Load-balancing Parallel I/O of Compressed Hierarchical Layouts

Ke Fan
kefan@uab.edu
University of Alabama at Birmingham
Birmingham, Alabama, USA

Duong Hoang
kefan@uab.edu
University of Utah
Salt Lake City, Utah, USA

Steve Petruzza
steve.petruzza@usu.edu
Utah State University
Logan, Utah, USA

Thomas Gilray
gilray@uab.edu
University of Alabama at Birmingham
Birmingham, Alabama, USA

Valerio Pascucci
pascucci@sci.utah.edu
University of Utah
Salt Lake City, Utah, USA

Sidharth Kumar
sid14@uab.edu
University of Alabama at Birmingham
Birmingham, Alabama, USA

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 an 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 enables random-access and multiresolution queries. While multiresolution layouts make it possible to interactively explore 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 which 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 then applied, the compressed data chunks may end up of very 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.

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

KEYWORDS

Parallel I/O, load-balancing, multiresolution, precision

ACM Reference Format:
Ke Fan, Duong Hoang, Steve Petruzza, Thomas Gilray, Valerio Pascucci, and Sidharth Kumar. 2018. Load-balancing Parallel I/O of Compressed Hierarchical Layouts. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 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 the user can efficiently work with the data by loading only the necessary bits. Unfortunately, writing a multiresolution data layout while fully utilizing the hardware bandwidth of a parallel I/O system is hampered by a non-scalable gathering step to collect coarse resolution data which tends to involve communication among processes across the whole simulation domain [16].

We observe that a global hierarchy can also be implicitly constructed by having multiple independent local hierarchies, one for each localized patch of data. This approach avoids the global need to gather coarse resolution data at write time because the patches can be written independently instead. Furthermore, since each patch encapsulates its own hierarchy, I/O schemes have complete freedom on how to organize the patches for optimal I/O performance without the risk of disturbing a global hierarchy. At query time, data at any resolution level can be retrieved and assembled from multiple patches. By tuning the patch size, we can ensure the performance of retrieving coarse resolution data is minimally impacted. Based on these observations, we propose a patch-based data layout for large regular grids which 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, for example with filtering, feature reduction, and compression. In our experiments, to achieve a compact layout as well as high-quality

data filtering at coarse resolution levels, we employ the wavelet transform, followed by compressing the wavelet coefficients with ZFP [19]. Such transformations, intervening between simulation output and I/O, present a load-balancing problem: some processes may need to do significantly more work than others because they host more patches or more data heavy patches. We solve these problem with a novel scheme that (i) distributes data patches among simulation processes so as to ensure processes have similar numbers of patches, and then (ii) performs a load-balancing aggregation phase to balance the data load during I/O across a relatively small number of aggregator processes, while retaining some coarse spatial locality.

The problem addressed by our distribution phase is that of balancing the number of data patches per process, a mapping that greedy approaches can do poorly. Then, as the data itself quickly becomes overwhelmingly large, reduction techniques such as lossy compression, filtering, or feature extraction are being increasingly used before disk I/O. Such techniques create a further load-balancing problem for I/O, as reduction rates may (and often do) vary widely across the simulation domain. As a result, I/O speed depends on the process with the most data to write, resulting in non-optimal performance. To mitigate this problem, our second phase performs load-balanced aggregation. This distributes compressed and reduced data patches across aggregators so as to be even in terms of bytes. Here, we use a relatively smaller number of aggregators before final disk I/O to collect data from simulation processes, while also preserving spatial locality of the patches. Spatial locality is important as nearby patches are likely to be queried together at read time, so storing them close together on disk typically reduces read latency. Beside 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 or particles. 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, beside proposing a solution to this problem in the form of novel distribution and aggregation phases, we also propose 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 smaller 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 supports random access, multiresolution queries.
(2) A patch distribution phase that facilitates load balancing for data transformation while minimizing data movement. Empirical experiments show 2.5x improvement over a non-balancing approach.
(3) An aggregation phase that balances non-uniform data. Empirical experiments show 1.3x 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 extensive experimental evaluation of different I/O pipelines including traditional I/O approaches (e.g., file per process and collective I/O), and more advanced solutions using non-balanced and balanced aggregation schemes.

2 RELATED WORK

Parallel I/O for grid-based and structured datasets have been widely explored, however, few works have focused on parallel I/O for non-uniform load distributions, and in particular for compressed multiresolution data. This is partly due to the challenges of working with non-uniform distribution data loads. 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. This pushes the challenge on the parallel file systems where limited efforts exist to alleviate load unbalance on I/O servers [9]. Popular I/O libraries such as PnetCDF [18], parallel HDF5 [1] and ADIOS [20] are built on top of MPI collective I/O [31], which defaults to data aggregation using two phase I/O. The two-phase I/O in MPI-I/O defaults to one shared file, which often results in sub-optimal performance. On the other hand, our proposed data aggregation system supports subfiling and is designed to work for non-uniform data distributions, where aggregator count and size are selected based on the distribution of data load across processes.

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, PnetCFD, Parallel HDF5, and PIDX, use a suite of I/O transformations to effectively translate distributed-application data layout to file level bit streams. 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 write data to a single shared file. It is well-known in the parallel I/O community that both writing to a single shared file or using the file-per-process mode will lead to sub-optimal performance [15]. 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 [6].

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 [8, 17, 20, 21, 31] 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 needs to access the parallel file system. Subfiling [4, 11] works in a similar manner, though does not necessarily aggregate the data over the network to 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. Sub-filing controls the number of files generated while two-phase I/O with data aggregation controls the total number of processes that access the parallel file system. These two schemes balance between file per process and single shared file approaches to provide a set of portable, scalable, and tunable I/O strategies.

To improve I/O performance and reduce latency for visualization and analysis tasks, a wide range of visualization focused file formats have been proposed. A common trend among all such formats is that they provide both a hierarchical representation, and reorganize the data for better spatial coherence. To visualize large cosmological datasets, Fraedrich et al. [10] construct a multiresolution hierarchy as a post-process task, which optimizes for level of detail and fast read performance. Similar efforts by Reichl et al. [24], Schatz et al. [28] and Rizzi et al. [25] have presented interactive, level of detail based visualization methods for large scale cosmology simulations, all based on either a post-process data conversion [24, 28] or a re-sorting process, which is run when the visualization task starts [25].

Finally, in terms of performance assessment, several tools exist to characterize the message passing capabilities of a system [23] [22] and assess their peak performance. A few of them, like MADBench2 [3] assess I/O performance using application-driven I/O loads, for example simulating Cosmic Microwave Background data analysis. One of the most common tool used to perform I/O benchmarks is IOR [29], which allows to define 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 perform two-phase I/O and subfiling together is very limited and benchmarks for uneven data distribution is generally not supported.

3 COMPRESSED HIERARCHICAL LAYOUT

In this section we introduce our novel hierarchical layout for scientific data and discuss how it is designed to facilitate fast parallel writes while allowing flexible multiresolution reads. We start with the observation that traditional hierarchical layouts [16] 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 must avoid such global synchronization and let processes perform I/O operations in parallel as much as possible [14].

Patch-based design. To mitigate the global synchronization problem, we design our data layout to be patched-based. 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 ($L \leq \log_2(\min(p_x, p_y, p_z)) + 1$). Data samples are assigned to files in units of patches (i.e., samples from the same patch are always written in the same file). In each file, the samples are sorted by resolution levels (coarse to fine), then by patch number. $p_{x,y,z}$ and L are controlled by the user. 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 own I/O library sorts the patches in Morton order [32], enforces that patch numbers in the same file are contiguous in this order, and allows the user to choose the number of files F. Figure 1 illustrates a 2D example of such a layout with $p_x = p_y = 4$, $L = 3$ and $F = 4$; it clearly shows that compared to a global hierarchy approach, our patch-based design enables I/O libraries to achieve localized network communication when gathering data for each file, resulting in more scalable I/O.

A possible reason for concern with our patch-based design is that since coarse resolution data samples are spread across more files, a query for low-resolution 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. This process may be very slow depending on the size of each chunk. However, it is easy to appropriately choose reasonable values for $p_{x,y,z}$, L and the number of chunks per file 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. 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.

Data access and reconstruction In each file, we store the patch numbers and their offsets for each resolution level. 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.

Note that as described, only data up to the requested resolution level is loaded from disk, and that random access is supported at the patch level (i.e., patches can be loaded independently but same-resolution data within a patch is always read together). Because data samples in each file are sorted first by resolution level, progressive reads of the files are possible and fast, as a linear scan of each file from beginning to end corresponds to progressively loading finer and finer resolution levels for the corresponding patches. Furthermore, since the patches can be independently refined, our data layout supports adaptive reconstruction of the whole domain at patch level, when certain regions containing quantities of interest may require more resolution to resolve than others. For certain analysis tasks such as volume rendering, such adaptive reconstruction can significantly reduce the amount of data loaded and processed.

Although not mandated by the data layout, in practice data is often transformed and compressed in some way before being stored on disk. For example, our own I/O library performs the wavelet transform followed by lossily compressing the wavelet coefficients (Section 4.2). In particular, at write time we transform and compress each patch independently of one another, and at read time each is independently decompressed and inverse transformed back. 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. On the other hand, they must be large enough to not incur too much metadata that we need in order to support random access and adaptive refinement. We have found 32 and 64 to be good choices for $p_{x,y,z}$, that is, a typical patch contains 32^3 or 64^3 samples. In our experiments, with such patch sizes the metadata takes only X percent 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.

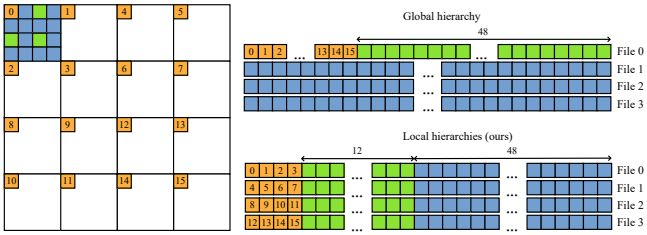


Figure 1: As an example, a 16×16 simulation domain is divided into 16 patches of size 4×4 each. The grid samples are transformed into three resolution levels (orange, green, and blue). For clarity we only show the individual grid samples for the first patch. In the global hierarchy (top right), when four files are written, the first file contains coarse-level data samples from the entire domain, forcing expensive global communication. In our layout consisting of four local hierarchies (bottom right), each file only gathers data from four neighboring processes, making network communication more scalable.

4 PARALLEL I/O

As discussed in Section 3, we have chosen to design our layout to be patch-based, so as to avoid global communication often found in other multiresolution schemes. 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 [13]. We have also chosen the B-spline multilinear basis [7], because they are fast to compute and offer great compression opportunities. Each patch then becomes our unit of computation, as each 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 straddled between two or more processes which necessitates data communication during data transformation. The simulation code however may distribute data in ways that result in many straddling 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 a whole number of patches.

Ideally in a two-phase I/O system, the aggregation phase can serve as a patch distribution phase as well, and subsequent transform 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 patch distribution phase, with the aim of distributing the patches evenly among all processes so that the subsequent transform 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 compress to different sizes, depending on the data being compressed. 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 4.1)
(2) Parallel wavelet transform and compression (Section 4.2)
(3) Balanced data aggregation and file I/O (Section 4.3)

4.1 Balanced patch distribution

We start by distinguishing *regular patches* and *shared patches*. A regular patch is fully contained within a process before distribution, while a shared patch is shared among two or more processes (i.e,. each process holds some portion of the patch). To avoid excessive data movement, regular patches are not moved by the distribution process. Shared patches, on the other hand, may need to be moved from its original process to a new process for balancing purpose. Suppose the total number of processes is N and the number of patches is M. With a balanced distribution, on average each process holds N/M patches (typically $N > M$).

For each process, we compute the target number of patches that the process will hold after distribution. First, let each process hold $\lfloor N/M \rfloor$ patches. We divide the remaining N mod M patches equally among the nodes. Suppose each node receives K more patches, then for each node, these K patches are then assigned evenly to the first K processes in the node (each such process receives one more patch). With this scheme we aim to distribute the patches as evenly as possible to the processes and nodes, as the target number of patches for each process either hold $\lfloor N/M \rfloor$ or $\lfloor N/M \rfloor + 1$. To execute the scheme, we iterate through the shared patches one by one and at each step, assign the current shared patch to one process among the processes that share the patch. The chosen process must currently contains fewer patches than its target number of patches

(either $\lfloor N/M \rfloor$ or $\lfloor N/M \rfloor + 1$, computed as above). If there are multiple candidates, the process with the smallest rank is chosen.

We divide the set of processes sharing a patch into senders and receiver. The receiver is the one process that the patch is assigned to, the rest are senders. Each sender then sends the region of patch that it holds to the receiver, which at the end contains the entire patch. This algorithm is run on every process, so each one knows exactly whether it is a receiver or sender for each shared patch, and exactly what data to send or receive. `MPI_Type_create_subarray` is used to define these sub-patch regions that are sent and received over the network, and MPI's non-blocking point to point API (`MPI_Isend` and `MPI_Irecv`) is used for data transfer. In Figure 2, we give a small example where we number the sub-patch regions to help visualizing them. After all the sub-patch regions are sent and received, every process holds a whole number of patches and each patch 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 [16], albeit not for balancing the data transformation per patch, but to minimize interleaving of data samples among processes in aggregation buffers. However, in Figure 2 we show that the greedy scheme leads to very imbalanced patch distribution. In Section 5 we also show through experiments that in practice, the greedy scheme does indeed result in significantly longer computation time for wavelet transform and compression.

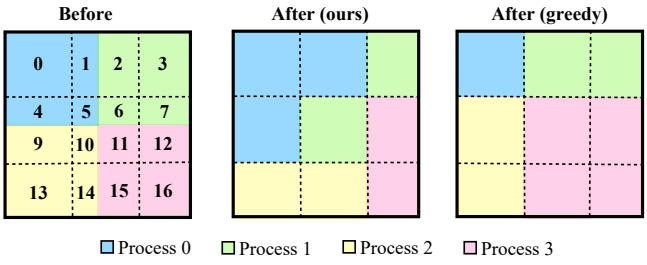


Figure 2: A 2D example illustrating our patch distribution scheme. 3×3 patches (separated by dash lines) are distributed among four processes (distinguished by colors). Before the distribution, some patches are shared among processes, resulting in many sub-patch regions which are numbered. After the distribution using our scheme, the processes have exactly 3, 2, 2, 2 patches respectively. In constrast, the greedy scheme, which picks the process with the largest sub-patch region for each shared patch, distributes the patches unevenly. Note that boundary patches may not have the same dimensions as the rest of the patches, we nevertheless treat them in the same way.

4.2 Wavelet transform and compression

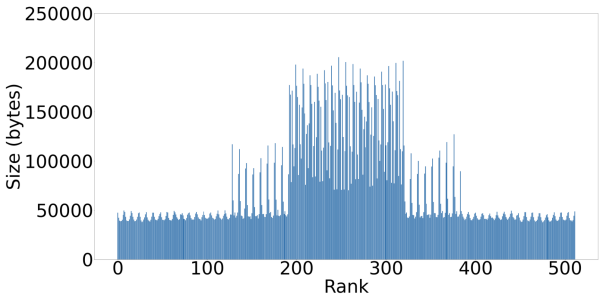
After patches are evenly distributed to the processes, wavelet transform is performed independently for each patch. The transform updates and separates the data samples in each patch into a set of resolution levels, each captures information at a particular scale. Note that since the patches are processed independently, once the patch distribution is done, we do not need communication among processes, for example to exchange boundary data, for the wavelet transform to happen. This deliberate design is to extract the most parallelism from the machine for the transformation step, but it also means wavelet coefficients at patch boundaries are incorrectly computed. To handle the discontinuity at patch boundaries, common solutions include 0-padding and mirroring. A better solution is the linear-lifting extrapolation approach introduced in [13], which has been shown to be very effective at eliminating artificially large wavelet coefficients which produce blocky artifacts when data is reconstructed for each patch independently.

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 [27, 30] but they tend to be slow. For performance reason, we use ZFP [19] – a fast compressor for floating-point arrays – to compress wavelet coefficients in each resolution level independently. Prior work [13] 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 5, we provide detailed timings of this step with our implementation.

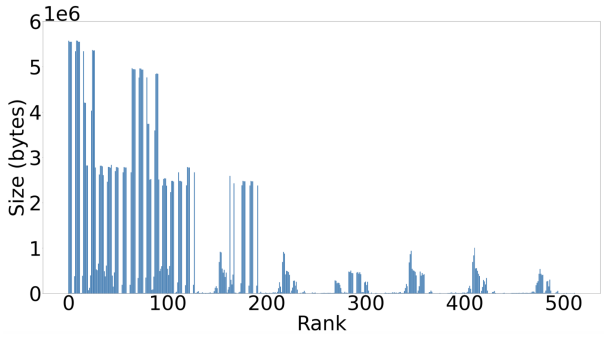
4.3 Layout-aware balanced data aggregation

The next step after every patch is transformed and compressed is to write the compressed data to disk, in the file layout described in Section 3. For this purpose, existing parallel I/O schemes suffer from load-balancing problems unique to writing compressed data layouts such as ours. 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 process 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 with time, necessitating an adaptive parallel I/O system. We show this imbalance for real world scientific datasets in Figure 3. As can be seen in the figure, most of the datasets exhibits considerable variation in load across processes.

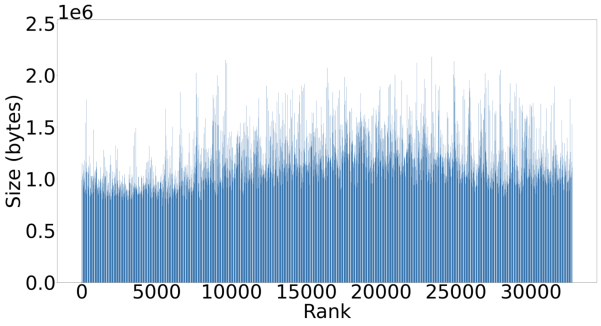
We have designed our data layout in a manner that inherently supports writing to a tunable number of files (F). Although MPI has support for collective I/O which internally does data aggregation, it explicitly does not support sub-filing. In order to achieve sub-filing with MPI's collective I/O, one needs to create local sub-groups of processes that have their own communicators. The processes with the subgroup then coordinates access to write to a shared file collectively. In this setting, the total number of local communicators



(a) Magnetic resonance dataset [12]



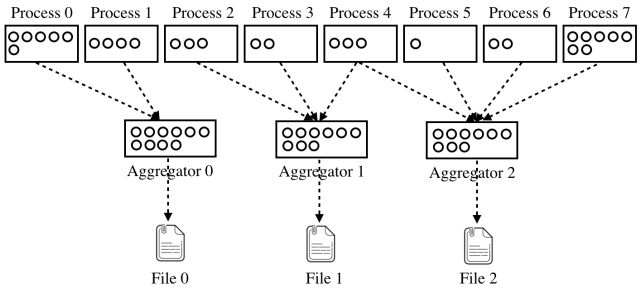
(b) S3D flame dataset [33]



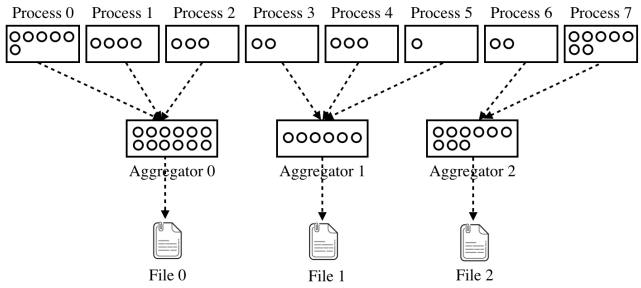
(c) Turbulence dataset [26]

Figure 3: Size of data held by processes after wavelet transform and ZFP compression for three different datasets demonstrating the imbalanced nature of data distribution. While the size distribution pattern varies across datasets, we see considerable range in size across processes for all of them. For example, the S3D flame dataset has a range of 0.1 – 5 megabytes and the turbulence dataset has a range of 1 – 2 megabytes.

equals the total number of files written. This scheme 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 [5]. However, this method does not directly translate well for non-uniform data distributions, as it leads to fewer aggregators writing more data than other methods, causing sub-optimal performance. In order to extract maximum available bandwidth from the I/O hardware, we must have a uniform distribution of I/O load across aggregators.



(a) Balanced aggregation



(b) Non-balanced aggregation

Figure 4: Diagram showing our two aggregation phases. Both approaches yields same number of files, however, balanced aggregation generates files of same size as opposed to non-balanced scheme where files have varying sizes.

In order to deal with load-balancing challenges, we have design a customized two-phase IO 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 aggregators. Following are the steps in the aggregation phases:

(1) Aggregator selection
(2) Patch assignment
(3) Patch transfer

Aggregator selection. A key step in tailoring aggregation is to select appropriate number of aggregators and files. Given that we have designed our data layout to support a flexible number of files, we keep that as a tunable parameter that is set by the user. The only constraint we impose is that the total number of files outputted should be less than equal or equal to the total number of processes. In practice, the number of file 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. We allow a file to be written by only one aggregator process, therefore making the aggregator count equal to the total number of file being written. 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 makes unaligned

accesses to a file. In order to extract maximum I/O bandwidth, we place the aggregators uniformly across the rank space. In Section 5, we demonstrate the impact of number of aggregators/files.

Patch assignment. In order 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 different number of patches to achieve a balanced amount of data per-file. In order to end up with similar-sized files we enable the aggregator to receive data from a varying number of processes: many lighter-weight processes can be aggregated into the same file while fewer heavier-weight processes are likewise collated into a single file. We begin by collecting patch sizes across processes using MPI's `MPI_Allgather`. As we wish to write out patches in Morton order [32], to preserve the spatial locality of patches in the file, we first sort all patches in Morton order, then scan over this sorted list allocating patches to aggregator processes in a balanced fashion. We keep a running sum of compressed patch sizes and progress to the next aggregator process whenever this value passes 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.

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 uses the same step to identify the patches and correspondingly the ranks it is going to receive the data from. This allows the aggregator processes to correctly allocate buffers to accomodate the receiveing 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 in Figure 4(a). It can be seen that our scheme can supports both sub-filing and also perform uniform sized I/O writes. We compare the performance of our approach with a non-balanced aggregation scheme, that generates the same number of files as the balanced aggregation approach, but writes to non-uniform sized files. In the non-uniform aggregation scheme, every aggregator receives patches from same number of processes, and since the processes have varying data loads, the aggregators end up getting a non-uniform load distribution. This approach can be seen in Figure 4(b).

5 EVALUATION

We begin by evaluating the efficacy of our data balanced data distribution and balanced aggregation phases. We have used a mix of synthetic and real simulation datsets (see Table 1) for our experiments. All our experiments are performed on the Theta Supercomputer [2] 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 of online disk storage. The supercomputer has Dragonfly network topology and a Lustre filesystem.

Dataset Name	Resolution	Size after compression	PSNR
Magnetic resonance [12]	512 x 512 x 512	64311392	37.7
S3D flame [33]	2025 x 1600 x 400	786616672	35.6
Turbulence data [26]	4096 x 4096 x 4096	38989451056	33.9

Table 1: Datasets used in our experiments

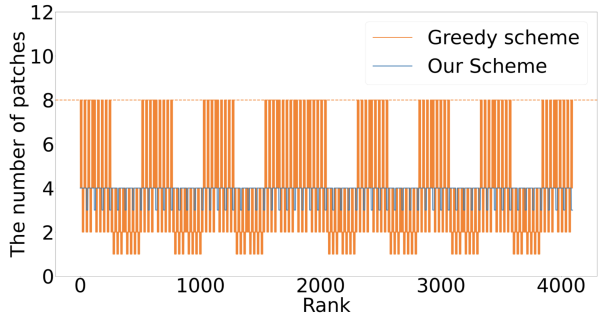
5.1 Patch distribution phase

In our I/O pipeline, patch distribution happens first, followed by a parallel-wavelet transform and compression (for each patch). Here we compare our balanced patch distribution scheme with the greedy scheme discussed in Section 4.1. Patch distribution using the greedy scheme has been used previously, for example in [16], but not for balancing the data transformation, per patch, but to minimize interleaving of data samples among processes in the aggregation buffer. 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 of resolution 1600^3 and perform a strong-scaling experiment while varying the total number of processes from 512 to 4,096. The patch dimensions are 64^3, resulting in a total of $1600^3/64^3 = 15{,}625$ patches.

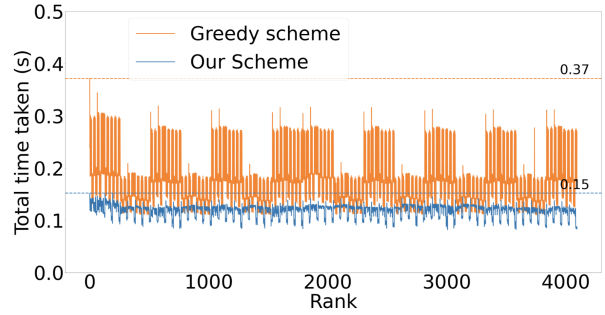
At 4,096 processes, our scheme distributes the patches more uniformly (Figure 5a). With the greedy scheme, approximately a quarter of the processes hold eight patches each while a significant number of processes holds only one patch, as opposed to our scheme, where every process holds either three or four patches. The near perfect balance not only makes the data distribution phase fast, it also directly impacts the following wavelet-transform and compression phases (Figure 5b). With the greedy scheme, processes with patches must perform more computation overall, causing performance degradation. We plot the results for strong scaling in Figure 5c. Our approach takes 1s at 512 processes and 0.13s at 4,096 processes, demonstrating that it achieves near perfect scaling efficiency, while the greedy approach does not.

5.2 Load-balanced data aggregation

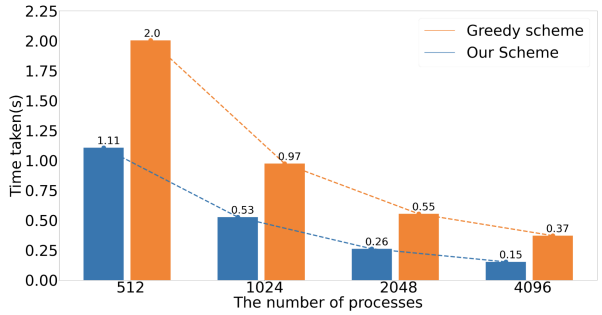
Here we demonstrate the efficacy of our scheme for load-balanced aggregation. We compare our method with a non-balanced aggregation approach that simply assign an equal number of patches to each aggregator without considering the (compressed) patch size. Recall that our balanced aggregation scheme ensures that aggregators have similar I/O loads and so, as a consequence, files also have similar sizes; this is not true for non-balanced aggregation schemes. We evaluate our scheme on S3D flame and Turbulence dataset listed in Table 1, the experiments are performed at 4,096 and 32,768 processes, respectively. For the $nprocs = 32{,}768$ process experiments, we vary the total number of aggregators/files from n (file-per-process) to $n/8$. For the $nprocs = 4{,}096$ process experiments, we vary the total number of aggregators/files from n



(a) With out scheme, patches are evenly distributed across the process ranks (4, 096 process run)

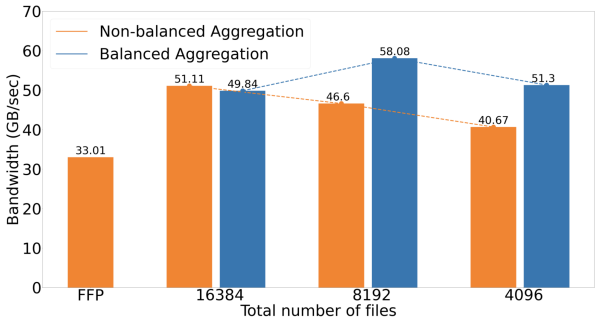


(b) The balanced patch distribution results in a significant speedup factor (4, 096 process run)

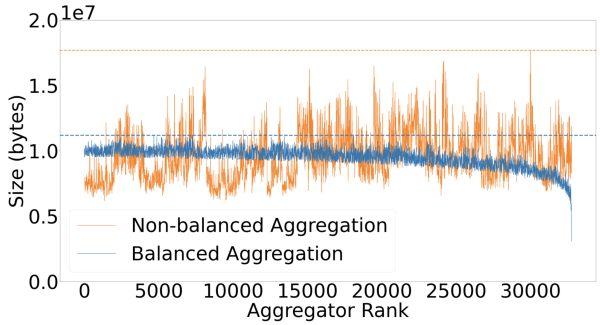


(c) Our distribution scheme also achieves near perfect strong scaling

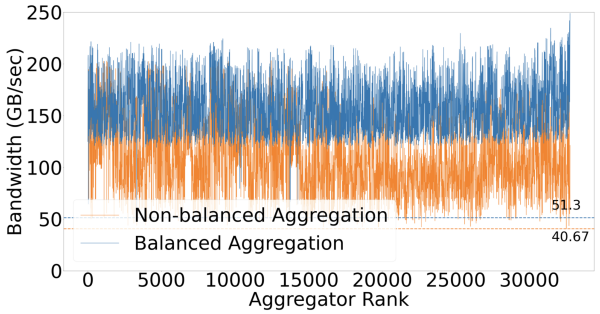
Figure 5: Compared to the greedy patch distribution scheme, our balanced scheme reduces the total combined time of the distribution, wavelet transform, and compression steps by more than half. It also exhibits near perfect strong scaling behavior.



(a) Write bandwidth for turbulence [26] dataset at 32,768 processes. FFP stands for file-per-process I/O mode.



(b) Size of I/O load for all aggregators for the 4096 aggregator run



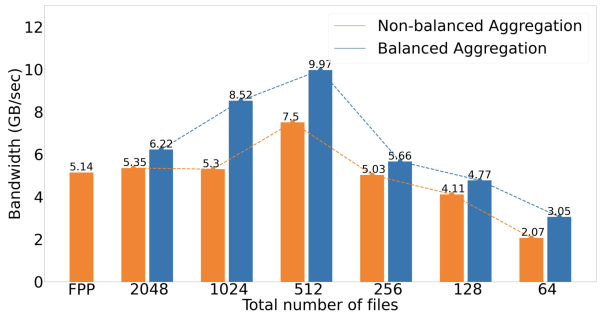
(c) Write bandwidth for all aggregators for the 4096 aggregator run

Figure 6: Results showing the impact of balanced data aggregation for writing turbulence [26] dataset.

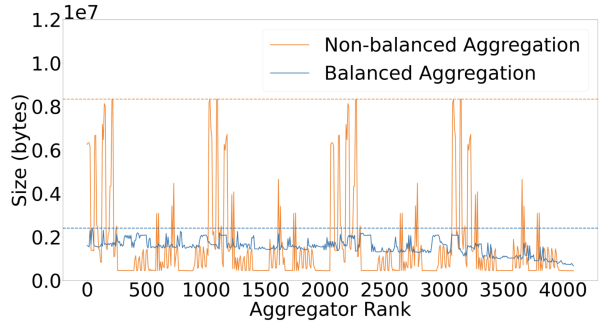
(file-per-process) to $n/16$. We perform 10 iterations for all experiments and plot the bandwidth of the median run in Figure 6(a) and 7(a).

For the 4096^3-resolution turbulence dataset our load-balanced aggregation approach yields a peak throughput of 58 GiB/second while the unbalanced approach yields a peak throughput of 51 GB/second. File-per-process I/O mode yields a throughput of 33 GiB/second. These numbers correspond to a 13% improvement in performance over the non-balanced approach and a 90% improvement over file-per-process I/O. For the S3D flame dataset, we observe a maximum bandwidth of 9.97 GiB/second with load-balanced aggregation, and 7.5 GiB/second with the non-balanced aggregation, while file-per-process I/O yields a throughput of 5.14 GB/second. These correspond to a 30% improvement in performance over non-balanced aggregation and around 2× improvement over file-per-process I/O.

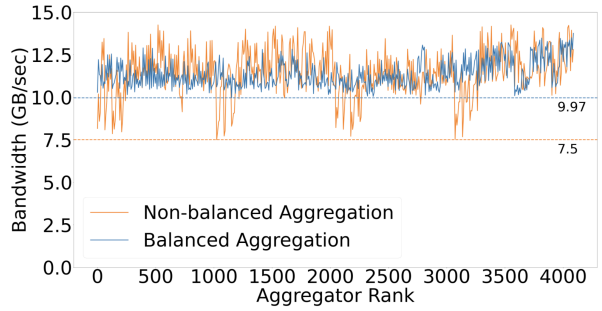
The improvement (percentage) in the flame dataset is more than in the turbulence dataset. The result can be explained from Figure 6(b-c) where we plot the data load and bandwidth of all aggregators for the 4,096-aggregator run of the turbulence dataset, and Figure 7(b-c) which plots similar metric for the 256-aggregator run of the S3D flame datset. We observe that the degree of non-uniformity of data distribution varies for the two datasets. With



(a) I/O bandwidth for S3D flame dataset [33] at 4,096 processes. FPP stands for file-per-process I/O mode.



(b) Size of I/O load for all aggregators for the 512 aggregator run



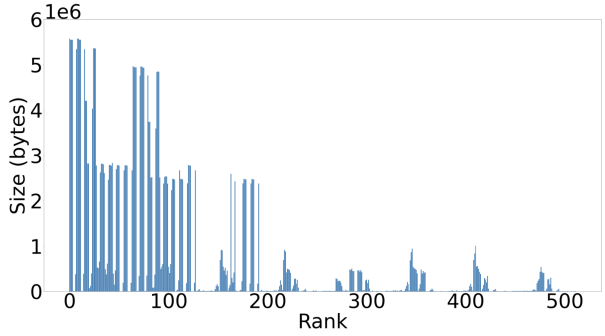
(c) Write bandwidth for all aggregators for the 512 aggregator run

Figure 7: Results showing the impact of balanced data aggregation for writing turbulence [33] dataset.

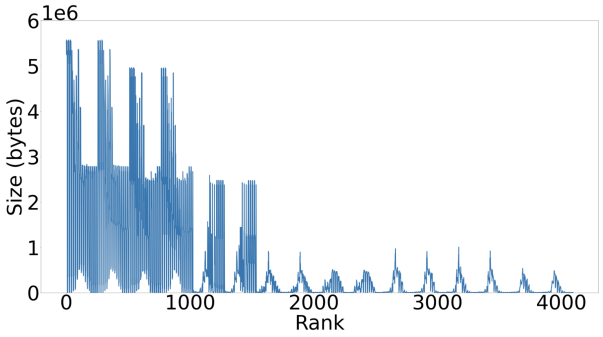
the flame dataset, the non-balanced scheme results in groups of aggregators having proportionally much more data than the others. Whereas, in the turbulence data, the degree of imbalance in data distribution across aggregators is more moderate, even using the non-balanced scheme. This difference in non-uniformity of data in initial distributions leads to different degrees of performance improvement, but in both cases we may note that our approach leads to a more uniform distribution of data across aggregators.

We also observe that with our balanced scheme, we attain a maxima in performance at aggregator count 8,192 and 512 for the two datasets. Finding the right number of aggregators is a difficult problem, as it often depends on the scale of the experiment and in our case also on the load distribution pattern. However, we recognize that the Lustre filesystem is more suited to write data in file-per-process mode and is adept at handling 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 factor, 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.

5.3 Micro-benchmarks



(a) Original data distribution of S3D flame data at 512 processes



(b) Data distribution generated by our method at 4096 processes

Figure 8: Micro-benchmark load distributions. (a) load distribution of the S3D flame dataset after parallel wavelet transform and compression using 512 ranks, (b) load distribution generated from our micro-benchmark tool for 4096 processes. The generated data preserves the load distribution patterns in the original data representing a great candidate for benchmark at scale.

Traditional I/O benchmarks use static, sometimes uniform, data distributions replicated across cores to assess systems I/O bandwidth. Otherwise they would require a simulation to run and generate representative data loads, or an extensional database requiring significant storage. In this work, we introduce a micro-benchmark technique that allows us to better mimic a real scientific application's data-load distribution patterns at scale. The main idea is to extrapolate data distribution patterns from existing applications and reproduce them at different scales. This approach is inspired by common practices in simulation design, where scientists initially prototype small scale simulation runs using a coarse resolution

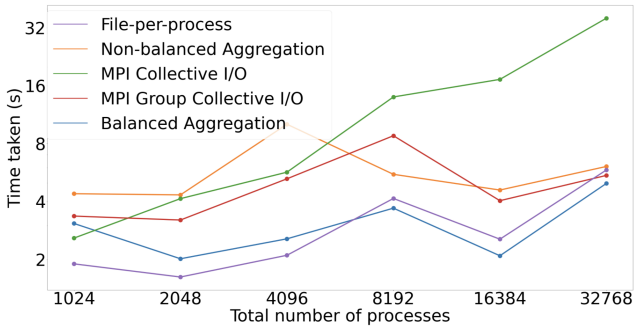


Figure 9: Execution time of different I/O pipelines. In this weak scaling study the total size of the dataset starts at 1,024 ranks with 9.5GB. These results demonstrate that our balanced aggregation presents the best performance at scale while other approaches lose efficiency.

grid and only later increase the resolution of each patch to perform large-scale simulations. For our micro-benchmarks we first consider a scientific dataset and collect the data size that each rank holds in the original 3D grid domain at small scale. With this information we launch our micro-benchmark at a larger scale (i.e., using a larger number of cores) and assign to each rank in the new 3D grid domain a data size. This data size is computed in parallel on the fly with trilinear interpolation using the data size values in the original grid. At this stage each rank allocates a buffer of the assigned data size and execute a given I/O pipeline.

In Figure 8 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.

We performed micro-benchmark 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 mimic a S3D simulation that creates 16 variables, using our I/O simulator generating a non-uniform load distribution across processes as shown in Figure 8(a). We vary the total number of processes from $1,024$ to $32,768$, while varying the total I/O load goes from 9.5 GB at to 297 GB per timestep. We perform 10 iterations of all our runs and report the median in Figure 9.

We observe that our balanced aggregation schemes outperforms all other methods at $32,768$ cores. We report a throughput of 60 GB/seecond at $32,768$ processes, compared to 51 GB/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 uses an internal heuristic to manage the aggregators. From the experimental result we can notice how the those two approaches have similar performance trend 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 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. It is important to consider 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 lower scale. Furthermore, if we consider post-process analysis and visualization tasks (generally executed using a smaller set of computational resources) using a large number of files would probably reduce the I/O read performance.

6 CONCLUSION

We've 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× 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] [n.d.]. HDF5 Home Page. http://www.hdfgroup.org/HDF5/.
[2] [n.d.]. Introducing Argonne's Theta Supercomputer. ([n. d.]). https://www.osti.gov/biblio/1371569
[3] Julian Borrill, Leonid Oliker, John Shalf, and Hongzhang Shan. 2007. Investigation of Leading HPC I/O Performance Using a Scientific-Application Derived Benchmark. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing* (Reno, Nevada) *(SC '07)*. Association for Computing Machinery, New York, NY, USA, Article 10, 12 pages. https://doi.org/10.1145/1362622.1362636
[4] Suren Byna, Mohamad Chaarawi, Quincey Koziol, John Mainzer, and Frank Willmore. [n.d.]. Tuning HDF5 subfiling performance on parallel file systems. ([n. d.]). https://www.osti.gov/biblio/1398484
[5] Suren Byna, Mohamad Chaarawi, Quincey Koziol, John Mainzer, and Frank Willmore. 2017. Tuning HDF5 subfiling performance on parallel file systems. In *Cray User Group*.
[6] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. 2009. Small-File Access in Parallel File Systems. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*

(IPDPS '09). IEEE Computer Society, USA, 1–11. https://doi.org/10.1109/IPDPS.2009.5161029

[7] A. Cohen, Ingrid Daubechies, and J.-C. Feauveau. [n.d.]. Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics* 45, 5 ([n. d.]), 485–560.

[8] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. 1993. Improved Parallel I/O via a Two-phase Run-time Access Strategy. *SIGARCH Comput. Archit. News* (1993).

[9] Bin Dong, Xiuqiao Li, Qimeng Wu, Limin Xiao, and Li Ruan. 2012. A dynamic and adaptive load balancing strategy for parallel file system with large-scale I/O servers. *Journal of Parallel and distributed computing* 72, 10 (2012), 1254–1268.

[10] Roland Fraedrich, Jens Schneider, and Rüdiger Westermann. 2009. Exploring the Millennium Run–Scalable Rendering of Large-Scale Cosmological Datasets. *IEEE Transactions on Visualization and Computer Graphics* 6 (2009).

[11] K. Gao, W. Liao, A. Nisar, A. Choudhary, R. Ross, and R. Latham. 2009. Using Subfiling to Improve Programming Flexibility and Performance of Parallel Shared-file I/O. In *2009 International Conference on Parallel Processing*. 470–477. https://doi.org/10.1109/ICPP.2009.68

[12] Fan Guo, Hui Li, William Daughton, and Yi-Hsin Liu. 2014. Formation of Hard Power-laws in the Energetic Particle Spectra Resulting from Relativistic Magnetic Reconnection. *Physical Review Letters* 113 (2014). Issue 15. arXiv:arXiv:1405.4040

[13] D. Hoang, B. Summa, H. Bhatia, P. Lindstrom, P. Klacansky, W. Usher, P. T. Bremer, and V. Pascucci. 2021. Efficient and Flexible Hierarchical Data Layouts for a Unified Encoding of Scalar Field Precision and Resolution. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2021), 603–613. https://doi.org/10.1109/TVCG.2020.3030381

[14] S. Kumar, D. Hoang, S. Petruzza, J. Edwards, and V. Pascucci. 2017. Reducing Network Congestion and Synchronization Overhead During Aggregation of Hierarchical Data. In *IEEE International Conference on High Performance Computing (HiPC '17)*. 223–232.

[15] Sidharth Kumar, Steve Petruzza, Will Usher, and Valerio Pascucci. 2019. Spatially-Aware Parallel I/O for Particle Data. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 84, 10 pages. https://doi.org/10.1145/3337821.3337875

[16] 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. 2012. 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*. 1–11. https://doi.org/10.1109/SC.2012.54

[17] Sidharth Kumar, Venkatram Vishwanath, Philip Carns, Brian Summa, Giorgio Scorzelli, Valerio Pascucci, Robert Ross, Jacqueline Chen, Hemanth Kolla, and Ray Grout. 2011. PIDX: Efficient parallel I/O for multi-resolution multi-dimensional scientific datasets. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE.

[18] Jianwei Li, Wei keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. 2003. Parallel netCDF: A High-Performance Scientific I/O Interface. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. 39–39. https://doi.org/10.1109/SC.2003.10053

[19] P. Lindstrom. 2014. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2674–2683.

[20] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, Manish Parashar, Nagiza Samatova, Karsten Schwan, Arie Shoshani, Matthew Wolf, Kesheng Wu, and Weikuan Yu. 2014. Hello ADIOS: The Challenges and Lessons of Developing Leadership Class I/O Frameworks. 26, 7 (May 2014), 1453–1473. https://doi.org/10.1002/cpe.3125

[21] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. 2009. Adaptable, metadata rich IO methods for portable high performance IO. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–10. https://doi.org/10.1109/IPDPS.2009.5161052

[22] WilliamD NORCOTT. 2003. Iozone filesystem benchmark. *http://www. iozone. org/* (2003).

[23] Rolf Rabenseifner and Alice E. Koniges. 2000. Effective File-I/O Bandwidth Benchmark. In *Euro-Par 2000 Parallel Processing*, Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1273–1283.

[24] Florian Reichl, Marc Treib, and Rüdiger Westermann. 2013. Visualization of Big SPH Simulations via Compressed Octree Grids. In *IEEE International Conference On Big Data*.

[25] Silvio Rizzi, Mark Hereld, Joseph Insley, Michael E. Papka, Thomas Uram, and Venkatram Vishwanath. 2015. Large-Scale Parallel Visualization of Particle-Based Simulations Using Point Sprites and Level-Of-Detail.

[26] D. Rosenberg, A. Pouquet, R. Marino, and P. D. Mininni. 2015. Evidence for Bolgiano-Obukhov scaling in rotating stratified turbulence using high-resolution direct numerical simulations. *Physics of Fluids* 27, 5 (2015), 055105. https://doi.org/10.1063/1.4921076 arXiv:https://doi.org/10.1063/1.4921076

[27] A. Said and W. A. Pearlman. 1996. A New, Fast, and Efficient Image Codec based on Set Partitioning in Hierarchical Trees. *IEEE Transactions on Circuits and Systems for Video Technology* 6, 3 (1996), 243–250.

[28] Karsten Schatz, Christoph Muller, Michael Krone, Jens Schneider, Guido Reina, and Thomas Ertl. 2016. Interactive Visual Exploration of a Trillion Particles. In *LDAV*.

[29] Hongzhang Shan and John Shalf. 2007. *Using IOR to analyze the I/O performance for HPC platforms*. Technical Report. Ernest Orlando Lawrence Berkeley NationalLaboratory, Berkeley, CA (US).

[30] David S. Taubman and Michael W. Marcellin. 2001. *JPEG 2000: Image Compression Fundamentals, Standards and Practice*. Springer.

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

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

[33] Chun Sang Yoo, Edward S. Richardson, Ramanan Sankaran, and Jacqueline H. Chen. 2011. A DNS study on the stabilization mechanism of a turbulent lifted ethylene jet flame in highly-heated coflow. *Proceedings of the Combustion Institute* 33, 1 (2011), 1619–1627.