Load-balancing Parallel I/O of Compressed Hierarchical Layouts

Ke Fan*, Duong Hoang†, Steve Petruzza‡, Thomas Gilray*, Valerio Pascucci†, Sidharth Kumar*
* University of Alabama at Birmingham, Birmingham, AL, USA
† SCI Institute, University of Utah, Salt Lake City, UT, USA
‡ Utah State University, Logan, Utah, USA
Email:kefan@uab.edu

***Abstract*—Scientific phenomena are being simulated at ever-increasing resolution and fidelity thanks to advances in modern supercomputers. These simulations produce a deluge of data, putting unprecedented demand on the end-to-end data-movement pipeline that consists of parallel writes for checkpoint and analysis dumps and parallel localized reads for exploratory analysis and visualization tasks. Parallel I/O libraries are often optimized for uniformly distributed large-sized accesses, whereas reads for analysis and visualization benefit from data layouts that enable random-access and multiresolution queries. While multiresolution layouts enable interactive exploration of massive datasets, efficiently writing such layouts in parallel is challenging, and straightforward methods for creating a multiresolution hierarchy can lead to inefficient memory and disk access.**

In this paper, we propose a compressed, hierarchical layout that facilitates efficient parallel writes, while being efficient at serving random access, multiresolution read queries for post-hoc analysis and visualization. To efficiently write data to such a layout in parallel is challenging due to potential load-balancing issues at both the data transformation and disk I/O steps. Data is often not readily distributed in a way that facilitates efficient transformations necessary for creating a multiresolution hierarchy. Further, when compression or data reduction is applied, the compressed data chunks may end up with different sizes, confounding efficient parallel I/O. To overcome both these issues, we present a novel two-phase load-balancing strategy to optimize both memory and disk access patterns unique to writing non-uniform multiresolution data. We implement these strategies in a parallel I/O library and evaluate the efficacy of our approach by using real-world simulation data and a novel approach to microbenchmarking on the Theta Supercomputer of Argonne National Laboratory.

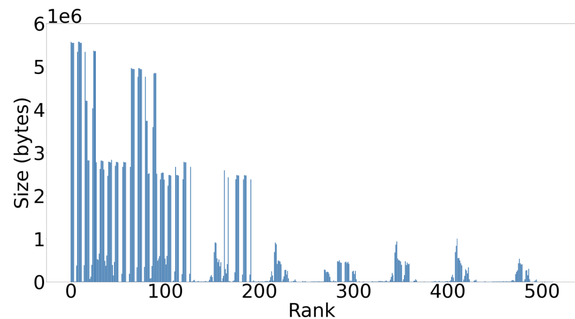
***Index Terms*—Parallel I/O, load-balancing, multiresolution, precision, compression, data layout, aggregation.**

I. Introduction

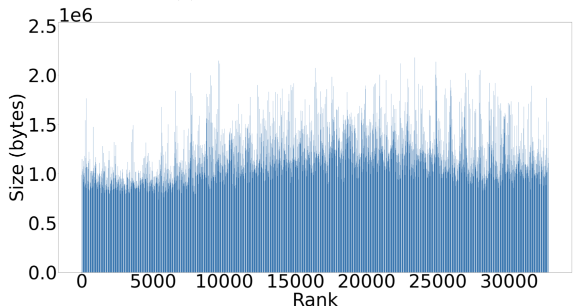
Effectively managing the deluge of data we expect at exascale is critical in ensuring scientific progress across domains. Thanks to massive hardware improvements and larger clusters, scientists are performing more complex and accurate simulations, generating enormous data sets in the process (now hundreds of gigabytes per time step or more). At the same time, post-hoc analyses of such datasets are often performed with modest computational resources. This necessitates compact data formats that support both low latency random-access reads and progressive, multiresolution queries, so that users can efficiently work with data by loading only the necessary bits. In particular, progressive multiresolution data layouts allow exploration of large volumes of data with low latency. Coarser resolutions of a scientific dataset can be accessed almost instantaneously while further details can be loaded progressively (and asynchronously). Progressive data layout thus enables interactive visualization and exploration of very large volumes of data, and has been successfully used in simulation-visualization pipelines [1]. However, writing traditional multiresolution data layouts directly from applications in parallel is challenging, mainly due to the communication overhead involved in gathering initial coarser resolution data that is spread across the entire spatial domain. Previous studies using multiresolution data layouts [2] have demonstrated that parallel I/O performance is often limited by large collective communication operations among processes (i.e., required to collect coarse level resolution data).

We observe that a global multiresolution hierarchy can also be implicitly constructed by having multiple independent local hierarchies, one for each localized chunk of data, termed *patch* in this paper. This approach avoids gathering coarse-resolution data globally at write time because the patches can be written independently instead. Furthermore, because each patch has its own hierarchy, I/O schemes have complete freedom in organizing patches for optimal I/O performance without the risk of disturbing the global hierarchy. With the patch resolution (size) being configurable and not being dependent on global resolution or the number of processes, there is additional flexibility in controlling the optimal file size (and hence, I/O burst size) during parallel I/O. At query time, data at any resolution level can be retrieved and assembled from multiple patches. Based on these observations, we propose a patch-based data layout, where multiresolution hierarchies are created independently and locally within every patch, instead of creating a global multiresolution hierarchy spanning the entire spatial domain. This approach is more amenable to parallel I/O than traditional hierarchical layouts while still allowing for fast multiresolution access.

In practice, data is often transformed before being written to relieve I/O stress, for example with filtering, feature reduction, and compression. In our framework, to achieve both a multiresolution and compact layout, we employ the wavelet transform, followed by compressing the wavelet coefficients with ZFP [5]. Such transformations, interposed between simulation output and I/O, present a load-balancing problem:



(a) S3D flame dataset [3]



(b) Turbulence dataset [4]

Fig. 1: Size of data held by processes after wavelet transform and ZFP compression for two different datasets demonstrates the imbalanced nature of data distribution. The S3D flame data is spread across 512 processes and the turbulence dataset across $32{,}768$ processes. It can be seen that the degree of imbalance is greater for the S3D (compared to turbulence).

some processes may need to do significantly more work than others because they host more patches or more data-heavy patches. We solve these problems through (i) an optimal patch distribution phase that ensures processes end up with a similar number of patches, and then (ii) performing a load-balancing aggregation phase to balance the data load during I/O across a small number of aggregator processes, while retaining some coarse spatial locality.

The patch distribution phase addresses the problem of balancing the number of patches per process, a mapping that greedy approaches do poorly. After the patches are distributed uniformly across processes, wavelet transformation and zfp compression are applied independently to them. This further creates a load-balancing problem, as reduction rates may (and often do) vary widely across the simulation domain (see Figure 1), creating uneven, non-uniform data loads across processes. This imbalance in load percolates to the parallel I/O layer, causing sub-optimal performance. We mitigate this problem with our load-balancing aggregation scheme, which distributes compressed patches across aggregators to be even in terms of bytes. We use a relatively smaller number of aggregators before final disk I/O to collect data from simulation processes, while also preserving the spatial locality of the patches. Spatial locality is important as nearby patches are likely to be queried together at reading time, so storing them close together on a disk typically reduces read latency. Besides ensuring near-uniform data distribution across aggregators, our scheme endeavors to assign nearby patches to the same aggregator so that they are written to the same file on disk.

Load balancing at I/O time is a key problem to solve, not only because data reduction techniques are becoming increasingly universal, but also because certain simulation data types are inherently imbalanced, such as AMR grids [6] or particles [7]. Most I/O libraries today either are designed to write near-uniform data distributions (e.g., raw regular grids) or have not adequately addressed the balancing problem. As such, except for proposing solutions to this problem in novel distribution and aggregation phases, we also introduce an I/O benchmark that simulates real-world non-uniform data distribution patterns at large scales. We achieve this goal by extrapolating post-compression patch sizes from relatively small data sets to any scale. Our benchmark is useful for testing and tuning parallel I/O libraries because it does not rely on a computationally expensive simulation being run or any large-scale extensional dataset to be used while retaining important statistics of compressed patches originating from real-world data. In summary, we make the following specific contributions to the literature:

1) A compressed, patch-based, hierarchical data layout that is amenable for parallel I/O while effectively supporting random access, multiresolution queries.
2) A patch distribution phase that facilitates load balancing for data transformation while minimizing data movement. Empirical evaluation shows a $2.5\times$ improvement over a non-balancing approach.
3) An aggregation phase that balances non-uniform data. Experiments show a $1.3\times$ improvement over a non-balancing approach.
4) A novel I/O benchmark that simulates non-uniform patch size distributions found in scientific data at scale and an experimental evaluation of different I/O pipelines, including traditional I/O approaches.

Our patch distribution phase is specific to multi-resolution data layouts and can be used by any parallel I/O library [8] that supports such data. However, our load-balanced aggregation technique is general-purpose and can be used for any non-uniform data distribution (irrespective of its source), and thus can also be adopted by existing parallel I/O libraries such as PnetCDF [9], HDF5 [10] and ADIOS [11].

II. Related work

Parallel I/O for grid-based and structured datasets has been widely explored. However, few studies [**?**] have focused on parallel I/O for non-uniform load distributions and in particular for compressed multiresolution data. It is partly due to the challenges of working with non-uniform data distributions. Developers of simulation runtimes and scientific applications are generally more focused on load balancing the computation workload and much less on improving the file I/O pipelines. It shifts the challenge to parallel file systems, where there are

few efforts to alleviate load imbalance on I/O servers [12]. Popular I/O libraries, such as PnetCDF [9], parallel HDF5 [10] and ADIOS [11], are built on top of MPI collective I/O [13], which uses two-phase I/O by default. The two-phase I/O in MPI-I/O defaults to one shared file, which often results in sub-optimal performance.

Parallel I/O libraries are often tunable, as some strategies work better on different networks, file system configurations, or levels of parallelism. State-of-the-art parallel I/O libraries such as ADIOS, PnetCDF, Parallel HDF5, and PIDX use a suite of I/O transformations to effectively translate distributed-application data layouts to file-level bitstreams. Two factors are key while writing data in parallel: how many processes are accessing a file, and how many files are being written in total. Common strategies used by these libraries are: file per process, single shared file, two-phase I/O, and subfiling. In file-per-process, every process writes its data to an independent file, whereas with shared file I/O processes, they write data to a single shared file. It is well-known in the parallel I/O community that writing to a single shared file or using the file-per-process mode will lead to sub-optimal performance [7]. While file-per-process I/O suffers from metadata overhead due to the massive number of files produced, shared-file I/O typically suffers from file-locking contention when every process attempts to write at once. Similarly, allowing every process to perform its own I/O leads to sub-optimal performance [14].

The latter two strategies, two-phase I/O and subfiling, balance between file per process and single shared file approaches, to provide portable, scalable, and tunable I/O strategies. Two-phase I/O strategies [1], [13], [15], [16] begin by assigning a configurable number of processes to be data aggregators. The non-aggregator processes are assigned a data aggregator to send their data to, forming a subgroup. The processes then send their data over the network to their assigned aggregator, which writes a single file out after it has received data from the processes in its subgroup. This scheme restricts the number of processes that need to access the parallel file system. Subfiling [17], [18] works similarly, though it does not necessarily aggregate the data over the network into an aggregator process before writing it out. Subfiling strategies group processes into subgroups, then perform single shared file writes within the subgroups, outputting a file per subgroup. Subfiling controls the number of files created while two-phase I/O with data aggregation controls the total number of processes that access the parallel file system. These two schemes provide a set of portable, scalable, and tunable I/O strategies.

Finally, in terms of performance assessment, several tools exist to characterize the message passing capabilities of a system [19] [20] and assess its peak performance. A few of them, like MADBench2 [21], assess I/O performance using application-driven I/O loads, for example, simulating Cosmic Microwave Background data analysis. One of the most common tools used to perform I/O benchmarks is IOR [22], which allows defining a per-process buffer size and various settings to perform file I/O, ultimately relying on POSIX or MPI I/O APIs. This allows for experiments using file-per-process and collective I/O patterns for a uniform data distribution among the ranks. The ability to create uneven data distribution across processes is not supported.

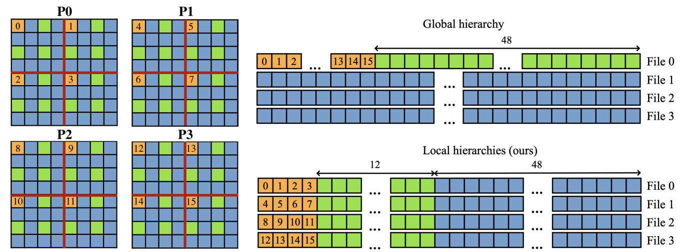


Fig. 2: A 16×16 simulation domain, for example, is divided into 16 patches of 4×4 each. Each of the four processes contains four patches. Each patch's grid samples are transformed into three resolution levels (orange, green, and blue). When four files are written in the global hierarchy (top right), the first file contains coarse-level data samples from all patches, forcing expensive global communication. Each file in our layout, which consists of four local hierarchies (bottom right), only gathers data locally from its own patches.

III. Compressed hierarchical layout

In this section, we introduce our compressed hierarchical layout for scientific data and discuss how it is designed to facilitate fast parallel writes. We start with the observation that traditional hierarchical layouts [2] tend to require I/O libraries to gather coarse-resolution data samples across the whole simulation domain, which incurs very expensive global communication at write time. To make parallel I/O scalable, we designed our data layout to be patch based. The patch-based layout avoids the collective communication overhead of creating the global hierarchy at write time because patches can be written independently. As each patch encapsulates its own hierarchy, I/O schemes can flexibly organize the patches for optimal I/O performance without the risk of disturbing the global hierarchy. Furthermore, the ability to tune the patch resolution helps meet the optimal file size (I/O burst size) requirement with more flexibility. If the patch size is fixed to the initial process load, then there is less room to control the I/O burst size. This is because, typically, we will have more patches of smaller sizes (compared to the total number of processes), that can be packed more uniformly across aggregators targeting an optimal I/O burst size.

A. *Patch-based design*

A patch is typically a localized chunk of data. In particular, the simulation grid is partitioned into patches of dimensions $p_x \times p_y \times p_z$, where each dimension is a power of two. Each patch is transformed independently to form a hierarchy of L resolution levels ($L = \log_2(\min(p_x, p_y, p_z)) + 1$). Data samples are assigned to files in units of patches (i.e., samples from the same patch are always written in the same file). The order in which samples are written within each

file is flexible. They can be stored in two ways: (i) sorted by patch indices, then by resolution levels (coarse to fine), or (ii) sorted by resolution levels (coarse to fine), then by patch indices. The first scheme prioritizes random access over progressive multiresolution access, while the second one prioritizes multiresolution access over random access. $p_{x,y,z}$ and storage schemes are controlled by the user, with $p_{x,y,z}$ being a tunable parameter. Smaller $p_{x,y,z}$ values result in a more balanced data-load distribution during the aggregation phase, which improves I/O performance. The patch-based layout also explicitly facilitates efficient random access reads. For parallel I/O, the layout itself does not enforce any policy for assigning patches to files to give I/O libraries more opportunities for optimization. As an example, our I/O library sorts the patches in Morton order [23], enforces that patch indices in the same file are contiguous in this order, and allows the user to choose the number of files (F). Figure 2 illustrates a 2D example of such a layout with $p_x = p_y = 4$, $L = 3$ and $F = 4$; it clearly shows that when compared to a global hierarchy approach, our patch-based design allows I/O libraries to operate with no collective communication overheads, resulting in more scalable I/O.

A possible reason for concern with our patch-based scheme is that since coarse resolution data samples are spread across more files, a query for these samples across the whole domain may require reading a large number of small chunks (one from each file) instead of one big chunk from a single file (as with the global hierarchy scheme). This process may be very slow depending on the size of each chunk. However, it is easy to appropriately choose reasonable values for $p_{x,y,z}$ and the number of files so that each I/O request is for a sizable chunk (tens or hundreds of kilobytes). For example, if $p_x = p_y = p_z = 64$, $L = 5$, each patch contains $4^3 = 64$ data samples at the coarsest resolution level (level-5). If each sample is a float64 and a file contains at least $8^3 = 512$ patches, then the coarsest level occupies a chunk of $512 \times 64 \times 8$ = 256 KB in each file, which is large enough for high throughput I/O.

Although not mandated by the data layout, in practice, data is often transformed and compressed in some way before being stored on the disk. Our I/O library, for example, performs the wavelet transform followed by lossy compression of the wavelet coefficients (Section IV-B). At write time, we transform and compress each patch independently of one another, and at read time, we decompress and inverse transform each patch independently. Therefore, the patch dimensions $p_{x,y,z}$ must be chosen small enough to allow fine-grained random access and to facilitate parallel transform and compression/decompression. They must, however, be large enough to avoid accumulating too much metadata in order to support random access and adaptive refinement. Also, while the patch resolution is independent of the global or the local per-process resolution, a special case can occur when its resolution is set to be the same as the initial per-process resolution. This would avoid any communication cost in the patch distribution phase (every process will have a patch), but, we will lose the flexibility to control the optimal file-size when performing parallel I/O. However, we have found that, in our experiments, 32 and 64 are good choices for $p_{x,y,z}$ (32^3 or 64^3 samples per patch). With such patch sizes, the metadata accounts for only about 2% of the total compressed data, and there are enough patches to distribute among processes so that they can be transformed and compressed efficiently in parallel, resulting in good parallel I/O performance.

B. Data access and reconstruction

In each file, we store the total number of patches it contains, their patch indices, and offsets for all resolution levels of each patch. We also compute for each file a bounding box of all the patches in the file, and store these bounding boxes in a metadata file. At read time, the user can issue a query for data at a certain resolution level and for a certain region of the domain (the region may also be the whole domain). Given such a query, we can quickly intersect the queried region with these bounding boxes to locate the files in which the possibly relevant patches are stored. Since a bounding box for a file may have holes (i.e., patches that are stored in another file), we next intersect the queried region with the extents of the patches themselves to filter out irrelevant patches. Once the relevant patches have been identified, they are loaded from their respective files. We know exactly where to load these patches because their offsets are stored as metadata.

IV. Parallel I/O

Figure 3 illustrates our end-to-end pipeline. To create a compressed multiresolution hierarchy, we apply the discrete wavelet transform, then compress the wavelet coefficients for each patch, following the approach taken by [24]. We have also chosen the B-spline multilinear basis [25] because it is fast to compute and offers great compression opportunities. Each patch is a computational unit, which is independently transformed and compressed. For efficient memory access during these computations, it is important that each patch is worked on by only one process: we want to avoid having a patch shared by two or more processes that necessitate data communication during transformation. The simulation code, however, may distribute data in ways that result in many shared patches – after all, the simulation in general does not work with patches of the same dimensions as our layout. Therefore, before wavelet transform and compression, we need to distribute the patches so that each process contains non-overlap patches.

Ideally, in a two-phase I/O system, the aggregation phase can serve as a patch distribution phase as well, and subsequent transformation and compression are done on aggregator processes. However, such an approach can severely limit the parallelism of those computations, because the number of aggregators is typically very small compared to the number of processes. To leverage the computational power from all processes involved in the simulation, we propose a separate phase, called the patch-distribution phase, to distribute the patches evenly among all processes so that the subsequent

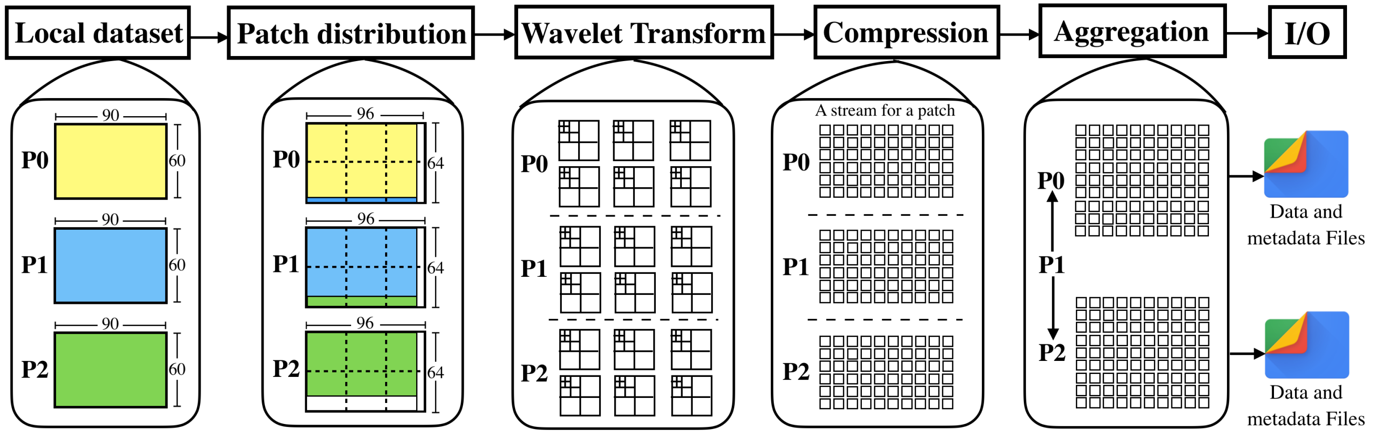


Fig. 3: The end-to-end pipeline of our parallel-I/O library. Local per process data is partitioned across a *tunable* number of patches–this step takes place in the patch distribution phase (section IV-A). We then apply wavelet transform on each of the patches, creating local multiresolution hierarchies (section IV-B). The wavelet coefficients are then compressed with zfp to get rid of redundancies in the data (section IV-B). Finally, the compressed patches, are buffered on few chosen aggregator processes which writes the data (and meta data) to files (section IV-C).

transformation and compression are well load-balanced. After (lossy) compression, we face another balancing issue, this time for disk I/O, which arises due to non-uniform load distributions created by uneven compression ratios across processes. This is because different regions of the domain may be compressed to different sizes, depending on the data being compressed (see Figure 1 for example). The process with the most data after compression is likely to be the last one finishing file I/O operations, degrading the overall performance. Even without compression, scientists often perform data filtering or feature extraction before disk I/O to reduce the amount of data written, which also causes this balancing issue. We tackle this challenge by devising a layout-aware balanced aggregation strategy. In short, our parallel I/O pipeline consists of three distinct phases:

1) Balanced patch distribution (Section IV-A)
2) Parallel wavelet transform and compression (Section IV-B)
3) Balanced data aggregation and file I/O (Section IV-C)

A. Balanced patch distribution

We start by distinguishing between *regular-patches* and *shared-patches*. A *regular-patch* is fully contained within a process before distribution, while a *shared-patch* is shared by two or more processes (i.e,. each process holds some portion of the patch). For example, in Figure 4, out of the 9 patches, 4 are *regular-patches* (ids: 0, 2, 6 and 8) and 5 are *shared-patches* (ids: 1, 3, 4, 5, and 7); *shared-patch* id 4 is shared by all four processes. *Regular-patches* are not moved during the patch distribution phase, hence reducing data movement costs. *Shared-patches*, on the other hand, may need to be moved from their original process to a new process for balancing purposes.

With a total of M patches to be distributed across N processes, a perfectly balanced distribution of patches is ensured when every process gets exactly $\lfloor M/N \rfloor$ patches and the remaining $M \bmod N$ patches are spread-out uniformly across all N processes. A process will therefore either hold $\lfloor M/N \rfloor$ or $\lfloor M/N+1 \rfloor$ patches, and we call this the *target patch count*. For example, in Figure 4 the *target patch count* for processes, blue, green, pink, and yellow are 3, 2, 2 and 2. The *regular-patches* are not moved and are thus assigned to their host process itself. The *shared-patches* that are spread across processes must be assigned to a target process. We go through all of the *shared patches*, assigning each one a target process. Again, to minimize data movement, we *attempt* to assign a *shared-patch* to one of the processes that share that patch—we choose the process that has currently been assigned fewer patches than its *target patch count*, which is either $\lfloor M/N \rfloor$ or $\lfloor M/N+1 \rfloor$. If there are multiple candidates, the process with the smallest rank is chosen. If a target process for the *shared-patch* is found this way, then at-least a chunk of the *shared-patch* will be locally copied (instead of being sent across the network), hence reducing data movement. Alternatively, if all processes that share the *shared-patch* have already reached their *target patch count*, we scan through all processes and assign the patch to the first process that has not reached its *target patch count*. As an example, the first *shared-patch* (id 1) in Figure 4 will be assigned to the blue process, since both blue and green processes have not met their target-patch-counts yet, we chose the process with smaller rank (assuming rank order: blue $<$ green $<$ pink $<$ yellow). The scheme ensures a perfectly uniform distribution of patches.

In our implementation, we divide the set of processes sharing a patch into senders and receivers. The receiver is the one process that the *shared-patch* is assigned to, and the rest are senders. Each sender then sends the region of the patch that it holds to the receiver. This algorithm is run on every process, so each one knows exactly whether it is a receiver or sender for each *shared-patch* and what data to send or receive. `MPI_Type_create_subarray` is used to

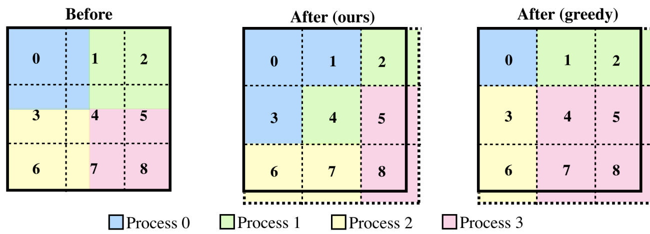


Fig. 4: A 2D example of our patch distribution scheme. 3×3 patches (separated by dash lines and numbered) are distributed among four processes (distinguished by colors). Before the distribution, some patches are shared among processes, resulting in many sub-patch regions. After the distribution using our scheme, the processes have exactly 3, 2, 2, 2 patches respectively. In contrast, the greedy scheme distributes the patches unevenly. Note that boundary patches (have smaller dimensions) may also be padded to regular patches.

define these sub-patch regions that are sent and received over the network, and MPI's non-blocking point to point API is used for data transfer. In Figure 4, we give a small example where we number the patches and color the processes to help to visualize them. After all the sub-patch regions are sent and received, every process ends up with *target-patch count* patches, each of which is stored in a separate contiguous memory block, ready to be transformed.

Perhaps a more straightforward method for choosing the receiver for a *shared-patch* is to pick the one with the largest sub-patch region among the processes that share the patch (i.e., the process that originally contains the most data from the patch). Patch distribution using this greedy scheme has been used, for example in [2], albeit not for balancing the data transformation per patch, but to minimize interleaving of data samples among processes in aggregation buffers. However, in Figure 4 we show that the greedy scheme leads to a very imbalanced patch distribution. In Section V we also show through experiments that in practice, the greedy scheme does indeed result in significantly longer computation time.

B. Wavelet transform and compression

Following patch distribution, the wavelet transform is applied to each patch independently. The transform updates and divides the data samples in each patch into a set of resolution levels, each of which captures information at a specific scale. Because the patches are processed independently, once the patch distribution is completed, there is no need to communicate among processes, such as exchanging boundary data. This deliberate design is intended to extract the most parallelism from the machine for the transformation step. In all our studies, we keep the patch resolution to be powers of two, ensuring that wavelet coefficient at patch boundaries are computed correctly.

Linear and higher-order wavelets are not only useful for low-pass filtering but also for lossy compression. One way to compress is to treat the fine-scale wavelet coefficients as being 0 and do not store them. More sophisticated wavelet encoding schemes such as SPIHT and JPEG2000 exist [26], [27], but they tend to be slow. For performance reasons, we compress wavelet coefficients at each resolution level independently using ZFP [5], a fast compressor for floating-point arrays. Prior work [24] has shown that ZFP in fixed accuracy mode can be used very efficiently as a wavelet compressor. To allow data retrieval of individual resolution levels, we compress each resolution level independently. The compression accuracy (or absolute error tolerance) is a parameter controlled by the user. Because patches can be processed independently of one another, it is possible for I/O libraries to achieve very high degrees of parallelism when performing data transformation, namely, wavelet transform and compression. In Section V, we provide detailed timings of this step with our implementation.

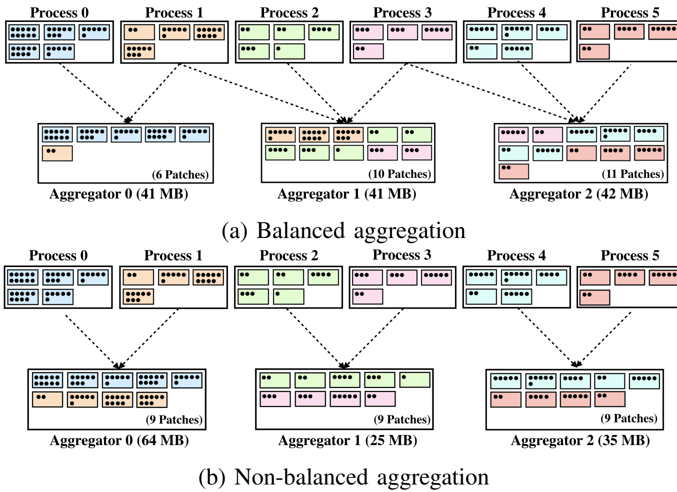
C. Layout-aware balanced data aggregation

The next step is to write the compressed data to disk, in the file layout described in Section III. Lossy data compression may create load imbalance across processes, as different regions of the spatial domains are compressed to different sizes. This can be attributed to the nature of the dataset in question, as some processes have regions with more coherent data, which gets compressed more than processes with regions of higher randomness. This imbalance in load percolates to the I/O layer, causing sub-optimal performance. The problem is more prominent with time-varying simulation datasets, where the degree of imbalance changes over time, necessitating an adaptive parallel I/O system. Figure 1 depicts this imbalance for real-world scientific datasets, demonstrating that most of the datasets show significant variation in load across processes.

We have designed our data layout in a manner that inherently supports writing to a tunable number of files (F). Although MPI has support for collective I/O that internally does data aggregation, it explicitly does not support sub-filing. Sub-filing allows one to use collective access within each communicator group and also write data to a hierarchy of files. This scheme of doing parallel I/O is popular and is widely used by I/O libraries like parallel HDF5 [18]. However, this method does not directly translate well for non-uniform data distributions, as it leads to fewer aggregators writing more data than others, causing sub-optimal performance. To extract the maximum available bandwidth from the hardware, we must have a uniform distribution of I/O load across aggregators.

To deal with load-balancing challenges, we have designed a customized two-phase I/O strategy, that facilitates both sub-filing and ensures that aggregators have similar I/O loads to write. Our aggregation phase takes into account the global view of data distribution across processes while assigning the size and extent of each aggregator. Following are the steps in the aggregation phase: 1) aggregator selection, 2) patch assignment, and 3) patch transfer.

1) Aggregator selection: A key step in tailoring aggregation is to select an appropriate number of aggregators. We allow only one aggregator to write a file, so the aggregator count is equal to the total number of files written. Given that we have designed our data layout to support a flexible number of files,



(a) Balanced aggregation

(b) Non-balanced aggregation

Fig. 5: Our balanced aggregation scheme vs. the non-balanced scheme. Both approaches yield the same number of files (same number of aggregators). However, balanced aggregation generates files of similar size as opposed to the non-balanced scheme where files have varying sizes but the same number of patches. (A filled black circle represents 1M).

we keep that as a tunable parameter that is set by the user. The only limitation is that the total number of files outputted must be less than or equal to the total number of processes. In practice, the number of files generated should be a fraction of the total number of processes. For example, in our evaluation section, we vary the number of files from $nprocs$ (file-per-process) to $nprocs/16$. This aggregator count and file count configuration is in line with sub-filing, and also avoids any file locking contention which can happen with processes that make unaligned accesses to a file. To extract the maximum I/O bandwidth, we place the aggregators uniformly across the rank space. In Section V, we demonstrate the impact of the number of aggregators/files.

2) Patch assignment.: To ensure a balanced I/O phase, we need to assign patches to aggregators so that every aggregator manages roughly the same volume of data. This step would be trivial if processes had similar-sized data loads. However, as our data patches are all of different sizes, a file must contain a different number of patches to achieve a balanced amount of data per-file. To attain similar-sized files, we allow aggregators to receive data from a varying number of processes. We keep the smallest unit of data exchange to be a patch (i.e. a process can send its patches to multiple aggregators), and in the extreme case, it can transmit to as many aggregators as it has patches. This configuration facilitates creating a near-uniform data load across the aggregators. Additionally, it also provides extra flexibility in controlling the file-level layout. For example, we write out patches in Morton order [23] to preserve the spatial locality of patches in the file.

We begin by collecting patch sizes across processes using MPI's `MPI_Allgather`. This allows processes to construct a consistent global view of all patches, which is used to independently identify the aggregators they need to send their data to. As we intend to write out patches to files in Morton order [23], we first sort all patches in Morton order, then scan over this sorted list, allocating patches to aggregator processes in a balanced manner. We keep a running total of compressed patch sizes and progress to the next aggregator process whenever this value exceeds a running average (the remaining data divided by the number of remaining aggregators). When the running sum runs over our target aggregator size, we assign the patches in the running sum to the current aggregator and reset our running sum to 0, progressing to the next aggregator. This approach yields a small dip in the per-aggregator data at the very end as this average declines. This does not create significant balancing problems but may be remediated by undershooting the average at first, or by alternating between going over the average and staying under the average at each aggregator. We plan to experiment further with such improvements in the future.

Dataset Name	Resolution	Type	Size after compression (bytes)	PSNR
Synthetic data	1600 x 1600 x 1600	float	1540263880	37.7
S3D flame [3]	2025 x 1600 x 400	double	786616672	35.6
Turbulence data [4]	4096 x 4096 x 4096	float	38989451056	33.9

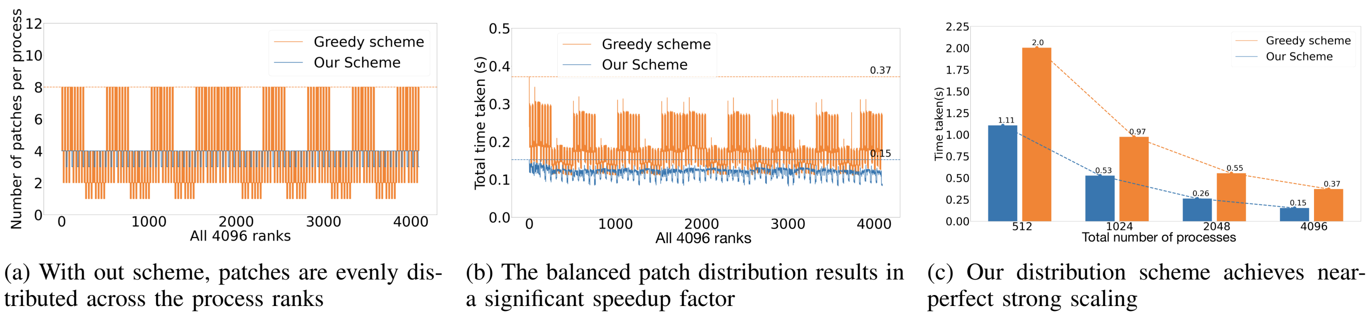
TABLE I: Datasets used in our experiments

3) Patch transfer: The patch assignment step is executed independently by every process, at the end of which every patch is assigned to a target aggregator. Target aggregators use the same step (patch assignment) to identify the patches and, correspondingly, the ranks they are going to receive the data from. This allows the aggregator processes to correctly allocate buffers to accommodate the receiving patches. Processes then transfer their patches to the aggregators using MPI's non-blocking point-to-point communication.

We show an example of our *balanced* aggregation scheme in Figure 5(a). It can be seen that our scheme can support both sub-filing and also perform uniform-sized I/O writes. We compare our approach's performance to that of a *non-balanced* aggregation scheme (Figure 5(b)), which generates the same number of files as the balanced aggregation scheme but writes to non-uniform-sized files. In the *non-balanced* aggregated scheme, every aggregator receives the same number of patches, and since the patches have varying data loads, the aggregators end up with a non-uniform load distribution.

V. Evaluation

We begin by evaluating the efficacy of our balanced patch distribution and aggregation phases. We have used a mix of synthetic and real simulation datasets (see Table I) for our experiments. All our experiments are performed on the Theta Supercomputer [28] at the Argonne Leadership Computing Facility (ALCF). Theta is a Cray machine with a peak performance of 11.69 petaflops, 281,088 compute cores, 843.264 TiB of DDR4 RAM, 70.272 TiB of MCDRAM, and 10 PiB

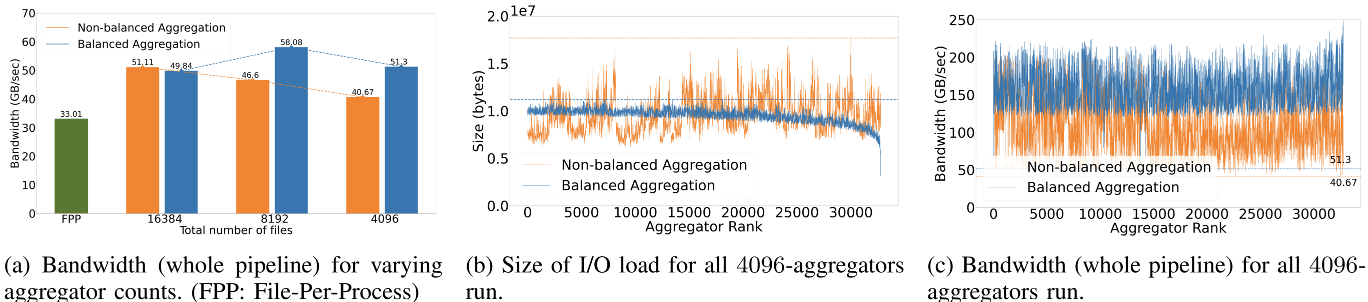


(a) With out scheme, patches are evenly distributed across the process ranks

(b) The balanced patch distribution results in a significant speedup factor

(c) Our distribution scheme achieves near-perfect strong scaling

Fig. 6: Compared to the greedy patch distribution scheme, ours reduces the total combined time of the distribution, wavelet transform, and compression steps by more than half. It also exhibits near-perfect strong scaling behavior. $(4,096$ ranks)

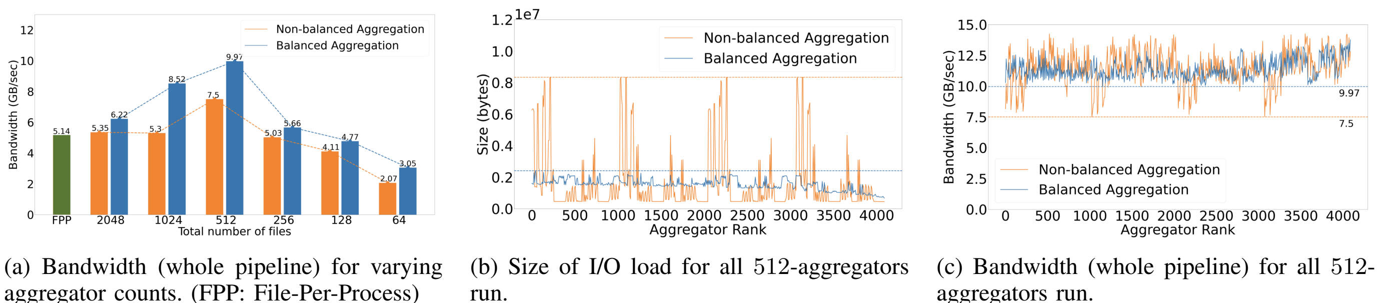


(a) Bandwidth (whole pipeline) for varying aggregator counts. (FPP: File-Per-Process)

(b) Size of I/O load for all 4096-aggregators run.

(c) Bandwidth (whole pipeline) for all 4096-aggregators run.

Fig. 7: Results showing the impact of balanced data aggregation for writing turbulence [4] dataset. $(32,768$ ranks)



(a) Bandwidth (whole pipeline) for varying aggregator counts. (FPP: File-Per-Process)

(b) Size of I/O load for all 512-aggregators run.

(c) Bandwidth (whole pipeline) for all 512-aggregators run.

Fig. 8: Results showing the impact of balanced data aggregation for writing S3D flame [3] dataset. $(4,096$ ranks)

of online disk storage. The supercomputer has a Dragonfly network topology and a Lustre filesystem.

A. Patch distribution phase

In our I/O pipeline (Figure 3), patch distribution happens first, followed by a wavelet-transformation and compression (for each patch). Here we compare our balanced patch distribution scheme with the greedy scheme discussed in Section IV-A. Patch distribution using the greedy scheme has been used extensively in the past, for example, in [2]. In addition to measuring the total time taken to perform patch redistribution, we also time the wavelet-transformation and compression steps to measure the real impact of the patch distribution phase. We use a grid with a resolution 1600^3 and run a strong-scaling experiment with a total number of processes ranging from 512 to $4,096$. The patch dimensions are 64^3, resulting in a total of $1600^3/64^3 = 15{,}625$ patches. We repeated each experiment 10 times, and plotted the medians in Figure 6.

At $4,096$ ranks, our scheme distributes the patches more uniformly (Figure 6a). With the greedy scheme, approximately a quarter of the processes hold eight patches each while a significant number of processes hold only one patch, as opposed to our scheme, where every process holds either three or four patches. The near perfect-balance not only makes the data distribution phase fast, but it also directly impacts the following wavelet-transform and compression phases (Figure 6b). With the greedy scheme, processes with patches must perform more computation overall, causing performance degradation. Figure 6c depicts the results for strong scaling. Our approach takes 1.11s at 512 processes and 0.15s at $4{,}096$ processes, demonstrating near-perfect scaling efficiency, while the greedy approach does not.

B. Load-balanced data aggregation

In this section, we demonstrate the efficacy of the load-balanced aggregation scheme. We compare our method with a *non-balanced aggregation* approach in which each aggregator is given an equal number of patches. Recall that our *balanced* aggregation scheme ensures that aggregators have similar I/O loads and so, as a result, files have similar sizes; this is not true for *non-balanced* aggregation schemes.

We evaluate our scheme on the S3D flame and Turbulence dataset listed in Table 1. The experiments are performed at $4{,}096$ and $32{,}768$ processes (n), respectively. For the S3D flame experiments, we varied the total number of aggregators $(=$ files) from n (file-per-process) to $n/8$. For the Turbulence experiments, we varied the total number of aggregators $(=$ files) from n (file-per-process) to $n/64$. We repeated each experiment 10 times and plotted the bandwidth of the median in Figure 7(a) and 8(a). For the 4096^3-resolution turbulence dataset, our *balanced* aggregation approach yields a peak throughput of 58.08 GiB/second, while the *non-balanced* approach yields a peak throughput of 51.11 GiB/second. *File-per-process* I/O mode yields a throughput of 33.01 GiB/second. These figures represent a 12% improvement in performance over the *non-balanced* approach and a 43.16% improvement over *file-per-process* I/O. For the S3D flame dataset, we observe a maximum bandwidth of 9.97 GiB/second with the *balanced* aggregation, and 7.5 GiB/second with the *non-balanced* aggregation, while *file-per-process* I/O yields a throughput of 5.14 GiB/second. These correspond to a 24.77% improvement in performance over *non-balanced* aggregation and around $2\times$ improvement over *file-per-process* I/O. The *file-per-process* I/O approach can be considered a baseline for comparison with no communication overhead. This comparison demonstrates that although our optimizations have an extra communication phase, they ultimately improve the overall performance.

The improvement (%) in the flame dataset is more than in the turbulence dataset. Figure 7(b-c) depicts the data load and bandwidth of all $4{,}096$-aggregators runs of the turbulence dataset, and Figure 8(b-c) depicts a similar metric for the 256-aggregators run of the S3D flame dataset. From these figures, we observe that the degree of non-uniformity of data distribution varies between the two datasets. With the flame dataset, the *non-balanced* scheme results in groups of aggregators significantly much more data than the others. In the turbulence data, however, the degree of imbalance in data distribution across aggregators is more moderate, even using the *non-balanced* scheme. This difference in load distribution across aggregators for the two datasets can be directly attributed to the initial load distribution across processes (as seen in Figure 1). Clearly, the S3D flame dataset has more variance than the turbulence dataset that percolates to the aggregation layer. This inherent difference in the degree of non-uniformity between the two datasets leads to the S3D data set benefiting more from our balanced aggregation strategy than the turbulence dataset (which has less inherent imbalance). Furthermore, the differences between the two aggregation schemes becomes less discernible for larger aggregator counts, mainly because each aggregator gathers a smaller number of patches. This trend is even more clear for datasets with near-uniform load distribution, such as turbulence. Additionally, the patch assignment overhead of the balanced aggregation scheme is slightly higher than that of the non-balanced one because it must calculate the running average size (Section IV-C2). As a result, in Figure 7(a), the non-balanced aggregation scheme slightly outperforms the balanced one.

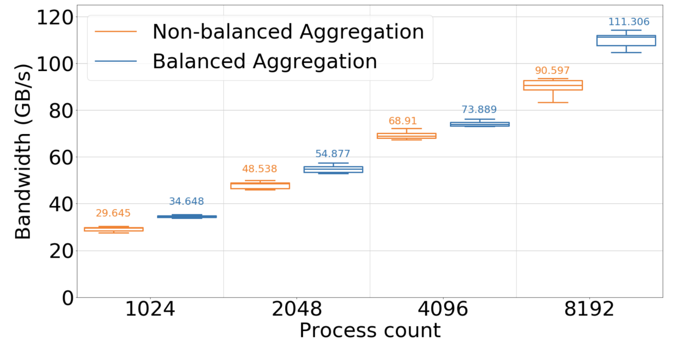


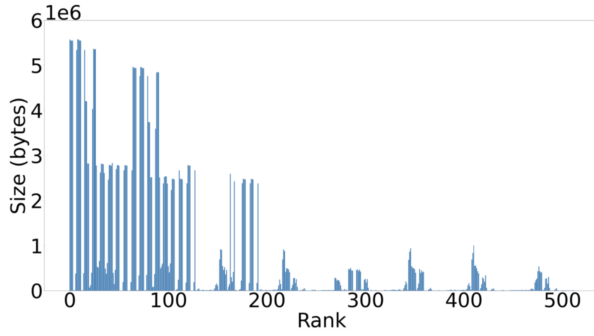
Fig. 9: Strong scaling results for writing the S3D dataset with both balanced and non-balanced aggregation schemes. Balanced aggregation scheme consistently outperforms the other scheme at all process counts.

We also observe that our *balanced* scheme achieves the best performance at aggregator counts of $8{,}192$ and 512 for the two datasets. Finding the optimal number of aggregators is a difficult problem, as it often depends on the scale of the experiment, and in our case, the load distribution pattern. However, we recognize that the Lustre filesystem is more suited to writing data in file-per-process mode and is adept at handling a large number of files. This is further seen in our experiments in this section and in the following section. For our experiments, we have therefore experimented with smaller aggregation factors, leaning more towards the file-per-process I/O spectrum. We believe that our system is flexible in its design, and can be effectively tuned for different filesystems.

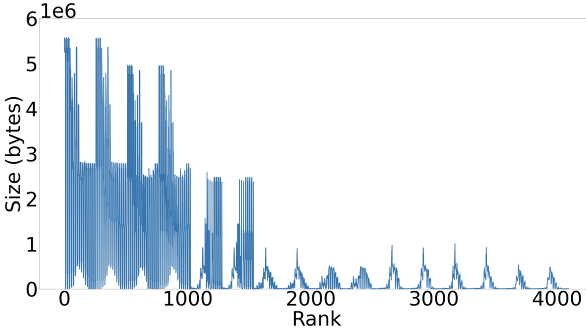
We also perform strong scaling experiments for the S3D dataset and plot the results in Figure 9. Here, we also see our load-balanced aggregation scheme consistently outperform the imbalanced scheme at all process counts.

C. Micro-benchmarks

Traditional I/O benchmarks such as IOR [22] use static, uniform, data distributions replicated across processes to assess system I/O bandwidth. Generating a realistic non-uniform data distribution at scale would require a real simulation to be run (computation overhead), or would need an extensional database (storage overhead). In this work, we introduce a simple micro-benchmark that can mimic the non-uniform data distribution pattern (across processes) of a real scientific application at any arbitrary scale, without any computation or storage overhead. The main idea is to extrapolate the per-



(a) S3D flame dataset ($P = 512$).



(b) Generated data ($P = 4096$)

Fig. 10: Micro-benchmark load distributions. The generated data preserves the load distribution patterns in the original data, representing a good candidate for benchmarking at scale.

process data *size* instead of the actual data. We first consider a scientific dataset and collect the data size that each rank holds in the original 3D grid domain at a small scale (say, at process count $P_x \times P_y \times P_z$). This data size is then stored in a 3D *size*-grid of dimensions P_x, P_y, P_z. With this information, we launch our micro-benchmark at a larger scale (i.e., using a larger number of processes) and assign a new data size to each rank — computed on the fly using trilinear interpolation on the original 3D *size*-grid. At this stage, each rank allocates a buffer with the assigned new data size and runs an I/O pipeline. This approach is inspired by common practices in simulation design, where scientists first prototype small-scale simulation runs using a coarse resolution grid and then increase the resolution of each patch to perform large-scale simulations.

In Figure 10, we can see how the load distribution of the S3D flame dataset (after parallel wavelet transform and compression) using 512 ranks is very similar to our generated load distribution using 4,096 ranks. These plots demonstrate how our micro-benchmark technique preserves the load-distribution patterns (in a 3D domain) of a scientific dataset at scale.

Using our micro-benchmark, we can perform weak-scaling experiments to assess and compare our load-balanced aggregation approach against traditional file I/O. In particular, we ran our experiments at scale with five I/O pipelines: (i) file per process I/O; (ii) MPI collective (parallel write to a single file); (iii) MPI Group Collective (parallel write to a target number of files); (iv) non-balanced aggregation; (v) balanced aggregation (our approach). We configure schemes (iii), (iv) and (v) to generate $nprocs/8$ files. In this set of micro-benchmarks, we allocate data buffers on each rank to mimic a real scientific simulation load. For weak-scaling experiments, we use our I/O simulator to simulate an S3D simulation with 16 variables, resulting in a non-uniform load distribution across processes, as shown in Figure 10(a). We vary the total number of processes from $1,024$ to $16,384$, while the total I/O load varies from 9.5 GiB to 153 GiB per timestep. We repeated each experiment 10 times and plotted the median in Figure 11.

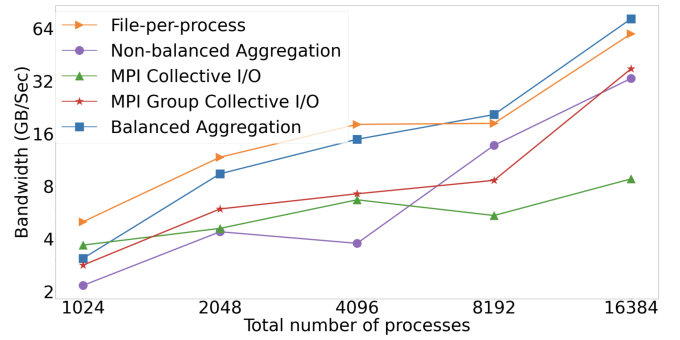


Fig. 11: Weak scaling results of different I/O pipelines, where the per-process workload is created using our micro-benchmark following the distribution of S3D data (Figure 10). The total size of the dataset starts at $1{,}024$ ranks with 9.5GiB.

We observe that our balanced aggregation scheme outperforms all other methods at $16,384$ cores. We report a throughput of 72.75 GiB/second at $16,384$ processes, compared to 59.65 GiB/second for file-per-process I/O. These results demonstrate that our balanced aggregation strategy outperforms other schemes at scale. In particular, the *non-balanced* aggregation and MPI Group Collective I/O experiments both perform a *non-balanced* two-phase I/O pipeline producing the same number of files equal to $nprocs/8$. In the *non-balanced* aggregation pipeline, the number of aggregators is equal to the number of files, while the MPI Group Collective manages the aggregators using an internal heuristic. From the experimental result, we can see that the two approaches have similar performance trends at scale. Unsurprisingly, the MPI Collective I/O presents the worst performance at scale due to the global communication overhead in the data aggregation phase and also by having a large number of processes writing to the same file. Finally, *file-per-process* I/O maintains overall good performance but starts losing efficiency at scale due to the increasingly higher number of files. That's because the Lustre file system can handle large numbers of files but only achieves saturating performance at very high process counts. As a result, aggregation strategies are expected to be ineffective at lower process counts. It is important to note that the tunability of the proposed I/O library permits configuration, so it may perform *file-per-process* I/O and achieve the best performance at a lower scale. Furthermore, if we consider post-process analysis and visualization tasks (typically run with a smaller set of computational resources),

using a large number of files would probably reduce the I/O read performance.

VI. Conclusion

We have presented a compressed hierarchical data layout and efficient parallel I/O scheme suitable for a variety of HPC applications. Hierarchical formats allow fast access to data at different scales, making it favorable for interactive analysis tasks. Writing traditional multi-resolution layouts that create global hierarchies is challenging as it involves expensive synchronization during the aggregation phase. Our layout solves this problem by creating a grid of local hierarchies, called patches, that can be processed and written in parallel without any global synchronization. We identify two load-balancing challenges associated with writing our data layout in parallel, one for per-patch data transformation and compression, and the other for parallel writing of compressed patches. We present a technique to facilitate balanced patch distribution across processes, and a novel aggregation strategy that incorporates sub-filing and creates uniform I/O loads across aggregators. We report an $8\times$ improvement in performance over the default MPI collective I/O at scale. Our proposed balanced aggregation technique can also be applied to other HPC applications that produce non-uniform or sparse data loads. The presented techniques are generic and so can also be integrated with existing I/O libraries such as PnetCDF, parallel HDF5 and ADIOS.

References

[1] 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 *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE, 2011.

[2] S. Kumar, V. Vishwanath, P. Carns, J. A. Levine, R. Latham, G. Scorzelli, H. Kolla, R. Grout, R. Ross, M. E. Papka, J. Chen, and V. Pascucci, "Efficient data restructuring and aggregation for I/O acceleration in PIDX," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11.

[3] C. S. Yoo, E. S. Richardson, R. Sankaran, and J. H. Chen, "A DNS study on the stabilization mechanism of a turbulent lifted ethylene jet flame in highly-heated coflow," *Proceedings of the Combustion Institute*, vol. 33, no. 1, pp. 1619–1627, 2011.

[4] D. Rosenberg, A. Pouquet, R. Marino, and P. D. Mininni, "Evidence for Bolgiano-Obukhov scaling in rotating stratified turbulence using high-resolution direct numerical simulations," *Physics of Fluids*, vol. 27, no. 5, p. 055105, 2015.

[5] P. Lindstrom, "Fixed-rate Compressed Floating-Point Arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[6] 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 *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 413–423.

[7] S. Kumar, S. Petruzza, W. Usher, and V. Pascucci, "Spatially-Aware Parallel I/O for Particle Data," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019.

[8] 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 *Networking, Architecture and Storage (NAS), 2013 IEEE Eighth International Conference on*, 2013.

[9] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A High-Performance Scientific I/O Interface," in *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, Nov 2003, pp. 39–39.

[10] "HDF5 Home Page," http://www.hdfgroup.org/HDF5/.

[11] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, M. Parashar, N. Samatova, K. Schwan, A. Shoshani, M. Wolf, K. Wu, and W. Yu, "Hello ADIOS: The Challenges and Lessons of Developing Leadership Class I/O Frameworks," vol. 26, no. 7, p. 1453–1473, May 2014.

[12] B. Dong, X. Li, Q. Wu, L. Xiao, and L. Ruan, "A dynamic and adaptive load balancing strategy for parallel file system with large-scale I/O servers," *Journal of Parallel and distributed computing*, vol. 72, no. 10, pp. 1254–1268, 2012.

[13] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Seventh Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[14] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig, "Small-File Access in Parallel File Systems," in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, ser. IPDPS '09, 2009, p. 1–11.

[15] 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*, 1993.

[16] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Adaptable, metadata rich IO methods for portable high performance IO," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–10.

[17] K. Gao, W. 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 *2009 International Conference on Parallel Processing*, Sep. 2009, pp. 470–477.

[18] S. Byna, M. Chaarawi, Q. Koziol, J. Mainzer, and F. Willmore, "Tuning HDF5 subfiling performance on parallel file systems," in *Cray User Group*, 2017.

[19] R. Rabenseifner and A. E. Koniges, "Effective File-I/O Bandwidth Benchmark," in *Euro-Par 2000 Parallel Processing*, A. Bode, T. Ludwig, W. Karl, and R. Wismüller, Eds., 2000, pp. 1273–1283.

[20] W. NORCOTT, "Iozone filesystem benchmark," *http://www. iozone. org/*, 2003.

[21] J. Borrill, L. Oliker, J. Shalf, and H. Shan, "Investigation of Leading HPC I/O Performance Using a Scientific-Application Derived Benchmark," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, ser. SC '07, 2007.

[22] H. Shan and J. Shalf, "Using IOR to analyze the I/O performance for HPC platforms," Ernest Orlando Lawrence Berkeley NationalLaboratory, Berkeley, CA (US), Tech. Rep., 2007.

[23] P. van Oosterom and T. Vijlbrief., "The spatial location code." in *International Symposium on Spatial Data Handling (SDH '96)*, 1996, pp. 1–17.

[24] D. Hoang, B. Summa, H. Bhatia, P. Lindstrom, P. Klacansky, W. Usher, P. T. Bremer, and V. Pascucci, "Efficient and Flexible Hierarchical Data Layouts for a Unified Encoding of Scalar Field Precision and Resolution," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 2, pp. 603–613, 2021.

[25] A. Cohen, I. Daubechies, and J. Feauveau, "Biorthogonal bases of compactly supported wavelets," *Communications on Pure and Applied Mathematics*, vol. 45, no. 5, pp. 485–560.

[26] A. Said and W. A. Pearlman, "A New, Fast, and Efficient Image Codec based on Set Partitioning in Hierarchical Trees," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 3, pp. 243–250, 1996.

[27] M. Rabbani, "JPEG2000: Image compression fundamentals, standards and practice," *Journal of Electronic Imaging*, vol. 11, no. 2, p. 286, 2002.

[28] "Introducing Argonne's Theta Supercomputer," vol. 2, no. 3, 7 2017. [Online]. Available: https://www.osti.gov/biblio/1371569