Reducing network congestion and synchronization overhead during aggregation of hierarchical data

Sidharth Kumar* ‡, Duong Hoang* ‡, Steve Petruzza*, John Edwards†, Valerio Pascucci*
*Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT, USA
†Informatics and Computer Science Department, Idaho State University, Pocatello, ID, USA
{sidharth, duong, spetruzza, pascucci}@sci.utah.edu, edwajohn@isu.edu
‡*Authors contributed equally*

Abstract—**Hierarchical data representations have been shown to be effective tools for coping with large-scale scientific data. Writing hierarchical data on supercomputers, however, is challenging as it often involves all-to-one communication during aggregation of low-resolution data which tends to span the entire network domain, resulting in several bottlenecks. We introduce the concept of indexing templates, which succinctly describe data organization and can be used to alter movement of data in beneficial ways. We present two techniques, domain partitioning and localized aggregation, that leverage indexing templates to alleviate congestion and synchronization overheads during data aggregation. We report experimental results that show significant I/O speedup using our proposed schemes on two of today's fastest supercomputers, Mira and Shaheen II, using the Uintah and S3D simulation frameworks.**

I. Introduction

Large-scale scientific simulations run on supercomputing systems with tens or hundreds of thousands of processes and produce data hundreds of gigabytes or terabytes in size, posing serious challenges to post-processing data analysis tasks which are often performed on machines with much less computing power and memory. Multiresolution data representations have been shown to be a promising solution to this problem [1], [2], [3], [4], as they allow scientists to decide the scale at which a given analysis task will be performed, avoiding the need to read or even to store data at finer scales. In particular, they enable interactive visualization of large scale HPC data since data can be accessed at varying levels of resolution with low latency. Also, simulation data written directly into multi-resolution format can be used immediately in efficient data analytics without data duplication or slow post-processing.

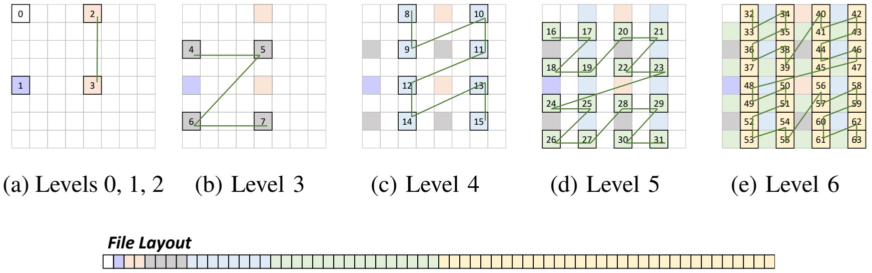
Two phase I/O [5], [6] is a commonly used technique in which data is moved (or aggregated) across the fast interconnect onto a few chosen *aggregator* processes before being written to disks. Data movement during the aggregation phase is challenging, because network congestion can happen due to a skewed distribution of aggregators among processes so that traffic converges to a small set of links. Also, when the number of sending processes become too large, links leading to aggregator nodes become congested, and the cores on these nodes themselves become bottlenecks due to a high number of receive calls to process. These two issues happen often in aggregating coarse levels of hierarchical data, which span the entire domain. Yet another source of bottlenecks comes from MPI's collective calls such as `MPI_Win_fence()` which incur synchronization overheads that grow rapidly as the number of processes in an MPI communicator grows [7], [8], [9].

In this paper we introduce novel techniques that are collectively used to alleviate congestion and global communication overheads. We focus our study primarily on HZ-order, a hierarchical multi-resolution data format that is efficient to compute and has been shown to have excellent locality both spatially and hierarchically [2]. HZ-based multi-resolution data format has successfully been used in [4] for interactive Poisson editing of gigapixel images, and in [10] for interactive analysis of multi-terabyte climate simulation data. There has been work on efficient writing of HZ-ordered grids ([11], [12], [13], [14]), but none has addressed in-depth the issue of efficient aggregation for hierarchical data, which is the main topic of this paper. We generalize the HZ-order format such that it enables uniform distribution of aggregators in rank space, avoiding congestion. Further, we demonstrate that our approach is effective not only for hierarchical formats, but also for non-hierarchical.

Our contributions include:

- The concept of *indexing templates* that are used to create a range of multi-resolution decompositions, partitioning schemes, and aggregation patterns. The template provides flexibility in terms of both data organization and movement. (Section II).
- A localized aggregation scheme based on altering a prefix of the indexing template, which facilitates efficient data movement by distributing aggregators uniformly in rank space across the multi-resolution hierarchy. (Section III).
- A domain partitioning scheme where data is divided among a number of partitions in a way that enables each partition to write an independent dataset while maintaining the global hierarchy, alleviating the overheads associated with global communication. The partitioning planes follows a prefix of the indexing template. (Section IV).

We report the results of extensive experiments for each



(a) Levels 0, 1, 2 (b) Level 3 (c) Level 4 (d) Level 5 (e) Level 6

Figure 1: HZ ordering for an 8×8 dataset with the $yxyxyx$ indexing template.

of the proposed schemes on two systems with very different architectures: ALCF's Mira [15] (an IBM BlueGene/Q) and KAUST's Shaheen II [16] (a Cray XC40). We tested our methods using both micro-benchmarks and simulations using the Uintah and S3D simulators. Our methods result in significant I/O speedup compared to traditional methods with uniform aggregator placement and no partitioning.

II. Indexing templates

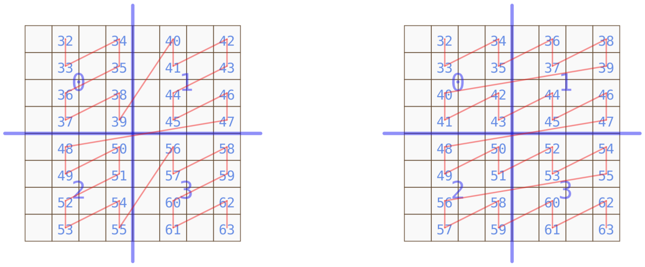
Samples can be ordered in different ways when written to a file. *Indexing* is the process of reordering samples from one scheme, typically row-major, to another scheme. Different orderings achieve different objectives. For example, row-major order is popular because it is simple and easy to implement it, but it shows poor data locality. Z-order [17] is efficient to compute and shows excellent data locality. HZ indexing (Hierarchical Z-order) [2] adds hierarchical, or resolution, locality to Z-order.

We introduce *indexing templates* as succinct descriptors of sample ordering. Index templates have useful properties that we will be able to use in domain partitioning and aggregation. We motivate indexing templates using HZ-order and then generalize to other structured data orderings.

In this section we summarize HZ indexing. Detailed descriptions can be found in [2] and [18]. HZ-order imposes both resolution levels and ordering within resolution levels. For example, in 1D, two levels of resolution can be created by splitting the samples into even and odd samples. More levels can be added by recursively splitting the even samples in the same way. In higher dimensions, each split occurs in a specific dimension. A sequence of splits can be described as a string of x, y, and z characters, each corresponding to the dimension in which to split. For example, consider splitting the samples of an 8×8 grid using the string $yxyxyx$ that is to be parsed from right to left, where x and y are the split dimensions. We call this string the *indexing template*. Using the template $yxyxyx$ we get the hierarchy shown in figure 1. The last resolution level (level 6) consists of all odd X samples because the last character is x, while the second-last resolution level has all odd Y samples among the remaining samples. We continue recursively, each time extracting a character from the right and extracting a resolution level from the remaining samples until the string is exhausted and only one sample is left.

With hierarchy defined we now turn to sample ordering within a resolution level. At hierarchy level l, we define the GZ (Generalized Z) template to be the leftmost $l - 1$ characters of the indexing template. Consider level 6 in the example. We order the samples according to the GZ template $yxyxy$ (i.e., the first five characters of the indexing template) as follows: we interleave the bit representations of the sample's x, y and z Cartesian coordinates according to the string in consideration (e.g., $yxyx$ means taking the first bit from y, the second bit from x, the third bit from y, and the fourth bit from x). The bit interleaving pattern dictated by $yxyxyx$ is the same as the Morton code, or Z-order [17].

Indexing templates generalize the Z- and HZ-orders. We call the new orderings GHZ-order (Generalized HZ-order). Similarly, at a given hierarchy level, we say the samples are in GZ-order. See figure 2. Indexing templates are powerful tools in that they control the shape of the GZ curves at different hierarchy levels which becomes important when matching samples to processes. Certain GHZ-orderings cause a process to contain non-contiguous chunks of samples, while others cause a contiguous chunk of samples to span non-contiguous chunks of processes, both of which are undesirable for efficient aggregation (explained later). For an illustration of the first problem, see figures 2a and 2b, where the domain is spanned by four processes, each occupying one quadrant of the 8×8 grid. In figure 2b, the sample-to-process distribution is at odds with the $yyxxyx$ indexing template, causing each process to contain a non-contiguous range of samples in HZ space, whereas in 2a there is no such problem. We say a GZ curve is rank-conforming if the curve visits processes in strictly increasing order of rank. A GHZ-order given by an indexing template is rank-conforming if the GZ-curve at each hierarchical level is rank-conforming. We show in the next section that rank-conforming GHZ-orderings are desirable for efficient aggregation.



(a) $yxyxyx$ template

(b) $yyxxyx$ template

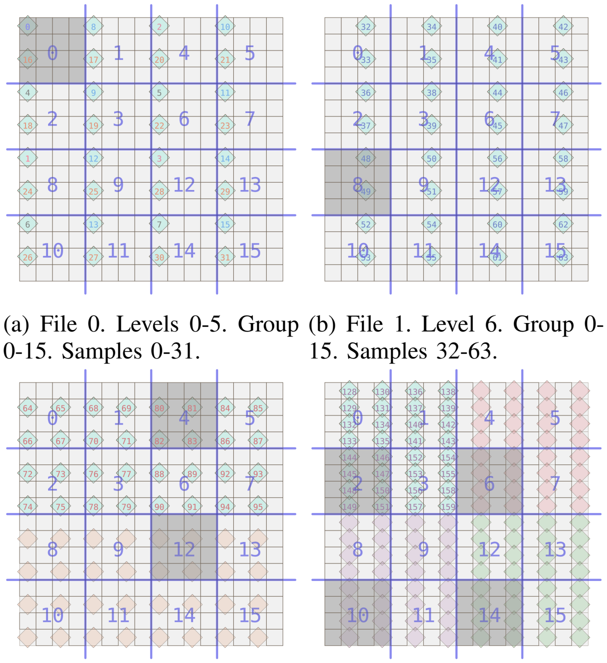
Figure 2: GHZ indexing using two different indexing templates. Only the last level is shown. Blue lines show process boundaries while big letters show process IDs. Red lines highlight the GHZ curve at level 6. The GZ curve in (a) is rank-conforming since the curve visits the processes in rank order. The indexing in (b) is not rank-conforming since each process is visited twice.

III. Efficient aggregation for hierarchical data

We start the discussion of efficient aggregation by defining the concept of *aggregation groups*, which will play a crucial role in this section and the rest of the paper. In our I/O framework, each aggregator is responsible for writing only one file. The set of processes that aggregate data for a common file is called an aggregation group for that file (see figure 3). Each file consists of a fixed number of contiguous samples in GHZ space. The aggregation groups for two files may overlap if the files belong to different resolution levels, while files on the same level always have non-overlapping aggregation groups.

A. Challenges for efficient aggregation of hierarchical data

Several works have studied the problem of placing aggregators carefully to achieve better utilization of the network (see [19], [20], [21]), but none considers multi-resolution data. Among the conditions for efficient aggregation, a uniform distribution of aggregators has been identified to be particularly important because it takes full advantage of parallelism in the network. However this condition is non-trivial to enforce for multi-resolution data due to overlapping spatial extents across resolution levels. Prior works on efficient I/O for HZ-indexed grids such as [11] and [12] use a *uniform distribution* scheme, where aggregators are chosen uniformly and are assigned to files in an one-to-one manner, starting from file 0. We have found that this simple aggregation scheme introduces a staggered and uneven communication pattern: on every level, the aggregators are clustered in rank space, causing underutilization of the network and congestion on the links close to these aggregators (see figure 6). We have confirmed this inefficiency through experiments that will be discussed later in this section.



(a) File 0. Levels 0-5. Group 0-15. Samples 0-31.

(b) File 1. Level 6. Group 0-15. Samples 32-63.

(c) File 2. Level 7. Group 0-7. Samples 64-95.

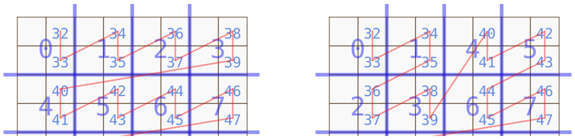
(d) File 4. Level 8. Group 0-3. Samples 128-159.

Figure 3: Aggregator groups and aggregators for indexing template $yxyxyxyx$ The 16 processes are in Z-order and there are 32 samples per file. Aggregators are highlighted in gray background. Diamond samples with the same color belong to the same file. Sample indices are shown for the given file. *Level* is the GHZ-order level of the samples. *Group* is the aggregation group. *Samples* are the GHZ indices of the samples to be written to the file.

B. Localized aggregator selection

Our proposed aggregation scheme avoids the pitfalls of the uniform distribution scheme by enforcing the "locality" constraint: the aggregator for each file is chosen from the aggregation group for the same file. In particular, to find an aggregator for a given file, we calculate the Cartesian coordinates of its first and last samples. From those coordinates we obtain the range of processes spanning the file (i.e., its aggregation group). In the case of a single simulation field, the aggregator is the process with index $\frac{1}{2}(f + l)$ where the aggregation group ranges from process f to process l. Generalizing to n simulation fields, the aggregator for field i is $\frac{i+1}{n+1}(f+l)$. If the first and second file share the same range of processes, as seen in figure 3, we apply a fixed offset to the aggregators for the first file so that they start at rank 0. An example of this scheme in 2D is shown in figure 3, where we show that our algorithm can pick aggregators uniformly at each level and across levels.

The localized aggregator selection requires that the GHZ curve be rank-conforming in order to guarantee a uniform



(a) Row-major prefix ($yyxx$) and row-major rank distribution

(b) Morton prefix $yxyx$ and Morton rank distribution

Figure 4: Altering indexing template prefix to match process rank distribution. In both cases, the GZ curve visits the processes in strictly increasing rank order.

distribution of aggregators in rank space. We ensure this by *restructuring* the distribution of samples to processes so that each process holds a $2^{p_x} \times 2^{p_y} \times 2^{p_z}$ block of samples, for some appropriately chosen integers p_x, p_y, and p_z. This restructuring is equivalent to using the last (rightmost) $p = p_x + p_y + p_z$ characters in the indexing template to index intra-process samples. The best way to set these last p characters is to make them fully interleaved (i.e., rotating among z, y and x in turns), to maximize spatial locality.

Assuming the indexing template has n characters, the remaining $n - p$ characters are used to index the processes themselves. To ensure the GHZ curve visits the processes in increasing rank order, we alter this $(n-p)$-character prefix of the indexing template to match the process rank distribution in the spatial domain. As an example, consider the 8×8 grid in figure 4, where the first 4 characters of the indexing template are used to index the $2^4 = 16$ processes. A row-major ordering of ranks requires the template to start with $yyxx$ (figure 4a), while a Morton ordering of ranks requires the $yxyx$ prefix instead. In general, the prefix that works for row-major rank ordering is of the form $zz \dots zyy \dots yxx \dots x$, while for column-major ordering it is $xx \dots xyy \dots yzz \dots z$, and for Morton ordering $zyxzyx \dots zyx$.

The fact that we alter only a prefix of the template is important, because modifying the indexing template in any way affects the structure of the GHZ hierarchy and, therefore, locality. Since the last p characters of the template are not altered, the last $p+1$ levels are left almost intact: the set of samples on each of these levels stays the same, but the GZ curve changes, potentially resulting in worse locality. Our blocking mechanism thankfully alleviates this loss in locality, as the samples within each block are still spatially coherent, only the blocks themselves are less so. For coarse levels that correspond to the altered template prefix, the fact that these levels can be jumbled turns out not to be a problem in practice. It is generally not beneficial to work with very low-resolution versions of the data. So the first few resolution levels can actually be merged into one level without affecting the practical usage of the data.

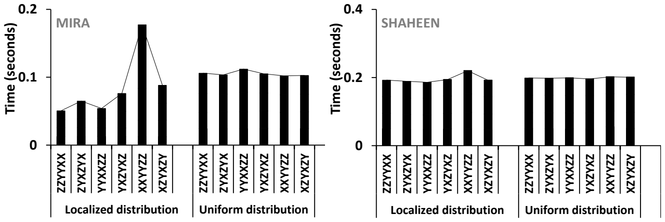


Figure 5: Performance comparison of the three aggregation strategies for different indexing template patterns on Mira (left) and Shaheen (right). The standard deviations on Mira are on the order of 10^{-5}s, and on Shaheen 10^{-2}s.

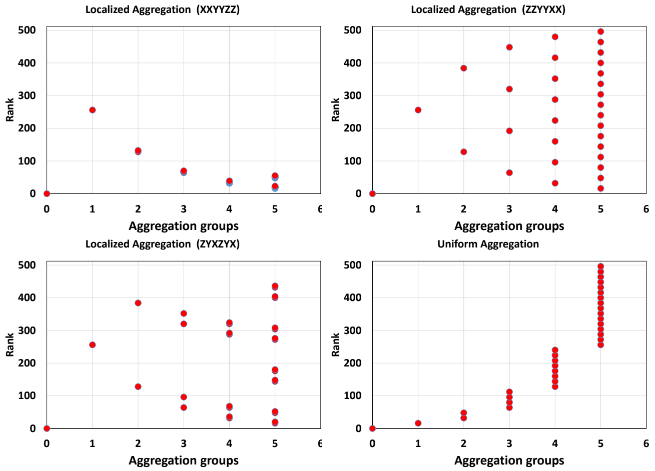


Figure 6: Aggregator distribution using 512 processes and 32 aggregators, separated by levels on the horizontal axis. Row-major rank distribution is used. The $zzyyxx$ template prefix (top right) matches the rank distribution, yielding uniform distribution in rank space.

C. Comparison of aggregation selection schemes

In this section we present a series of micro-benchmarks run on Mira and Shaheen to compare six aggregation strategies: uniform aggregation, and localized aggregation using six template prefix patterns, which are: $zzyyxx$ (short for $zz \dots zyy \dots yxx \dots x$) (Z-axis major or row major), $yyxxzz$ (Y major), $xxyyzz$ (X major), $zyxzyx$ (Z-major Morton), $yxzyxz$ (Y-major Morton) and $xzyxzy$ (X-major Morton). The experiment was carried out with 4096 processes, each containing 32^3 samples. The process ranks are laid out in row-major order in all cases, so we expect that the $zzyyxx$ would perform the best, which is confirmed by experimental results. The total amount of data being aggregated for each of these template configurations is fixed at 1 GB. The results are plotted in figure 5.

Our experiment shows that on Shaheen there is virtually no difference among the schemes, possibly due to the high connectivity of the Dragonfly network. On Mira, uniform aggregation performs very poorly compared to localized

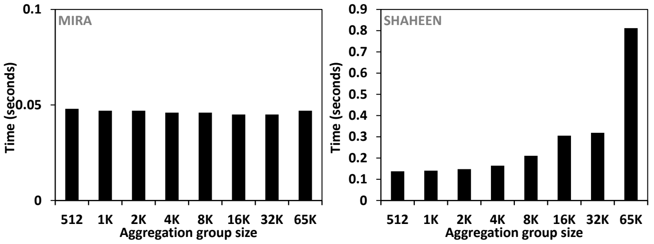


Figure 7: Aggregation timings at 65536 processes for all group sizes, on Mira (left) and Shaheen (right).

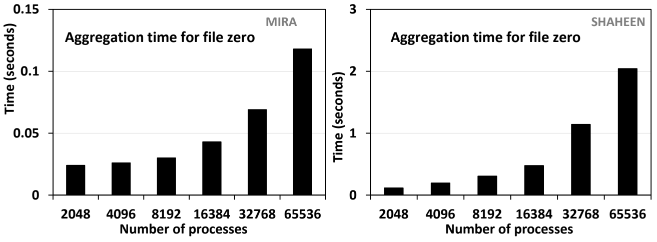


Figure 8: Scaling results for the aggregation of the first file for Mira (left) and Shaheen (right)

aggregation in general. Among the localized aggregation template prefixes, $xxyyzz$ is the worst by a large margin, while $zzyyxx$ is the best. To see the reason, we visualize the distribution of aggregators in rank space (figure 6). As can be seen in the figure, uniform aggregation and template indices $xxyyzz$ and $zyxzyx$ result in clustering and thus underutilization of the network. Only the $zzyyxx$ prefix distributes aggregators uniformly on all levels. Note that both uniform aggregation and localized aggregation using the $zzyyxx$ template prefix select aggregators uniformly in rank space as a whole, but only the latter selects aggregator uniformly on each level as well, which is crucial for performance.

In figure 7 we separate the timings for different aggregation group sizes for a 65,536 process run. We keep the same configuration as the runs in figure 5. Although the amount of data aggregated in each group is constant, lower-resolution files span across more processes, therefore having a bigger group size. On Mira, time is constant for all aggregation group sizes, while on Shaheen time increases for larger aggregation group sizes, possibly because as the communicating nodes span more than one network group, the slower inter-group links become the bottlenecks. In contrast, on a torus network using shortest-Manhattan-distance routing such as Mira's, the number of routing paths grows in proportion with the number of communicating nodes to maintain a constant network throughput.

Finally we note that the timing numbers reported here are for one simulation field. The timing differences among schemes will grow proportionally with the number of fields, which can range from tens to hundreds, in real simulation.

IV. Domain partitioning

The data aggregation phase is often plagued by scalability challenges at high core counts. A common problem at high core counts is MPI synchronization overheads when too many processes are involved in one communicator. This problem can be solved by splitting one global communicator into several communicators, each responsible for many fewer processes. However, for multi-resolution data we also face the problem where the aggregation for low-resolution files becomes a bottleneck due to these files straddling across too many processes (all processes in the case of the first two files). In this section we first describe the aforementioned two challenges in detail, and present a unified solution in the form of a domain partitioning scheme.

A. Limited scalability of group size

In figure 3a we see an example of a file that strides across the entire domain, resulting in an all-to-one communication pattern during aggregation. This is typical in any multi-resolution scheme at low resolutions, and it results in network bottleneck, as the aggregator has to process a large number of network messages, each incurring overhead in the network stack. This problem is illustrated in figure 8, which plots the aggregation time for the first file with increasing number of participating processes. The size of the first file is fixed, so a fixed amount of data is moved across the network during the aggregation phase, but the all-to-one communication pattern renders the aggregation phase non-scalable.

B. Limited scalability of the number of groups

The other cause of network congestion is increasing cost of MPI synchronization when too many processes are involved in one communicator. In our implementation, MPI one-sided communication is used for data transfer, and `MPI_Win_fence()` is used for synchronization. An MPI fence is commonly implemented as a barrier on one process that waits for confirmation signal from all other processes in the same communicator. This communication pattern creates serious bottlenecks as the number of processes in the communicator increases. This problem is illustrated in figure 9 with a weak scaling experiment. We vary the total number of processes and measure timings for three aggregation group. For a given group size (holding the number of processes in a group fixed), time increases with increasing number of groups suggesting communication bottlenecks caused by `MPI_Win_fence()`. This problem indeed has been identified by works such as [7], [8], and [9].

C. Partitioning schemes

We propose to tackle both challenges by partitioning the domain and having each partition write its own dataset, restricting communication and MPI synchronization to within

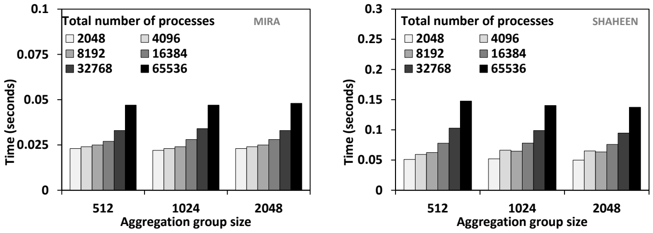
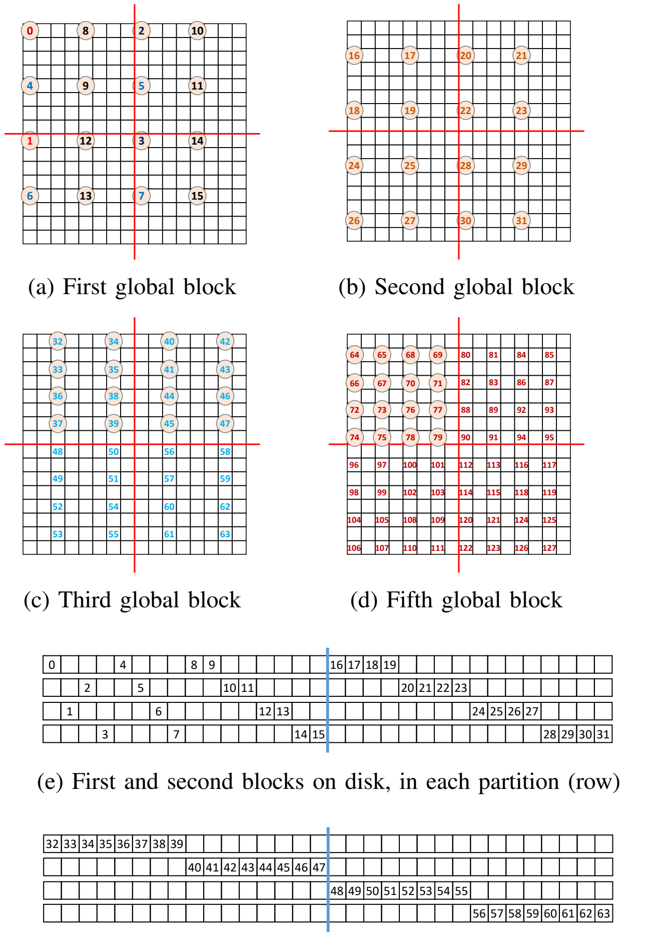


Figure 9: Aggregation timing for different aggregation group sizes on Mira (left) and Shaheen (right). Number of cores is from 2K (8 GB) to 65K (256 GB). For all core counts, the amount of data moved for an aggregation group size is fixed, while the total number of aggregation groups changes.

a partition. Partitioning a regular and non-hierarchical grid is straightforward, as demonstrated in [22]. For a hierarchical grid however, if each partition writes its own dataset, low-resolution samples will be scattered across files. We propose two solutions to this problem. In the first solution, the partitions write data in the global index space, which puts most of the data in its correct block on disk, and we use a serial routine to merge the low-resolution samples from across partitions. In the second approach, each partition writes completely in its own local index space.

1) Writing data in global index space: Figure 10 illustrates the global index space approach. Four partitions are imposed on a 16×16 grid and each file block contains 16 samples. The first two blocks straddle across all partitions, the third and fourth each straddles across half the partitions, but the fifth block onwards are contained within a partition's boundary. In general, if there are R partitions, and B blocks ($B > R$), there will be two all-partition blocks (the first and the second one), $R - 2$ multi-partition blocks, and $B - R$ single-partition blocks. Multi-partition blocks that are written in global index space will result in holes in the files, and will be replicated on each of the partitions (see figure 10, (e) and (f)). The number of replicated global blocks is R.

For each of these R blocks, we use a single-process merge routine that reads and merges the replicas in memory into a single block and writes it back to disk. The actual amount of data that needs to be read from disk and merged is $BR \log_2 R$ where B is the size of a block in bytes. The number of replicas that are written decreases exponentially as one progresses through resolution levels, so the number of replica blocks is typically a small fraction of the total number of blocks. As an example, for a 512^3 data set consisting of 4096 blocks and 4 partitions, the first 4 global blocks will be written as 12 blocks on disk, for an overhead of $8/4096 \approx 0.2\%$. Empirically we observe that merging 2, 4, 8 and 16 blocks (from 512 KB to 4 MB in size respectively) from 2, 4, 8 and 16 partitions respectively took 0.1, 0.4, 0.5 and 0.7 seconds on a Mira login node.



(a) First global block

(b) Second global block

(c) Third global block

(d) Fifth global block

(e) First and second blocks on disk, in each partition (row)

(f) Third and fourth blocks on disk, in each partition (row)

Figure 10: Partitions writing in global index space. Samples in circles belong to the same block. Red lines indicate partition boundaries. In (e) and (f) each row shows the data as written on disk, for one partition. Blue lines separates adjacent blocks. Nothing is written in empty cells.

2) Writing data in local index space: Alternatively, each partition can write a separate dataset in its local index space, with no knowledge of the global dataset (see figure 11). The advantage of this approach is that there is no data replication or post-process merge as seen in the global index space approach. However, the reader of data now has to know that the different datasets belong to one single, global dataset, so that data queries can be answered and displayed correctly. Another problem with this approach, as figure 11 shows, is that if we keep the size of a block fixed for I/O efficiency reasons, the data can no longer be queried below certain resolution levels. For example, in figure 10a we can read the first block of the data and obtain a coarse representation consisting of samples in the first five resolution levels. However in figure 11a when data is written in per-partition local index space, the first block gives one-quarter of a finer representation that corresponds to the first seven levels instead of five. Two levels are "lost", due to the fact that the domain is composed of $2^2 = 4$ partitions. Fortunately this is not a big problem in practice, for two reasons. First, as mentioned in the last paragraph of section III-B, the first few resolution levels give too little information to be useful. Second, the number of partitions in practice is not so large that too many coarse levels are lost.

As writing in global and local index space both have their pros and cons, we consider both as viable solutions.

D. *Partitioning based on the indexing template*

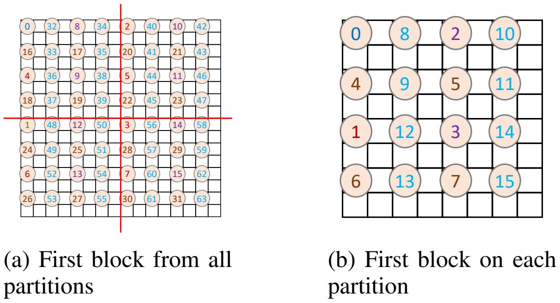
For both global and local index space partitioning to work as intended, the partitioning axes must follow the indexing template so that each partition stores only samples that are contiguous in GHZ space. When this condition is satisfied, most blocks written under domain partitioning will contain exactly the same data as if they were written without domain partitioning, so in the global index space approach, few samples need merging, and in the local index space approach, fewer resolution levels are lost. To enforce this condition we split the indexing template into two pieces. Given R partitions, the first $\log_2 R$ characters are the *partition template* and the remaining characters on the right are the *local indexing template*.

The partition template determines the partitioning planes. For example, if the goal is to create 8 partitions and the indexing template is $yzxxyz$ then the partition template is yzx (the first $3 = \log_2 8$ characters). The first partition plane must divide the domain in half along Y, the next along Z and the last one along X. After the partitions are identified, we assign one sub-communicator to each group of processes sharing a partition. The global communicator is not used anymore for aggregation, and there is also no aggregation of data across the partitions, as each writes data in complete isolation from other partitions, either in global or in local index space.

The remaining characters of the indexing template after removing the partition template constitute the local indexing template. The local indexing template is used to write in local index space. When writing in global index space we use the original, global indexing template.

E. *Evaluation*

In this section we report several experiments that quantify the effects of domain partitioning on data aggregation and file I/O at scale. We adopt the global index space approach here, but do not expect the performance of the local index space approach to differ significantly. In figure 12 (top row) we vary the number of partitions and measure total aggregation time. Aggregation time reduces significantly every time the number of partition doubles (solid black trend line), indicating that domain partitioning is effective in mitigating communication bottlenecks at high core counts.



(a) First block from all partitions

(b) First block on each partition

0	4	8	9	16	17	18	19	32	33	34	35	36	37	38	39
2	5	10	11	20	21	22	23	40	41	42	43	44	45	46	47
1	6	12	13	24	25	26	27	48	49	50	51	52	53	54	55
3	7	14	15	28	29	30	31	56	57	58	59	60	61	62	63

(c) First block stored on disk for each partition.

Figure 11: Partitions writing in local indexing space. Each partition writes its own dataset in complete isolation. (a) shows that all the four first blocks combine to "consume" the first 7 HZ levels, as opposed to figure 10a, where the first block only consumes only the first 5 HZ levels.

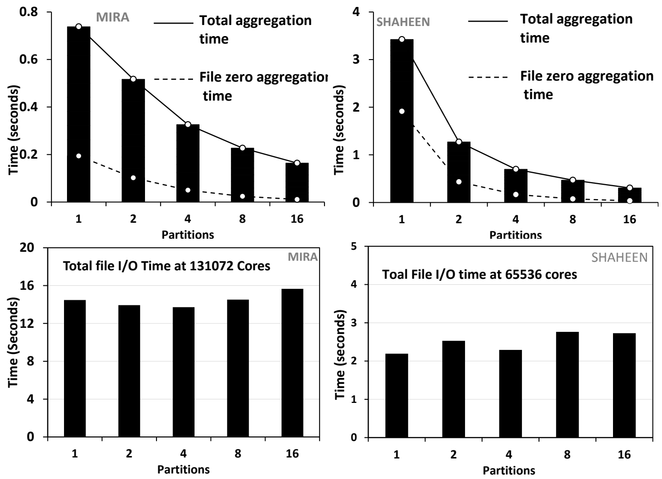


Figure 12: Effects of domain partitioning on the aggregation (top) and file I/O (bottom) with varying partition counts on Mira (left) and Shaheen (right). 131K processes are used on Mira and 65K on Shaheen. Number of partitions varies from 1 (no partition) to 16. Per-process load is 4 MB.

The effects of domain partitioning on file I/O performance can be seen in figure 12 (bottom row). The performance is largely unaffected by the number of partitions, most likely because our localized aggregator scheme successfully distributes the load uniformly across the I/O nodes.

We also run weak scaling experiments to understand the efficacy of partitioning at scale. Figure 13 shows that aggregation time remains fixed at all scales. It is therefore possible to achieve a high degree of scalability by adopting domain

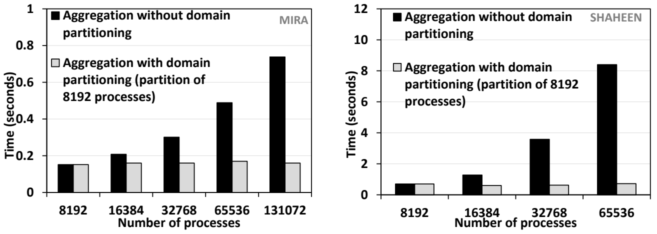


Figure 13: Weak scaling results of the aggregation phase using domain partitioning on Mira (a) and Shaheen (b). Number of processes: 8K to 131K on Mira, 8K to 65K on Shaheen; data per process: 4 MB. The number of partitions varies from 1 (at 8K processes) to 16 (at 131K processes).

partitioning. Depending on the application and the platform under consideration, a good partition size in practice can range from 1024 to 8192 processes.

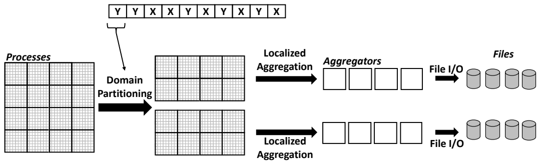


Figure 14: End-to-end data movement framework.

V. End-to-end data movement pipeline

In this section, we present how localized aggregation and domain partitioning fit together in our data movement framework (figure 14). The first step is deriving an optimized indexing template. In the example in the figure, there are 4×4 processes laid out in row-major order, each containing 8×8 samples giving 32×32 total samples. Note that if each process holds a non-constant or a non-power-of-two number of samples in the beginning, we need to impose a restructuring step that re-distributes the samples properly. The 8×8 samples per process gives rise to the interleaved template suffix $yxyxyx$, while the row-major ordering of the processes suggests using the template prefix $yyxx$. So the whole indexing template is $yyxxyxyxyx$. We can break the domain into two partitions along the Y axis, because the first template character is y. If we were to create 4 partitions, we would use yy, and so on. These partitions write data in parallel, each using localized aggregation.

A. Comparison of optimization techniques

We developed a micro-benchmark to quantify the performance improvement due to our two proposed techniques. Our baseline aggregation scheme uses uniform distribution of aggregators similar to [23]. The second implementation uses localized aggregation alone, and the third one uses both localized aggregation and domain partitioning. It can be seen in figure 15 that the domain partitoning is necessary on both machines to get scalable performance. Localized aggregation brought no improvement on Shaheen II, as opposed to a 20% improvement on Mira. This can be attributed to the differences in network topologies of the two machines, as explained in Section III-C.

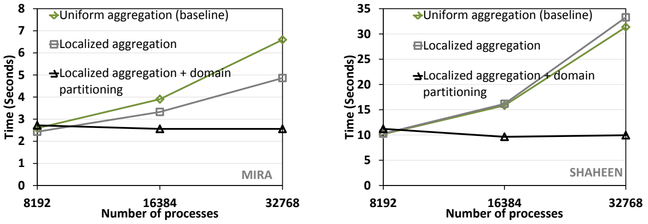
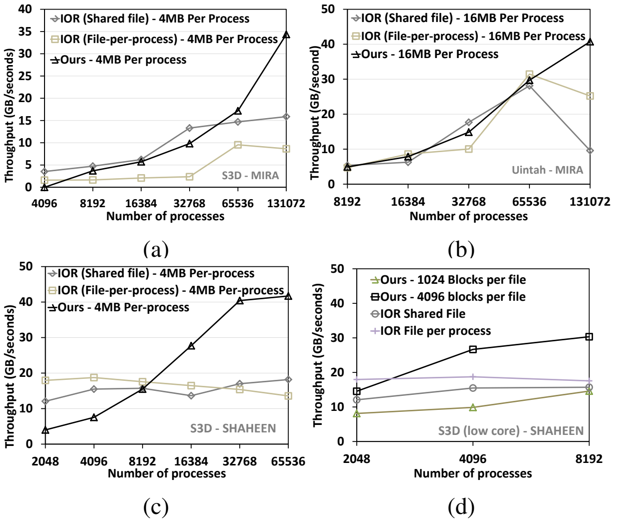


Figure 15: Efficacy of localized aggregation and domain partitioning. Per-process resolution is 32^3, 16 variables, using doubles. In domain partioning, number of partition is 1 (8192), 2 (16384) and 4 (32768).



(a)

(b)

(c)

(d)

Figure 16: Results for weak scaling of PIDX I/O on Mira for S3D (a) and Uintah (b) simulations. (c) Results for weak scaling of PIDX I/O on Shaheen using S3D. (d) Scaling result of PIDX for low core counts on Shaheen.

B. Evaluation using S3D and Uintah

In this section we use weak scaling to evaluate the performance of our I/O framework with real applications, namely S3D [24] and Uintah [25]. With S3D, each process writes a 32^3 block and 16 fields of double precision data (4 MB). The amount of data written varies from 8 GB (4K processes) to 256 GB (131K processes) on Mira, and 4 GB (2K processes) to 128 GB (65K processes) on Shaheen. We also report IOR [26] timing numbers for both file-per-process and shared file I/O using MPI collective I/O.

On Mira, the partition size is 4096 processes, so the number of partitions varies from 1 to 32, at 4K and 131K processes respectively. In contrast to Mira, on Shaheen we observe that the aggregation time dominates the actual file write time. Hence, we reduce the partition size to 1024 processes, as smaller partition sizes reduce aggregation time without affecting file write time (see section IV-E). On Mira, figure 16a shows that our I/O framework scales well up to 131K processes, and it performs better than IOR for most process counts. A similar trend is seen on Shaheen (figure 16c), except at low process counts. To achieve better performance at these low process counts we further increase the number of partitions to 1024 and use bigger block sizes of 1024 and 4096 instead of 256 samples, for more efficient disk access (figure 16d).

Uintah uses task-based parallelism, but from I/O point of view differs from S3D only in terms of the number of fields. For Uintah simulation, each process holds a 32^3 block of double precision data (16 MB) consisting of 64 fields. On Mira, we varied the number of processes from 8K (128 GB) to 131K (2 TB). Each partition contains 4096 processes. It can be seen in figure 16b that IOR scales up to 65K processes compared to 13K for our framework. Overall the promising results suggest that the techniques introduced in this paper can help our I/O framework achieve good scalability with large-scale simulations.

VI. Conclusion and Discussion

We have presented two techniques for efficient aggregation of multi-resolution data, namely domain partitioning and localized aggregation. Both are designed to take advantage of the fractal-like structure of the hierarchical Z indexing scheme, which we generalized using the concept of the indexing template. With indexing templates we can optimize the distribution of aggregators to match well with the underlying rank distribution of processes, as well as find the optimal partitioning axes. Our overall data movement framework exposes parameters (indexing template, number of partitions, and number of aggregators) that control the flow of data over network, and can be tuned depending on the current configuration. Our techniques are generic, and can be easily applied to regular, non-hierarchical grids (by simply replacing HZ with Z indexing), as well as to other I/O systems. For example, in other experiments we have seen an 8x improvement in throughput using domain partitioning on parallel HDF at 32K processes on both Mira and Shaheen.

Our generalized HZ indexing can also be used to index wavelet transform subbands, thanks to the discrete wavelet transform using the exact even-odd style of splitting samples into subbands [27]. Figure 17 (left half) illustrates the correspondence between HZ indexing and kdtree-style wavelet subband decomposition. We can also obtain an indexing scheme corresponding to the more common octree-style subband decomposition for wavelet (figure 17, right half), by changing the HZ indexing formula slightly. This means the techniques described in this paper can be used for writing large grids of wavelet-transformed data on supercomputers, which is valuable as wavelets have proved to be very effective in compression and progressive streaming of large scientific data [3], [28], [29].

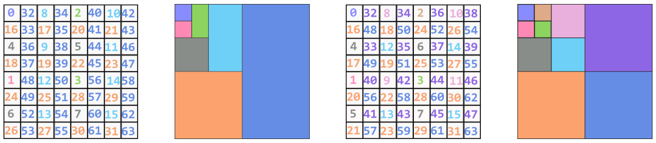


Figure 17: Correspondence between wavelets subband decomposition and hierarchical indexing. (Left) kd-tree indexing corresponds to (middle-left) kd-tree subband decomposition. (Middle-right) Oct/quad-tree indexing corresponds to (right) oct/quad-tree subband decomposition. Samples are colored by wavelet subbands.

Acknowledgement: This work is supported in part by NSF: CGV: Award:1314896, NSF:IIP Award :1602127 NSF:ACI:award 1649923, DOE/SciDAC DESC0007446, CCMSC DE-NA0002375, and PIPER: ER26142 DE-SC0010498. The research used (under INCITE program) resources of the ALCF, a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We also thank the Supercomputing Laboratory at KAUST in Saudi Arabia. This work was also supported by the Department of Energy Office of Science, Office of Advanced Scientific Computing Research under the guidance of Dr. Lucy Nowell and Richard Carlson.

References

[1] Y. Tian, S. Klasky, W. Yu, B. Wang, H. Abbasi, N. Podhorszki, and R. Grout, "Dynam: Dynamic multiresolution data representation for large-scale scientific analysis," in *2013 IEEE Eighth International Conference on Networking, Architecture and Storage*, July 2013, pp. 115–124.

[2] V. Pascucci and R. J. Frank, "Global static indexing for real-time exploration of very large regular grids," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, ser. SC '01. ACM, 2001, pp. 2–2.

[3] C. Baldwin, G. Abdulla, and T. Critchlow, "Multi-resolution modeling of large scale scientific simulation data," in *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, ser. CIKM '03. New York, NY, USA: ACM, 2003, pp. 40–48.

[4] 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.

[5] 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, no. 5, pp. 31–38, Dec. 1993.

[6] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in romio," in *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, ser. FRONTIERS '99. IEEE Computer Society, 1999.

[7] P. Balaji, A. Chan, R. Thakur, W. Gropp, and E. Lusk, "Toward message passing for a million processes: characterizing mpi on a massive scale blue gene/p," *Computer Science - Research and Development*, vol. 24, no. 1, pp. 11–19, 2009.

[8] J. Lofstead and R. Ross, "Insights for exascale io apis from building a petascale io api," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. ACM, 2013.

[9] R. Thakur, W. Gropp, and B. Toonen, "Optimizing the synchronization operations in message passing interface one-sided communication," *The International Journal of High Performance Computing Applications*, vol. 19, no. 2, pp. 119–128, 2005.

[10] C. Christensen, J. Lee, S. Liu, P. Bremer, G. Scorzelli, and V. Pascucci, "Embedded domain-specific language and runtime system for progressive spatiotemporal data analysis and visualization," in *6th IEEE Symposium on Large Data Analysis and Visualization, LDAV 2016, Baltimore, MD, USA, October 23-28, 2016*, 2016, pp. 1–10.

[11] 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.

[12] S. Kumar, A. Saha, V. Vishwanath, P. Carns, J. A. Schmidt, G. Scorzelli, H. Kolla, R. Grout, R. Latham, R. Ross, M. E. Papkafa, J. Chen, and V. Pascucci, "Characterization and modeling of pidx parallel I/O for performance optimization," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. ACM, 2013, pp. 67:1–67:12.

[13] S. Kumar, C. Christensen, J. Schmidt, P.-T. Bremer, E. Brugger, V. Vishwanath, P. Carns, H. Kolla, R. Grout, J. Chen, M. Berzins, G. Scorzelli, and V. Pascucci, "Fast multiresolution reads of massive simulation datasets," in *Supercomputing*, ser. Lecture Notes in Computer Science, J. Kunkel, T. Ludwig, and H. Meuer, Eds. Springer International Publishing, 2014, vol. 8488, pp. 314–330.

[14] S. Kumar, J. Edwards, P.-T. Bremer, A. Knoll, C. Christensen, V. Vishwanath, P. Carns, J. A. Schmidt, and V. Pascucci, "Efficient I/O and storage of adaptive-resolution data," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 413–423.

[15] ANL. (2017) Mira home page. [Online]. Available: https://www.alcf.anl.gov/mira

[16] KAUST. (2017) Shaheen II home page. [Online]. Available: https://www.hpc.kaust.edu.sa/content/shaheen-ii

[17] G. M. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.

[18] V. Pascucci and R. J. Frank, "Hierarchical indexing for out-of-core access to multi-resolution data," Lawrence Livermore National Laboratory, Tech. Rep. UCRL-JC-140581, 2001.

[19] H. Bui, H. Finkel, V. Vishwanath, S. Habib, K. Heitmann, J. Leigh, M. Papka, and K. Harms, "Scalable parallel I/O on a blue gene/q supercomputer using compression, topology-aware data aggregation, and subfiling," in *Proceedings of the 2014 22Nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, ser. PDP '14.

[20] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, "Topology-aware data movement and staging for I/O acceleration on blue gene/p supercomputing systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. ACM, 2011, pp. 19:1–19:11.

[21] F. Tessier, P. Malakar, V. Vishwanath, E. Jeannot, and F. Isaila, "Topology-aware data aggregation for intensive I/O on large-scale supercomputers," in *Proceedings of the First Workshop on Optimization of Communication in HPC*, ser. COM-HPC '16. IEEE Press, 2016, pp. 73–81.

[22] W. Yu and J. Vetter, "Parcoll: Partitioned collective I/O on the cray xt," in *2008 37th International Conference on Parallel Processing*, Sept 2008, pp. 562–569.

[23] S. Kumar, V. Vishwanath, P. Carns, J. Levine, R. Latham, G. Scorzelli, H. Kolla, R. Grout, R. Ross, M. Papka, J. Chen, and V. Pascucci, "Efficient data restructuring and aggregation for I/O acceleration in PIDX," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012, pp. 1–11.

[24] 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.

[25] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins, "Investigating applications portability with the Uintah DAG-Based runtime system on PetScale supercomputers," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013.

[26] 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 the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2008, pp. 1–12.

[27] W. Sweldens, "The lifting scheme: A construction of second generation wavelets," *SIAM Journal on Mathematical Analysis*, vol. 29, no. 2, pp. 511–546, 1998.

[28] J. Woodring, S. Mniszewski, C. Brislawn, D. DeMarle, and J. Ahrens, "Revisiting wavelet compression for large-scale climate data using jpeg 2000 and ensuring data precision," in *2011 IEEE Symposium on Large Data Analysis and Visualization*, Oct 2011, pp. 31–38.

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