, suboptimal file accesses from each process into large, contiguous accesses performed by a few chosen aggregator processes.

A. Regular Datasets

The technique for writing IDX data in parallel, described in [5], begins by performing HZ encoding of the local data at each compute node. Each node calculates an *HZ range*, which is the smallest and largest HZ index that will be produced by HZ encoding the local data. The *HZ bound* is the difference between the upper and lower values in the HZ range. Each process allocates enough memory to hold HZ bound elements of data in order to guarantee that sufficient buffer space is available to encode the data, even though this buffer may be larger than the actual amount of data encoded. This approach has a compelling advantage, in that data elements within the resulting HZ buffer map directly to the data in an IDX file, whereas a dense HZ buffer would require explicit, costly tracking of the appropriate file offset for every element.

HZ encoding is logically based on even powers of 2; Hence, for regular datasets, the number data samples after HZ encoding always equals the HZ bound. This is further illustrated in Fig. 2a, which shows parallel conversion of a regular 8×8 2D row major dataset to IDX format using four processes. Each process, indicated by a different color, handles

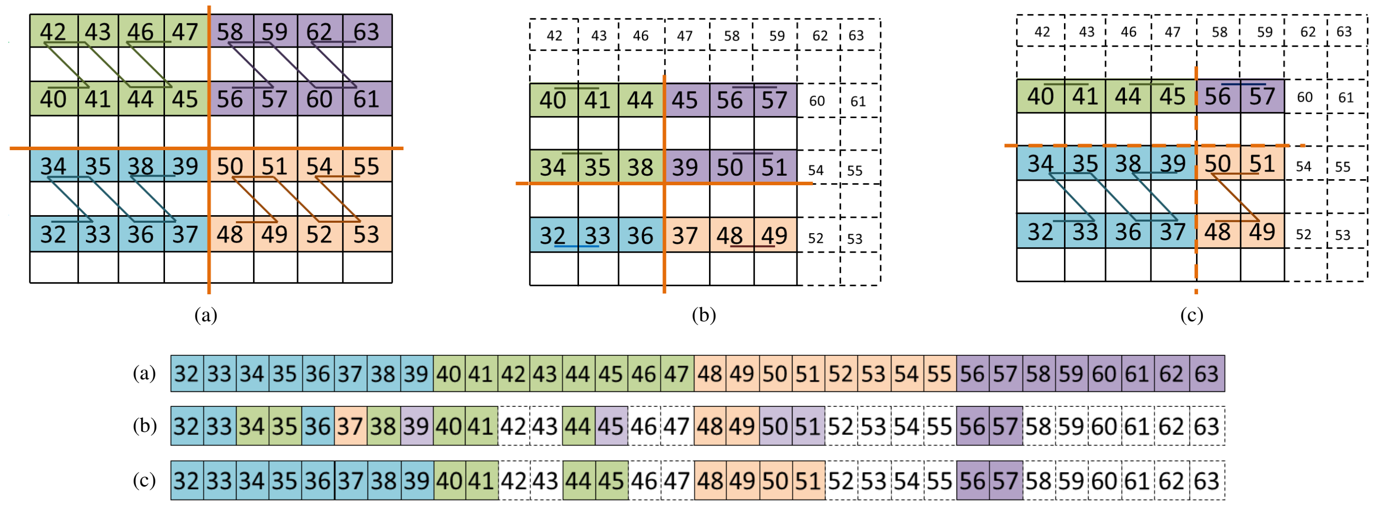


Fig. 2. (a) HZ computation and traversal of a 8 × 8 2D regular data performed in parallel by 4 processes. Numbers in the box stand for HZ indices of the highest resolution, while empty boxes are for lower-resolution data (not shown). Within every process, all HZ indices are spanned by one Z curve, producing a dense and continuous disk layout (below). (b) HZ computation and traversal of a 6 × 6 irregular dataset. Dashed boxes are for nonexistent data points. Since HZ encoding requires dimensions in powers of 2; a sparse, interleaved, and overlapping disk level layout is produced (below). (c) By restructuring the data, the interleaving of each process's data is reduced despite the fact that loads on the boundary have different sizes.

TABLE I
HZ RANGES FOR THE DATA LAYOUTS SHOWN IN FIGURE 2

	HZ Range			Sample # (HZ Bound)		
Process	(a)	(b)	(c)	(a)	(b)	(c)
P0	32:39	32:36	32:39	8(8)	3(5)	8(8)
P1	40:47	34:44	40:45	8(8)	6(11)	4(5)
P2	48:55	39:57	56:57	8(8)	6(19)	2(2)
P3	56:63	37:49	48:51	8(8)	3(13)	4(4)

a 4 × 4 block of data. For the purpose of simplicity and clarity, we have shown only indices for the highest resolution level (HZ level 6). The remaining HZ levels are not shown here; they are represented by empty boxes - the parallel conversion scheme can be similarly applied to the lower HZ levels. From the figure one can see that for all four processes (**P0**, **P1**, **P2**, **P3**), the number of sample points (8) is equal the HZ bound of 8. This is a dense and continuous mapping from the two-dimensional Cartesian space to the HZ space. This mapping ensures allocation of a minimal memory (8 samples) to store the HZ-encoded data. Table I additionally indicates this in the regular dataset, in which it shows the HZ ranges and bounds are such that each process for the data layout in Fig. 2a requires a tight usage of memory. In addition, the HZ ranges of all processes are always nonoverlapping. This approach provides the opportunity for large, noncontentious I/O operations.

B. Irregular Datasets

Unlike regular datasets, the HZ encoding of irregular local data produces a sparse, noncontiguous HZ buffer at each process. Furthermore, the HZ indices stored in that buffer are logically interleaved with the HZ indices stored at other processes. This configuration causes two problems. First, it is wasteful of limited memory resources at each compute node. Second, it leads to inefficient memory access and a high volume of small messages when the PIDX library attempts to aggregate the encoded data for two-phase I/O.

Figure 2b illustrates this by showing a parallel conversion of an irregular 6 × 6 two-dimensional row order data to IDX format by four processes. Each process, indicated by a different color, handles a 3×3 irregular block of data. Since HZ encoding is based on even powers of 2, we must use the closest power-of-2 (regular) box (8 × 8) to compute the HZ indices, matching the HZ indexing scheme in Fig. 2a, but skipping the dashed boxes. Examining the arrangement of data within each process, it is apparent that it does not lie on a single, connected piece of the Z curve. For example, process **P0** has only three sample points, sparsely placed in two pieces. Examining the disk-level layout clearly shows the sparse, noncontiguous and overlapping data pattern of the four processes. In addition, the HZ bounds of the processes (and therefore the amount of memory consumed by each process) is much larger than the actual number of samples that they hold, as summarized in Table I. For the regular layout (a), each process stores 8 samples using a range of memory values. For irregular data (b), each process stores less data but requires a larger range of HZ indices. After restructuring (c), samples are redistributed, but the maximum size of each HZ range shrinks because they are nonoverlapping.

IV. THREE-PHASE I/O

The two-phase aggregation algorithm for PIDX [5] distributes data to a subset of processes after the HZ encoding step. This technique is efficient only in cases in which the local HZ encoding at each process produces a dense buffer. To effectively handle irregular data sets that do not exhibit this characteristic, we must introduce an additional *restructuring phase* to the algorithm before HZ encoding and two-phase

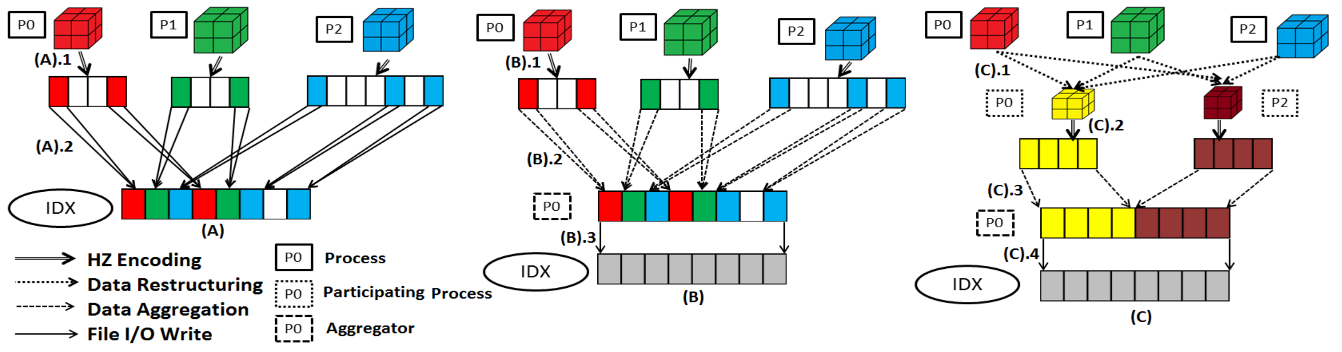


Fig. 3. **(A)** One-Phase I/O: **(A).1** HZ encoding of irregular data set leads to sparse data buffers interleaved across processes. **(A).2** I/O writes to underlying IDX file by each process, leading to a large number of small accesses to each file. **(B)** Two-Phase I/O: **(B).1** HZ encoding of irregular data set leads to sparse data buffers interleaved across processes. **(B).2** Data transfer from in-memory HZ ordered data to an aggregation buffer involving large number of small sized data packets. **(B).3** Large sized aligned I/O writes from aggregation buffer to the IDX file. **(C)** Three-Phase I/O: **(C).1** Data restructuring among processes transforms irregular data blocks at processes P0, P1 and P2 to regular data blocks at processes P0 and P2. **(C).2** HZ encoding of regular blocks leading to dense and non-overlapping data buffer. **(C).3** Data transfer from in-memory HZ ordered data to an aggregation buffer involving fewer large sized data packets. **(C).4** I/O writes from aggregation buffer to a IDX file.

aggregation. This additional restructuring phase alters the distribution of data among processes while the data is still in its initial multidimensional format. The goal is to distribute the dataset such that each process holds a regular, power-of-2 subset of the total volume. This distribution can be performed by using efficient, large, nearest-neighbor messages. Once the data has been structured in this manner, the subsequent local HZ encoding at each process will produce a dense HZ-encoded buffer that can be written to disk efficiently using two-phase aggregation.

Figure 2c, further elucidates the effect of the data restructuring phase. Comparing Figure 2c with Figure 2b, one can see how the restructuring phase transforms the dimensions of process **P0**, from a *(3 × 3) irregular* box to a *(4 × 4) regular* box. The HZ encoding of data in this layout produces an efficient, dense memory layout. The only exceptions are relatively small boundary regions of the data, which do not fit into even power-of-2 subvolumes; but even in that case, the restructuring at least eliminates interleaving of HZ-encoded data across processes.

Figure 3 illustrates the three I/O phases in detail, compared with the one-phase or two-phase I/O. Data-restructuring is a localized phase involving only neighboring processes, at the end of which we have a set of processes handling regular boxes. With this scheme of data-restructuring, barring a few boundary blocks, almost every irregular data block is transformed into a regular one. In our current scheme we use independent I/O of IDX block sizes to write the irregular edge blocks. Our implementation is both scalable and efficient because of the localized nature of communication among processes. For any any imposed regular box, only a group of neighboring process can participate in the restructuring phase.

To restructure data, we impose a virtual grid of regular boxes over the entire volume set. These regular boxes are then used to redistribute the irregularly stored data on each process. To resolve the different boundaries, we judiciously exchange data among processes. We use point-to-point MPI communication (using MPI_Irecv and MPI_Isend) to transfer data among processes. After this data-restructuring, we end up with a grid of regular data boxes stored on some subset of processes.

Algorithm 1 Data Restructuring

```
 1: All-to-all communication of data extents.
 2: Compute dimension of regular box.
 3: Compute All intersecting regular boxes.
 4: for All intersecting regular boxes do
 5:     Compute all other intersecting processes.
 6:     for All intersecting processes do
 7:         Assign receiver and sender process set.
 8:         Find offset and counts of transferable data chunks.
 9:     end for
10:     Proceed with data communication.
11: end for
```

The pseudocode of the restructuring algorithm is given in Algorithm 1. The process consists of four steps. Initially, each process communicates its data extents (line 1) using MPI_Allgather to exchange data-extents information (global offset and count of data it is handling). In doing so, each process builds a consistent picture of the entire dataset and where each piece is held. Next (lines 2-3), these extents are used to compute (1) the extents of the imposed regular boxes (by rounding irregular boxes up to the closest power-of-2 number) and (2) all other imposed regular boxes that intersect that process's unique piece of the data. The third step (lines 5-7), involves selecting which process chooses to receive data for each imposed box. In the current scheme, the process that has the maximum intersection volume with the regular box is chosen as the receiver, this is a greedy scheme that minimizes the amount of data-movement. Finally (lines 8-10), every process calculates the extents of the pieces it will send to receivers, while the receiver allocates a buffer large enough to accommodate the data it will receive. MPI point-to-point communication (using MPI_Irecv and MPI_Isend) is used among processes for data exchanges.

V. Algorithm Evaluation

In this section, we evaluate the performance of the restructuring algorithm compared to previous algorithms, and we measure the overhead of writing irregular versus regular datasets. We also perform a sensitivity study to evaluate the impact of key tuning parameters.

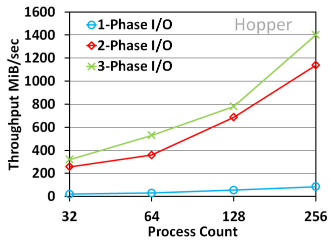
The experiments presented in this work have been performed on both the Hopper system at the National Energy Research Scientific Computing (NERSC) Center and the Intrepid system at the Argonne Leadership Computing Facility (ALCF) Hopper is a Cray XE6 with a peak performance of 1.28 petaflops, 153,216 cores for running scientific applications, 212 TB of memory, and 2 petabytes of online disk storage. The Lustre [15] scratch file system on Hopper used in our evaluation consisted of 26 I/O servers, each of which provides access to 6 object storage targets (OSTs). Unless otherwise noted, we used the default Lustre parameters which stripe each file across two OSTs. Intrepid is a Blue Gene/P system with a total of 164K cores with 80 terabytes of RAM, and a peak performance of 557 teraflops. The storage system consists of 640 I/O nodes that connect to 128 file servers and 16 DDN 9900 storage devices. We used the GPFS [16] file system on Intrepid for all experiments. The Intrepid file system was nearly full (95% capacity) during this evaluation. We believe that this significantly degraded I/O performance on Intrepid.

A. Comparison of PIDX I/O Algorithms

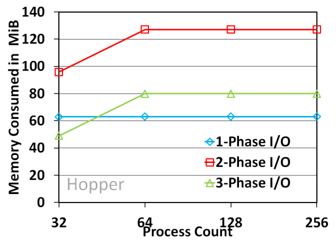
We developed a microbenchmark to evaluate the performance of PIDX for various dataset volumes. We then compared three techniques for writing IDX data in parallel. The first is a naive implementation, in which each process performs an HZ encoding and writes its data directly to disk. The second uses two-phase aggregation after the HZ-encoding step. The third algorithm introduces a data-restructuring phase prior to HZ encoding. The aggregate performance on Hopper and the per process memory load of each implementation are shown in Figure 4. The per process data volume was set to 60^3, with each element containing two variables, with four floating-point samples per variable. This configuration produced 13.18 MiB of data per process.

The naive one-phase implementation performed poorly because of the combination of both discontiguous memory buffers after HZ encoding and discontiguous file access with small write operations. At 256 processes it achieved 85 MiB/s. Each process consumed approximately 64 MiB of memory (nearly five times the size of the original data volume) because of the sparse HZ encoding.

A two-phase I/O aggregation scheme mitigates the disk access inefficiencies due to small, noncontiguous file access by aggregating buffers into larger block-aligned buffers. This optimization improved throughput from 85 to 1100 MiB/s. The implementation performance still suffers from small network transfers during the aggregation phase due to the sparse HZ

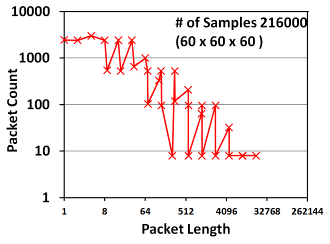


(a)

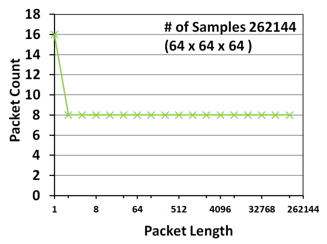


(b)

Fig. 4. (a) [Microbenchmark] Throughput for weak scaling of the three PIDX implementation phases, with 60^3 block size (log-linear scale). (b) [Microbenchmark] Per process memory footprint of weak scaling of the three PIDX implementation phases, with 60^3 block size (log-linear scale)



(a)



(b)

Fig. 5. [Microbenchmark] Size and number of data packets sent during the aggregation phase by process with rank 0, at process count 256 (from Fig. 4a), for both (a) 2-phase and (b) 3-phase I/O. The effect of the data restructuring phase can be seen as all small data packets gets bundled into a small number of packets and the maximum packet length increases.

encoding. This is illustrated in Figure 5a, which shows the frequency and size of messages sent from rank 0 as an example. There are almost 10,000 messages of less than four bytes. In addition, the scheme consumes even more memory than the previous implementation because of the extra buffer used for aggregation. The aggregation buffer size varied from 32 to 64 MiB depending on the overall data volume.

By restructuring the data before aggregation and using a three-phase I/O scheme, significant performance gains can be made; specifically, additional 25% performance improvement over the two-phase aggregation algorithm, reaching 1400 MiB/s. Figure 5b illustrates the frequency and size of messages sent from rank 0 during aggregation after the restructuring phase. The message size is significantly larger, leading to fewer total messages and significantly improved overall performance. In addition, the memory consumption per process is reduced in comparison to that of the two-phase algorithm because of the dense HZ encoding that is performed on restructured, power-of-2 data.

B. Performance of Regular and Irregular Datasets

In our next experiment, we compared the performance of PIDX when writing an irregular dataset with a comparably sized regular one. We used the same microbenchmark and variable configuration as in previous experiments, but we varied the per process volume from 128^3 (a regular volume) to 126^3 (an irregular volume) and measured the impact at larger

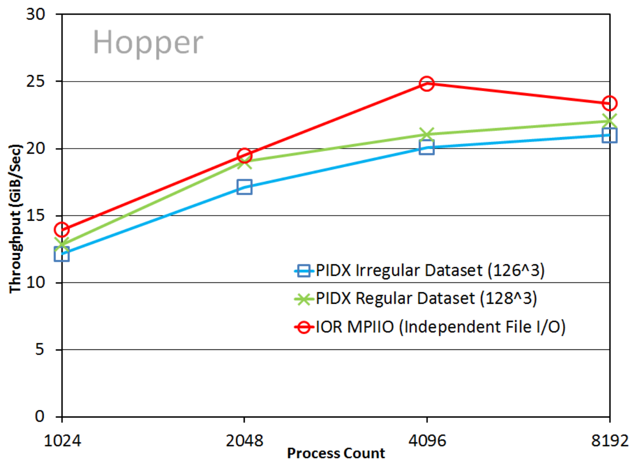


Fig. 6. [Microbenchmark] Weak scaling performance comparison of irregular dataset (per process block 126^3) with a comparably sized regular dataset (per process block 128^3)

scales. We also compared PIDX to the IOR benchmark [17] configured to write a similar volume to unique files. IOR does not perform any data encoding and is used in this case to provide a baseline for I/O performance.

The weak scaling results for all three cases are shown in Fig. 6. From the figure we see that at 8,192 processes, PIDX achieves a maximum throughput of 21 GiB/s to write the irregular dataset (0.95 terabyte per timestep) and a comparable maximum throughput of 22.08 GiB/s achieved to write the regular one (1 terabyte per timestep). At the same process count for both regular and irregular datasets, we have respectively achieved 94% and 89% of the IOR throughput.

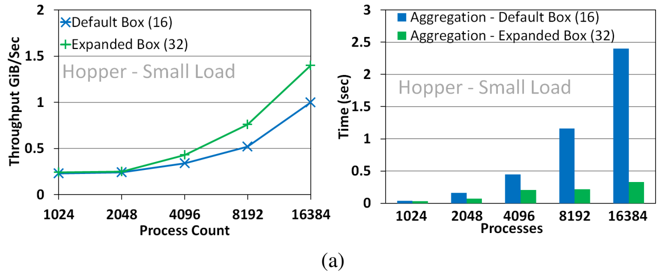
C. PIDX Parameter Study

The two tunable parameters in PIDX that have the most profound impact on runtime behavior are the box size used for power-of-2 restructuring and the number of IDX blocks per file. The former parameter affects network traffic, while the latter affects file system traffic. The two parameters can therefore be tuned independently to reflect the characteristics of the network and storage infrastructure on a given system.

1) Imposed Box Size: Our API provides some flexibility for controlling the size of the imposing regular box during the data restructuring phase. Specifically, we have a parameter that switches between the default imposed box and an expanded box. The default imposed box size is calculated at runtime by rounding up the data volume at each process to the nearest power-of-2. The expanded box size doubles the size of the box in each dimension. This parameter affects both the restructuring phase of the algorithm and the aggregation phase of the algorithm as follows:

1) Data-Restructuring Phase: change in the time needed to distribute the data as it is being transmitted to different numbers of intermediate nodes.
2) Data-Aggregation Phase: change in time needed to send the data to the aggregator nodes as the data is coming from different numbers of intermediate nodes.

As an example to illustrate the impact of this parameter, consider an experiment in which 4,096 processes each hold a 15^3 volume. The default number of participating processes holding the distributed data after the restructuring phase can be calculated; $4096 \times (15^3))/(16^3) = 3375$. If an expanded box size is used instead, then the number of participating processes after restructuring will be $4096 \times (15^3)/(32^3) \approx 422$, with each of those processes handling a larger load. This parameter can therefore be used to help strike a balance between load balancing and message size needed for communication.



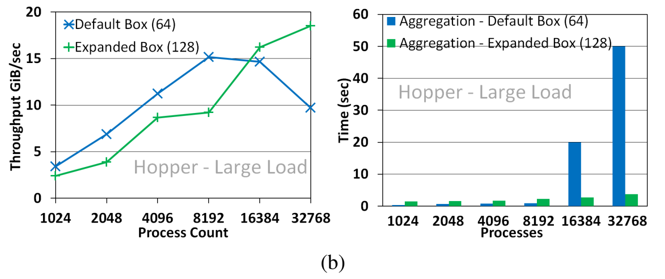


Fig. 7. [Microbenchmark] [log-linear scale] Weak scaling results on Hopper as the restructuring box size is varied. (a) A small load of 15^3 per process with a default and expanded box size of 16^3 and 32^3, respectively. (b) A larger load of 60^3 per process with default and expanded box size of 64^3 and 128^3 respectively.

To understand the behavior of this PIDX parameter, we ran a set of experiments varying both scale and load. We used a small per process load of 15^3 as well as a relatively larger load of 60^3. As in earlier experiments, each element consisted of two variables, each in turn containing four floating-point values. The small and large volumes were 0.2 MiB and 13.18 MiB, respectively. Figure 7 shows the performance of these two scenarios as the microbenchmark is scaled from 1,024 to 16,384 processes for the smaller load, and from 1,024 to 32,768 for the larger load; the box size is toggled between the default and expanded setting.

We found that the time consumed by the restructuring phase was not greatly affected by the box size. For example, at 16,384 processes it varied from 0.012 seconds with a default box and 0.018 seconds with an expanded box. The reason is that this algorithm is well distributed and involves communication with neighboring processes. As a result, we focused our attention on the impact of the aggregation phase. Figure 7 illustrates both the aggregate performance of the algorithm and the time consumed in the aggregation phase of the algorithm.

For the smaller load, both box sizes follow similar trends, but the performance gap between the default and expanded setting increases at scale. This result can best be explained by looking at the histogram of time spent during the aggregation phase with both the default and expanded box (Fig. 7a). From the graph one can clearly see the increasing amounts of time spent performing aggregation with the default box. At process count 4,096, for example, the aggregation phase with the expanded box involves fewer processes (422) with relatively larger loads (2 MiB), as opposed to the default box where aggregation involves too many processes (3,375) each with a smaller load (256 KiB). Thus, the difference in performance can be attributed to the extra overhead caused by having too many nodes transmitting small data volumes during aggregation.

The larger load exhibits a different performance trend. In this case, the default box performs better than the expanded box up to process counts of 8,192, after which its performance starts to decline. The expanded box, on the other hand, continues to scale at higher process counts. Looking at the adjoining histogram of time spent during aggregation, we see that when fewer processes are involved, aggregation with the default box requires less time than with the expanded box, creating a relatively higher throughput. In this example, with a larger dataset, the best strategy at small scale is to use the default box in order to involve enough processes in aggregation to distribute the load evenly. The best strategy at larger scale is to use an expanded box to limit message contention.

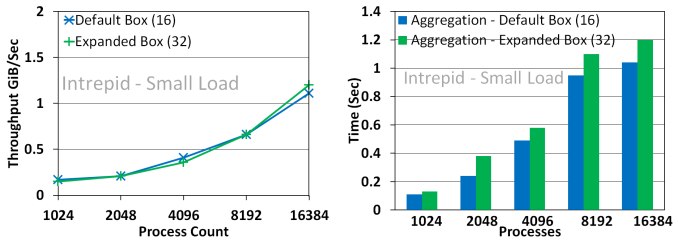


Fig. 8. [Microbenchmark] [log-log scale] Intrepid (BG/P) results for weak scaling with per process data block of size 15^3. The default box size is 16^3, and the expanded box size is 32^3

The results from a single system (Hopper) seem to indicate that this parameter could be tuned automatically based simply on the data volume and scale of an application. However, we find that this is not necessarily the case when we repeat the experiment on Intrepid, which has a different network infrastructure. The results of this experiment are shown in Figure 8. In this case, the performance improvement from using an expanded box is negligible as the scale is increased. This indicates that the Intrepid network is less sensitive to variations in the number and size of network messages.

These results indicate that the restructuring box size parameter should be selected to reflect characteristics of both the data set and the computing system.

2) Blocks Per File: The second notable PIDX parameter is the number of blocks per file used in the IDX file format. Varying this parameter changes the number of underlying files stored as part of an IDX dataset. In the case of two-phase aggregation in PIDX, it also alters the number of aggregators involved in writing data. PIDX uses n aggregators per file, where n is the total number of samples per data element. Each aggregator writes to a contiguous, block-aligned portion of the file.

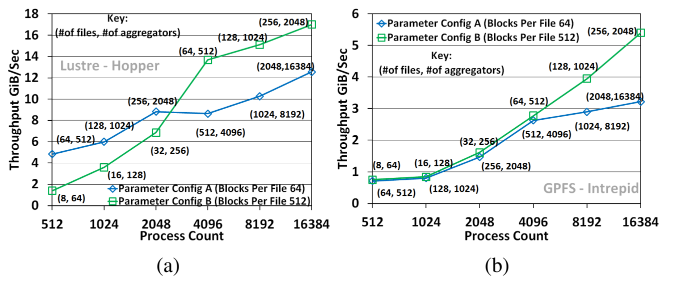


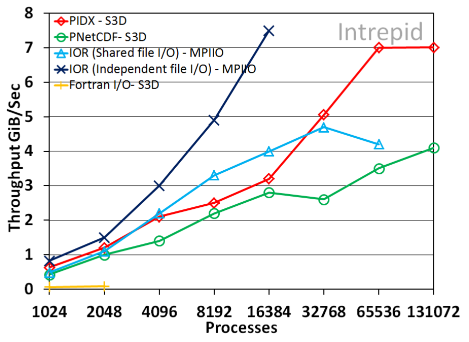
Fig. 9. [Microbenchmark] Weak scaling on (a) Hopper and (b) Intrepid with per process data block of size 60^3 done with parameter configuration A (64 blocks per file) and parameter configuration B (512 blocks per file)

To illustrate the effect of this parameter, we executed a weak-scaling microbenchmark on both Hopper and Intrepid and varied the blocks per file from 64 to 512. The data volume per process was set to 60^3 and the number of variables in each element was set to the same configuration as in previous experiments. The results are shown in Figure 9. Each data point is also labeled with the number of files and number of aggregators that were involved at each scale, based on the global volume and the blocks-per-file parameter.

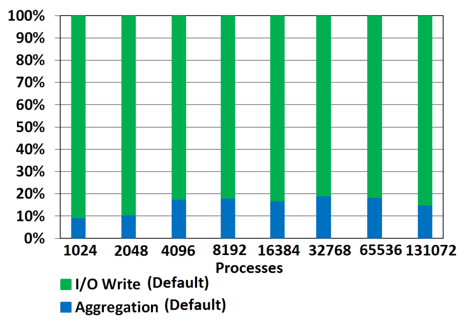
For Hopper, the performance of the lowest scale (512) is most optimal with 64 blocks per file, creating 64 files and using all 512 processes as aggregators. However, at the largest scale (16,384) the best performance used 512 blocks per file (creating 256 files using only 2,048 aggregators). On Intrepid, in contrast, 512 blocks per file consistently outperforms 64 blocks per file. As in the sensitivity study of the imposed box size parameter, we find that the blocks per file parameter should be chosen based not just on characteristics of the run-time data set but also characteristics of the system itself. This example illustrates the difference in tuning strategy for different file systems (Lustre and GPFS) and different storage hardware.

VI. S3D

S3D is a continuum-scale, first-principles, direct numerical simulation code that 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 [6]. Each rank in S3D is responsible for a portion of the three-dimensional dataset; all MPI ranks have the same number of grid points and the same computational load under a Cartesian decomposition. S3D can be compiled with support for several I/O schemes, which are then selected at runtime via a configuration parameter. In this study we used an I/O kernel derived from S3D (S3D-IO) that



(a)



(b)

Total Processes	Processes with Restructured Data	Aggregator Processes
1024	843	64
2048	1688	128
4096	3375	256
8192	6750	512
16384	13500	1042
32768	27000	2048
65536	54000	4096
131072	108000	8192

(c)

Fig. 10. [S3D I/O Benchmark] (a) Intrepid results for weak scaling of different I/O mechanisms including PIDX, Fortran I/O, and PnetCDF generating irregular datasets with block size 30^3. The total data volume ranges from 3.29 GiB to 422 GiB. (b) The proportion of time taken by the aggregation phase and the I/O write phase at each scale. (c) The number of processes responsible for restructured data and the number of aggregator processes at each scale

allows us to focus our study on the checkpointing performance of S3D. We modified S3D-IO to include support for PIDX alongside the other supported I/O modules. Unless otherwise noted, we configured the I/O kernel to operate on a volume of size 30^3 per process, which approximates the data volume per process used at scale for S3D production use at NERSC.

In all cases we compared PIDX performance with that of both the Fortran I/O and PnetCDF modules in S3D. In the case of Fortran I/O, data is written in its native format to unique files from each process. In the case of PnetCDF, data is written to a single, shared file in structured, NetCDF format. In terms of file usage, PIDX lies somewhere between these two approaches in that the number of files generated is based on the size of the dataset rather than on the number of processes. In addition to S3D-IO results, we show IOR results in which IOR has been configured to generate a similar amount of data to that produced by S3D. These results are intended to provide a baseline for shared-file and file-per-process performance. Default file system striping parameters were used in all cases, except for the PnetCDF and shared-file IOR results on Hopper, in which the Lustre striping was increased to span all 156 OSTs available on that system.

A. Weak Scaling

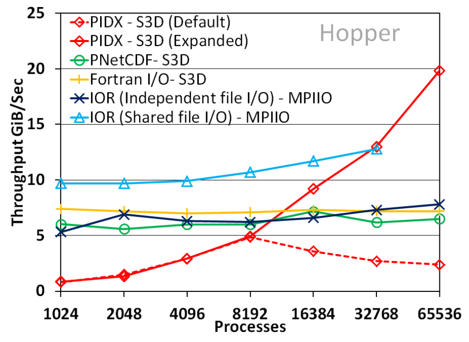
In this section we evaluated the weak scaling performance of PIDX when writing irregular S3D datasets on both Intrepid and Hopper. In each run, S3D writes out 20 timesteps wherein each process contributes a 30^3 block of double-precision data (3.29 MiB) consisting of 4 variables; pressure, temperature, velocity (3 samples), and species (11 samples). On Hopper, we varied the number of processes from 1,024 to 65,536, thus varying the amount of data generated per timestep from 3.29 GiB to 210.93 GiB. On Intrepid, we varied the number of processes from 1,024 to 131,072, thus varying the amount of data generated per timestep from 3.29 GiB to 421.875 GiB. We respectively used 256 and 512 blocks per file for all our experiments with PIDX on Intrepid and Hopper.

Weak scaling results for PIDX and other parallel file formats on Intrepid are shown in Figure 10a. At the time of these experiments, the Intrepid file system was nearly full (95% capacity) which is believed to have seriously degraded I/O performance for all experiments. While each of the output methods showed scaling, none of the output methods approached the expected throughput of Intrepid at scale. We used the default restructuring box size of 32^3 in all cases.

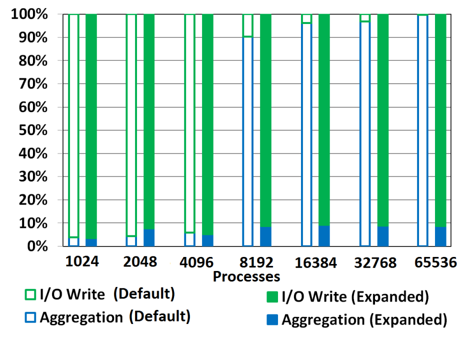
Looking at the performance results in Figure 10a, we observe that PIDX scales well up to 65,536 processes, and that for all process counts, it performs better than PnetCDF and Fortran I/O. This behavior is an artifact of PIDX aggregation, which finds a middle ground between shared-file and file-per-process I/O. Fortran I/O uses a unique file per process and places all files in the same subdirectory. This approach caused a high degree of metadata contention and serialization. In contrast, PIDX creates a hierarchy of subdirectories at rank 0 and coordinates file creation to avoid directory contention. The IOR unique file case appears to perform well because the measurements did not include directory creation time. IOR was configured with the *uniqueDir = 1*, which creates a separate subdirectory for each file in an attempt to show throughput in the absence of metadata contention at file creation time. However, this simply shifted the contention to directory creation time (which is not measured by IOR) instead of file creation time. As a result, we were able to run IOR with unique files only up to a scale of 16,384 despite its apparent performance because subdirectory creation was taking as much as three hours to complete at the largest scale.

Figure 10c shows the number of processes that are responsible for restructured data as well as the number of I/O aggregators. Both numbers grow linearly as the application scales. Scalability of aggregation on Intrepid can also be seen from Figure 10b, which shows the normalized timings of aggregation and I/O write phases. The restructuring, HZ encoding, and file creation phases are not shown because they are small in comparison. These indicate a low overhead of the PIDX network phases, as most of the PIDX write time is spent on disk I/O.

Weak scaling results for Hopper are shown in Figure 11a. As described in Section V-C1, aggregation scaling on Hopper is affected by the size of the imposed regular box. Using an expanded box during the data-restructuring phase changes



(a)



(b)

Total Processes	Processes with Restructured Data (Def.)	(Exp.)	Aggregator Processes
1024	843	105	32
2048	1688	210	64
4096	3375	422	128
8192	6750	843	256
16384	13500	1688	512
32768	27000	3375	1024
65536	54000	6750	2048

(c)

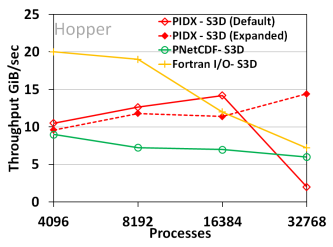
Fig. 11. [S3D I/O Benchmark] (a) Hopper results for weak scaling of different I/O mechanisms including PIDX, Fortran I/O, and PnetCDF generating irregular datasets with block size 30^3. (b) The proportion of time taken by the aggregation phase and the I/O write phase as we scale from 1,024 processes (3.29 GiB) to 65,536 processes (210.93 GiB) for both default and expanded box. (c) The number of processes responsible for restructured data when using the default (Def.) and expanded (Exp.) imposed box as compared with the number of aggregator processes.

the distribution of data, effectively altering the number of processes participating in aggregation. Hence, on Hopper we performed two set of experiments, using both a default imposed box and an expanded one, with dimensions 32^3 and 64^3, respectively.

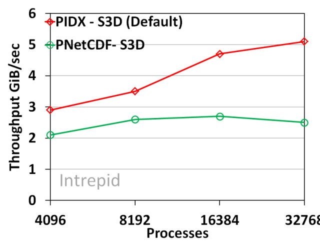
We see comparable performance for default and expanded boxes at lower process counts. This behavior corresponds to the case when the aggregation phase takes a similar amount of time, regardless of the number of processes participating, also supported by the phase-wise partition graph in Figure 11b. With increasing process counts, aggregation using the default box fails to scale, whereas aggregation with the expanded box continues to scale even at higher process counts. Figure 11c indicates how the number of processes responsible for restructured data varies significantly with the default and the expanded box. For instance, at process count 32,768, the data is redistributed to 27,000 processes as opposed to only 3,375 processes with the expanded box. These two different process counts lead to aggregation phases with very different runtimes. As Figure 11b shows, aggregation at 32,768 processes take the majority of time when using the default box, whereas with the expanded box it takes under 10% of the total PIDX write time.

Compared with other file formats PnetCDF and Fortran I/O, PIDX appears to lag in the lower process count ranges ($\leq$ 8192); but as the number of processes increases PIDX outperforms them. This lag in performance for the lower process counts can be attributed to lack in aggregators. We saw in Figure 9a how at lower process counts Hopper yields higher throughput with relatively large number of aggregators controlled by blocks per file. For lower process counts, PIDX performed optimally with 64 blocks per file, transcending into a larger set of aggregators, whereas in the experiments here we used 512 blocks per file. Comparing performance numbers, we see at process count 65,536 that PIDX achieves a throughput of around 19.84 GiB/sec which is approximately three times that of Fortran I/O (7.2 GiB/sec) as well as PnetCDF (6.5 GiB/sec). We also remark that throughput performance for file formats that use one file per process (IOR unique and Fortran I/O) is lower than expected because of the small amount of data (3.29 MiB) written to each file relative to the cost of creating the file.

B. Strong Scaling



(a)



(b)

Fig. 12. [S3D I/O Benchmark] Strong scaling for PIDX, PnetCDF, and Fortran I/O for generating irregular datasets of dimension 960^3 on (a) Hopper and (b) Intrepid BG/P.

In this section, we compared the strong scaling results of PIDX with Fortran I/O and Parallel NetCDF on both Hopper and Intrepid. In all our experiments we used a total 3D domain size of dimension 960^3. We varied the number of processes from 4,096 to 32,768, thereby strong-scaling the block size per process from 60^3 (4,096 processes) down to 30^3 (3,2768 processes). The datasize per time-step is 105.46 GiB, and we configured S3D to write out 20 timesteps. The results of this evaluation are shown in Figure 12.

Similar to weak scaling, for all our experiments on Hopper we used both a default and expanded imposed box during

TABLE II
NUMBER OF PROCESSES RESPONSIBLE FOR RESTRUCTURED DATA IN S3D STRONG SCALING EXAMPLES.

Total Processes	Imposed Box Dimension (Default)	(Expanded)	Processes with Restructured Data (Default)	(Expanded)
4096	64	128	3375	422
8192	64	128	3375	422
16384	64	128	3375	422
32768	32	64	27000	3375

the data restructuring phase, as opposed to just using the default box on Intrepid. Table II shows the effects of using the two different imposed boxes in terms of number of processes that are responsible for restructured data once it has been distributed. For process counts 4,096, 8,192 and 16,384 with the default box, this number equals to 3,375 processes. This setting yields throughput in the range of 10 GiB/sec to 14 GiB/sec for the three process counts, whereas the expanded box causes distribution of data across far fewer processes (422) at similar process counts, leading to less throughput compared with that of the default box. This performance pattern is similar to the one observed in the weak scaling results. Looking further at Table II, for process count 32,768, the default box causes the entire data to be distributed across 27,000 processes. This approach renders aggregation unscalable and causes aggregation time to increase, resulting in poor performance throughput. Similar to weak scaling, the use of an expanded box at this process count leads to a relatively small set of processes participating in aggregation: 3,375. This number, obtained by the use of an expanded box at 32,768 processes, exactly matches the number of processes obtained by using the default box at process counts 4,096, 8,192 and 16,384. With aggregation settings similar to the smaller process counts, using an expanded box yields comparable throughput of 14.2 GiB/sec.

PIDX outperforms PnetCDF at all process counts on both Hopper and Intrepid. On both platforms, at process count 32,768, PIDX achieves an almost two-fold improvement over PnetCDF, achieving a throughput of 5.1 GiB/sec and 14.2 GiB/sec on Intrepid and Hopper as opposed to 2.6 GiB/sec and 6.02 GiB/sec achieved by PnetCDF. On Hopper, compared with PIDX, Fortran I/O yields higher performance at process counts 4,096 and 8,192, but decreases steeply as we scale to higher process counts, and PIDX outperforms at process count 16,384 and 32,768. Since Fortran I/O performs unique file I/O, at higher process counts (16,384 and 32,768) the data volume per process becomes so small that the file creation takes a larger fraction of the time and hence the performance drops steeply. PIDX, on the other hand, writes fewer files and does not face such issues.

VII. Future Work and Conclusion

In this work we have discussed a new parallel algorithm to write irregular datasets in the IDX file format. With S3D I/O we have shown PIDX to scale up to 131,072 processes on Intrepid BG/P and up to 65,536 on Hopper, with performance competitive to other commonly used parallel file formats, Fortran I/O and PnetCDF. In addition to its performance, we think that one of the major benefits of PIDX is that it strikes a balance between efficient read I/O for multiresolution visualization as well as effective write I/O for large-scale simulation.

Since PIDX implements a multiphase scheme, understanding the behavior of its parameters is important. Exploring the nature of how these parameters need to be chosen has indicated that they can be both application dependent and architecture dependent. In fact, knowing the effects of a combination of these two factors is often necessary in order to achieve high amounts of efficiency.

For data that is unevenly distributed among processes, PIDX offers a generic scheme for evening out the distribution. This applies to data consisting of irregularly sized blocks, regularly sized blocks, and their combinations.

We plan to improve the aggregation strategy of PIDX in future work. One limitation of the current algorithm is that the number of aggregators can be modified only by changing the number of blocks per file. A dynamic aggregation scheme would improve upon this algorithm by providing more fine-grained control over the number of aggregators and the aggregation buffer size. In this work, however, we favored using a more straightforward scheme and still achieved positive results. Similarly, we think that additional levels of control over the size of the imposed box are important. In this work, we experimented with two options, a default and an expanded box size, which were most appropriate for our target application of S3D. It is possible that an even larger imposed box would be appropriate in other use cases, especially where the per-process volume is small.

We think that our algorithm would benefit from more sophisticated schemes for data-restructuring as well as nominating which process acquires the data block. Our current implementation selects the process that owns the largest piece of the original data. In the case of conflicts, we resolve this scheme in a way by which some processes may have multiple blocks if they happen to own the majority in two. In this scheme, balancing the load would make more sense; however, to do so efficiently requires good measurements of data locality as well as providing good measures to resolve conflicts.

Acknowledgments This work was supported by 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] 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*, 2001.

[2] V. Pascucci, D. E. Laney, R. J. Frank, F. Gygi, G. Scorzelli, L. Linsen, and B. Hamann, "Real-time monitoring of large scientific simulations," in *ACM Symposium on Applied Computing*, 2003, pp. 194–198.

[3] 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.*, vol. 30, pp. 7:1–7:13, April 2011.

[4] 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.

[5] S. Kumar, V. Vishwanath, P. Carns, B. Summa, G. Scorzelli, V. Pascucci, R. Ross, J. Chen, H. Kolla, and R. Grout, "PIDX: Efficient parallel I/O for multi-resolution multi-dimensional scientific datasets," in *IEEE International Conference on Cluster Computing*, 2011.

[6] C. S. Yoo, R. Sankaran, and J. H. Chen, "Three-dimensional direct numerical simulation of a turbulent lifted hydrogen jet flame in heated coflow: flame stabilization and structure," *Journal of Fluid Mechanics*, pp. 453–481, 2009.

[7] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo, "FLASH: An adaptive mesh hydrodynamics code for modelling astrophysical thermonuclear flashes," *Astrophysical Journal Supplement*, vol. 131, pp. 273–334, 2000.

[8] B. Palmer, A. Koontz, K. Schuchardt, R. Heikes, and D. Randall, "Efficient data I/O for a parallel global cloud resolving model," *Environ. Model. Softw.*, vol. 26, no. 12, pp. 1725–1735, Dec. 2011.

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

[10] "HDF5 home page," http://www.hdfgroup.org/HDF5/.

[11] K. Gao, W.-K. 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*, sept. 2009, pp. 470 –477.

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

[13] 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*, june 2005, pp. 11 – 223.

[14] 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*, vol. 21, pp. 31–38, December 1993.

[15] "Lustre home page," http://wiki.lustre.org/index.php/Main_Page.

[16] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, 2002, pp. 231–244.

[17] 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.